Application-Specific Protocols for User-Level Shared Memory*

Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, Ioannis Schoinas,
Mark D. Hill, James R. Larus, Anne Rogers†, David A. Wood

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

Abstract

Recent distributed shared memory (DSM) systems and proposed shared-memory machines have implemented some or all of their cache coherence protocols in software. One way to exploit the flexibility of this software is to tailor a coherence protocol to match an application's communication patterns and memory semantics. This paper presents evidence that this approach can lead to large performance improvements. It shows that application-specific protocols substantially improved the performance of three application programs—appbt, em3d, *and* barnes—*over carefully tuned transparent shared memory implementations. The speed-ups were obtained on* Blizzard, *a fine-grained DSM system running on a 32-node Thinking Machines CM-5.*

1 Introduction

A shared address space is central to many parallel languages and models of parallel computation. It provides the global names for data that enable a processor to reference any datum, regardless of its location in a parallel system. The difficulty, of course, comes in efficiently detecting a reference to a remote datum and communicating its current value. Shared address space systems implement these operations with one of three different techniques. First, custom hardware in a shared-memory machine directly validates a referenced address, fetches, and possibly caches remote data. Second, a distributed shared memory (DSM) system relies on a machine's existing virtual memory protection to detect a remote access and system software to transfer data with explicit messages. Finally, a compiler implements an address space by detecting remote accesses with static program analysis or run-time tests and transfers data through explicit messages.

These systems have gained a reputation for performing poorly in many circumstances and being overly sensitive to programs' spatial locality and false sharing [12]. One reason for shared memory's poor performance is that it typically implements only a single coherence protocol. The policy embodied in a protocol controls a system's response to a remote reference and therefore the message traffic between processors. Most systems provide a single, fixed coherence policy that must work in all situations. A general policy can result in excess traffic in simple situations. For example, the widely-used write-invalidate protocol requires four messages to transfer a value in a producer-consumer relationship (request from consumer, invalidate to producer, acknowledge from producer, and response to consumer). A more appropriate, write-update protocol can transfer a value with one or two messages, but is not generally applicable as it does not ensure sequential consistency.

One solution is to select a coherence protocol to match an application's semantics and sharing pattern. This approach has been used to improve the performance of DSM systems, such as Munin [2], which allows programmers to annotate data structures with expected access patterns (e.g., read-only or migratory) that Munin uses to select an appropriate coherence protocol. DSM systems, unfortunately, perform access control at the page level, which can cause considerable false sharing.

Stanford FLASH [11] and Wisconsin Typhoon [21] are two recent "hardware" shared-memory systems that implement their protocols in software running on custom coprocessors, which allows protocol spe-

*This work is supported in part by NSF PYI/NYI Awards MIP-8957278, CCR-9157366, and CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, an AT&T Ph.D. Fellowship, 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.

†On leave from Princeton University.

cialization. In many ways, these systems are an evolutionary unification of DSM-like custom protocols with hardware, cache-coherent systems, which move data in cache blocks (not pages). FLASH and Typhoon provide software with full control over coherence policy, unlike earlier hardware-software protocol hybrids (e.g., MIT Alewife [3] and Wisconsin Dir_1SW [7, 26]).

A second advantage of software protocols is that they lead to a natural unification of message passing with shared memory. A shared-memory program can send messages when a user or compiler wants to bypass shared-memory overheads and can determine in advance how data will be used [5, 10]. A third advantage is the opportunity to develop language-specific memory models, such as Loosely Coherent Memory [14].

Thus far we have argued the benefits of software coherence protocols without specifying whether the software runs in the system (in kernel mode) or at user level. In fact, the systems discussed above, except Munin and Typhoon, run protocol software in kernel mode. As with the tradeoffs between monolithic operating systems and micro-kernels, incorporating protocols into a system has costs and benefits. First, system-provided protocols are necessarily limited and may not match an application's needs. Second, system-provided protocols must be parameterized and formulated in a general manner (for example, to allow a variety of cache block sizes), which introduces unnecessary run-time costs [27]. Finally, system-level protocols make experimentation difficult and time-consuming and restrict who can try new protocols.

Protocols that run at user-level (as part of an application) overcome these drawbacks. New protocols can be written at will, specialized for specific uses (by a programmer or compiler), and tested without operating system changes. User-level protocols also have disadvantages. First, specializing protocols may be too difficult or expensive for most programmers to write, particularly if a protocol must be re-written for every machine. Second, specialized protocols may not yield performance benefits that exceed the costs of running at user-level (e.g., address translation overhead). Finally, the system must ensure that incorrect or malicious user-level protocols do not violate protection or deadlock the system. This paper addresses the first two concerns, but not the third.

The *Tempest* interface [21] (Section 2) addresses the first concern. Tempest provides a standard, system-independent interface to mechanisms that enable programmers, compilers, and program libraries to implement and use message passing, transparent shared memory, and hybrid combinations of the two. Tempest's mechanisms are low-overhead messages, bulk data transfer, virtual memory management, and fine-grain access control. Its most novel mechanism—fine-grain access control—allows user software to tag blocks (e.g., 32 bytes) as read-write, read-only, or invalid, so local memory can cache remote data transparently [22]. We have implemented Tempest both in a simulation of custom hardware [21] and directly on a CM-5 [22]. We are currently porting Tempest to a network of workstations.

Section 3 describes *Blizzard*, our Tempest implementation on a Thinking Machines CM-5 [22]. Blizzard consists of a modified version of the CM-5 operating system and a user-level library. Blizzard uses the CM-5's ECC bits and page-level address translation hardware to synthesize fine-grain access control (at cache block granularity). A Tempest application program is compiled with a standard compiler (e.g., *gcc*), linked with the Blizzard library and a Tempest-compliant user-level protocol (e.g., Stache [21]), and runs directly on a CM-5.

Section 4 addresses our second concern, namely, whether tailored protocols will yield performance gains. We optimize protocols for three applications running on Blizzard. For each application, we begin with a transparent shared memory program that uses a standard, library-provided protocol. Through successive refinement, we identify performance bottlenecks and alter both the application and protocol. Custom protocols improve the performance of the three programs (*appbt*, *em3d*, and *barnes*) by factors of 5.7, 16, and 1.4, respectively (ignoring initialization and the first iteration).

We achieved significant improvements with three variations of an update protocol. The three protocols differ greatly, however, depending on whether the communication pattern is known at compile-time (*appbt*), fixed for each run (*em3d*), or dynamic (*barnes*). While three application programs cannot provide a definitive answer, our opinion is that system-provided protocols are unlikely to be both general and fast enough to compete with application-specific protocols.

2 Tempest

2.1 Tempest Mechanisms

Application-specific protocols use software to implement memory system policies, but rely on system-supplied mechanisms for memory access detection and communication. The Tempest mechanisms [21] are general enough that programmers, protocol libraries, or compilers can use them to support shared memory, message passing, or hybrid (i.e., combina-

tion) applications. The four types of Tempest mechanisms are:

Low-Overhead "Active" Messages. Tempest supports an active message abstraction, in which each message specifies a destination node, handler address, and a string of arguments [25]. When a message arrives at its destination, it creates a thread that runs the handler atomically with respect to other message handlers. Nothing guarantees atomicity between a handler and the destination node's computation thread, except explicit (user-level) synchronization.

Bulk Data Transfer. Tempest includes a collection of synchronous and (logically) asynchronous bulk data transfer mechanisms. Variants include a DMA-like transfer—in which the initiator specifies a length, initial virtual address on the source node, and initial virtual address on the destination node—and a channel-based transfer—similar to the CM-5's CMMD library.

Virtual Memory Management. Virtual memory management enables user-level code to manage virtual memory pages allocated in a special segment. The user can map a virtual page into the physical memory of more than one processor.

Fine-Grained Memory Access Control. Finally, Tempest allows a user to protect an aligned memory block (e.g., 32 bytes—much smaller than a page) as *ReadWrite*, *ReadOnly*, or *Invalid*. Inappropriate accesses to a block (e.g., a store into a *ReadOnly* block) generate faults that are vectored to a user-level handler.

2.2 Shared Memory with Tempest

Tempest does not force programmers to use a shared address space. The underlying process model is a conventional distributed memory, one process-per-processor model, in which each process has its own private address space. Consequently, Tempest can support most message-passing programs with its active message and bulk data transfer mechanisms. The power of Tempest, however, comes from its support for a shared address space. While Tempest does not implement the shared address space directly, it provides mechanisms that enable user-level software to do so. Applications can manage their address space explicitly or, more likely, through a standard library.

For example, the default Tempest library provides transparent shared-memory semantics using the *Stache* allocation policy [21], which is similar to Li and Hudak's fixed distributed manager [16]. The library allocates a region of each process's private address space for a shared segment. Pages in this segment are assigned a unique "home" processor node, which provides the physical memory. Home nodes can be allocated flexibly. The current library supports allocating a set of shared pages on a particular node or round-robin across all nodes. The library maintains a user-level page table to map a shared address to its home node.

The Stache policy allows a page's home node to access it directly. However, all other nodes must first map it to their *Stache*—a region of local memory that the library manages as a second- (or third-) level cache for remote data. On the first reference to a remote page, the library (invoked by the page fault handler) allocates a corresponding "stache" page to hold local copies of requested data blocks. Initially, all blocks in a stache page have their access control tags set to *Invalid*.

For both home and stache pages, the library installs a unique set of default handlers that dictate the protocol actions. The default coherence protocol—*Stache128*—closely resembles an all-software implementation of LimitLESS [3], except that the block size is 128 bytes. For example, when a processor tries to read an *Invalid* stache block, the Stache128 handler sends an active message to the home node requesting a read-only copy. The message handler performs any necessary protocol actions before responding with the requested data block.

The Stache128 protocol provides sequentially-consistent shared-memory semantics, with 128-byte blocks. However, the protocol is easily modified, because it is implemented in user-level handlers. As a trivial example, changing the block size to 32 bytes—to reduce false sharing in a fine-grain application—simply requires recompiling the Stache128 protocol. Furthermore, Tempest allows multiple protocols to co-exist by specifying different handlers for different pages. Thus an application could use Stache128 to maintain most data structures, but specify Stache32—or some other customized protocol—for a data structure that exhibits false-sharing.

2.3 Implementing Tempest

Tempest's messaging and virtual memory support are largely conventional. Active message abstractions can be implemented very efficiently with custom hardware [6, 18], but also have reasonable performance on existing machines [25]. Tempest's virtual memory mechanisms can be implemented as a user-level library on a system that provides `mmap()` and `munmap()` or with custom kernel modifications [19].

The key challenge in implementing Tempest is supporting fine-grain access control. Schoinas, et al. [22], showed that many good access control techniques exist, even for existing machines. At one extreme, the

proposed Wisconsin Typhoon system adds a custom network interface processor to the cache-coherent bus of a workstation-like processing node to get (simulated) shared-memory performance comparable to a conventional directory-based cache-coherent machine (e.g., Stanford DASH [15]). At the other extreme, fine-grain access control can be supported with no hardware changes by rewriting the executable files [13] to insert software tests.

3 Blizzard

This section describes *Blizzard*, the Tempest implementation that we use to study the performance of application-specific protocols. Blizzard implements the Tempest mechanisms on a Thinking Machines CM-5 [22]. The CM-5 is a distributed-memory, message-passing parallel computer, in which each processing node contains a 33MHz SPARC microprocessor with a 64KB direct-mapped unified cache, memory management unit, up to 128MB of memory, a custom network interface chip, and optional custom vector units [8]. Blizzard does not use the vector units.

Blizzard consists of an augmented version of the CM-5's CMOST operating system, a user-level library containing the interface to the Tempest mechanisms, and libraries of default user-level protocols. Blizzard uses a variant of the "executive interface" extensions developed for the Wisconsin Wind Tunnel [19]. This interface provides user-level virtual memory management routines and fine-grain access control functionality needed by Tempest. Blizzard's executive interface implementation is faster than the Wind Tunnel's, because handlers run in the same address space (context) as the application and because we tuned its performance (e.g., by eliminating SPARC register window overflows on commonly-executed paths).

To make Tempest's message and fault handlers atomic (with respect to each other) without invoking system calls to re-enable interrupts, we use an interrupt masking scheme similar to Stodolsky et al.'s [24]. Critical sections set a "software-disable" flag on entry and check a "deferred-interrupt" flag on exit. If an interrupt encounters a "software-disable" flag, the handler queues the interrupt and sets the "deferred-interrupt" flag.

Tempest's active messages are more general than those in CMAML [25] or TMC's CMMD message-passing library. In particular, Tempest allows messages larger than the CM-5 packet size. This generality makes Tempest protocols independent of a particular implementation, but degrades performance on the CM-5 because messages must be divided into packets and reassembled. Blizzard mitigates the effect of this feature by buffering packets only when they arrive out of order, which is typically 10-20% of packets in our applications.

Blizzard uses ECC bits to implement fine-grain memory access control (as does the Wisconsin Wind Tunnel [20]). It synthesizes the *Invalid* state by intentionally setting incorrect ECC values on affected locations. The *ReadOnly* state requires write protecting the page containing the block. Writes to *ReadWrite* blocks on the same page as a *ReadOnly* block cause a protection exception that the system detects and suppresses to complete the write operation. Surprisingly, this use of ECC bits does not reduce ECC coverage [20].

ECC bits, however, are not required to implement Tempest. Schoinas, et al. [22], showed that a Blizzard variant that added an explicit, software test before shared-memory loads and stores runs from 108% slower to 2% faster than Blizzard using ECC. The software version is not always slower, because it does not have to pay the ECC trap overhead on each miss. The performance of this variant could also be improved by optimizing the tests.

To provide a reference point to gauge the absolute performance of Blizzard, Schoinas, et al. [22] compared a 32-node Blizzard to a Kendall Square Research KSR-1 [9]. On six benchmarks Blizzard ranged from 18% to 120% of the speed of a KSR-1. The variation occurs because the ratio of computation to communication differed in each program. These results are encouraging given that the CM-5 was not designed to run Tempest and that, without vector units, the CM-5 has much lower floating-point performance than a KSR-1.

4 Custom Protocols

This section describes the three applications that we studied. In each case, we first used a variety of techniques to improve a program's performance under transparent shared memory as far as possible. We then implemented a custom protocol for the application and compared the resulting performance against the best shared memory version.

For each application, we include a table that reports timings—for 32 processors—for a series of implementations of that application. In these tables, a line separates the results for best transparent shared memory implementation from those for implementations that rely on Tempest-specific mechanisms. All performance improvements given in this section ignore the first iteration, because production runs of

all three applications would require a large number of iterations.

4.1 APPBT

Appbt is one of the NAS Parallel Benchmarks [1] produced by NASA Ames as representative of the computation and communication patterns in three-dimensional computational fluid dynamics applications. At each time step, *appbt* performs three computation phases in each of three dimensions. In phase one, it calculates a block tridiagonal matrix A. In the second phase, it solves $Ax = b$ for x using Gaussian elimination. In the third phase, it recomputes the right hand side vector b. The first computation has no dependences in all three dimensions. In the second and third phase, the values of a grid point are computed based on the values from neighboring grid points in the dimension being solved. Our results are for 60 iterations with 32x32x32 cubes.

4.1.1 Transparent Shared Memory

As part of a parallel computation course project at Wisconsin, Doug Burger and Sanjay Mehta converted *appbt* to C and parallelized it for shared memory in the following manner. First, they reorganized matrix A to make processors responsible for grid points in the interior of a subcube, so communication occurs only on subcube faces. Next, they parallelized the first and third phase by having each processor work on its subcubes independently. Wherever possible, they computed bordering elements redundantly to reduce communication. Finally, they parallelized the second phase using the natural parallelism in two dimensions and pipelining to overcome the dependences in the third dimension.

Our initial implementation, *appbt-spin*, preserves dependences in the second phase by spin waiting on a shared array of counters. To understand how this works, consider the case of a single vector where each element resides on a separate processor and each element, except the first, depends on the value of the previous element. These dependences are preserved with a counter, c, which is initially 0. The processor that owns element x cannot execute until $c = x$. Once $c = x$, the processor performs its computation and then increments the counter to $x + 1$. A simple extension to three dimensions requires N^2 counters. Blocking the matrix into subcubes of size $BxBxB$ reduces this to N^2/B^2.

4.1.2 Synchronization

The counters that manage the pipeline are simple and effective, but require more message traffic than necessary. Consider what happens when a processor updates a counter. First, invalidation messages are sent to all processors to its right in the pipeline. These processors immediately request new copies of the counter, because they are spinning on it. Only one processor terminates its spinning as a result of the update. Our second implementation, *appbt-sigwait*, uses `signal` and `wait` to exploit this property. When a processor finishes, it signals the next processor in the pipeline, which previously posted a wait. `Signal` uses active messages provided by the Tempest interface. *Appbt-sigwait* is 1.5 times faster than *appbt-spin*.

Version	Performance ms/iter (1)	Performance ms/iter (2–60)	Summary
Spin	13328	10731	Transparent shared memory, uses counters
Sigwait	9818	7162	Replace counters with signal/wait
512	7460	5861	Uses longer cache lines
Update	7865	1866	Send data with signal

Table 1: Blizzard Execution Time (ms/iteration) for *appbt*.

4.1.3 Block Size

Grid points consist of either a vector of 5 double precision numbers or a 5x5 matrix. Processors request this data in 128-byte cache lines. Our third implementation, *appbt-512*, replaces the default Stache128 protocol with Stache512 to reduce the number of requests. *Appbt-512* is 22% percent faster than *appbt-sigwait*. Stache512 is an inappropriate protocol for counters, because long lines increase both the unused data in a line and the effects of false sharing. To use Stache512 effectively, counters (and other similar data types) must be allocated on a separate page managed with Stache32 or replaced by `signal` and `wait`.

4.1.4 User-level Update Protocol

Our final implementation, *appbt-update*, forwards data at synchronization boundaries, to exploit the completely static sharing patterns in *appbt*. In the second phase, the data is simply forwarded along with a signal. In the other two phases, barrier synchronization is replaced with calls to forward data and perform pairwise synchronization. *Appbt-update* uses the Stache protocol in the first iteration to allocate the necessary pages in each processor's stache, adding a slight (5%) overhead to the execution time.

In subsequent iterations, processors call protocol routines that copy the data directly from the producing processor into the stache of the consuming processor, giving *appbt-update* a factor of 3.1 times performance improvement over *appbt-512*.

We made only three changes to the program to support this update protocol. First, we peeled the first iteration off the loop and added a call to the protocol library to mark its end. Second, we replaced the `signal` with a call to `signal+update`. Third, we replaced barrier synchronizations with calls to routines that forward data and perform pairwise synchronization. This version is not a message-passing program, because the address of the transmitted data remains the same on both the sender and the receiver.

In summary, *appbt-update* is a factor of 5.7 times faster than the *appbt-spin*.

4.2 EM3D

EM3D models the propagation of electromagnetic waves through objects in three dimensions [4]. The problem is formulated as a computation on a bipartite graph with directed edges from E nodes, which represent electric fields, to H nodes, which represent magnetic fields, and vice versa. The computation models the changes in the fields over time.

EM3D consists of an initialization phase and a main computation phase. The initialization builds the graph and performs some precomputation to improve the performance of the main loop. To build the graph, each processor allocates a set of E nodes and a set of H nodes. The edges are generated randomly with a user-specified fraction crossing processor boundaries. Edges that have a source on one processor and a sink on another processor are called *remote* edges. The main loop computes the change in E and H values over time. In each iteration, new E values are computed from the weighted sum of neighboring H nodes, and then new H values are computed from the weighted sum of neighboring E nodes.

We first wrote two implementations of *EM3D* for transparent shared memory. Then, we improved the program's performance dramatically with three successive refinements of a user-level update protocol. Table 2 summarizes the different versions of *EM3D* and its execution times for the compute phase (100 iterations) of a problem with 1000 H nodes and 1000 E nodes per processor and 10 edges per node, of which 20% are remote. We only list numbers for the compute phase because it should dominate the computation time of practical runs of this application.

Version	Performance		Summary
	ms/iter (1)	*ms*/iter (2–100)	
TSM	746	693	Naive shared memory
Value	516	448	Optimized shared memory
Update	758	71	Update protocol, single values
Vector	731	38	Update protocol, bulk messages
Channel	630	28	Update protocol, Tempest channels
MP	28	32	Message-passing

Table 2: Execution Time (ms/iteration) for *EM3D*

4.2.1 Transparent Shared Memory

Our first implementation, which we call *EM3D-TSM*, is a naive parallelization of *EM3D*. First, each processor allocates the appropriate number of E and H nodes and randomly generates both local and remote edges. These data structures are shared among the processors. Then, each processor executes the main loop, which computes new values for its nodes. References to values corresponding to remote edges are handled by the Stache protocol. This version is very easy to understand, but it is much slower than *EM3D-MP*, a carefully tuned message passing implementation for two reasons. First, *EM3D-TSM* does not exploit spatial locality for remote nodes, since a node's value field is intermixed with the rest of its fields, even though the main loop only accesses the value field of remote nodes. As a result, most of the data transmitted on Stache misses are unused. And second, an invalidate protocol (like Stache) performs poorly for producer-consumer communication, because it requires four messages to update a remote value. Consider an H node on Processor P that has an edge to an E node on Processor Q. In the first iteration, Processor Q incurs a remote miss to retrieve the initial value of H (two messages). When Processor P is ready to compute a new value for H, it invalidates the copy of H in Processor Q's cache (two messages), which causes Processor Q to miss on its reference to H at the beginning of the next iteration.

Our second implementation, *EM3D-value*, improves spatial locality by moving the value field out of the node into a separate vector. This optimization improves performance by 55% in the main loop. We use this improved version as the basis for the other implementations.

4.2.2 User-level Protocols

Below, we consider three increasingly complex user-level protocols to improve the performance of *EM3D*.

Our first refinement, *EM3D-update*, uses a user-level update protocol to correct the difficulty that invalidate protocols have with producer-consumer communication.[1] As mentioned earlier, Stache requires four messages to update a remote value. An update protocol uses only one message, once the sharing pattern is determined. The protocol for *EM3D-update* has two phases. In the first iteration of the main loop, the protocol looks very much like the Stache protocol. References to remote data cause a cache miss that is serviced by the owner of the remote data. In addition to sending back the necessary data, the owner records the requested address and the identity of the requester for later use. Because the graph does not change over time, the sharing patterns from the first iteration hold for the rest of the loop. Subsequent iterations switch to using an update protocol. At the end of each half step, each processor uses its sharing information from the first iteration to update remote copies with newly computed values.

This implementation required the largest change to the original program. First, we added calls to routines provided by the new protocol to initialize the sharing data structure before the start of the main computation and to signal the end of the first iteration. Second, we replaced a barrier in the original program with a call to a protocol routine that signals that the updates can proceed. However, since *EM3D-update* is 6.3 times faster than *EM3D-value*, these changes were amply rewarded.

EM3D-update updates each value in a separate message. Our next implementation, *EM3D-vector*, improves performance with the old observation that bulk transfer is the most efficient way to communicate large amounts of data. With this protocol, a processor batches all updates destined for another processor into one message. This improves performance by sending 37% fewer network packets and by reducing the frequency of message handler invocation on the receiver from once per remote value to once per pair of processors. *EM3D-vector* is 86% faster than *EM3D-update*.

Our final implementation, *EM3D-channel*, uses Tempest virtual channels,[2] rather than bulk sends, to transfer updates. Channels reduce handshaking overhead and eliminate buffering of out-of-order packets. *EM3D-channel* is 36% faster than *EM3D-vector* and its performance is nearly identical to the message-passing version of this program (*EM3D-MP*). In summary, *EM3D-channel* is a factor of 16 faster than our best transparent shared memory implementation, *EM3D-value*.

[1] An earlier paper on the proposed Typhoon system also used *EM3D* as an example of an application-specific protocol [21]. This paper significantly extends that work by looking at a wide spectrum of user-level protocols and evaluating them on the Blizzard system.

[2] Tempest's virtual channels, which are part of the bulk data transfer library, are similar to the Virtual Channels provided by the CM-5's CMMD Library. They reduce overhead by allowing a pair of processors that repeatedly communicate to handshake once and then to transfer data without further handshaking in the message library.

4.3 BARNES

Barnes, one of the SPLASH benchmarks [23], simulates the evolution, over time, of bodies in a gravitational system. Each body is modeled as a point mass and exerts gravitational force on other bodies in the system. At each time step, the program computes new positions and velocities for all bodies. The naive algorithm for this problem requires N^2 pairwise force calculations. *Barnes* reduces the work by using the observation that the force one body exerts on another falls rapidly with distance. As a result, the force exerted by a collection of bodies beyond a certain distance can be approximated by the force exerted by a point mass at the center of mass of the collection.

The main data structure in *Barnes* is an oct-tree that represents the location of bodies in 3-space. Interior tree nodes, called *cells*, represent regions in space that hold more than one body. A cell represents one octant of its parent's region. The root of the tree represents the entire space. The tree leaves contain bodies. The tree is not balanced and is much deeper in a region that contains many bodies. Also, the program does not incrementally modify the tree, but rather rebuilds it in every iteration to track bodies moving in space.

To calculate the force on a particular body B, the algorithm performs a depth-first traversal of the tree from the root. Upon encountering a body, the algorithm computes the force that the body exerts on B. Upon encountering a cell, the algorithm first determines if the bodies in the cell are distant enough to be approximated as a point mass. This calculation compares the ratio of the length of the cell's region and the distance from B to the center of mass of the bodies contained in the cell's region with a user-specified threshold. If the ratio is less than the threshold, the bodies in the region are approximated as a point mass located at the region's center of mass. Otherwise, the algorithm traverses the cell's eight subtrees.

Each iteration is composed of two phases. The first rebuilds the tree. Each processor is responsible for a set of bodies. Processors insert bodies into the tree, in parallel, with the following algorithm: Start at the root of the tree, which is a cell containing eight slots, and determine which octant should contain the body. If the slot corresponding to an octant is empty, store

Version	Performance		Summary
	sec/iter (1)	sec/iter (2–4)	
MCS	81	80	MCS locks + grouped bodies
ML	42	37	ML locks
Chunk	33	26	Chunk allocation + MCS locks
Chunk+ML	32	25	Chunk allocation + ML locks
FSE	26	21	Split body structure
Update	26	15	Update protocol

Table 3: Blizzard Execution Time (sec/iteration) for *Barnes*.

the body into the slot and process the next body. If the slot contains a cell, recursively apply this algorithm. If the slot contains a body, allocate a new cell, insert the old body into the new cell, store the cell in the slot, and insert the original body into the new cell using this algorithm.

The second phase consists of a force calculation, which computes new accelerations and velocities for the bodies, and a position update, which computes bodies' new positions. Again, each processor is responsible for its group of bodies. As the computation progresses, each processor references portions of the tree, which copies these nodes into its local stache. Since the tree does not change in this phase, subsequent references are local, until the tree is rebuilt in the next phase.

Our initial implementation, *Barnes-MCS*, differs from the SPLASH version in that ours assigns responsibility, in the first iteration, for groups of bodies rather than assigning a single body at a time. The program runs on 16,384 bodies for four iterations with 0.01250 as the time step and 0.50 as the tolerance.

4.3.1 Synchronization

In our initial implementation, locks protect a pool of free cells and the cells in the tree. Contention for the free cell pool is very high. We tried two approaches to improve performance: reduce the cost of lock operations and reduce the frequency of locking.

Our second implementation, *Barnes-ml*, replaced MCS locks with message passing locks. MCS locks use memory reads and writes to pass messages [17]. As a result, they are expensive in high contention situations (requiring as many as eight messages for a lock/unlock pair; however, successive lock/unlock pairs by the same processor on an MCS lock do not require any communication). Message-passing locks, on the other hand, require only four messages for a lock/unlock pair. This change merely required replacing the MCS lock library with a message passing lock library. The result is worth the effort, as *Barnes-ml* ran 2.2 times faster than *Barnes-MCS*.

Barnes-chunk, our next implementation, changed the way in which cells are allocated. Originally, the free cell pool was a vector and each time a processor needed a new cell, it acquired the pool's lock, took the next free cell, and released the lock. Our first change split the vector into 32 regions and allowed each processor to allocate cells only from its region. Our second change put each processor's region in its local portion of shared memory. This change used the local allocation mechanism, which allocates space in the global address space with the provision that the memory's home processor will be the allocating processor. The first change eliminated a lock, since each processor allocates only from its own pool. The second change improved locality, because the processor that allocates a cell makes most of the references to that cell. These changes improve performance dramatically. *Barnes-chunk*, which still used MCS locks for tree cells, ran 3.1 times faster than *Barnes-MCS*.

Barnes-chunk-ml combined these the optimizations of distributing the free cell pool and using message passing locks for the tree cells. Since contention for tree cell locks is low, the two optimization only improved performance by 4% over *Barnes-chunk* alone.

4.3.2 False Sharing Elimination

During the force calculation, a body's owner writes the acceleration, velocity, phi, and cost fields of a body. During the same phase, both the body's owner and other processors read the mass and position fields. To reduce the false sharing caused by these reads, our next implementation, *Barnes-fse*, split the body data structure into two pieces: mass, position, and type; and acceleration, cost, phi, and velocity. In addition, we deferred writing the cost, which is computed during the calculation phase, until the position update phase. These changes improved performance by 19% over *Barnes-chunk-ml*.

4.3.3 User-level Update Protocol

Following the force calculation, each processor updates its bodies' positions, which invalidates the copies cached in other processors. Because bodies do not move very far in any iteration, it is very likely that a processor will touch the same bodies in subsequent iterations. So, bodies invalidated at the end of one iteration will be re-requested in the next iteration. A similar phenomenon occurs with each body's cost.

Our final implementation, *Barnes-update*, used an update protocol to eliminate these unnecessary inval-

idations (and subsequent requests) and split bodies in a slightly different way than *Barnes-fse*, to allow more fields to be updated.

In our protocol, a processor's first reference to a body causes a miss that is handled by the Stache protocol. These references occur during force calculation. At the end of the force calculation, the program updates its bodies and calls a protocol routine. This routine forwards updates to the Stache directory that is responsible for the body. The directory then forwards the update to any processor that requested the body in the past. As a result, a processor misses only on its first reference to a body. Some processors that no longer reference a body continue to receive updates under this scheme. This is not a serious problem in our current implementation, because bodies move slowly and we simulated only four time steps. If it becomes an issue, we see two solutions. The first is to clear the directory update list periodically. The second is to have each processor track bodies that it touches and periodically flush unnecessary ones.

The field split in *Barnes-update* is different than in *Barnes-fse*. A body is now split into three pieces: type and mass; cost and position; and acceleration, phi, and velocity. The first two are allocated as shared structures. The third is allocated in private memory. These fields are only used by the current owner of the body and their current values move from processor to processor via explicit requests when the owner of a body changes; effectively, we implement a migratory protocol using private memory and explicit messages. In addition, this version automatically invalidates the stached cells at the end of every iteration. This eliminates the invalidations that need to be sent when the cells are used again to build the tree during the next iteration. *Barnes-update* is 40% faster than *Barnes-fse*, our best transparent shared memory implementation.

5 Discussion and Conclusions

Most cache-coherent shared-memory systems provide a single coherence policy for all programs and compilers. A "one size fits all" protocol is a serious bottleneck when it causes excess communication because of a program's sharing patterns. The Munin [2] and Stanford FLASH [11] systems enabled a programmer to select from a collection of system-provided protocols. The Tempest interface, implemented in hardware like Typhoon or as a fine-grain DSM system like Blizzard, combines the flexibility of DSM systems, which implement coherence protocols in software, with the fine-grain access control of hardware shared memory.

This paper examined the performance improvement made possible by the Tempest interface, which permits a coherence protocol to be tailored to an application. Application-specific protocols improve the performance of *appbt*, *em3d*, and *barnes* by factors of 5.7, 16, and 1.4, respectively (ignoring initialization and the first iteration which is slowed by at most 18%) over the best transparent shared memory version. These results indicate that significant performance gains are possible by moving beyond transparent shared memory and customizing a protocol to an application. Not surprisingly, we also found that writing protocols from scratch is difficult and error-prone, so we are investigating ways of encapsulating user-level protocols to make customization easier.

These large performance improvements are due, in part, to the high cost of an access-control miss on the Blizzard system. The performance improvements would likely be smaller on systems like FLASH or Typhoon that have extensive hardware support. On the other hand, clusters of workstations will have higher communication latencies than the CM-5 and could benefit even more than Blizzard from the reduction in communication made possible by these protocols.

The other issue raised in this research is whether user-level protocols are necessary or whether all important protocols can be incorporated into a system. For our three applications, the dramatic improvements came from update protocols. These three update protocols, however, were quite different. *Appbt's* communication pattern is known at compile-time. Its update protocol allocates pages in the first iteration and then uses explicit messages to distributed updates in subsequent iterations. *EM3D's* nodes share information through a list of nodes that is fixed for the duration of a computation, but is unknown at compile-time. *EM3D's* update protocol finds the sharers in the first iteration and then uses the list to communicate directly in subsequent iterations. *Barnes'* sharing is dynamic, which requires the least common denominator protocol that sends each update to a home node, which forwards it to the current list of sharers.

Perhaps, then, only three update protocols are necessary in a system? This seems unlikely, since our three update protocols differ in whether updates occur immediately or are batched at synchronization points and whether they update one, two, or many data fields. It seems impossible to anticipate all useful combinations in efficient, even if parameterized, protocols. This dilemma is a variant of the well-known policy versus mechanisms distinction. System-provided policies often omit an important feature needed by some users or incur the cost of generality by running slower [27]. Mechanisms allow users to build

the policies appropriate for their programs. While three application programs cannot provide a definitive answer, our opinion is that system-provided protocols are unlikely to be both general and fast enough to compete with application-specific protocols.

Acknowledgements

The other members of the Wisconsin Wind Tunnel Project, especially Shubu Mukherjee, Brad Richards and Doug Burger, provided assistance and support for this work. David Culler provided the original EM3D source code. Madhu Talluri provided helpful comments on this paper. Jun Makino alerted us to an error in our tables.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] Matthew I. Frank and Mary K. Vernon. A Hybrid Shared Memory/Message Passing Parallel Machine. In *Proceedings of the 1993 International Conference on Parallel Processing (Vol. I Architecture)*, pages 232–236, August 1993.

[6] Dana S. Henry and Christopher F. Joerg. A Tightly-Coupled Processor-Network Interface. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 111–122, October 1992.

[7] 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.

[8] 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.

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

[10] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating Message-Passing and Shared-Memory: Early Experience. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 54–63, May 1993.

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

[12] James R. Larus. Compiling for Shared-Memory and Message-Passing Computers. *ACM Letters on Programming Languages and Systems*, 2(1–4):165–180, March–December 1994.

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

[14] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, October 1994. To appear.

[15] 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.

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

[17] John M. Mellor-Crummey and Michael L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[18] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A Multithreaded Massively Parallel Architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, May 1992.

[19] 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.

[20] 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.

[21] 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.

[22] 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)*, October 1994. To appear.

[23] 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.

[24] 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.

[25] 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.

[26] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993.

[27] William A Wulf. Compilers and Computer Architecture. *IEEE Computer*, 14(7):41–47, July 1981.