Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations

Ioannis Schoinas
Babak Falsafi
Mark D. Hill
James R.Larus
Christopher E. Lukas
Shubhendu S. Mukherjee
Steven K. Reinhardt
Eric Schnarr
David A. Wood

Technical Report #1307

March 1996

Submitted for publication. Contact first author for citation information.

Implementing Fine-Grain Distributed Shared Memory On Commodity SMP Workstations

Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, Christopher E. Lukas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnarr, David A. Wood

Computer Sciences Department
University of Wisconsin
1210 West Dayton Street
Madison, WI 53706 USA
wwt@cs.wisc.edu

This paper reports our experience implementing the Blizzard fine-grain distributed shared memory system on a network of unmodified dual-processor workstations running a commercial operating system. The paper describes and measures: three fine-grain access control mechanisms (optimized software, commodity hardware, and custom hardware); a low-latency, user-level communication layer; kernel support in a commercial operating system; and techniques to exploit multiprocessor (SMP) nodes. The results show that a network of workstations can effectively support fine-grain shared memory, but that high performance requires either custom network hardware or custom coherence protocols.

1 Introduction

This paper reports our experience implementing the Blizzard fine-grain distributed shared memory system on the COW (Cluster of Workstations). The COW consists of 40 Sun SPARCStation 20, each of which contains two 66 Mhz Ross HyperSPARC processors [32] and a Myricom Myrinet [4] interface.[1] This paper explores alternative implementation techniques for fine-grain distributed shared memory on commodity hardware and software. Specifically, the paper describes and measures: three fine-grain access control mechanisms (optimized software, commodity hardware, and custom hardware); a low-latency, user-level communication layer; kernel support in a commercial operating system; and techniques to exploit multiprocessor (SMP) nodes. The results show that a network of workstations can effectively support fine-grain shared memory, but that high performance requires either custom network hardware or custom coherence protocols.

A previous fine-grain distributed shared memory system—Wisconsin Blizzard [33]—ran on a Thinking Machines CM-5 massively parallel processor. That platform, unlike COW, consisted of single processor nodes running a minimal kernel without virtual memory or multiprogramming. However, CM-5 communication had much lower latency and overhead. Blizzard's platform is a group of unmodified commercial

1. Shortcomings in first-generation Myrinet switches limited measurements in this paper to 16 SMP nodes. These switches will be replaced by new switches (after this paper deadline!), which we hope will permit larger configurations.

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, NSF PYI Award CCR-9157366, NSF Grant MIP-9225097, an AT&T graduate fellowship, two IBM COOP fellowships, and donations from A.T.&T. Bell Laboratories, Digital Equipment Corporation, Sun Microsystems, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School. 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.

workstations running an unmodified operating system. The systems differ in other ways as well. For example, Blizzard's software access control introduced 15 instructions (18 cycles) before loads and stores, while Blizzard's software access control requires 3 instructions (3 cycles) before far fewer instructions. Furthermore, Blizzard also can use a small amount of custom hardware for access control. This paper describes the techniques—efficient fine-grain access control, operating system support for access control, efficient user-level communication, and multiprocessor (SMP) nodes—necessary to implement fine-grain distributed shared memory on a network of workstations.

Networks of workstations (NOWs) are often considered a distant—and poor—cousin of massively parallel computers. However, the demise of MPP vendors and the advent of fast microprocessors and low latency, high bandwidth networks has created the potential of using NOWs as affordable parallel computers. Shared memory simplifies the difficult task of programming these machines by providing a shared address space that enables programmers to uniformly reference data throughout a system. Shared memory typically requires hardware support. However, previous distributed shared memory systems, such as Ivy, Munin, and Treadmarks [21,9,11], relied instead on virtual memory page protection hardware to detect and control shared references. The large size of memory pages typically causes excessive false sharing, which later systems mitigate with relaxed memory consistency models. Unlike these other systems, Blizzard provides fine-grain access control, which permits user-level software to detect and regulate memory access at cache-block granularity (e.g., 32–64 bytes) and to implement sequentially consistent (transparent) shared memory.

This paper make several contributions:

- **Fine-grain access control alternatives.** This paper extends previous work with several new architectural and compiler techniques for fine-grain access control (Section 4). We implemented ECC-based access control under a commercial operating system, showed that compiler-style program analysis can greatly reduce the overhead of software-based access control, and used a simple hardware access control accelerator [31]. Since these alternatives shared a common platform, this paper can compare their performance on both microbenchmarks and full applications.

- **Commodity operating system support.** Throughout this work, a constant constraint was to use unmodified commodity hardware and software. Blizzard runs under a Solaris 2.4 kernel, which it extends with loadable device drivers. Section 5 discusses this approach and the trade-offs that it entails.

- **Efficient user-level network support.** A serious impediment to NOWs is the long latency and high overhead of commodity communication hardware and software. Long latencies favor bulk communication, which is the antithesis of transparent shared memory's fine-grain communication. This paper pursues two alternatives. The first is to implement a network layer with moderately low latency. We used Myricom's Myrinet hardware and Berkeley's AM software [13]. By carefully considering hardware-software trade-offs, such as interrupts or polling upon message arrival, we achieved latencies as low as 40 μsec round-trip and remote cache miss times as low as 77 μsec (Section 6). The other alternative, which we only touch on in this paper, is to use custom coherence protocols to tolerate latency.

- **Symmetric multiprocessor nodes.** Another unique aspect of this system is its SMP nodes. Using two processors complicated the system (Section 7), but increased performance on 16 processors by 0–32%. Space does not permit detailed analysis of this alternative, so measurements are for a configuration in which one processor runs protocol software and the other runs the computation thread.

In summary, this paper describes Blizzard, which implements the Tempest interface [30] on a network of workstations. The experiences and measurements reported in this paper should prove useful to other groups implementing distributed shared memory systems, low-latency communication, or fine-grain access control on commodity platforms.

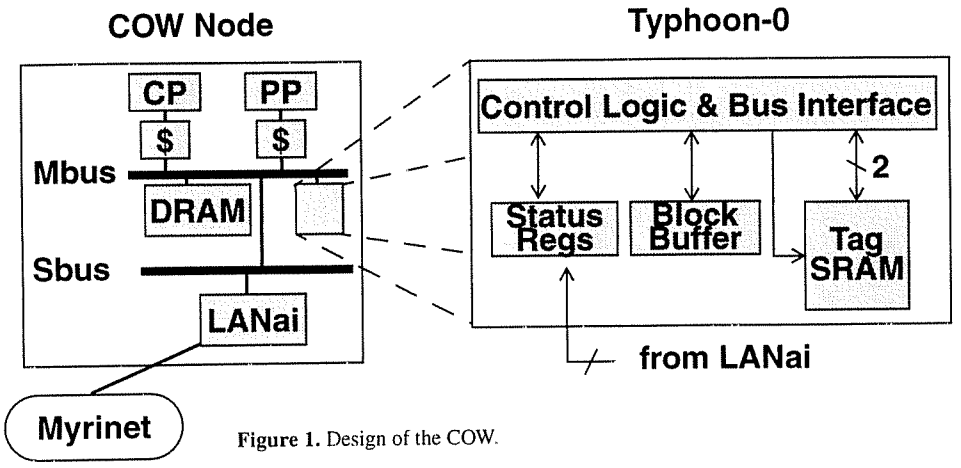

Figure 1. Design of the COW.

2 Related Work

This system differs from page-based distributed shared memory systems, such as Ivy [21], Munin [9], or Treadmarks [1], in several respects. First, Blizzard supports fine-grain access control, which permits efficient transparent shared-memory semantics. Second, Blizzard supports user-level coherence protocols. Munin supported a large, but fixed collection of protocols, but Blizzard provides mechanisms that enable an application to implement a custom protocol in unprivileged software. Third, Blizzard exploits low-latency communication to transfer small data blocks. Earlier systems amortized latency with page-granularity transfers. Finally, Blizzard runs on multiprocessor nodes, using one processor to speed communication and protocol processing. Karlsson and Stenstrom simulated a page-based distributed shared memory system running on an SMP [17].

Blizzard differs from object-based distributed shared memory system, such as CRL [16], by running unannotated shared-memory applications. It differs from another fine-grain distributed shared memory system, Wisconsin Blizzard [33], in several ways. Blizzard ran on a massively parallel processor (CM-5) with an expensive, custom network that offered low-overhead user-level communication. Blizzard uses commercial workstations and a commercial network. CM-5 nodes ran a small, simple kernel that did not support virtual memory or multiprogramming. Blizzard runs under an unmodified Solaris 2.4 kernel. Blizzard also supports more efficient fine-grain access control, including optimized software and dedicated hardware. Again, the final difference is that Blizzard exploits multiprocessor nodes.

Blizzard's uses Tempest active messages [15,38,27]. Tempest communication differs from other active message system [13,25,37] since it does not constrain a program to request-response protocols and interrupts computation on message arrival. These unique aspects are necessary for coherence protocols, but complicate the communication layer. Even so, Blizzard's communication is nearly as fast as a simpler, less general active message system.

3 COW

This section describes the platform for these experiments. COW consists of 40 dual-processor Sun SPARCStation 20s (Figure 1). Each contains two 66 Mhz Ross HyperSPARC processors [32] with a 256 KB L2 cache memory and 64 MB of memory. The cache-coherent 50 Mhz MBus connects the processors and memory. I/O devices are on the 25 Mhz SBus, which a bridge connects to the MBus. The COW nodes run the same version of Solaris 2.4 as other workstations at our institution.

Each COW node contains a Typhoon-0 card [26] and a Myrinet network interface[4]. The Typhoon-0 card plugs into the MBus and performs fine-grain access control by snooping bus transactions (Section 4.3). Node also contain a Myrinet interface, which consists of a slow (7–8 MIPS) custom processor (LANai) and 128 KB of memory. The LANai performs limited protocol processing and schedules

DMA transfers between the network and LANai memory or LANai memory and SPARC memory. The LANai processor cannot access SPARC memory directly, however it can interrupt the host processor. For this paper, we used 16 COW nodes, each connected to a Myrinet 8x8 switch (three total). The Myrinet switches were fully connected to each other.

SPARCStations 20s provide an I/O virtual address space for Sbus devices with a separate MMU in the MBus-to-SBus bridge. The I/O MMU maps 32 bit Sbus addresses to 36 bit Mbus addresses using a one-level page table. Under Solaris 2.4, the I/O MMU directly maps kernel virtual addresses to Sbus virtual addresses, which limits DMA operations to the kernel address space.

4 Fine-Grain Access Control

Fine-grain access control is the Tempest mechanism that detects reads or writes to invalid blocks or writes to read-only blocks and traps to user-level code, which uses the resulting exceptions to execute a coherence protocol action [15]. We implemented three techniques for fine grain access control on commodity systems.

4.1 Blizzard-S: Software Access Control

Blizzard-S uses the EEL executable editing library [20] to insert access control tests before shared-memory loads and stores in a fully compiled and linked program. These tests use an instruction's effective address to index a table, which contains two bits per cache block (the bits for 4 blocks are stored in one byte of memory) that determines if the memory reference should incur an exception. The Blizzard-S system required 15 instructions (18 cycles) before every load and store instruction [33]. Through the optimizations described below, we reduced this overhead to 3 instructions (3 cycles) at most loads and 8 instructions (9 cycles) at most stores (Table 1).

	Appbt		DSMC		EM3D		Moldyn		Barnes	
none	373.6	1.0	453.9	1.0	44.6	1.0	314.396	1.0	90.1	1.0
no opt	149.1	4.0	1951.9	4.3	95.8	2.1	901.305	2.9	338.1	3.7
cc reg	1171.7	3.1	1596.3	3.5	80.1	1.8	642.670	2.0	218.0	2.4
slice	1124.7	3.0	880.0	1.9	77.9	1.7	533.313	1.7	171.2	1.9
sentinel	723.5	1.9	717.5	1.6	56.4	1.2	434.607	1.3	147.6	1.6

Table 1: Overhead of software access control. The left entries in each column are the total execution time (in secs) on a uniprocessor. Each row corresponds to a different software fine grain access control technique. The right entries are the times normalized against the uninstrumented program ("none"). "No opt" uses Blizzard's instrumentation, which does not affect the condition code registers. "Cc reg" uses program analysis to insert a faster test at points at which the condition code registers are not live. "Slice" uses program slicing to eliminate unnecessary tests before non-shared memory references. "Sentinel" uses a sentinel value to speed access control of full-word loads. Optimizations are applied cumulatively.

The first optimization replaced the original test, which did not use the SPARC's condition code registers, with a 15 instruction (10 cycles) sequence that uses test and branch instructions.[1] The original snippet which required 14 cycles on the HyperSparc CPU, avoided using condition codes since these registers may contain live values that cannot be saved and restored quickly in user-level code. EEL's live register analysis identified the few (1–4%, measured dynamically) load and store instructions at which the condition codes contained live values. We used the old sequence for these instructions. This optimization reduced the overhead of access tests by 15–35% in the sequential execution of the five benchmarks described in Section 8.

The next optimization used static program analysis to determine that a load or store instruction could not access a shared-memory address, and consequently did not require an access control test. The analysis examined instructions in a backward slice [39] from a memory referencing instruction, to find the instructions that produced the reference's effective address. If this computation involved a stack pointer or pro-

1. All instruction and cycle counters are best-case estimates (discounting cache misses) for multiprocessor snippets. The single processor snippets are slightly shorter. For example, the equivalent uniprocessor snippet was 13 instructions (8 cycles).

duced a static address in the text or data segment, the reference could not access shared memory, which resides in a separate segment. The analysis is conservative and only examines values in registers (not memory) but still eliminated 9–66% of the dynamic tests in the benchmarks. Applied after the previous optimizations, it further reduced the overhead of access control by 3–45%.

The final optimization used sentinel values to replace the in-line table lookup before loads with a fast, possibly inaccurate test followed, when necessary, by a complete table lookup [10]. The sentinel value was a small integer corresponding to an IEEE floating point NaN. After loading a word into an integer or floating point register, two instructions tested if the word matched the sentinel. If so, the sequence dispatched to a handler, which performed a complete table lookup to distinguish access control violations from innocent uses of this value. Although we used this fast test only for word or double word loads, it applied to 24–70% of the memory references in the benchmarks. Applied after previous optimizations, it further reduced the overhead by 14–36%.

Overall, after applying these optimizations, sequential programs with software access control tests ran only 25–94% slower than uninstrumented programs, as compared to 200–400% slower without optimization.

4.2 Blizzard-E: Commodity Hardware Access Control

Blizzard-E uses deliberately incorrect error-correcting code (ECC) bits to detect invalid references to memory. The approach is similar to previous implementations [29, 33]. As before, the ECC bits implement valid/invalid access control, while read/write access control depends on virtual memory page table protection. Since page table granularity is larger than a cache block, writes to writable blocks on a page containing a read-only block trap. This forces the kernel to execute these writes within the kernel trap handler at a cost of 5.91 μsecs. The cost of setting invalid ECC is 55 μsecs for a 64-byte block (75 μsecs for 128 byte block). Section 5.2 discusses kernel support for Blizzard-E and Section 7 discusses the complications introduced by multiprocessors.

4.3 Blizzard-T: Custom Hardware Access Control

Typhoon-0's custom board snoops memory bus transactions, performs fine-grain access control tests through a SRAM lookup, and coordinates intra-node communication. The board contains two FPGAs, two SRAM, and some minor logic. Typhoon-0 only detects fine-grain access control violations. Protocol software run on a SPARC processor (not the Typhoon-0 board) and communicates through the Myrinet interface (on the I/O bus). However, Typhoon-0 also accelerates intra-node communication with a single cachable memory location on which a processor can poll to find the status of both the fine-grain access control and the network interface.

5 Kernel Support

Two forms of access control in Blizzard rely on hardware mechanisms (ECC and Typhoon-0). For both, Solaris 2.4 must provide an interface to hardware resources beyond those it was designed to support. This section describes the constraints that limited the virtualization of these resources.

5.1 Device Drivers and Kernel Modularity

Our goal—based on an earlier, unfavorable experience with adding code to production software—was to support fine-grain access control under a standard kernel. Fortunately, Solaris 2.4 is structured in an object-oriented manner, and, like other systems, supports dynamically loadable device drivers. Much of the system is invoked through function pointers embedded in well-defined data structures. Blizzard loadable drivers replace these function pointers to intercept and extend kernel calls.

5.2 Operating System Support for Fine-Grain Access Control

An operating system can support fine-grain access control in two ways. The first approach treats access control as a direct extension of virtual memory semantics. An access fault, like a page fault, is a synchronous exception raised at an invalid access. A fault invokes a handler, which runs in the process's address space. In this framework, virtualizing access control is simple. When a page is swapped out of physical memory, its access bits are saved [28]. When a page comes back from disk, its access bits are restored. To implement this approach, the operating system abstractions must be extended to encompass fine-grain access control. The benefits of this approach are clear. Fine-grain access control becomes a first-class citizen within an operating system, so it can be used for operations such as memory mapped files and IPC regions and to support kernel shared memory. Unfortunately, virtual memory is a fundamental abstraction, so the change would affect every aspect of the system and would be impossible with loadable drivers.

Instead, we adopted a simpler approach that treats fine-grain access control as an attribute of select memory regions. The operating system supports these regions through a limited interface. The kernel treats the physical memory underlying these regions specially, by delegating responsibility to a Blizzard device driver. User-level software coordinates with the device driver to virtualize this memory region. We implemented this approach with limited resources and without rewriting the kernel. The resulting shared memory is efficient and provides enough functionality to support application-level shared address spaces.

This approach proved surprising powerful. Consider an example. Both hardware access control techniques use virtual memory aliases to access memory with different attributes. The ECC scheme requires an uncached alias to data pages to clear invalid ECC, and a writable alias to modify blocks on read-only pages. The Typhoon-0 scheme requires both cached and uncached aliases for its control registers as well as an uncached alias to the fine-grain access control tag array. Unfortunately, these aliases violate the kernel's assumption that physical memory (e.g., the tag array) is always cachable and device memory is always uncachable. To solve this problem, Blizzard driver adds a wrapper function that intercepts calls to the low-level function that modifies hardware page tables. At each page table modifications, the wrapper consults private data structures to determine the correct cacheability attribute for a page.

On the other hand, some functionality is difficult or impossible to add from a device driver. The current implementation locks shared-memory pages in physical memory. This is necessary, because the Solaris does not notify a segment driver when a page is swapped. Extending the page table subsystem is difficult, as it relies on many static data structures and direct calls that do not go through a function table. Our solution—as yet unimplemented—is to delegate paging in these regions to a system-wide daemon, which also provides global—intranode—memory management, a desireable policy for parallel machines [8].

A larger complication is kernel and device accesses to shared-memory regions. The kernel typically accesses a user address space at clearly defined points. When it does, the kernel can incur access faults. The kernel deals with these faults as it handles any invalid argument and returns an error code. A more complete alternative would invoke user handlers at these faults to make an application's fine-grain access controlled memory indistinguishable from hardware shared memory. This would require a large change, of questionable value as we can provide the same functionality by wraping functions around system calls. A wrapper functions copies data in or out of local memory and always pass pointers in local memory to a system call. Another unresolved problem is DMA, which, for example, resets ECC bits. Since DMA to user pages only occurs when a user process directly accesses block devices, such as disk device drivers, we have not yet addressed this issue. However, a wrapper around the system calls could apply here as well.

5.3 Performance Enhancement to Solaris

Blizzard uses some features of Solaris in a manner for which they were not intended and so ran into performance bottlenecks. Consider an example. Access control hardware generates traps which must be delivered to a user-level handler. Solaris 2.4 (like other commodity systems) is not optimized to deliver synchronous traps quickly to user-level code. Only the signal interface can propagate traps to a process.

This interface is general, which makes it too costly for fine-grain access control. The measured time for a synchronous signal, such as SIGBUS, from a faulting instruction to the signal handler and back is 101 µsecs, which is longer than a roundtrip across the network. A loadable device driver implements a fast dispatch interface similar to ones discussed elsewhere [28,36]. The round trip time for the interface, including saving and restoring global registers in the user handler, is 4.56 µsecs.

Another performance bottleneck is virtual memory page protection, which ECC-based access control uses to enforce read-only access. Changing a page's protection, when the first read-only block is created, is costly, particularly for a multiprocessor kernel. This operation requires modifying page tables and flushing all CPUs' TLBs. The measured cost of the mprotect() system call is 170 µsecs. As a minor optimization, we extended the ioctl() interface to allow setting of invalid ECC and changing access protection in the same operation. This saves a second kernel call, but the overall cost of the combined operation is still 195 µsecs (instead of 225 µsecs for two separate calls).

6 Communications

The overhead of traditional, kernel-arbitrated access to a network is too large to support the fine-grain communication necessary for shared memory. To reduce latencies, a network device's memory can be mapped to the user address space. This approach permits a program to communicate without kernel intervention and is used by most recent low-latency communication layers [25,13,37,3]. For Myrinet hardware and Tempest active messages, it presents interesting hardware-software trade-offs.

6.1 Tempest Active Messages

Tempest messages are similar to other active message models [38], but they differ in two respects: messages are not constrained to follow a request-reply protocol and are delivered without explicit polling by an application program. The differences are necessary to use Tempest messages to implement *transparent* shared memory protocols. For example, a common protocol operation is for a block's home node to forward a request for the block, on behalf of a third node, to the node that has the exclusive copy. With a pure request-reply protocol, the operation would have to split in two phases; one to contact the home node and get the current block owner, and one to contact the owner requesting the block. In addition, Tempest message handlers must appear interrupt driven, since they run in a software layer below an application program, which is unaware of the underlying communication and can hardly be expected to poll for it. To optimize large transfers, Tempest sends small and large messages. The first are optimized for transferring cache blocks, the latter for more general message operations. A large message can specify the remote virtual address where the data is deposited. If unspecified, a user handler must pull the data from the network.

6.2 Communication Architecture

With Myrinet hardware, the host (SPARC) processor and Myrinet LANai processor cooperate to send and receive data. The division of labor is flexible since the LANai processor is programmable. However, the SPARC processor is far faster, which effectively limits the LANai processor to simple tasks [25]. Blizzard's communication library started from the LANai Control Program (LCP) used in Berkeley's LAM library [13]. We modified the LCP to fit the Tempest active message model and to improve performance. The changes are small, and, in fact, our modified LCP is still compatible with Berkeley's LAM library. The protocol supports small messages up to 8 words and large messages up to 4 KB.

A Solaris device driver maps the Myrinet interface's local memory into a program's address space. To permit DMA between the LANai and the address space, the driver allocates a kernel buffer mapped into the user address space. This approach does not virtualize the network, which limits network access to cooperating processes. However, the kernel could timeshare the Myrinet interface. We will address this issue in the future, when we implement synchronized network preemption on the cluster level.

The LCP implements, in LANai memory, separate send and receive queues. Each queue contains 256 entries consisting of a message header and up to 8 words of data. The host processor uses loads and stores to move the word data—the complete message for small messages and the header for large messages—in and out of LANai memory. The LANai processor uses DMA to move larger messages through intermediate kernel/user buffers. This organization lowers the latency of small messages and increases the throughput of large messages.

6.3 Implementing Tempest Active Messages

Unlike most earlier schemes [38], Tempest Active Messages need not follow a request-reply protocol. A key advantage of request-reply protocols is that they offer a simple buffer allocation policy. A system can preallocate buffers at each node for each sender. A sender uses a simple credit-based flow-control scheme to ensure it never has more outstanding requests to another node than preallocated buffers. A node must always send replies, which implicitly acknowledge the corresponding request and indicate that its buffer has been freed.

Since Tempest does not force request-reply communication, the messaging software must handle buffer allocation in a more general way. Blizzard breaks it into two subproblems: flow control and deadlock detection. Blizzard uses a similar credit-based flow control mechanism to guarantee that buffer space exists before sending a message[1]. Unfortunately, this policy can cause deadlock if a user overcommits network resources, for example, if a message handler sends more messages than the low-level network layer can buffer. To prevent this type of deadlock, the system must extend the buffer space by copying messages to a software queue in main memory [5,22]. While not scalable, this approach more than suffices for moderate parallel machines.

Earlier systems aggressively buffer messages when the sender blocks [33, 5] or after it has been blocked for a timeout interval [18]. However, the extra copies this entails can potentially degrade performance [23]. Blizzard, instead, uses a conservative deadlock detection scheme, which only buffers messages when a deadlock may have occurred. Our implementation uses a conservative, local condition to detect potential deadlocks on a send to node i:

$$msgs\_sent\_to\_node(i) - acks\_received\_from\_node(i) \geq window\_size$$
and $\exists$ *node j* such that
$$msgs\_received\_from\_node(j) - acks\_sent\_to\_node(j) \geq window\_size$$

This condition is necessary for two processors to be blocked on message sends. If $i=j$, then a cycle of length two exists; if $i<>j$, then a larger cycle *may* exist. In either case, the LANai copies messages from the low-level receive queue into a software queue in user virtual memory.

The deadlock condition is not sufficient to indicate a deadlock. Certain communication patterns can cause unnecessary buffering. For example, two-way, full-speed communication between two nodes can trigger the condition if the window is too small. If one node fails to drain the network rapidly enough, the algorithm will detect a spurious deadlock. However, in practice, this can be avoided if the window size is kept larger than the bandwidth-delay product.

6.4 Polling vs. Interrupts

To avoid false deadlocks, and reduce round-trip latencies, a node must quickly respond to arriving messages. Conceptually, an arriving message causes an interrupt, which invokes a coherence protocol's message handler. In practice, however, hardware interrupts are far too expensive (Table 2) and Solaris does not provide a fast mechanism to deliver asynchronous interrupts. Blizzard resorts to two forms of polling.

1. The LAM library enforces request-reply semantics, but allows for requests without replies. In that case, the receiver automatically sends a null reply. In one-way transfers, this approach generates a null reply—an explicit acknowledgment—for every message. By contrast, Blizzard's scheme sends one reply for every *window_size* messages.

Because our applications are shared-memory codes, which are oblivious to the underlying communication, we cannot expect them to explicitly poll. Instead, one approach uses EEL to insert explicit polling code on backwards loop edges and potentially recursive procedure calls. This code consists of 14 instructions (10 cycles) that read a location to test if a message arrived. The second approach, on SMP nodes, uses the node's second processor to poll and execute the message dispatch loop. In addition, this processor handles incoming traffic by running protocol handlers and sending replies. With this approach, both processors can communicate concurrently since the send and receive queues are independent. However, it does require additional locking.

In either approach, polling can be implemented with one of four techniques: poll on the LANai receive queue, poll on a cachable memory location that the LANai updates by interrupting the kernel or via a DMA transfer, or use the Typhoon-0's hardware support for detecting message arrivals. The original Berkeley library polls on the receive queue head, which requires an uncached memory access across the SBus-to-MBus bridge. These accesses are costly (0.38 µsecs) and polling consumes bandwidth on the MBus and SBus-to-MBus bridge.

The second and third alternatives poll on a cacheable location in the kernel-user data buffer[1]. This reduces the overhead of a "failed" poll to a cache hit and eliminates unnecessary memory bus traffic. Conversely, updating the flag is more expensive. We tried two approaches to update the flag when a message arrives. For the first, the LANai interrupts the host processor, which runs the device driver updates the flag. This requires 14 µsecs to update the flag. A better approach, requiring only 4 µsecs, is for the LANai to directly update the flag with a one-word DMA. The LANai keeps a local copy of the flag and only sets the host flag (or signals an interrupt to set the host flag) when a message arrives *and* its local copy is not already set. The host clears both its flag and the LANai copy once the message queue is empty. Thus, only the first message in a burst incurs the cost of an update.

The final alternative uses the message bit in Typhoon-0's status register. The Typhoon-0 board is connected via a jumper to the Myrinet interface board's LED signal. The LANai toggles the LED by writing to a register upon message arrival, which in turn sets the Typhoon-0 message bit. Since the Typhoon-0 status register is cachable, "failed" polls are quick and do not cause bus traffic. However, a message arrival causes a cache miss followed by a handshake to clear the register. This approach requires 1.3 µsecs.

6.5 Network Performance

This section presents microbenchmarks that attempt to determine the performance of the network.

6.5.1 Small Message Latency

The small message benchmark measures the round-trip cost of small (4 byte) messages. In a timing loop, one COW node sends a message to another node, which responds with a reply message (Table 2). Sending the message from the compute thread—as opposed to sending from the protocol handler, on the other processor, which received the reply message—incurs an extra overhead of 4–6 µsecs.

The cost to notify the compute thread depends on the polling mechanism, since mechanisms differ in their actions after message arrival, which affects the latency to notify the compute thread. Some of this overhead is overlapped, since the handler dispatch loop runs on a separate processor. The DMA method incurs the least overhead, because it only clears a cachable memory location. The LANai memory polling method comes next, with extra overhead due to the protocol processor's continuous polling across the MBus-to-SBus bridge, which slows the subsequent send. Interrupts are similar to the LANai, except the interrupt is serviced within the kernel, which pollutes the cache of the processor that handles the interrupt. Finally, Typhoon-0 requires two uncached stores and two Mbus transactions to invalidate the message bit and read a fresh copy of the control register. Overall, polling the LANai memory achieves minimum

1. Solaris 2.4 normally maps device memory as uncachable, which increases the cost of loading the poll flag from 1 to 18 cycles. However, since SPARCStation 20's support cache-coherent DMA, we used our wrapper function to map the flag as cachable.

Polling	Sending Thread	Time (μsecs)
DMA	Compute	53.8
	Protocol	49.9
INT	Compute	76.6
	Protocol	71.1
LANai	Compute	44.9
	Protocol	39.9
Typhoon-0	Compute	51.9
	Protocol	46.0

Table 2: Small message latency. This table records the round-trip time for a 4 byte message. The protocol thread polls for incoming messages in one of four ways: on a location set by the LANai by a DMA transfers ("DMA"), on a location set by a processor interrupted by the LANai ("INT"), on an uncachable location on in the LANai's memory ("LANai"), or a cachable control register provided by the Typhoon-0 board ("Typhoon-0"). The message originates from either the compute thread ("Compute") or the protocol handler that received the reply ("Protocol").

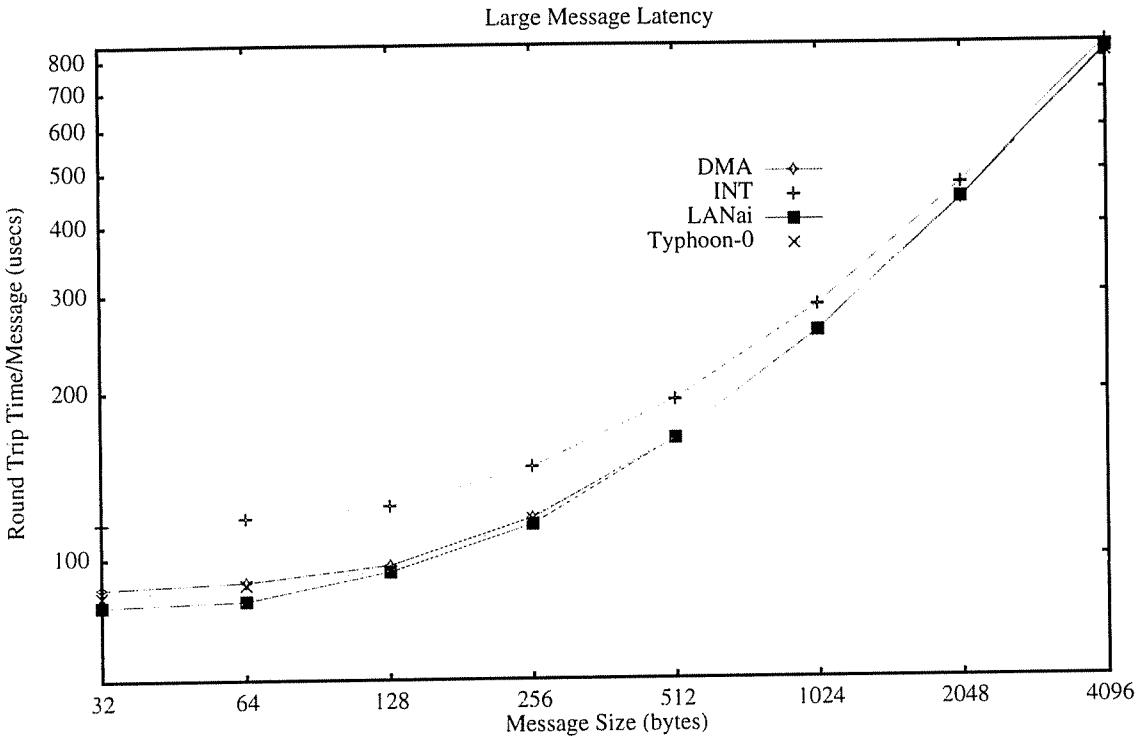

Figure 2. Large message latency.

latency (40 μsecs) but failed polls consume bus bandwidth. Despite the more general functionality, this latency is only 4 μsecs higher than the measured latency of the original Berkeley library on COW and only 2 μsecs higher than the measured latency of the Berkeley library using our modified LCP.

6.5.2 Large Message Latency

The large message network benchmark measures the round-trip time for a large message (Figure 2). We varied the data block from 32–4K bytes (the maximum size supported). Again the benchmark examines the effect of the polling mechanisms on message latency. We present only data for send operations from the protocol thread. Sending from the compute thread is similar, except for the additional cost of notifying the compute thread (as for small messages).

The minimum round trip latency for a 32-byte block is 82 μsecs with LANai polling. This method produces the lowest latency for blocks up to 512 bytes. At that point, Typhoon-0 and DMA polling become

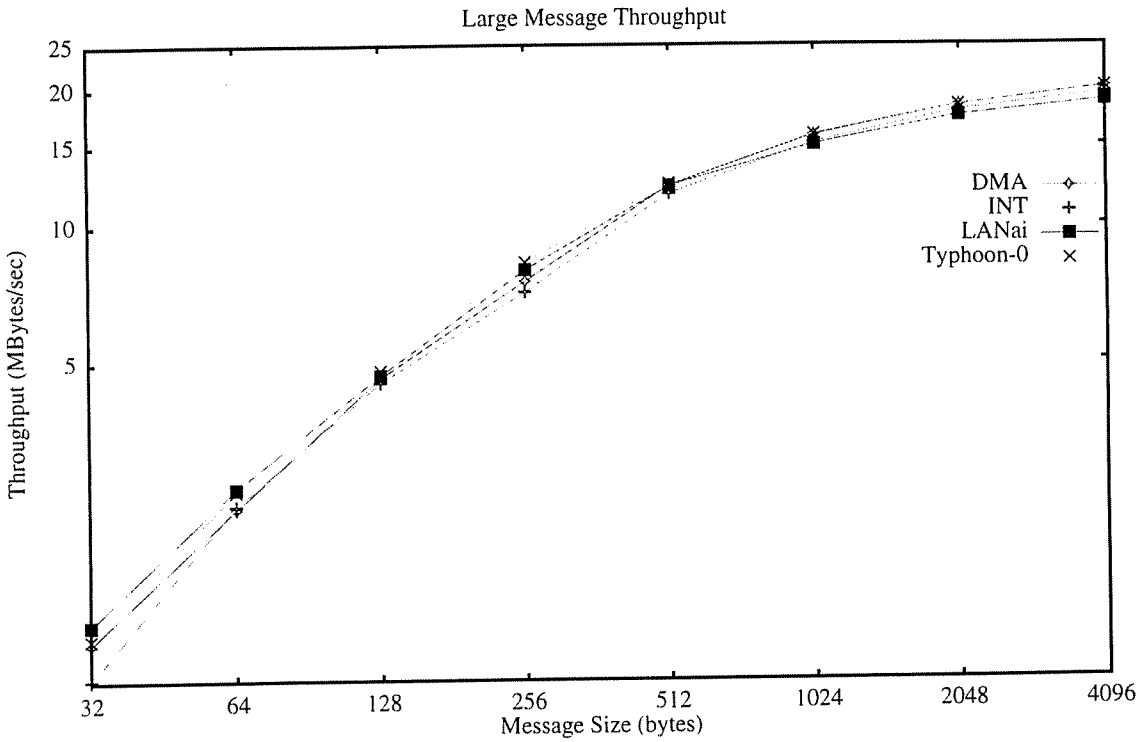

Figure 3. Large message throughput.

slightly faster. Continuous polling of the LANai slows down the DMA to and from the network interface. Typhoon-0 a slight advantage (less that 3 µsecs) over DMA polling because of contention for the DMA hardware, which it uses both for transfering data and setting the flag.

The measured latencies for LANai polling are lower than the original Berkeley LAM library for all block larger than 32 bytes. The original LCP was optimized for high throughput with larger message. The Berkeley LAM library, with our modified LCP, is faster (by 13 µsecs) for 32 byte blocks, but Blizzard's network layer catches up at 256 bytes and is slightly faster (by 53 µsecs) for 4K blocks. The difference in the latency for small messages can be attributed to the synchronization overhead required to coordinate accesses to the network structures by the two processors. On the other hand, for large messages the optimized (for blocks aligned at a 32-byte boundary) memory copy routine in Blizzard explains this result.

6.5.3 Throughput

The throughput benchmark measures the maximum throughput of the network layer for large messages. The achieved value of 20 MB/sec is close to reported Myrinet measurements [13,25]. The Berkeley LCP was optimized for throughput and achieves 25% higher throughput for messages larger than 256 bytes.

LANai memory polling method achieves the highest bandwidth for small messages. As the message size increases, its bandwidth becomes lower than other techniques since polling on LANai memory consumes bus bandwidth, which becomes scarce with large messages. Even interrupts perform better for large messages. There is little difference between polling DMA and the Typhoon-0 message bit. Both achieve the best results for large message.

In a separate experiment, we measured the maximum bandwidth for a sender and receiver by saturating a receiver with messages from many senders and a sender by sending to many receivers. The measured receive bandwidth was 27 MB/sec and the measured send bandwidth was 21 MB/sec. This indicates that the bottleneck in the network layer is the sender.

7 SMP Nodes

Systems that run on uniprocessors can implement critical sections in protocol handlers surpressing interrupts, either in software [35] or hardware. These mechanisms do not work for multiprocessors in which the second processor runs the protocol. Blizzard ensures mutual exclusion with locks, which complicated the atomicity of a software or ECC access control test and the subsequent reference.

7.1 Blizzard-S

In Blizzard-S, the access control tests executes before the memory reference:

```
if (!lookup(Address)) CallHandler(Address, AccessType);
load-or-store(Address, Value);
```

Between a test and store, a remote reference could invoke a message handler, which changes the referenced block's status and causes the stored value to be lost. Although the window of vulnerability is small, typically a few cycles, unless an uncommon event such as a page fault or an interrupt occurs, access test must execute quickly, since it is a common operation.

Blizzard-S uses two techniques. The first is a flag that indicates an access in progress. The access test sets the flag and resets it after the memory reference. The Tempest routine that modifies access tags spins on this flag after it changes a tag to ensure that the protocol action sees the updated memory value and further tests use the new tag value. This flag introduces two extra stores per access test. Fortunately, load's more efficient sentinel access tests (Section 4.1) does not require a flag, since the data and access control value are loaded simultaneously and the access test appears atomic.

7.2 Blizzard-E

In Blizzard-E, the kernel must execute a store to a writable block in a read-only page, which again separates an access control test from a memory reference. Although these stores from the kernel are relatively infrequent, the kernel cannot wait to acquire a lock.

Blizzard uses a memory location in the user address space as a lock. Before accessing the tag bits, the kernel tries to acquire the lock. If it fails, it returns from the trap handler, and the access is retried. By acquiring the lock, protocol code can temporarily stop the kernel from performing these writes. Although more powerful, this mechanism is used in the same way as the Blizzard-S flag. The Tempest routine that modifies access permissions, acquires and immediately releases the lock after changing the tags.

Setting invalid ECC is more complicated on a SMP than a uniprocessor. To change ECC, the memory controller must be in a diagnostic mode in which the controller does not compute ECC for subsequent writes. Instead, the ECC value comes from a register. Since this mechanism applies to all writes, all other memory traffic must stop—including the second processor and DMA operations. We stop everything by disabling bus arbitration for every MBus master except processor zero. To reset the ECC, uncached double word writes suffice.

8 Performance Evaluation

This section presents measurements of microbenchmarks and applications running on Blizzard.

8.1 Miss Benchmarks

This section presents a microbenchmark evaluation of three aspects of the Blizzard system. In these benchmarks, two COW nodes share a memory region. One node owns the region and services all cache misses. The benchmarks measure the other node's non-loop time to access all blocks (Table 1). For DMA,

Miss Times (μsecs/miss)	Event	Block Size			
		64	128	256	512
S	Read Miss	76.7	85.4	106.1	147.4
E		77.2	84.4	101.5	135.1
T		76.8	92.6	122.7	184.9
S	Write Miss	77.4	86.0	107.2	149.7
E		181.6	196.2	250.7	356.3
T		76.8	91.2	120.1	177.4
S	Read Miss; Write Miss	127.0	137.2	157.9	205.7
E		702.7	728.9	784.3	881.6
T		129.6	144.2	174.7	237.4

Table 3: Miss microbenchmarks (LANai memory polling). Read Miss" measures the cost (in μsecs) of a cache miss on a read access to a cache block on a remote node. Write Miss" measures the cost of a remote miss on a memory write."Read Miss; Write Miss" measures the cost of a remote miss on a read access, immediately followed by a write to the same block.

LANai or Typhoon-0 polling, the miss cost is directly attributable to the cost of communication plus access control (Table 4). However, this relation does not hold for interrupts, since the interrupt handler invokes the kernel, which pollutes the caches and increases the miss times.

The first benchmark measures read miss latency for a block in the Idle state. At a miss, the faulting pro-

Fine Grain Access Control Overhead	Event	Block Size			
		64	128	256	512
S	Read Miss	7%	9%	18%	24%
E		8%	8%	14%	17%
T		8%	16%	29%	39%
S	Write Miss	8%	10%	19%	25%
E		61%	60%	65%	68%
T		7%	15%	28%	36%
S	Read Miss; Write Miss	13%	15%	20%	26%
E		84%	84%	84%	83%
T		14%	19%	27%	36%

Table 4: Calculated fraction of overhead for fine-grain access control in a remote miss. The entries are computed from the difference between the miss times (Table 3) and the communication cost divided by the miss times.

cessor sends a request to the block's home node, which replies with the block. Both nodes set the block's access tag to read-only (from read-write and invalid, respectively). All access control methods perform comparably and the cost of fine-grain access control is only 8% of the total miss time (64 byte block). As the block size increases, the relative cost of fine-grain access control increases, since the communication cost does not increase linearly with block size. In Blizzard-S and Blizzard-E, the cost of the fine grain access control increases slower than in Blizzard-T. When a cache block arrives, Blizzard-S copies the data and sets the access bits. Most of these memory reference run at full speed and hit in the processor's cache. Blizzard-E sets the access bits and copies the data with uncached doubleword stores, which exploit the HyperSPARC's write buffer to run at full speed for small blocks. On the other hand, Blizzard-T manipulates the Typhoon-0 access control hardware, which supports 32-byte blocks and does not accelerate multi-block operations.

The second benchmark is similar, except that a write causes the miss. In this case, the protocol requires the writer to have exclusivity and prevents access by the home node. The home node changes the block's access tag from read-write to invalid. In Blizzard-E, setting a tag to invalid requires an ioctl() system call,

which more than doubles the total miss time and increases the overhead of fine-grain access control to over 60%. Blizzard-S and Blizzard-T perform as previously.

The final benchmark reads, then immediately writes the block, which causes two misses. The first miss is identical to the first benchmark. The second miss causes an upgrade request to the home node, which invalidates the block and replies confirming that the requesting node has exclusive access. For the software and Typhoon-0 systems, fine-grain access control comprises less than 14% of the miss time. However, this benchmark is a worst case for Blizzard-E, as the two misses require two page protection on each node.

9 Application Measurements

To better understand the system's performance, we ran five shared-memory programs on Blizzard. We report speedups, with respect to a sequential program running on a COW workstation (Figure 4 and Appendix A). The datasets ensured that the sequential program fit entirely in memory. Blizzard polled using the Typhoon-0, which resulted in slightly better performance than polling the LANai memory.

Appbt is a NAS Parallel Benchmark [2], which NASA produced as an exemplar of computation and communication pattern in three-dimensional computational fluid dynamics applications. The program ran for 30 iterations (times exclude the first iteration) on 32x32x32 cubes (Table 6). The sequential run required 320 seconds.

The transparent shared memory version[7] suffers because processors spin-wait on a shared array of counters. Contention for counters causes a many misses, which gives Blizzard-S an advantage of Blizzard-E since software access control handles cache miss faster (Table 3). Overall, Blizzard-E shows little speedup since the number of system calls to modify ECC increases faster than the number of processors. Moreover, even efficient access control (Blizzard-T) cannot remedy the mismatch between the program and the high network latencies.

The custom protocol version reduces contention by implementing signal-wait primitives with active messages [14]. In addition, it modifies the coherence protocol to better fit the producer-consumer sharing pattern. The hardware access control mechanisms produce good speedups for this version. Blizzard-S's instrumentation overhead is high relative to other applications since the program mainly references shared data, so that program slicing eliminates instrumentation for only 9.2% of memory references. For 2–4 nodes the speedup is superlinear since parallel execution partitions the program's large arrays among several nodes' memories.

Barnes is a SPLASH-2 benchmarks [34]. It simulates the evolution of bodies in a gravitational system. The program used the default input set of 8192 bodies and 8 iterations. The first iteration is excluded from the speedups. The sequential run required 43 seconds.

This program is a demanding application for a shared-memory system. Its primary data structure is a shared oct-tree, which represents bodies' locations in space. Processors cooperatively build the tree at the start of each iteration. Then, each processor computes the forces exerted on its subset of bodies by using the oct-tree to find near-by bodies and approximate distant bodies. The algorithm entails fine-grain sharing, which causes many misses. Even Blizzard-T's efficient access control cannot compensate for the large network latencies, so speedups were modest.

The custom protocol version is written in C** [19], a large-grain data parallel programming language. This version builds the oct-tree through data parallel steps that use user-defined reductions to add tree nodes. The program has communication-intensive phases interleaved with computation. Unfortunately, the current Myrinet switches cannot handle large data transfers. We introduced delays in the network software to avoid resetting the Myrinet switches during these runs. For 16 COW nodes, the delays made the results meaningless. (We hope that new switches will solve this problem.) Despite these problems and artificial delays, the custom protocol version performs up to 30% better than the shared memory version (43% efficiency on 8 nodes).

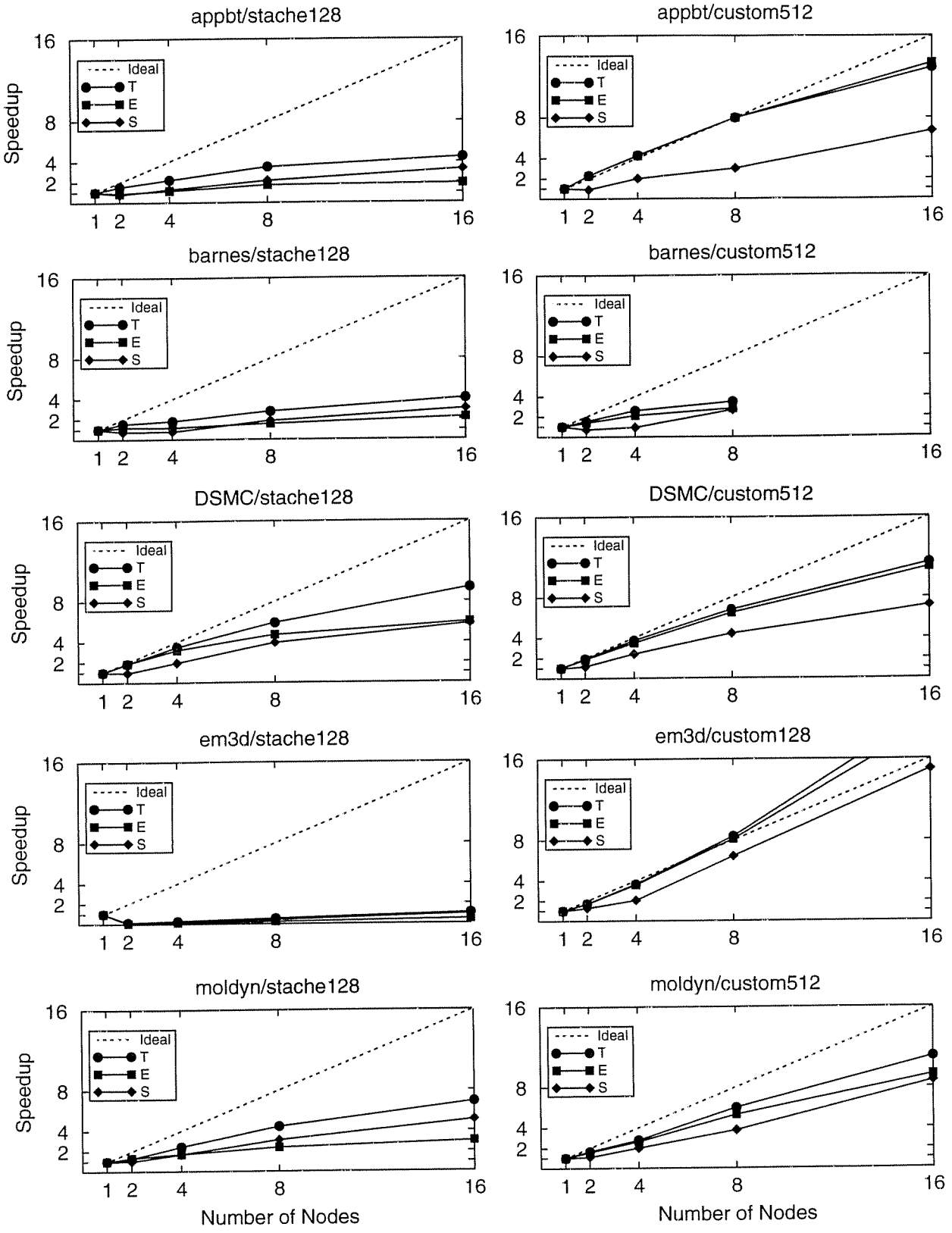

Figure 4. Speedup of the application benchmarks, with respect to the sequential program.

DSMC models properties of a gas by using a direct simulation Monte Carlo method to simulate the movement and collision of many particles in a three dimensional domain [40]. Each processor simulates collisions in its region of space. The results are for 300 iterations. The sequential run required 417 seconds.

DSMC performs well with transparent shared memory since the program has a high computation to communication ratio and because the code previously was optimized for shared memory [24]. Communication occurs during a relatively short phase in which molecules move to neighboring processors. Blizzard-S performs well relative to Blizzard-E, since the cost of modifying page protection increases with the number of COW nodes, although the total misses per node remains constant. The custom protocol version moves molecules in explicit messages. This eliminates misses when a molecule moves between processors and gives Blizzard-E a large boost. The custom protocol offers fewer benefits for Blizzard-S and Blizzard-T, since these systems handle misses more efficiently.

EM3D models the propagation of electromagnetic waves through objects in three dimensions [12]. Its primary data structure is a bipartite graph partitioned among the processors. The program ran with a graph containing 32K nodes for 100 iterations. In the parallel runs, 10% of edges connected nodes on different processors. The sequential run required 43 seconds.

The shared memory version does not achieve speedup. The cost of servicing a miss in every 10 iterations of the inner loop imposes a large penalty that is difficult to overcome. However, the system scales linearly (from its abysmal starting point) on all Blizzard systems. The custom protocol version uses an update protocol that performs well for the program's producer-consumer sharing [14]. This version achieves good speedups. The superlinear speedups are due to cache effects, as the graph does not fit in a processor's cache until it has been finely partitioned.

Moldyn is a molecular dynamics application [6]. Molecules are uniformly distributed in a cubical region and the system computes a molecule's velocity and the force exerted by other molecules. An interaction list (rebuilt every 20 iterations) limits interaction with molecules in a cut-off radius. The program ran with 8722 molecules for 60 iterations. The sequential run required 342 seconds.

The shared memory version makes effective use of shared memory [24]. The program's reduction phase performs operations on large arrays with minimum locking. However, in this phase, each processor accesses a large portion of the shared memory, which limits Blizzard's speedups. Blizzard-E was particularly hindered by a large number of page protection changes. Blizzard-S performed quite close to Blizzard-T due to low instrumentation overhead and the large number of misses. In this application, static slicing eliminated instrumentation for 66.1% of loads and stores. The custom protocol version used explicit messages to avoid cache misses in the reduction. It performs quite well for all access control methods including Blizzard-S.

Run Time (secs)		Appbt	Barnes	DSMC	EM3D	Moldyn
Blizzard-S	stache128	97.0	17.6	71.9	44.2	67.1
Blizzard-E		168.9	24.3	69.8	93.0	113.8
Blizzard-T		71.7	13.2	44.0	40.3	49.7
Blizzard-E		326.6	63.0	127.8	116.3	154.9
Blizzard-S	custom512	47.9		58.0	2.9	39.7
Blizzard-E		23.9		38.1	2.2	37.0
Blizzard-T		24.8		36.5	2.1	31.0
Blizzard-E		89.0	39.3	105.9	9.5	129.6

Table 5: Run time for the five benchmarks on the Blizzard systems and Wisconsin's Blizzard-E on a CM-5 (16 processors)

As a final experiment, we compared the programs against the Blizzard-E system running on a TMC CM-5 (Table 10). Although processors on the CM-5 are roughly four times slower than those on COW, CM-5's communication has lower latency and overhead. For shared-memory applications, Blizzard-T on COW ran

2.9–4.8 times faster. For custom protocols, Blizzard-T was 2.9–4.5 faster. Blizzard-S generally performed better than the ECC version for shared memory (the opposite held on the CM-5) and ran 1.8–3.6 faster than Blizzard. For custom protocols, it was 1.9–3.3 times faster.

9.1 Discussion

Access control using commodity hardware (Blizzard-E) performs well for applications and protocols that do not incur many cache misses or require many access control changes. Toggling access control and invalidating ECC are expensive, since both require a system call and the former requires flushing TLBs. The more portable, software approach (Blizzard-S) performs well as long as a program has a large number of cache misses and a significant fraction of the program's time is spent processing protocol events. However, instrumentation overhead slows computation. Further optimization, particularly optimizations integrated into a compiler, could further reduce this overhead and make this approach, which is currently competitive, even more attractive. The custom hardware version (Blizzard-T) consistently performs the best. Although simple, the hardware performs access checks without overhead and does not incur the performance problems caused by using ECC bits. Although communication on COW was slower than in a previous system on an MPP, the overall performance closely tracked the processor performance improvement on the newer platform.

10 Conclusions

This paper describes Blizzard, a fine-grain distributed shared memory system that runs on a network of workstations. Commodity hardware and software complicated the implementation of this system in many ways. This paper focuses on three key interactions: fine-grain access control and multiprocessor nodes, access control and a commercial operating system, and low-latency communication and applications. It also measures two new fine-grain access control mechanisms: optimized software and a hardware accelerator.

Overall, the network of workstations formed a good platform for fine-grain distributed shared memory. However, the high cost of communication, even with Myrinet's low-latency hardware, proved a serious impediment to the performance of naive transparent shared memory programs. Optimized shared memory programs performed better, sometimes achieving up to 50% efficiency. Besides better networks, another approach is custom coherence protocols, which offer a way of improving the performance of shared-memory programs without losing the shared-address space abstraction. These protocols performed well on COW, although they were developed on other systems, and achieved efficiencies of 70% or higher.

Acknowledgements

This work developed in the supportive environment provided by members of the Wisconsin Wind Tunnel Project (http://www.cs.wisc.edu/~wwt). We would like to thank Scott Pakin, Andrew Chien and the FM group at University of Illinois for providing the FM communication package, and Steve Swartz for porting it to an earlier implementation of our system. Many thanks to Rich Martin, David Culler and the NOW group at University of California for releasing the AM communication package for Myrinet used in the current version of our system. We would also like to thank Guhan Viswanathan for providing us with the custom protocol Barnes. Finally, we would like to thank Mark Dionne for supporting the application utilities on our network of workstations.

References

[1] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwanepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[2] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

[3] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathon 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.

[4] Nanette J. Boden, Danny Cohen, Robert E. Felderman, Alan E. Kulawik, Charles L. Seitz, Jakov N. Seizovic, and Wen-King Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, February 1995.

[5] Eric A. Brewer, Frederic T. Chong, Lok T. Liu, Shamik D. Sharma, and John Kubiatowicz. Remote Queues: Exposing Message Queues or Optimization and Atomicity. In *Proceedings of the Sixth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 42–53, 1995.

[6] B. R. Brooks, R. E. Bruccoleri, B. D. Olafson, D . J. States, S. Swamintathan, and M. Karplus. Charmm: A program for macromolecular energy, minimization, and dynamics calculation. *Journal of Computational Chemistry*, 4(187), 1983.

[7] Doug Burger and Sanjay Mehta. Parallelizing Appbt for a Shared-Memory Multiprocessor. Technical Report 1286, Computer Sciences Department, University of Wisconsin–Madison, September 1995.

[8] Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors. In *Proceedings of Supercomputing '94*, pages 590–599, November 1994.

[9] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.

[10] Derek Chiou, Boon S. Ang, Arvind, Michael J. Becherle, Andy Boughton, Robert Greiner, James E. Hicks, and James C. Hoe. StartT-ng: Delivering Seamless Parallel Computing. In *Proceedings of EURO-PAR '95*, Stockholm, Sweden, 1995.

[11] Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, and Willy Zwaenepoel. Software Versus Hardware Shared-Memory Implementation: A Case Study. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 106–117, April 1994.

[12] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[13] David Culler, Lok Tin Liu, Richard Martin, and Chad Yoshikawa. LogP Performance Assessment of Fast Network Interfaces. *IEEE Micro*, pages 35–43, February 1996.

[14] Babak Falsafi, Alvin Lebeck, Steven Reinhardt, Ioannis Schoinas, Mark D. Hill, James Larus, Anne Rogers, and David Wood. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[15] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[16] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 213–228, December 1995.

[17] Magnus Karlsson and Per Stenstrom. Performance Evaluation of a Cluster-Based Multiprocessor Build from ATM Switches and Bus-Based Multiprocessor Servers. In *Proceedings of the Second IEEE Symposium on High-Performance Computer Architecture*, February 1996.

[18] John Kubiatowicz and Anant Agarwal. Anatomy of a Message in the Alewife Multiprocessor. In *Proceedings of the 1993 ACM International Conference on Supercomputing*, 1993.

[19] James R. Larus. C**: a Large-Grain, Object-Oriented, Data-Parallel Programming Language. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages And Compilers for Parallel Computing (5th International Workshop)*, pages 326–341. Springer-Verlag, August 1993.

[20] James R. Larus and Eric 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.

[21] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[22] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and Frans Kaashoek. Fugu: Implementing Translation and Protection in a Multiuser, Multimodel Multiprocessor. Technical Memo MIT/LCS/TM-503, MIT Laboratory for Computer Science, October 1994.

[23] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent Network Interfaces for Fine-Grain Communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, page ?, May 1996.

[24] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed-Memory Machines. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 68–79, July 1995.

[25] Scott Pakin, Mario Laura, and Andrew Chien. High Performance Messaging on Workstations: Illinois Fast Messages (FM) for Myrinet. In *Proceedings of Supercomputing '95*, 1995.

[26] Robert W. Pfile. Typhoon-Zero Implementation: The Vortex Module. Technical report, Computer Sciences Department, University of Wisconsin–Madison, 1995.

[27] Steven K. Reinhardt. Tempest Interface Specification (Revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February 1995.

[28] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel Support for the Wisconsin Wind Tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, September 1993.

[29] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David 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.

[30] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, April 1994.

[31] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

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

[33] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David 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, October 1994.

[34] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[35] Daniel Stodolsky, J. Brad Chen, and Brian Bershad. Fast Interrupt Priority Management in Operating Systems. In *Second USENIX Symposium on Microkernels and Other Kernel Archtitectures*, pages 105–110, September 1993. San Diego, CA.

[36] Chandramohan A. Thekkath and Henry 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 (ASPLOS VI)*, pages 110–119, 1994.

[37] Thorsten von Eicken, Anindya Basu, Vineet Buch, and Werner Vogels. U-Net: A User-Level Network Interface for Parallel and Distributed Computing. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 40–53, December 1995.

[38] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active Messages: a Mechanism for Integrating Communication and Computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[39] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[40] Richard G. Wilmoth. Direct Simulation Monte Carlo Analysis of Rarefied Flows on Parallel Processors. *AIAA Journal of Thermophysics and Heat Transfer*, 5(3):292–300, July-Sept 1991.

Appendix A: Performance Measurements

Appbt	Coherence Protocol	Number of Processors			
		2	4	8	16
Blizzard-S	stache128	0.78	1.27	2.15	3.30
Blizzard-E		0.89	1.15	1.72	1.89
Blizzard-T		1.53	2.21	3.51	4.46
Blizzard-S	custom512	0.90	1.99	3.83	6.69
Blizzard-E		2.27	4.25	7.97	13.38
Blizzard-T		2.28	4.28	7.98	12.92

Table 6: Speedups over the sequential program of Appbt.

Barnes	Coherence Protocol	Number of Processors			
		2	4	8	16
Blizzard-S	stache128	0.76	0.77	1.87	3.05
Blizzard-E		1.19	1.17	1.56	2.21
Blizzard-T		1.55	1.83	2.79	4.08
Blizzard-S	custom512	0.71	0.95	2.66	
Blizzard-E		1.40	2.11	2.82	
Blizzard-T		1.54	2.59	3.49	

Table 7: Speedups over the sequential version of Barnes. The Myrinet switches could not handle the custom protocol on 16 COW nodes.

DSMC	Coherence Protocol	Number of Processors			
		2	4	8	16
Blizzard-S	stache128	0.97	1.96	3.93	5.80
Blizzard-E		1.86	3.19	4.72	5.98
Blizzard-T		1.85	3.49	5.90	9.36
Blizzard-S	custom512	1.18	2.40	4.42	7.19
Blizzard-E		1.87	3.48	6.48	10.96
Blizzard-T		1.92	3.72	6.80	11.43

Table 8: Speedups over the sequential program of DSMC.

EM3D	Coherence Protocol	Number of Processors			
		2	4	8	16
Blizzard-S	stache128	0.12	0.22	0.41	0.98
Blizzard-E		0.07	0.12	0.24	0.47
Blizzard-T		0.14	0.27	0.54	1.08
Blizzard-S	custom128	1.28	2.06	6.41	15.00
Blizzard-E		1.68	3.58	8.12	19.41
Blizzard-T		1.71	3.66	8.38	20.65

Table 9: Speedups over the sequential version of EM3D.

Moldyn	Coherence Protocol	Number of Processors			
		2	4	8	16
Blizzard-S	stache128	1.02	1.69	3.11	5.10
Blizzard-E		1.33	1.72	2.38	3.01
Blizzard-T		1.27	2.44	4.43	6.88
Blizzard-S	custom512	1.12	1.99	3.75	8.62
Blizzard-E		1.58	2.62	5.28	9.25
Blizzard-T		1.65	2.77	5.97	11.05

Table 10: Speedups over the sequential version of Moldyn.