Appears in: "ASPLOS VI," Oct. 1994. Reprinted by permission of ACM.

Fine-grain Access Control for Distributed Shared Memory*

Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck,
Steven K. Reinhardt, James R. Larus, David A. Wood

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

Abstract

This paper discusses implementations of fine-grain memory access control, which selectively restricts reads and writes to cache-block-sized memory regions. Fine-grain access control forms the basis of efficient cache-coherent shared memory. This paper focuses on low-cost implementations that require little or no additional hardware. These techniques permit efficient implementation of shared memory on a wide range of parallel systems, thereby providing shared-memory codes with a portability previously limited to message passing.

This paper categorizes techniques based on where access control is enforced and where access conflicts are handled. We incorporated three techniques that require no additional hardware into Blizzard, a system that supports distributed shared memory on the CM-5. The first adds a software lookup before each shared-memory reference by modifying the program's executable. The second uses the memory's error correcting code (ECC) as cache-block valid bits. The third is a hybrid. The software technique ranged from slightly faster to two times slower than the ECC approach. Blizzard's performance is roughly comparable to a hardware shared-memory machine. These results argue that clusters of workstations or personal computers with networks comparable to the CM-5's will be able to support the same shared-memory interfaces as supercomputers.

*This work is supported in part by NSF PYI/NYI Awards CCR-9157366 and CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, an AT&T Ph.D. Fellowship, and donations from Digital Equipment Corporation, 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 Univ. of Wisconsin Graduate School.

1 Introduction

Parallel computing is becoming widely available with the emergence of networks of workstations as the parallel "minicomputers" of the future [1]. Unfortunately, current systems directly support only message-passing communication. Shared memory is limited to page-based systems, such as TreadMarks [17], which are not sequentially consistent and which can perform poorly in the presence of fine-grain data sharing [11].

These systems lack *fine-grain access control*, a key feature of hardware shared-memory machines. Access control is the ability to selectively restrict reads and writes to memory regions. At each memory reference, the system must perform a *lookup* to determine whether the referenced data is in local memory, in an appropriate state. If local data does not satisfy the reference, the system must invoke a protocol *action* to bring the desired data to the local node. We refer to the combination of performing a lookup on a memory reference and conditionally invoking an action as *access control*. Access control *granularity* is the smallest amount of data that can be independently controlled (also referred to as the *block size*). Access control is fine-grain if its granularity is similar to a hardware cache block (32–128 bytes).

Current shared-memory machines achieve high performance by using hardware-intensive implementations of fine-grain access control. However, this additional hardware would impose an impossible burden in the cost-conscious workstation and personal computer market. Efficient shared memory on clusters of these machines requires low- or no-cost methods of fine-grain access control. This paper explores this design space by identifying where the lookup and action can be performed, fitting existing and proposed systems into this space, and illustrating performance trade-offs with a simulation model. The paper then focuses on three techniques suitable for existing hardware. We used these techniques to implement three variants of Blizzard, a system that uses the Tempest interface [32] to support distributed shared memory on a Thinking Machines CM-5. The first variant, Blizzard-S, adds a fast lookup before each shared-memory reference [22] by modifying the program's executable [21]. The second, Blizzard-E, employs the memory's error-correcting code (ECC) bits as block valid bits [30]. The third, Blizzard-ES, combines the two techniques.

Blizzard's performance—running six programs written for hardware cache-coherent shared-memory machines—is consistent with our simulation results. Blizzard-S's (software) performance ranged from slightly faster than Blizzard-E to twice as slow, depending on a program's shared-memory communication behavior. To calibrate Blizzard's absolute performance, we compared it against a Kendall Square Research KSR-1 shared-memory machine. For one program, Blizzard-E outperforms the KSR-1; for three others, it is within a factor of 2.4–3.6; and two applications ran 6–7 times faster on the KSR-1.

These results show that clusters of workstations or personal computers can efficiently support shared memory when equipped with networks and network interfaces comparable to the CM-5's [23]. Blizzard also demonstrates the portability provided by the Tempest interface. Tempest allows clusters to support the same shared-memory abstraction as supercomputers, just as MPI and PVM support a common interface for coarse-grain message passing.

The paper is organized as follows. Section 2 examines alternative implementations of fine-grain access control. In particular, Section 2.5 presents a simulation of the effect of varying access control overheads. Section 3 describes Blizzard. Finally, Section 4 concludes the paper.

2 Access Control Alternatives

Fine-grain access control performs a lookup at each memory reference and, based on the result of the lookup, conditionally invokes an action. The referenced location can be in one of three states: *ReadWrite*, *ReadOnly*, or *Invalid*. Program loads and stores have the following semantics:

load(*address*) =
  if (lookup(*address*) $\notin \{ReadOnly, ReadWrite\}$)
    invoke-action(*address*)
  perform-load(*address*)

store(*address*) =
  if (lookup(*address*) $\neq ReadWrite$)
    invoke-action(*address*)
  perform-store(*address*)

Lookup	**Action**		
	Dedicated Hardware	Primary Processor	Auxiliary Processor
Software		Orca (object) Blizzard-S	
TLB		IVY (page)	
Cache	Alewife[1] KSR-1	Alewife[1]	
Memory	S3.mp	Blizzard-E	FLASH
Snoop	DASH		Typhoon

[1] Location of action depends on protocol state.

Table 1: Taxonomy of shared-memory systems.

Fine-grain access control can be implemented in many ways. The lookup and action can be performed in either software, hardware, or a combination of the two. These alternatives have different performance, cost, and design characteristics. This section classifies access control techniques based on where the lookup is performed and where the action is executed. Table 1 shows the design space and places current and proposed shared-memory systems within it.

The following sections explore this taxonomy in more detail. Section 2.3 discusses the lookup and action overheads of the systems in Table 1. Section 2.4 discusses how the tradeoffs in the taxonomy affect a wide range of shared-memory machines. Section 2.5 presents a simulation study of the effect of varying access control overheads.

2.1 Where is the Lookup Performed?

Either software or hardware can perform an access check. A software lookup avoids the expense and design cost of hardware, but incurs a fixed overhead at each lookup. Hardware typically incurs no overhead when the lookup does not invoke an action. Lookup hardware can be placed at almost any level of the memory hierarchy—TLB, cache controller, memory controller, or a separate snooping controller. However, for economic and performance reasons, most hardware approaches avoid changes to commodity microprocessors.

Software. The code in a software lookup checks a main-memory data structure to determine the state of a block before a reference. As described in Section 3.2, careful coding and liberal use of memory makes this lookup reasonably fast. Our current implementation adds 15 instructions before each shared-memory load or store. Static analysis can detect and potentially eliminate redundant tests. However, the asynchrony in parallel programs makes it difficult to predict whether a cache block will remain accessible between two instructions.

Either a compiler or a program executable editing tool [21] can insert software tests. We use the latter approach in Blizzard so every compiler need not reimplement test analysis and code generation. Compiler-inserted lookups, however, can exploit application-level information. Orca [2], for example, provides access control on program objects instead of blocks.

TLB. Standard address translation hardware provides access control, though at memory page granularity. Nevertheless, it forms the basis of several distributed-shared-memory systems—for example, IVY [26], Munin [4], and TreadMarks [17]. Though unimplemented by current commodity processors, additional, per-block access bits in a TLB entry could provide fine-grain access control. The "lock bits" in some IBM RISC machines, including the 801 [7] and RS/6000 [29], provide access control on 128-byte blocks, but they do not support the three-state model described above.

Cache controller. The MIT Alewife [6] and Kendall Square Research KSR-1 [18] shared-memory systems use custom cache controllers to implement access control. In addition to detecting misses in hardware cache(s), these controllers determine when to invoke a protocol action. On Alewife, a local directory is consulted on misses to local physical addresses to determine if a protocol action is required. Misses to remote physical addresses always invoke an action. Due to the KSR-1's COMA architecture, any reference that misses in both levels of cache requires protocol action. A trend toward on-chip second-level cache controllers [15] may make modified cache controllers incompatible with future commodity processors.

Memory controller. If the system can guarantee that the processor's hardware caches never contain *Invalid* blocks and that *ReadOnly* blocks are cached in a read-only state, the memory controller can perform the lookup on hardware cache misses. This approach is used by Blizzard-E, Sun's S3.mp [28], and Stanford's FLASH [20].

As described in Section 3.3, Blizzard-E uses the CM-5's memory error-correcting code (ECC) to implement a cache-block valid bit. While effective, this approach has several shortcomings. *ReadOnly* access is enforced with page-level protection, so stores may incur an unnecessary protection trap. Also, modifying ECC values is an awkward and privileged operation. The Nimbus NIM6133, an MBUS memory controller co-designed by Nimbus Technology, Thinking Machines, and some of the authors [27], addressed these problems. The NIM6133 supports Blizzard-like systems by storing a 4-bit access control tag with each 32-byte cache block. The controller encodes state tags in unassigned ECC values, which requires no additional DRAM. On a block write, the controller converts the 4-bit tag to a unary 1-of-16 encoding. For each 64-bit doubleword in the block, it appends the unary tag, computes the ECC on the resulting 80-bit value, and stores the 64 data bits plus ECC (but not the tag). On a read, the controller concatenates 16 zeros to each 64-bit doubleword. The ECC single-bit error correction then recovers the unary tag value. Because the tag is stored redundantly on each doubleword in the block, double-bit error detection is maintained. Tag manipulations are unprivileged and the controller supports a *ReadOnly* state.

S3.mp has a custom memory controller that performs a hardware lookup at every bus request. FLASH's programmable processor in the memory controller performs the lookup in software. It keeps state information in regular memory and caches it on the controller.

Custom controllers are possible with most current processors. However, future processors may integrate on-chip memory controllers (as do the TI MicroSPARC and HP PA7100LC).

Bus snooping. When a processor supports a bus-based coherence scheme, a separate bus-snooping agent can perform a lookup similar to that performed by a memory controller. Stanford DASH [24] and Wisconsin Typhoon [32] employ this approach. On DASH, as on Alewife, local misses may require protocol action based on local directory state and remote misses always invoke an action. Typhoon looks up access control state for all physical addresses in a reverse-translation cache with per-block access bits that is backed by main-memory page tables.

2.2 Where is the Action Taken?

When a lookup detects a conflict, it must invoke an action dictated by a coherence protocol to obtain an accessible copy of a block. As with the lookup itself, hardware, software, or a combination of the two can perform this action. The protocol action software can execute either on the same CPU as the application (the "primary" processor) or on a separate, auxiliary processor.

Hardware. The DASH, KSR-1, and S3.mp systems implement actions in dedicated hardware, which provides high performance for a single protocol. While custom hardware performs an action quickly, research has shown that no single protocol is optimal for all applications [16] or even for all data structures within an application [3, 12]. High design costs and resource constraints also make custom hardware unattractive. Hybrid hardware/software

System	Lookup				Action		Remote miss time (approx.)
	Bytes/ Block	Where Performed	No Action	Action Needed	Where Executed	Action Invocation	
Alewife	8	cache	0	1 / ~10	hardware	0	30
					prim. proc.	13	
KSR-1	128	cache	0	~10	hardware	0	200
DASH	16	snoop	0	10 / ~20	hardware	0	100
FLASH	128	memory	0	4 / ~14	aux. proc.	0	~100
Typhoon	32	memory	0	3	aux. proc.	2	100
Blizzard-S	32	software	18	18	prim. proc.	25	6000
Blizzard-E r,w[1]	32	memory	0	~10	prim. proc.	250	6000
w[1]	32	software (OS)	230	230			
Blizzard-ES r	32	memory	0	~10	prim. proc.	250	6000
w	32	software	18	18		25	
Munin	4K	TLB	0	~50	prim. proc.	2.01 ms	??
TreadMarks	4K	TLB	0	~50	prim. proc.	2600	110,000

[1] Lookup cost for writes depends on whether there are *ReadOnly* blocks on the page (see Section 3.3).

Table 2: Overheads of fine-grain access control for various systems (in processor cycles).

protocols—e.g., Alewife's LimitLESS [6] and Dir_1SW [13]—implement the expected common cases in hardware and trap to system software to handle complex, infrequent events.

Primary processor. Performing actions on the main CPU provides protocol flexibility and avoids the additional cost of custom hardware or an additional CPU. Blizzard uses this approach, as do page-based DSM systems such as IVY and Munin. However, as the next section discusses, interrupting an application to run an action can add considerable overhead. Alewife addressed this problem with a modified processor that supports rapid context switches.

Auxiliary processor. FLASH and Typhoon achieve both high performance and protocol flexibility by executing actions on an auxiliary processor dedicated to that purpose. This approach avoids a context switch on the primary processor and may be crucial if the primary processor cannot recover from late arriving exceptions caused by an access control lookup in the lower levels of the memory hierarchy. In addition, an auxiliary processor can provide rapid invocation of action code, tight coupling with the network interface, special registers (e.g., Typhoon's home node and protocol state pointer registers), and special operations (e.g., FLASH's bit field instructions). Of course, the design effort increases as the processor is more extensively customized.

2.3 Performance

Table 2 summarizes the access control overheads and remote miss times for existing and proposed distributed-shared-memory systems [4, 5, 17, 20, 25, 32, 33]. Values marked with '~' are estimated.

The left side of Table 2 lists the overhead of testing a shared memory reference for accessibility. Software lookups incur a fixed overhead, while the overhead of hardware lookups depends on whether or not action is required. Hardware typically avoids overhead when no action is needed by overlapping the lookup and local data access. When action is required (e.g., a remote miss), the data cannot be used so the lookup counts as overhead. Alewife, DASH, and FLASH have two numbers in the "Action Needed" column because misses to remote physical addresses immediately invoke an action but misses to local addresses require an access to local directory state. For Munin and TreadMarks, this column reflects the overhead of a TLB miss and page-table walk to detect a page fault.

Table 2 also lists action invocation overheads. This overhead reflects the time required from when an access conflict is detected to the start of the protocol action (e.g., for software actions, the execution of the first instruction). Dedicated hardware incurs no overhead since the lookup and action mechanisms are tightly coupled. FLASH also has no overhead because the auxiliary processor is already running lookup code, so the overhead of invoking software is reflected in the "Action Needed" column. Typhoon's overhead is very low because, like FLASH, its auxiliary processor is customized for fast dispatch.

Systems that perform lookup in hardware and execute actions on the primary processor incur much higher invocation overheads. A noticeable exception to this rule is Alewife. Its custom support for fast context switching can invoke actions in 13 cycles. By contrast, TreadMarks requires 2600 cycles on a DECStation 5000/240 running Ultrix 4.3 [17]. Of course, the overhead is the fault of Ultrix 4.3, not TreadMarks. With careful kernel coding (on a different processor), Blizzard-E's invocation overhead is 250 cycles, including 50 cycles that are added to every

CM-5 trap by a workaround for a hardware bug.

The final column of Table 2 presents typical round-trip miss times for these systems. These times are affected by access control overheads and other factors, such as network overheads and latencies. The systems in the first group of Table 2 provide low-latency interconnects that are closely coupled to the dedicated hardware or auxiliary processors. At the other extreme, TreadMarks communicates through Unix sockets using heavy-weight protocols. Its send time for a minimum size message is 3200 cycles (80 μs) [17]. Blizzard benefits from the CM-5's low-latency network and user-level network interface. Blizzard's performance would be better if the network supported larger packets (as, for example, the CM-5E). To efficiently communicate, packets must hold at least a virtual address, program counter, and a memory block (40 bytes total on Blizzard). Our CM-5 limits packets to 20 bytes, which requires block data messages to be split into multiple packets. Our implementation buffers only packets that arrive out-of-order, which eliminates buffering for roughly 80% of all packets.

2.4 Discussion

Both the cost of implementing access control and its speed increase as the lookup occurs higher in the memory hierarchy and as more hardware resources (e.g., an auxiliary processor) are dedicated to protocol actions. Because of the wide range of possible implementation techniques, a designer can trade-off cost and performance in a family of systems.

In the high-end supercomputer market, implementations will emphasize performance over cost. These systems will provide hardware support for both the access control test and protocol action. An auxiliary processor in the memory system, as in FLASH and Typhoon, minimizes invocation and action overhead while still exploiting commodity (primary) processors. However, this approach requires either a complex ASIC or full-custom chip design, which significantly increases design time and manufacturing cost.

In mid-range implementations targeted toward clusters of high-performance workstations, the cost and complexity of additional hardware is more important because workstations must compete on uniprocessor cost/performance. For these systems, simple hardware support for the test—as in the Nimbus memory controller—may be cost-effective.

The low end of parallel systems—networks of personal computers—will not tolerate additional hardware for access control. For these systems, implementations must rely on software access control, like that described in Section 3.2.

These tradeoffs would change dramatically if access control was integrated into commodity processors. For example, combining an RS/6000-like TLB with Alewife's context switching support would permit fast access control and actions at low hardware cost. Unfortunately, modifying a processor chip is prohibitively expensive for most, if not all, parallel system designers. Even the relatively cost-insensitive supercomputer manufacturers are resorting to commodity microprocessors [19] because of the massive investment to produce competitive processors. Commodity processor manufacturers are unlikely to consider this hardware until fine-grain distributed shared memory is widespread. The solutions described in this paper and employed by Blizzard provide acceptable performance on existing hardware to break this chicken and egg problem.

2.5 Access Control Overheads

This section describes a simulation that studies the effect of varying the overhead of access control and action invocation on the performance of a fine-grain distributed shared-memory system. Our simulator is a modified version of the Wisconsin Wind Tunnel [31] modeling a machine similar to the CM-5. The target nodes contain a single-issue SPARC processor that runs the application, executes protocol action handlers on access faults, and handles incoming messages via interrupts. As on the CM-5, the processor has a 64 Kbyte direct-mapped cache with a 32-byte line size. Instruction cycle times are accurately modeled, but we assume a perfect instruction cache. Local cache misses take 29 cycles. Misses in the fully-associative 64-entry TLB take 25 cycles. Message data is sent and received using single-cycle 32-bit stores and loads to a memory-mapped network interface. Message interrupts incur a 100-cycle overhead before the interrupt handler starts. Fine-grain access control is maintained at 32-byte granularity. The applications run under the full-map, write-invalidate Stache coherence protocol with 32-byte blocks [32].

In the simulations of two programs shown in Figure 1, we varied the overhead of lookups and the overhead of invoking an action handler. The "ideal" case is an upper bound on performance. It models a system in which access fault handlers and message processing run on a separate, infinitely-fast processor. In particular, the protocol software runs in zero time without polluting the processor's cache. However, to make the simulation repeatable, message sends are charged one cycle. The ideal case is 2.2–2.8× faster than a realistic system running protocol software on the application processor with hardware access control that reduces lookup overhead to zero and invocation overhead near zero. The simulations show that lookup overhead has a far larger effect on system performance than invocation overhead. For example, in

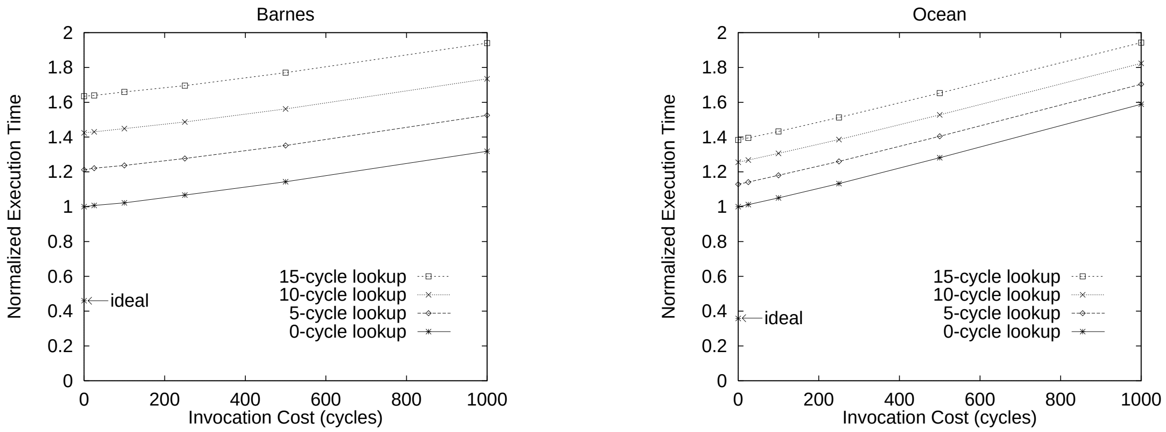

Figure 1: Simulation of fine-grain access control overheads.

Barnes, increasing the invocation overhead from 0 to 1000 cycles decreases performance less than increasing the lookup overhead from 0 to 5 cycles.

3 Access Control in Blizzard

Blizzard is our system that implements the Tempest interface on a Thinking Machines CM-5. Tempest is a communication and memory management interface [32] that can be supported on on a wide range of systems, ranging from multi-million-dollar supercomputers to low-cost clusters of workstations. Tempest provides four basic mechanisms necessary for both fine-grain shared memory and message passing [32]: active messages, bulk data transfer, virtual memory management, and fine-grain access control. This section presents an overview of Blizzard, with a focus on alternative implementations of fine-grain access control. Although we implemented these techniques for Tempest, they could also be used in other distributed-shared-memory systems.

Blizzard consists of a modified version of the CM-5 operating system and a user-level library. A shared-memory program is executed by compiling it with a standard compiler (e.g., *gcc*), linking it with the Blizzard library and a Tempest-compliant user-level protocol (e.g., Stache [32]), and running it on a CM-5 with the modified OS.

The next section describes our modifications to the CM-5 operating system. We then describe the three implementations of fine-grain access control: Blizzard-S, Blizzard-E, and Blizzard-ES.

3.1 Kernel Support for Blizzard

The Thinking Machines CM-5 [14] is a distributed-memory message-passing multiprocessor. Each processing node consists of a 33 MHz SPARC microprocessor with a 64 KB direct-mapped unified cache and a memory management unit, up to 128 MB of memory, a custom network interface chip, and optional custom vector units.

Blizzard uses a variant of the "executive interface" extensions developed for the Wisconsin Wind Tunnel [30]. These extensions provide protected user-level memory-management support, including the ability to create, manipulate, and execute within subordinate contexts. The executive interface also provides support for fine-grain access control using a memory tag abstraction. Although the executive interface provides the required functionality, there are several important differences discussed below.

First, the executive interface is optimized for switching contexts on all faults, which incurs a moderately-high overhead due to SPARC register window spills, etc. Tempest handles faults in the same address space and runs most handlers to completion. This change allowed a much faster implementation, in which exceptions (including user-level message interrupts) are handled on the same stack. Exceptions effectively look like involuntary procedure calls, with condition codes and other volatile state passed as arguments. In the common case, this interface eliminates all unnecessary stack changes and register window spills and restores. Furthermore, handlers can usually resume a faulting thread without entering the kernel. A kernel trap is only required in the relatively rare cases when the handler must re-enable hardware message interrupts or the SPARC PC and NPC are not sequential.

Second, Tempest requires that active message handlers and access fault handlers execute atomically. However, we use the CM-5's user-level message interrupt capability to implement our active message model. To preserve atomicity, we need to disable user-level interrupts while running in a handler. Unfortunately, the CM-5 does not provide user-level access to the interrupt mask, so it requires expensive kernel traps to both disable and re-enable interrupts.

Instead, we use a software interrupt masking scheme similar to one proposed by Stodolsky, et al. [35]. The key observation is that interrupts occur much less frequently than critical sections, so we should optimize for this common case. This approach uses a software flag to mark critical sections. The lowest-level interrupt handler checks this "software-disable" flag. If it is set, the handler sets a "deferred-interrupt" flag, disables further user-level hardware interrupts, and returns. On exit from a critical section, code must first clear the software-disable flag and then check for deferred interrupts. After processing deferred interrupts, the user-level handler traps back into the kernel to re-enable hardware interrupts. Stodolsky, et al.'s implementation uses a static variable to store the flags. To minimize overhead, our scheme uses a global register.

3.2 Blizzard-S: Software

Blizzard-S implements fine-grain access control entirely in software, using a variant of the Fast-Cache simulation system [22]. Fast-Cache rewrites an existing executable file [21] to insert a state table lookup before every shared-memory reference. The lookup table is indexed by the virtual address and contains two bits for each 32-byte block (the size is a compile-time constant). The state and reference type (i.e., `load` or `store`) determine the handler. When the current state requires no action (i.e., a `load` to a ReadWrite block) Blizzard-S invokes a special NULL handler which immediately resumes execution. Otherwise, it invokes a user handler through a stub that saves processor state. With the table lookup and null handlers, Blizzard-S avoids modifying the SPARC condition codes, which are expensive to save and restore from user code. Although Blizzard-S reserves address space for a maximum sized lookup table, it allocates the table on demand, so memory overhead is proportional to the data set size.

The lookup code uses two global registers left unused by programs conforming to the SPARC application binary interface (ABI). These registers are temporaries used to calculate the effective address, index into the lookup table, and invoke the handler. The current implementation adds 15 instructions (18 cycles in the absence of cache and TLB misses) before all load and store instructions that cannot be determined by inspection to be a stack reference. Simple optimizations, such as scavenging free registers and recognizing redundant tests could lower the average overhead, but these were not completed in time for inclusion in this paper.

To avoid inconsistency, interrupts cannot be processed between a lookup and its corresponding reference. Disabling and re-enabling interrupts on every reference would increase the critical lookup overhead. Instead, we permanently disable interrupts with the software flag described above, leaving hardware interrupts enabled, and periodically poll the deferred-interrupt flag. Because the deferred-interrupt flag is a bit in a global register, the polling overhead is extremely low. Our current implementation polls on control-flow back-edges.

3.3 Blizzard-E: ECC

Although several systems have memory tags and fine-grain access control, e.g., J-machine [10], most contemporary commercial machines—including the CM-5—lack this facility. In Blizzard-E, we synthesized the *Invalid* state on the CM-5 by forcing uncorrectable errors in the memory's error correcting code (ECC) via a diagnostic mode [30, 31]. Running the SPARC cache in write-back mode causes all cache misses to appear as cache block fills. A fill causes an uncorrectable ECC error and generates a precise exception, which the kernel vectors to the user-level handler. The Wisconsin Wind Tunnel [31] and Tapeworm II [36] both use this ECC technique to simulate memory systems.

This technique causes no loss of reliability. First, uncorrectable ECC faults are treated in the normal way (e.g., panic) unless a program specified a handler for a page. Second, the ECC is only forced "bad" when a block's state is Invalid, and hence the block contains no useful data. Third, the Tempest library and kernel maintain a user-space *access bit vector* that verifies that a fault should have occurred. The final possibility is that a double-bit error changes to a single-bit error, which the hardware automatically corrects. This is effectively solved by writing bad ECC in at least two double-words in a memory block, so at least two single bit errors must occur.

Unfortunately, the ECC technique provides only an *Invalid* state. Differentiating *ReadOnly* and *ReadWrite* is more complex. Blizzard-E uses the MMU to enforce read-only protection. If any block on a page is *ReadOnly*, the page's protection is set read-only. On a write-protection fault, the kernel checks the access bit vector. If the block is *ReadOnly*, the fault is vectored to the user-space Blizzard-E handler. If the block is *ReadWrite*, the kernel completes the write and resumes the application. Despite careful coding, this path through the kernel still requires ~230 cycles.

Protection is maintained in two ways. First, this check is only performed if the user has installed an access bit vector for the page. This ensures that write faults are only processed in this fashion on Blizzard-E's shared-data pages. Second, the kernel uses the SPARC MMU's "no fault" mode to both

Benchmark	Brief Description	Input
Appbt	Computational fluid dynamics	32^3, 10 iters
Barnes	Barnes-Hut N-body simulation	8192 bodies
Mp3d	Hypersonic flow simulation	24000 mols, 50 iters
Ocean	Hydrodynamic simulation	386 x 386, 8 days
Tomcatv	Parallel version of SPEC benchmark	1026^2, 50 iters
Water	Water molecule simulation	256 mols, 10 iters

Table 3: Benchmark descriptions.

Application	Blizzard-E	Blizzard-S	Blizzard-ES	Blizzard-P	KSR-1
Appbt	137 (1.00)	177 (1.29)	142 (1.04)	732 (5.35)	38 (0.28)
Barnes	48 (1.00)	60 (1.27)	51 (1.07)	288 (6.05)	7 (0.14)
Mp3d	134 (1.00)	132 (0.98)	147 (1.09)	716 (5.33)	24 (0.18)
Ocean	81 (1.00)	111 (1.37)	82 (1.01)	380 (4.67)	34 (0.42)
Tomcatv	78 (1.00)	162 (2.08)	87 (1.11)	478 (6.12)	94 (1.20)
Water	57 (1.00)	83 (1.47)	62 (1.09)	99 (1.74)	16 (0.28)

Table 4: Execution time in CPU seconds and (in parentheses) relative to Blizzard-E.

read the access bit vector and perform the store, allowing it to safely perform these operations with traps disabled.

3.4 Blizzard-ES: Hybrid

We also implemented a hybrid version of Blizzard that combines ECC and software checks. It uses ECC to detect the *Invalid* state for load instructions, but uses executable rewriting to perform tests before store instructions. This version—Blizzard-ES—eliminates the overhead of a software test for load instructions and the overhead introduced for stores to *ReadWrite* blocks on read-only pages in Blizzard-E.

3.5 Blizzard Performance

We examined the overall performance of Blizzard for six shared-memory benchmarks, summarized in Table 3. These benchmarks—four from the SPLASH suite [34]—were written for hardware shared-memory systems. Page-granularity DSM systems generally perform poorly on these codes because of their fine-grain communication and write sharing [9].

We ran these benchmarks on five 32-node systems: Blizzard-E, Blizzard-S, Blizzard-ES, Blizzard-P, and a Kendall Square KSR-1. The first three Blizzard systems use a full-map invalidation protocol implemented in user-level software (Stache) [32] with a 128-byte block size. Blizzard-P is a sequentially-consistent, page-granularity version of Blizzard. The KSR-1 is a parallel processor with extensive hardware support for shared memory. Table 4 summarizes the performance of these systems. It contains both the measured times of these programs and the execution time relative to that of Blizzard-E.

Blizzard-E usually ran faster than Blizzard-S (27%–108%), although for *Mp3d*, Blizzard-S is 2% faster. Blizzard-E's performance is generally better for computation-bound codes, such as *Tomcatv*, in which remote misses are relatively rare. Blizzard-S performs well for programs, such as *Mp3d* and *Barnes*, that have frequent, irregular communication and many remote misses. Surprisingly, Blizzard-ES is always worse than Blizzard-E. This indicates that writes to cache blocks on read-only pages are infrequent and that synthesizing Tempest's four memory states by a combination of valid bits and page-level protection is viable. Blizzard-P predictably performs worse than the fine-grain shared-memory systems (74% to 512% slower than Blizzard-E) because of severe false-sharing in these codes. Relaxed consistency models would certainly help, but we have not implemented them.

To provide a reference point to gauge the absolute performance of Blizzard, we executed the benchmarks on a commercial shared-memory machine, the KSR-1.[1] The KSR-1 ranges from almost 7 times faster to 20% slower than Blizzard-E. These results

[1] KSR operating system version R1.2.1.3 (release) and C compiler version 1.2.1.3-1.0.2.

are encouraging given the KSR-1's extensive hardware support for shared memory and relative performance of the processors. The KSR-1 uses a custom dual-issue processor running at 20 MHz, while the CM-5 uses a 33 MHz SPARC. Uniprocessor measurements indicate that the CM-5 has slightly higher performance for integer codes, but much lower floating-point performance. (We currently do not support the CM-5 vector units.)

The variation in KSR-1 performance can be explained by the ratio of computation to communication in each program. *Appbt*, *Ocean*, and *Water* are dominated by computation. On these benchmarks, Blizzard-E's performance is within a factor of four of the KSR-1, which is consistent with the difference in floating point performance. *Tomcatv* is also compute-bound and should behave similarly; we were unable to determine why it performs poorly on the KSR-1. Most of *Tomcatv*'s computation is on large, private arrays, and it is possible that the KSR-1 suffers expensive, unnecessary remote misses on these arrays due to cache conflicts. *Mp3d* incurs a large number of misses due to poor locality [8]. The high miss ratio explains both Blizzard-E's poor performance relative to the KSR-1and Blizzard-S's ability to outperform Blizzard-E. *Barnes* also has frequent, irregular communication that incurs a high penalty on Blizzard.

4 Summary and Conclusions

This paper examines implementations of fine-grain memory access control, a crucial mechanism for efficient shared memory. It presents a taxonomy of alternatives for fine-grain access control. Previous shared-memory systems used or proposed hardware-intensive techniques for access control. Although these techniques provide high performance, the cost of additional hardware precludes shared memory from low-cost clusters of workstations and personal computers.

This paper describes several alternatives for fine-grain access control that require no additional hardware, but provide good performance. We implemented three in Blizzard, our system that supports fine-grain distributed shared memory on the Thinking Machines CM-5. Blizzard-S relies entirely on software and modifies an application's executable to insert a fast (15 instruction) access check before each load or store. Blizzard-E uses the CM-5's memory error correcting code (ECC) to mark invalid cache-block-sized regions of memory. Blizzard-ES is a hybrid that combines both techniques. The relative performance of these techniques depends on an application's shared-memory communication, but on six programs, Blizzard-S ran from 2% faster to 108% slower than Blizzard-E.

We believe that the CM-5's network interface and network performance is similar to facilities that will be available soon for commodity workstations and networks, so Blizzard's performance is indicative of how these techniques will perform on widely-available hardware in the near future. We ran six applications, written for hardware shared-memory machines, and compared their performance on Blizzard and the KSR-1. The results are very encouraging. Blizzard outperforms the KSR-1 for one program. For three others Blizzard is within a factor of 2.4–3.6 times. Only two of the six applications run more than four times faster on the KSR-1, and none more than seven times faster, despite its hardware shared-memory support and faster floating-point performance.

While Blizzard on the CM-5 will not supplant shared-memory machines, these results show that programmers need not eschew shared memory in order to run on a wide variety of systems. A portable interface—such as Tempest—can provide the same shared-memory abstraction on a cluster of personal computers as on a supercomputer. The software approach of Blizzard-S provides an acceptable common denominator for widely-available low-cost workstations. Higher performance, at a higher price, can be achieved by tightly-coupled parallel supercomputers, either current machines like the KSR-1 and KSR-2 or future machines that may resemble Typhoon or FLASH. The widespread availability of shared-memory alternatives will hopefully motivate manufacturers to develop midrange systems using Blizzard-E-like technology (e.g., the Nimbus NIM6133).

Acknowledgments

This work was performed as part of the Wisconsin Wind Tunnel project, which is co-lead by Profs. Mark Hill, James Larus, and David Wood and funded by the National Science Foundation. We would like to thank Mark Hill, Anne Rogers, and Todd Austin for helpful comments on this research and earlier drafts of this paper. We would especially like to thank the Universities of Washington and Michigan for allowing us access to their KSR-1s.

References

[1] Tom Anderson, David Culler, and David Patterson. A Case for Networks of Workstations: NOW. Technical report, Computer Science Division (EECS), University of California at Berkeley, July 1994.

[2] Henri E. Bal, Andrew S. Tanenbaum, and M. Frans Kaashoek. Orca: A Language for Distributed Programming. *ACM SIGPLAN Notices*, 25(5):17–24, May 1990.

[3] John K. Bennett, John B. Carter, and Willy Zwaenepoel. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. In *Second ACM SIGPLAN*

Symposium on Principles & Practice of Parallel Programming (PPOPP), pages 168–176, February 1990.

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

[5] David Chaiken and John Kubiatowicz. Personal Communication, March 1994.

[6] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, April 1991.

[7] Albert Chang and Mark F. Mergen. 801 Storage: Architecture and Programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[8] David R. Cheriton, Hendrik A. Goosen, and Philip Machanick. Restructuring a Parallel Simulation to Improve Cache Behavior in a Shared-Memory Multiprocessor: A First Experience. In *International Symposium on Shared Memory Multiprocessing*, pages 109–118, April 1991.

[9] 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.

[10] William J. Dally and D. Scott Wills. Universal Mechanism for Concurrency. In *PARLE '89: Parallel Architectures and Languages Europe*. Springer-Verlag, June 1989.

[11] Susan J. Eggers and Randy H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 257–270, 1989.

[12] 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*, November 1994. To appear.

[13] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in ASPLOS V, Oct. 1992.

[14] W. Daniel Hillis and Lewis W. Tucker. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM*, 36(11):31–40, November 1993.

[15] Peter Yan-Tek Hsu. Designing the TFP Microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[16] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive Snoopy Caching. *Algorithmica*, (3):79–119, 1988.

[17] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. Technical Report 93-214, Department of Computer Science, Rice University, November 1993.

[18] Kendall Square Research. Kendall Square Research Technical Summary, 1992.

[19] R. E. Kessler and J. L. Schwarzmeier. CRAY T3D: A New Dimension for Cray Research. In *Proceedings of COMPCON 93*, pages 176–182, San Francisco, California, Spring 1993.

[20] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[21] James R. Larus and Thomas Ball. Rewriting Executable Files to Measure Program Behavior. *Software Practice & Experience*, 24(2):197–218, February 1994.

[22] Alvin R. Lebeck and David A. Wood. Fast-Cache: A New Abstraction for Memory System Simulation. Technical Report 1211, Computer Sciences Department, University of Wisconsin–Madison, January 1994.

[23] Charles E. Leiserson et al. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1992.

[24] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[25] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):41–61, January 1993.

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

[27] NIMBUS Technology. NIM 6133 Memory Controller Specification. Technical report, NIMBUS Technology, 1993.

[28] A. Nowatzyk, M. Monger, M. Parkin, E. Kelly, M. Borwne, G. Aybay, and D. Lee. S3.mp: A Multiprocessor in a Matchbox. In *Proc. PASA*, 1993.

[29] R. R. Oehler and R. D. Groves. IBM RISC System/6000 processor architecture. *IBM Journal of Research and Development*, 34(1):32–36, January 1990.

[30] 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.

[31] 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.

[32] 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.

[33] Rafael H. Saavedra, R. Stockton Gaines, and Michael J. Carlton. Micro Benchmark Analysis of the KSR1. In *Proceedings of Supercomputing 93*, pages 202–213, November 1993.

[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] Richard Uhlig, David Nagle, Trevor Mudge, and Stuart Sechrest. Tapeworm II: A New Method for Measuring OS Effects on Memory Architecture Performance. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, October 1994. To appear.