Hardware Support for Flexible Distributed Shared Memory

Steven K. Reinhardt,[1] Robert W. Pfile,[2] and David A. Wood[3]

Abstract

Workstation-based parallel systems are attractive due to their low cost and competitive uniprocessor performance. However, supporting a cache-coherent global address space on these systems involves significant overheads. We examine two approaches to coping with these overheads. First, DSM-specific hardware can be added to the off-the-shelf component base to reduce overheads. Second, application-specific coherence protocols can avoid some overheads by exploiting programmer (or compiler) knowledge of an application's communication patterns. To explore the interaction between these approaches, we simulated four designs that add DSM acceleration hardware to a collection of off-the-shelf workstation nodes. Three of the designs support user-level software coherence protocols, enabling application-specific protocol optimizations. To verify the feasibility of our hardware approach, we constructed a prototype of the simplest design. Measured speedups from the prototype match simulation results closely.

We find that even with aggressive DSM hardware support, custom protocols can provide significant speedups for some applications. In addition, the custom protocols are generally effective at reducing the impact of other overheads, including those due to less aggressive hardware support and larger network latencies. However, for three of our benchmarks, the additional hardware acceleration provided by our most aggressive design avoids the need to develop more efficient custom protocols.

Index terms: parallel systems, distributed shared memory, cache coherence protocols, fine-grain cache coherence, coherence protocol optimization, workstation clusters.

1 Introduction

Technological and economic trends make it increasingly cost-effective to assemble distributed-memory parallel systems from off-the-shelf workstations and networks [3]. Workstations (or, equivalently, high-end personal computers) use the same high-performance microprocessors found in larger systems—but at a far lower cost per processor, thanks to their much greater market volume. More recently, vendors have begun to advertise high-bandwidth, low-latency switched networks targeted specifically at the cluster market. We expect this trend to continue and for the bandwidth and latency characteristics of cluster networks to approach those of dedicated parallel system interconnects [8, 19].

In spite of these trends, custom parallel systems still hold an advantage over off-the-shelf clusters of workstations in their ability to provide a low-overhead, globally coherent shared address space. Custom-built distributed shared memory (DSM) systems such as MIT's Alewife [2], Stan-

1. EECS Dept., The University of Michigan, 1301 Beal Ave., Ann Arbor, MI 48109-2122. stever@eecs.umich.edu.
2. Yago Systems, 795 Vaqueros Ave., Sunnyvale, CA 94086. pfile@yagosys.com.
3. CS Dept., University of Wisconsin–Madison, 1210 W. Dayton St., Madison, WI 53706-1685. david@cs.wisc.edu.

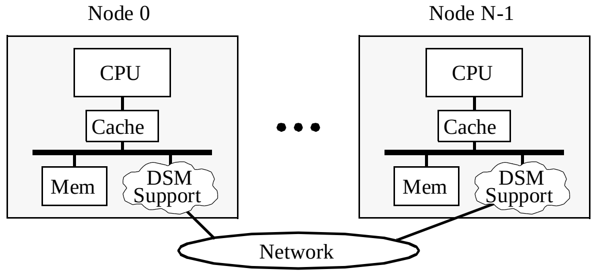


Figure 1. Workstation-based DSM system organization.

ford's FLASH [28], and the SGI Origin 2000 [30] integrate the memory and network control and datapaths at each processor–memory node, providing efficient coordination among processor requests, network messages, and DRAM accesses. In contrast, machines assembled from off-the-shelf workstations—even those that add custom hardware to accelerate DSM operations—must build on top of the workstations' existing memory systems. Nodes attach to the network via adapters that act as pseudo-processors or peripheral devices, as shown in Figure 1, forcing local and remote memory accesses to contend for memory bus bandwidth. These constraints increase the run-time overhead of providing a shared address space relative to a fully custom design.

We have explored two approaches to coping with the overheads of workstation-based DSM systems: DSM-specific hardware support and application-specific protocol optimizations. Adding DSM-specific hardware to off-the-shelf components reduces overheads by accelerating common operations. This approach increases performance at the expense of additional hardware design and manufacturing costs. Protocol optimizations avoid overheads by exploiting programmer or compiler knowledge of an application's communication patterns. For example, moving data proactively from producers to consumers both hides the latency of the data transfer and eliminates the overhead of handling the demand misses that would otherwise occur. This approach trades programmer effort for increased performance.

This paper discusses both options, focusing on the interaction between them. Can we build hardware that accelerates DSM performance yet provides flexibility to optimize communication? To what extent does hardware support make protocol optimizations unnecessary, and vice versa?

We examine these questions in the context of Tempest [44], an interface that lets user-level (unprivileged) software manage coherence at a fine granularity within a distributed shared address space. By default, applications use a standard software protocol that provides sequentially consistent shared memory, the consistency model generally preferred by most programmers for its simplicity [24]. However, for performance-critical data structures and execution phases, this default protocol may be replaced with arbitrary, potentially application-specific custom protocols, much like an expert programmer might recode a small performance-critical function in assembly language.[1] Although there are many approaches to optimizing shared-memory communication, Tempest uniquely allows programmers and compilers to adapt or reimplement the shared-memory abstraction to match a specific data structure or sharing pattern. This flexibility gives direct message-level control over communication *without* abandoning traditional shared-memory features such as pointer-based data structures and dynamic, transparent data replication.

Three Tempest-compatible system designs—Typhoon, Typhoon-1, and Typhoon-0—demonstrate one general approach to adding flexible DSM hardware to a cluster of workstations. These designs use logically similar components, but differ in the extent to which these components are integrated and customized for DSM operations. Using simulation, we compare the performance of both standard and custom-protocol versions of our benchmarks on these systems against a baseline hardware-supported DSM design (similar to Simple COMA [21]) that does not allow user protocol customization. While custom protocols provide less relative improvement over standard shared memory on more aggressive hardware, two benchmarks still show dramatic gains—86% and 384%—on the most aggressive Tempest system (Typhoon), outperforming the Simple COMA system by nearly the same amount. Custom protocols also effectively hide the higher overheads of less aggressive hardware; they decrease the performance gap between Typhoon-1 and Typhoon from 222% to 11% and the gap between Typhoon-0 and Typhoon from 427% to 22%. We expand on previous results [44, 45] by examining the effect of network latency and protocol-processor speed; we find that custom protocols are generally effective at reducing perfor-

1. Similarly, just as past improvements in compiler technology have reduced the need to code in assembly language for performance, we hope that future compilers will target Tempest to provide similar levels of optimization without expert programmer involvement [11].

mance sensitivity to these parameters as well. However, Typhoon executes the standard versions of three of our benchmarks nearly as quickly as the custom-protocol versions, and faster than the custom-protocol versions on the less aggressive systems. In these cases, the additional hardware acceleration avoids the need to develop more efficient custom protocols.

To demonstrate the feasibility of our designs, we built a prototype of the simplest system (Typhoon-0). We describe this prototype briefly and report performance measurements. Simulation results match measured prototype speedups within 6%.

The remainder of the paper is organized as follows. The next section reviews the Tempest model for fine-grain distributed shared memory, then discusses the benchmarks used in this study and the protocol optimizations that were applied to improve their performance. Section 3 describes the three Tempest designs, while Section 4 reports on their simulated performance. Section 5 reports on the Typhoon-0 prototype. Section 6 discusses related work, and Section 7 summarizes our conclusions and indicates directions for future work.

2 The Tempest interface

The Tempest interface provides a set of primitive mechanisms that allow user-level software—e.g., compilers and programmers—to construct coherent shared address spaces on distributed-memory systems [44]. Tempest's flexibility derives from its separation of system-provided mechanisms from user-provided coherence policies (protocols). Tempest also provides portability by presenting mechanisms in an implementation-independent fashion. This work exploits Tempest's portability by using optimized applications originally developed for an all-software implementation on a generic message-passing machine [50] to compare the performance of three different hardware-accelerated platforms.

Tempest provides four mechanisms—two types of messaging (*active messages* and *virtual channels*), *virtual memory management*, and *fine-grain memory access control*—to a set of conventional processes, one per processor–memory node, executing a common program text. Tempest's *active messages*, derived from the Berkeley model [55], allow the sender to specify a handler function that executes on arrival at the destination node. This user-supplied handler consumes the message's data—e.g., by depositing it in memory—and (if necessary) signals its arrival. *Virtual channels* implement memory-to-memory communication, potentially providing higher bandwidth for longer messages. *Virtual memory management* allows users to manage a portion of their virtual address space on each node. Users achieve global shared addressing by mapping the same virtual page into physical memory on multiple nodes, much like conventional shared virtual memory systems [32]. *Fine-grain memory access control* helps maintain coherence between these multiple copies by allowing users to tag small (e.g., 32-byte), aligned memory blocks as Invalid, ReadOnly, or Writable. Any memory access that conflicts with the referenced block's tag—i.e., any access to an Invalid block or a store to a ReadOnly block—suspends the computation and invokes a user-level handler function. Users may specify different sets of handlers for different virtual memory pages; thus multiple protocols may coexist peacefully, each managing a distinct set of shared pages. A complete specification of Tempest is available in a technical report [40].

2.1 Standard shared memory using Tempest

Tempest's mechanisms are sufficient to develop protocol software that transparently supports standard shared-memory applications. *Stache*, the default Tempest protocol included with the standard Tempest run-time library, is an example of such a protocol. The Stache library includes a memory allocation function, a page fault handler, and a set of access fault and active message handlers.

The memory allocation function, called directly by the application, allocates shared pages in the user-managed portion of the virtual address space. Stache assigns each page to a home node, which provides physical memory for the page's primary copy and manages coherence for the page's blocks. Processors on the home node access the primary copy directly via the global virtual address. Applications can specify a page's home node explicitly or use a round-robin or first-touch policy. For benchmarks that have an explicit serial initialization phase, the first-touch policy migrates the page to the node that first references it during the parallel phase [33].

Non-home nodes allocate memory for shared pages on demand via the page fault handler. Initially, these pages contain no valid data, so all their blocks are tagged Invalid. A reference to one of these Invalid blocks invokes an access fault handler that sends an active message requesting the block's data to the home node. The message handler on the home node performs any necessary protocol actions, then responds with the data. Finally, the response message's handler on the requesting node writes the data into the stache page, modifies the access tag, and resumes the local computation.

The Stache handlers maintain a sequentially consistent shared-memory model using a single-writer invalidation-based full-map protocol. A compile-time parameter sets the coherence granularity, which can be any power-of-two multiple of the target platform's access control block size. In practice, we maintain several precompiled versions of the Stache library supporting a range of block sizes.

2.2 Optimizing communication using Tempest

The Stache protocol effectively supports applications written to a standard, sequentially consistent shared-memory programming model. However, the real power of Tempest lies in the opportunity to optimize performance using customized coherence protocols tailored to specific data structures and specific phases within an application. Tempest also aids the optimization process itself by enabling extended coherence protocols that collect profiling information [34, 57]. This information drives high-level tools that help programmers identify and understand performance bottlenecks that may benefit from custom protocols. Unlike systems that provide relaxed memory consistency models [1, 18], the effects of these custom protocols are limited to programmer-specified memory regions and execution phases; the remainder of the program sees a conventional sequentially consistent shared memory.

At its simplest, Tempest's flexibility lets users select from a menu of available shared-memory protocols, as in Munin [10]. Programmers or compilers can further optimize performance by developing custom, application-specific protocols that optimize coherence traffic based on knowledge of an application's synchronization and sharing patterns. For example, a programmer might exploit a producer-consumer pattern in some inner loop, employing a protocol that batches the producer's updates into highly efficient bulk messages. Such protocols can often use application-specific knowledge to identify updated values without run-time overhead. This optimization does not change the shared-memory view of the data, but only the manner in which the producer and consumer copies are kept coherent. Thus, custom protocols typically do not require non-trivial modification of the original shared-memory source code. The custom protocols themselves currently require significant development effort; however, we seek to remedy this situation through software reuse, high-level tools for protocol development [12], and automatic compiler application of optimized protocols [11].

Although Tempest custom protocols use message passing to communicate between nodes, they directly support the higher-level shared-memory abstraction. In contrast, other systems that seek to integrate message passing and shared memory treat user-level message passing as a complementary alternative to—rather than a fundamental building block for—shared-memory communication [22, 27, 56]. To optimize communication in these systems, critical portions of the program must be rewritten in a message-passing style. Of course, if desired, Tempest programmers can also dispense with shared memory and use messages directly—for example, to implement synchronization primitives.

This section briefly describes the Tempest optimizations applied to six scientific benchmarks—Appbt, Barnes, DSMC, EM3D, moldyn, and unstructured—both to indicate typical optimizations and to provide background for following sections where these benchmarks are used to evaluate system designs. In each case, the programmer started with an optimized, standard shared-memory parallel program and further improved its communication behavior by developing custom protocols for critical data structures. Again, although these changes were performed manually at the lowest level, we expect future development tools to aid or automate this process. The original papers reporting on these optimizations (Falsafi et al. [16] for Appbt, Barnes, and EM3D, and Mukherjee et al. [37] for DSMC, moldyn, and unstructured) detail the applications and the evolutionary optimization process.

Appbt is a computational fluid dynamics code from the NAS Parallel Benchmarks [4], parallelized by Burger and Mehta [9]. Each processor works on a subcube of a three-dimensional matrix, sharing values along the subcube faces with its neighbors. In the original version, processors spin on counters to determine when each column of a neighbor's face is available, then fetch the data via demand misses. The Tempest-optimized version combines the communication and synchronization for each column into a single producer-initiated message. Approximately 100 lines of code were added or modified, a small fraction of the roughly 7,000 lines in the original program. Most changes were repetitive replacements of synchronization statements. The custom protocol itself required about 750 lines of C.

Barnes, from the SPLASH benchmark suite [51], simulates the evolution of an N-body gravitational system in discrete time steps. The primary data structure is an oct-tree whose interior nodes represent regions of three-dimensional space; the leaves are the bodies located in the region represented by their parent node. Computation alternates between two phases: rebuilding the tree to reflect new body positions, and traversing the tree to calculate the forces on each body and update its position. The standard shared-memory version we use incorporates several optimizations not in the original SPLASH code [16]. The Tempest-optimized version splits the body structure into three parts and applies a different protocol to each: read-only fields (e.g., the body's mass) use the default protocol, fields accessed only by the current owner use a custom migratory protocol, and the read-write position field uses a custom update protocol. Because the sharing pattern is dynamic, the updates are performed indirectly through the home node. Because the body data structure is split into multiple structures, source changes for Barnes were more widespread than for the other applications, making it difficult to quantify their scope.

DSMC simulates colliding gas particles in a three-dimensional space. Each processor is responsible for the particles within a fixed region of space. When a particle moves between regions, the source processor writes the particle's data into a buffer associated with the destination processor. Because the original version of the program performs well, we applied only one simple Tempest optimization: processors write particles into the destination processor's buffer using explicit messages instead of shared-memory writes. In addition to eliminating coherence overhead for the buffers, the atomic message handlers allow nodes to issue their writes concurrently without barriers or locking. This optimization modified only two lines in the original program. The remote-write protocol requires approximately 150 lines of code.

EM3D models electromagnetic wave propagation through three-dimensional objects by iteratively updating the values at each node of a graph as a function of its neighbors' values [13]. Each processor owns a contiguous subset of the graph nodes and updates only the nodes it owns. Where a graph edge connects nodes owned by different processors, each processor must fetch new values for the non-local nodes on every iteration. The standard shared memory version amortizes this overhead by placing multiple values in a cache block. This optimization modifies the graph node data structure, replacing the embedded value with a pointer into a packed value array. The Tempest version uses a custom deferred update protocol: each node sends all of the updated values required by a particular consumer in a single message over a virtual channel. The set of updates is static, but input dependent; the protocol determines the communication pattern by running a modified version of the Stache protocol in the first iteration. Graph nodes are fetched on demand, but the protocol records the addresses and processors involved. The second and subsequent iterations use the optimized protocol. Interfacing this custom protocol to the original program changed five lines of code. The custom protocol itself comprises over 1,000 lines of C. However, this value overstates the protocol's complexity somewhat; this code represents one of the earliest custom protocols, and was written without the aid of experience or protocol development tools.

Moldyn models force interactions between molecules. There are two main computation steps: the first identifies molecule pairs that interact significantly, and the second iterates over these pairs to compute forces and update the involved molecules' state. Most of the communication occurs in the second step, where each processor may generate a number of updates for a given molecule. The original version accumulates each processor's updates in a local array, then merges the updates in a synchronous, pipelined reduction phase. The Tempest version takes this optimization to its logical conclusion, using virtual channels to move array sections from node to node. This optimization changes only one line of the original code. The protocol itself involves less than

200 lines of C.

Unstructured iterates over a static, input-dependent mesh to calculate forces on a rigid body. The code partitions the mesh nodes across the processors. Unlike EM3D, processors may update as well as examine the values of mesh nodes owned by other processors. As in moldyn, the original version accumulates each processor's updates locally, then merges the updates via a separate reduction step. However, this reduction is less efficient than in moldyn because each processor updates a small fraction of the mesh nodes and each update involves only a small amount of computation. The custom protocol sets up virtual channels connecting processors that share edges and sends updates across these channels. Each processor then performs the reduction for its nodes locally. This optimization added five lines in the body of the program. The reduction protocol involves just over six hundred lines of C. A second custom protocol, similar to the one used in EM3D, propagates new node values from the owner to the referencing processors. This optimization modified 12 lines in the body of the program. The protocol itself is less than 600 lines of C.

2.3 Summary

By allowing user-level software to manage coherence, Tempest enables aggressive communication optimization within a shared address space. These optimizations typically require very few changes to the body of the application, because shared global data structures—for example, the pointer-based graph in EM3D—can be left intact.[1] Unfortunately, the optimizations described here involved significant protocol development effort. We believe that the magnitude of these efforts is due to our lack of experience and the intentionally low level of the Tempest interface, and will be dramatically reduced through software reuse, protocol development tools [12], and compiler support for protocol generation and selection [5, 11, 15]. Nevertheless, because Tempest enables a nearly unlimited spectrum of communication optimizations, it provides a good environment for investigating the interaction of these optimizations with hardware support.

3 Hardware support for Tempest

To examine the impact of hardware support on both standard and Tempest-optimized shared-

1. The leaf structure in Barnes is a notable exception, where the desire to apply different protocols to the structure's fields conflicts with Tempest's page-granularity protocol binding.

memory applications, we compare the performance of three system designs that add varying levels of hardware acceleration to a cluster of workstations. This section describes these three systems briefly. The following section (Section 4) analyzes the simulated execution of the benchmarks of Section 2.2 on these systems.

The three systems we study—Typhoon, Typhoon-1, and Typhoon-0—share the organization depicted in Figure 1. The DSM support hardware, represented by the “cloud” in Figure 1, consists of three components: logic implementing Tempest’s fine-grain access control, a network interface, and a processor to execute protocol code (message and access fault handlers). Tempest’s virtual memory management is supported by address translation hardware on the main CPU.

All three systems use a custom device at each node to implement some of these components; they differ in the number of components integrated into this device, as depicted by the shaded rectangles in Figure 2. Components not included in the custom device are implemented using off-the-shelf parts. These different integration strategies represent different trade-offs between performance and hardware design simplicity. Integrating a component into a custom device increases the device’s design time and design and manufacturing costs.[1] However, separating components into discrete devices significantly increases the overhead of communicating between them. Typhoon provides the highest performance—at the highest cost—by integrating all three components on a single custom device.[2] Typhoon-1 replaces Typhoon’s integrated processor with an off-the-shelf CPU, leaving the network interface and access control logic together in custom hardware. Typhoon-0 achieves the simplest design by splitting all three components across separate devices, two of which—the protocol processor and the network interface—can be purchased off the shelf. A relatively simple access control device is Typhoon-0’s only custom component. Typhoon-0’s simplicity allowed us to construct a prototype implementation, which we report on in Section 5.

1. In sufficient volume, the increase in the custom component’s cost may be offset by the system-level reduction in package count and board space.

2. Our focus on workstation-based systems precludes integrating components, such as the memory controller, that are already an integral part of the base workstation system.

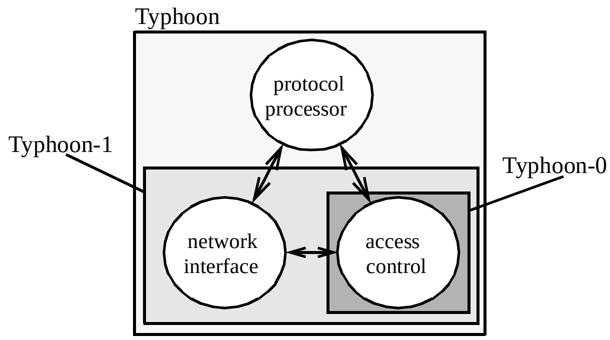


Figure 2. Component integration diagram.

The rest of this section describes the systems more fully. The first subsection covers common features; following subsections describe Typhoon, Typhoon-1, and Typhoon-0 in turn. Additional details can be found in previous publications [41, 44, 45].

3.1 Common features

To focus on the impact of the organizational differences in our designs, we assume similar technology for the three common components—the protocol processor, network interface, and access control logic.

The protocol processor is a general-purpose CPU. For convenience, the protocol processor uses the same instruction set as the primary (computation) processor. User-level protocol software running on the processor communicates directly with the access control logic and network interface via memory-mapped registers.

The network interface is a pair of memory-mapped hardware queues, one in each direction. Sending a message requires writing a header word indicating the destination node and message length, followed by the message data, into the send queue. A message is received by reading words out of the receive queue. A separate signal indicates when a message is waiting at the head of the receive queue. Credit-based flow control prevents deadlock on the hardware queues. To relieve protocol software from dealing with flow control, the run-time library buffers messages in main memory when the network overflows.

Tempest's active messages map directly to the interface: the first word of each message is used for the receive handler's program counter, and the handler reads the remainder of the message from the hardware queue. The messaging library implements Tempest's virtual channels in a straightforward fashion on top of the active message layer.

The access control logic monitors memory accesses by snooping on the memory bus and enforces access control semantics using the signals intended for local bus-based coherence. On every bus transaction caused by a processor cache miss, the device checks its on-board tag store in parallel with the main memory access. If the access conflicts with the tag, the device inhibits the memory controller's response (as if to perform a cache-to-cache transfer) and suspends the access. If the access does not conflict with the tag, the device allows the memory controller to respond. In the case of a read access to a ReadOnly block, the device asserts the "shared" bus signal to force the processor cache to load the block in a non-exclusive state. A subsequent write to the block will cause the processor to initiate an invalidation operation on the bus, which the device can then detect and suspend.

Once a block is loaded into the processor's cache, accesses that hit cannot be snooped; these hits must be guaranteed not to conflict with the access tag. For this reason, tag changes that decrease the accessibility of a block (e.g., from Writable or ReadOnly to Invalid) require a bus transaction to invalidate any copies that may be in the hardware caches. When a block is initially Writable, the bus transaction also retrieves an up-to-date copy, because the data could be modified in a hardware cache.

A *shadow space* [7, 22, 54] allows direct manipulation of access control tags from Tempest's user-level protocol software. A shadow space is a physical address range as large as, and at a fixed offset from, the machine's physical memory address range. Accesses to a shadow-space location are handled by the access control logic and interpreted as operations on the corresponding real memory location. The operating system allows a process to manipulate access control tags for its own data pages by mapping the corresponding shadow space pages into the process's virtual space. These shadow mappings prevent unauthorized accesses and implicitly translate authorized accesses to physical addresses.

3.2 Typhoon

Typhoon [44] combines the network interface, access control logic, and protocol processor on a

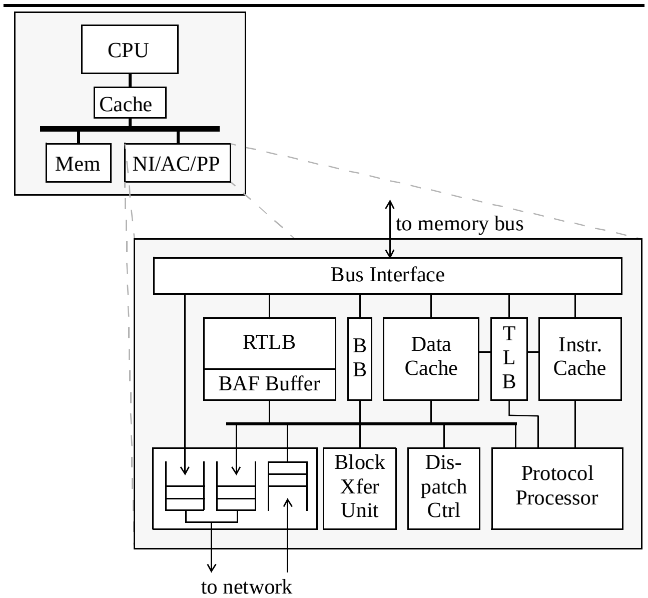


Figure 3. A Typhoon node [44].

single device (see Figure 3). Typhoon's full integration enables two categories of optimizations: those that enhance communication between the processor and other components, and those that take advantage of network interface/access control synergy.

Low-overhead, customized communication paths between the processor and the other components enable efficient invocation and execution of protocol handlers. Event dispatch logic provides memory-mapped registers containing the program counter and arguments for the next handler to be executed, transparently arbitrating between access fault and message handlers. If no events are pending, the dispatch register defers responding to reads, stalling the processor until an event occurs. While a handler executes, it enjoys single-cycle access to access control and network interface registers.

The access control logic combines tag enforcement and fault dispatch using a *reverse translation lookaside buffer* (RTLB). The RTLB caches per-page entries indexed by physical page number. Each entry contains the page's access tags, fault handler addresses, protocol data pointer, and the corresponding virtual page number. Each bus transaction is checked against the tags stored in the RTLB. On an access fault, the RTLB latches the appropriate handler address, the data pointer, and the virtual address of the faulting access. The RTLB derives its name from the "reverse" translation of physical bus addresses to the virtual addresses required by the user-level access fault handlers.

Integration of the network interface and access control logic allows efficient implementation of some common DSM operations. For example, in a single-writer protocol, a writing node must atomically give up write access and send the modified data to the next reader or writer. In this case, the Typhoon device issues a single read-for-ownership bus transaction that both invalidates cached copies (as required to downgrade the access tag) and fetches the current copy into the network send queue. Typhoon also accelerates receive operations using the *block buffer* (BB in Figure 3). The block buffer has address tags, like a cache, and participates in the local bus coherence protocol. When receiving data for a previously Invalid block, the block buffer stores the arriving data, sets the address tag to match the destination block's address, and sets the local protocol state to indicate exclusive ownership. (Note that the block buffer cannot safely assert ownership without the assurance that no other local cached copies exist, which comes from its knowledge that the block was previously tagged Invalid.) Until the block buffer replaces the data, it will satisfy subsequent requests for the block (possibly including the retry of the faulting access that initiated the block fetch) as cache-to-cache transfers without writing the data to main memory. The *block transfer unit* manages these combined block operations, which are invoked via writes to the shadow space.

3.3 Typhoon-1: off-the-shelf protocol processor

Typhoon-1 simplifies the Typhoon design by replacing Typhoon's integrated protocol processor with a general-purpose off-the-shelf CPU. This section discusses our motivation and methods for this separation of components. Because Typhoon-0 also uses an off-the-shelf protocol processor, the contents of this section apply to that design as well (unless stated otherwise).

In addition to reducing the custom device's design complexity and manufacturing cost, using a general-purpose CPU for protocol processing has two advantages. First, because the integrated processor will face die size constraints and incur design delay due to component integration and testing, an off-the-shelf processor will almost certainly have higher raw performance. (However, the integrated processor's effective performance can be increased by optimizing the microarchitecture for protocol handling, as in FLASH [28].) Section 4.4 investigates the performance impact of a slower integrated processor. Second, a general-purpose protocol processor may be used for the application's primary computation when there are no protocol events to handle. Similarly, any processor on a symmetric multiprocessor node may serve as the protocol processor. Falsafi and Wood [17] show that dynamically scheduling protocol processing tasks across all available processors is often more efficient than dedicating a protocol processor. Both Typhoon-1 and Typhoon-0 are capable of supporting this dynamic model. However, we assume a dedicated protocol processor in this study to provide a more direct performance comparison with Typhoon.

Of course, the primary drawback to a separate, off-the-shelf protocol processor is the increased overhead of communicating with the network interface and access control logic. Accesses to these other components must cross the memory bus, which is both much slower than an on-chip interconnect and subject to contention. This change in connectivity relative to Typhoon led to two changes in the interface between the processor and the other components.

First, we use a novel *cacheable control register* technique to accelerate dispatch of event handlers. A cacheable control register is a device register accessed using the local bus cache coherence protocol. When the register is read, the device responds with a cache block of data. Whenever the contents of the register change, the device issues a bus transaction to invalidate the cached copy. Both Typhoon-1 and Typhoon-0 use a cacheable control register called the *dispatch register* to transfer event information to the protocol processor. Because the dispatch register is a full cache block, it supplies both event notification and all available event parameters (e.g., the address of a block access fault) in a single bus transaction. Because the dispatch register's contents are stored in the protocol processor's cache until invalidated by the device, the processor can poll continuously while consuming bus bandwidth only when an event occurs.

Second, because the overhead of sending an event notification across the memory bus dominates handler invocation latency, we streamline the access control logic—shifting some access fault processing to software—without incurring a significant performance penalty. Specifically, the access control hardware maintains only the two-bit access tags for all of physical memory in

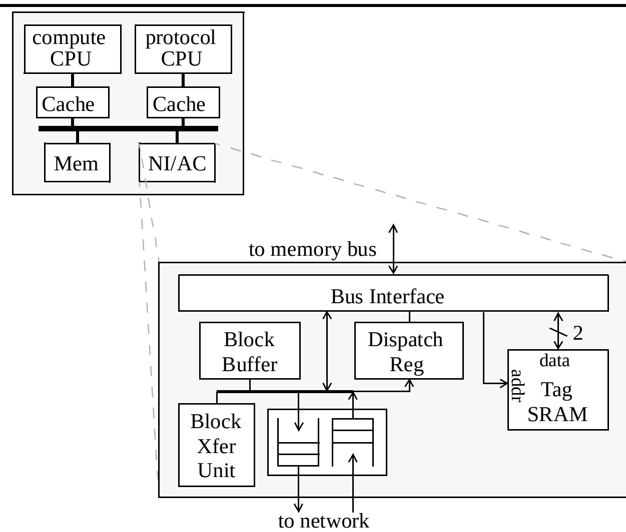


Figure 4. A Typhoon-1 node.

an on-board SRAM array. One megabyte of SRAM is sufficient to tag 128 Mbytes of physical memory at a 32-byte granularity. A software-managed inverted page table in main memory stores the other per-page information found in each Typhoon RTLB entry: the virtual page number, fault handler addresses, and protocol data pointer. On a block access fault, software obtains the physical page number from the hardware, indexes the inverted page table, determines the handler address, and forms the arguments. Because all fault information is transferred by the cacheable dispatch register in a single bus operation, and because the hardware formats this data to eliminate software shifting and masking, only eight instructions are needed from the detection of a block access fault to the invocation of the appropriate Tempest user handler. Assuming cache hits, these eight instructions require ten cycles on the Ross HyperSPARC.

In all other respects—other than the performance of inter-chip communication—the combination of Typhoon-1's protocol processor and custom device, shown in Figure 4, behave as the integrated Typhoon device. As in Typhoon, Typhoon-1 leverages access control/network interface integration, in cooperation with the block transfer unit and block buffer, to support combined block transfer and access tag change operations.

3.4 Typhoon-0: off-the-shelf network interface

Typhoon-0 further simplifies Typhoon-1's custom hardware device by replacing the integrated

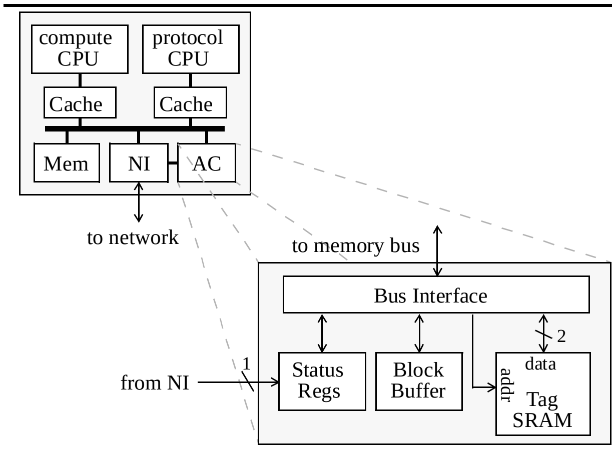


Figure 5. A Typhoon-0 node.

network interface with a separate, off-the-shelf network adapter, as shown in Figure 5. Just as Typhoon-1 leverages high-performance general-purpose CPUs, the Typhoon-0 design also capitalizes on emerging low-latency commercial networks such as Myricom's Myrinet [8] and DEC's Memory Channel [19]. The custom device contains only fine-grain access control logic. Section 5 describes an FPGA-based prototype of Typhoon-0 that demonstrates the feasibility and relative simplicity of this design.

Of course, the lack of network interface/access control integration increases Typhoon-0's overheads relative to Typhoon and Typhoon-1. These increases occur primarily in the areas of handler dispatch and combined block transfer/access tag change operations.

Although the custom device's dispatch register provides all the information required to invoke an access fault handler, it cannot do the same for message handlers. To limit the protocol processor's polling check to a single cacheable location, a dispatch register status bit reflects the state of the network interface's message arrival interrupt line, as shown in Figure 5. Unlike Typhoon and Typhoon-1, however, the processor must access the NI explicitly to fetch the message handler's program counter.

The combined block transfer and access tag change operations—handled by the block transfer unit and block buffer in Typhoon and Typhoon-1—require software sequencing in Typhoon-0. These operations must be performed carefully (and awkwardly) to avoid race conditions that lead to lost writes or undetected accesses to invalid data. For example, to revoke write access on a block and send the modified data, the processor first initiates the tag change via a write to the access control device's shadow space. The device performs a read-invalidate bus transaction to invalidate cached copies and fetch the current version, which is copied into the device's block buffer.[1] The processor then synchronizes with the device, using an uncached load, to guarantee that the transaction has completed. Finally, the processor copies the contents of the block buffer to the network interface. Similarly, when receiving data for a previously Invalid block, the processor first writes the data to memory via an uncached alias which bypasses the tag check, then changes the access tag.

4 Performance evaluation

This section compares the performance of these three designs via simulation. We first describe the simulation parameters and methodology. Section 4.1 presents results for a simple microbenchmark and a set of application macrobenchmarks. Section 4.3 examines the impact of network latency on application performance, and Section 4.4 examines the impact of integrating a lower-performance protocol processor in Typhoon.

The nodes of the simulated systems are based on the technology used for the Typhoon-0 prototype. Nodes are based on the Sun SPARCStation 20. Processors are modeled after the dual-issue Ross HyperSPARC [46] clocked at 200 MHz with a 1 Mbyte direct-mapped data cache. The instruction cache is not modeled; all instruction references are treated as hits.

A 50 MHz MBus connects the processor(s), memory, access control and network interface devices within each node. The MBus is a 64-bit, multiplexed address/data bus that maintains coherence on 32-byte blocks using a MOESI protocol [52]. On a cache miss, main memory returns the critical doubleword 140 ns (7 bus cycles or 28 CPU cycles) after the bus request is issued, with the remaining doublewords following in consecutive cycles. Miss detection, processor/bus clock synchronization, and bus arbitration add 11-14 CPU cycles to the total miss latency. The simulation accounts fully for occupancy, contention, and arbitration delays; the model is suf-

1. Typhoon-0's block buffer holds only data fetched by these read-invalidate transactions, so it does not have address tags as in Typhoon and Typhoon-1. Typhoon-0's block buffer is implemented as a cacheable register, so the processor can read the entire contents in a single bus transaction.

ficiently detailed and accurate that the same simulator was used for initial functional design of the Typhoon-0 prototype's access control device.

On a block access fault, the access control logic inhibits the memory controller and gives the requesting processor an MBus *relinquish and retry* response, forcing the processor to rearbitrate for the bus. The access control device masks the arbiter to keep the processor off the bus until the access can be completed [31]. Although this technique cannot be implemented on an unmodified SPARCstation 20, its performance is representative of more recent systems which support deferred responses, either explicitly (like the Intel P6 [20]) or using a split-transaction bus.

Timing parameters for the Typhoon-0 and Typhoon-1 access control devices are taken from the FPGA-based Typhoon-0 prototype implementation described in Section 5. To equalize the comparison with Typhoon-0 and Typhoon-1, which store all the access tags on the access control device, the Typhoon RTLB is assumed to be large enough to map all the active shared pages on each node. Because simulation limits the size of the data sets, it is unlikely that replacement overheads due to a finite RTLB would affect the results significantly.

The network interface queues transfer up to 64 bits per cycle, at the bus clock rate in Typhoon-1 and Typhoon-0 and at the protocol processor clock rate in Typhoon. Each node may have at most four messages outstanding to each other node. The simulated Typhoon-0's network interface is modeled after the CM-5 NI, with a message arrival signal that feeds the access control device's dispatch register.

Network contention is modeled at the interfaces, but not internally. As a result, the network wire latency, measured from the transmission of the tail from the sending network interface to the arrival of the head at the receiving interface, is constant. To emphasize the performance impact of DSM support, the default latency is a fairly aggressive 0.5 μs (100 processor cycles). Although current off-the-shelf networks are typically slower, the last few years have seen rapid performance advances in commercial system-area networks [8, 19]. We believe that networks with this level of performance will be available in the near future. Section 4.3 examines the overall performance impact of higher-latency networks.

The simulated network supports barrier synchronization in hardware, as in the CM-5. The latency from the last arrival to notification matches the one-way network latency. The benchmarks do not perform a significant amount of barrier synchronization, so the absence of this feature would not noticeably affect the results.

To quantify the performance impact of software protocols, we include a fourth system—an idealized implementation resembling Simple COMA [21]—as a baseline. This system is similar to Typhoon, but replaces the protocol processor with an idealized hardware implementation of Stache. This idealized protocol engine processes each access fault or message event with zero overhead, including manipulation of protocol state and the injection of an arbitrary number of messages. Events are processed at a maximum rate of 200 MHz. Messages observe latency due to network transport, potential queueing at the controller, and fetching data over the MBus. Due to the structure of the simulator, messages observe an additional cycle of pipelined latency between arrival and processing.

To obtain results, application codes are compiled and linked with portable software protocols (written in C using the Tempest interface) and platform-specific Tempest runtime software, exactly as they would be for an actual implementation. A rewriting tool (based on EEL [29]) processes the resulting SPARC binaries, replacing memory accesses with calls to the simulator and adding instrumentation to count instruction execution cycles. Direct execution of the modified binaries drives the detailed discrete-event simulator. To enable larger systems and data sets, the system nodes are simulated in parallel on a Thinking Machines CM-5 using the Wisconsin Wind Tunnel II [43, 36].

4.1 Microbenchmark

To gain insight into the overheads of these systems, we trace a simple remote read miss and break down the latency into its components. When the miss occurs, a cache page is already allocated on the caching node and the block is idle (unshared) at the home node. On the caching node, the miss access invokes a block access fault handler—part of the hardware state machine on Simple COMA, or software on the Typhoon systems—which sends a request to the home node. At the home, the message handler downgrades the block from Writable to ReadOnly and sends a copy to the requester. Back at the caching node, the response message handler writes the data to memory, changes the block's tag to ReadOnly, and signals the compute processor to retry the access.

Table 1. Remote miss latency breakdown for simulated systems.

		Latency (200 MHz cycles)			
	Step	**SC**	**Typh.**	**Typh.-1**	**Typh.-0**
Caching node	detect HW cache miss, issue bus transaction	10	10	10	10
Caching node	detect access fault, dispatch handler	0	6	101	101
Caching node	get fault state	0	16	18	18
Caching node	send msg	0	13	45	45
	request msg latency	100	100	100	100
Home node	dispatch msg handler	1	6	78	159
Home node	read msg	0	3	7	40
Home node	directory lookup, branch	0	20	20	20
Home node	send msg header	0	17	38	52
Home node	fetch data from memory, change tag, send	48	48	122	293
	response msg latency	100	100	100	100
Caching node	dispatch msg handler	1	6	78	159
Caching node	read msg header	0	3	7	40
Caching node	read msg data, change tag	0	12	20	261
Caching node	unmask CPU, reissue bus transaction	10	10	32	32
Caching node	fetch data, resume	31	31	31	31
Totals	200 MHz CPU cycles	301	401	807	1461
Totals	50 MHz bus cycles	76	101	202	366
Totals	microseconds	1.5	2.0	4.0	7.3
Totals	bus transactions	3	3	16	36

The results are presented in Table 1. The home node latency includes 2 bus cycles (8 processor cycles) to request and acquire the bus and 10 bus cycles (40 processor cycles) to fetch the block. (Block data is not pipelined into the network.) On the caching node, the final step ("fetch data, resume") includes 7 bus cycles (28 processor cycles) to fetch the critical word and 3 processor cycles to forward the data to the CPU and complete the load. The idealized Simple COMA system requires one additional cycle per message, for a total of 301 processor cycles, or about 1.5 µs. For comparison, the Stanford FLASH designers report remote read miss latencies of 1.11 and 1.45 µs, depending on whether the data is dirty in the remote processor's cache [23].[1] Because these fundamental latencies dominate, Typhoon takes only 33% longer to satisfy the miss despite the cost of

1. Because the systems described here always fetch data over the coherent memory bus, latencies are independent of the data's hardware cache status.

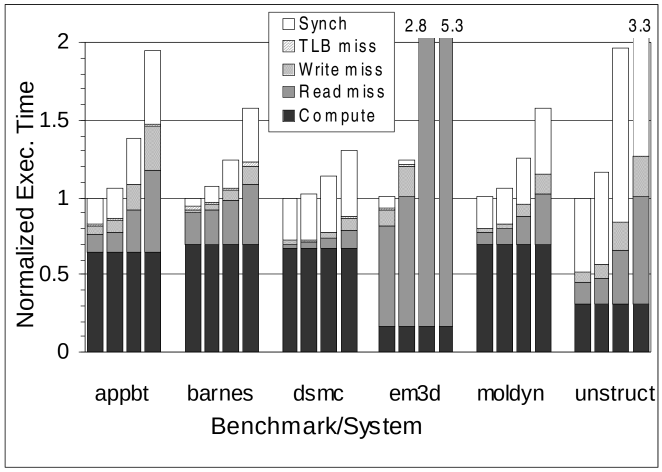


Figure 6. Application speedups on 32-node systems.

running software handlers. The less integrated designs do not fare as well in this comparison. Going from Typhoon to Typhoon-1, the miss latency roughly doubles; going to Typhoon-0, it nearly doubles again. As expected, this correlates with a large increase in the number of bus transactions needed to satisfy the miss.

4.2 Macrobenchmarks

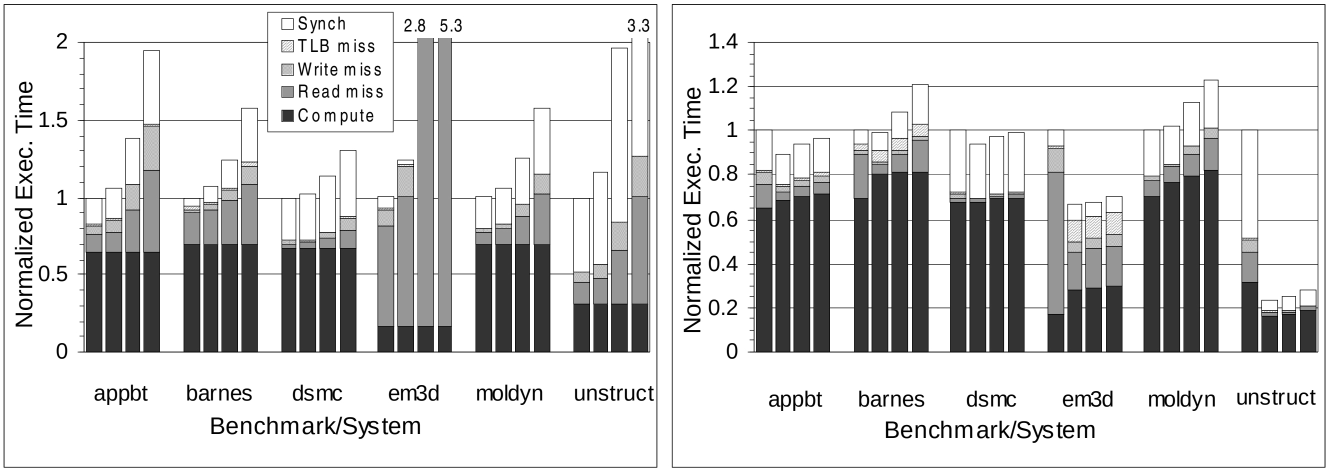
To determine how these overheads translate into application performance, we simulated the six shared-memory applications described in Section 2.2. All benchmarks are written in C and were compiled with gcc version 2.6.3 at optimization level -O2. Table 2 summarizes the applications and indicates the data sets used. All of the benchmarks except EM3D use the first-touch migrate-once scheme described in Section 2.1. EM3D explicitly allocates the graph so that writes are always to local pages. Although we simulated full applications, we report results for only the second and following computation iterations to focus on the portion of execution where a production version will spend most of its time. For most of the applications, iteration times are very regular, so meaningful results require only a few iterations. There are two exceptions. Moldyn rebuilds the molecule interaction list occasionally, resulting in an iteration that is an order of magnitude longer than the others; we simulate far enough to include the first of these rebuilds. DSMC simulates gas particles in a region with an incoming flow, so initially the number of particles increases with each iteration. It is impractical to simulate far enough to reach steady state; we arbitrarily chose to run for 400 iterations. As the number of particles increases, the speedup also increases, but very slowly; we do not expect results for a longer run to be qualitatively different.

Table 2. Benchmark applications and data sets.

Bench-mark	Application domain	Primary data structure(s)	Data set
appbt	CFD	3D array	32x32x32 array, 5 iterations
Barnes	hierarchical N-body	oct-tree	16,384 bodies, dtime=0.025, tstop=0.075 (4 iterations)
DSMC	Monte Carlo particle-in-cell	cell array, particle list	48,000 particles in 9720 cells, increasing to 72,000 particles, 400 iterations
EM3D	electro-magnetics	static bipartite graph	192,000 nodes, degree 5, 5% remote edges, 20 iterations
moldyn	molecular dynamics	molecule list, interaction list	8788 particles, 30 iterations, interaction list rebuilt once
unstruc-tured	CFD	static mesh	9428 nodes, 59863 edges, 5864 faces, 5 iterations

Figure 6 shows speedups for 32-node systems relative to our best sequential version on one node of the parallel system. (In this and following figures, SC stands for Simple COMA, T for Typhoon, T1 for Typhoon-1, and T0 for Typhoon-0.) We focus first on the shorter, darkly shaded bars in Figure 6, which indicate the speedups for standard shared memory using 64-byte coherence blocks. All the benchmarks achieve speedups of 19 or better on the Simple COMA system, with the exception of unstructured at under five. (Appbt and EM3D enjoy potentially large absolute speedups because 36% and 87% of their uniprocessor executions, respectively, are spent waiting on cache and TLB misses.) Figure 7a presents the benchmark execution times normalized to the Simple COMA system, breaking out the time spent by the main processor on computation and on read, write, TLB, and synchronization stalls. Two of the benchmarks—EM3D and unstructured—have poor processor efficiency, spending less than a third of the time computing even on the Simple COMA system. Unstructured fails to produce much speedup for any platform. EM3D achieves speedups only because its large (20 MB) data set thrashes the uniprocessor's cache and TLB, but fits in the caches and TLBs of the parallel systems.

We make two observations from the data in Figure 7. First, Typhoon's performance—which is at most 25% less than the idealized Simple COMA's, and is under 10% less for the four benchmarks with higher efficiencies—demonstrates that integrated hardware support can provide both flexible user-level software protocol processing and high performance. Second, although the



(a) Standard shared memory

(b) Custom protocols

Figure 7. Execution time breakdown.

higher overheads of Typhoon-1 and Typhoon-0 lead to significant increases in stall time, the impact on overall performance varies widely. On DSMC, for example, moving from Typhoon to Typhoon-0 increases stall time by 81%, but decreases performance by only 28%. On the low-efficiency benchmarks—EM3D and unstructured—the simpler designs show their weakness, turning in performance two to five times slower than the integrated systems.

Changing the coherence block size on the Tempest systems is as simple as linking with a different version of the Stache protocol library. The lighter shaded bars in Figure 6 show the speedups obtained by choosing the best coherence block size for each application between 32 and 512 bytes. Only EM3D gains significantly from this optimization. The Tempest systems also support multiple data-structure-dependent block sizes within a single application, which have been shown to improve speedups further [48]. Although the original Simple COMA proposal uses a fixed block size [21], we report the effect of choosing the best per-application block size for that platform as well.

Next we turn to the Tempest-optimized versions described in Section 2.2, whose speedups are indicated by the hatched bars in Figure 6. Although these application-specific protocols were written and optimized for a very different software-only Tempest platform—Blizzard-E [50] on the CM-5—they still provide some improvement over standard shared memory on our hardware-accelerated systems. (The lone exception is moldyn on Typhoon, where the custom protocol is 1%

slower than the standard version using the best block size, although it is 3% faster than the standard version using 64-byte blocks.) Two show dramatic improvement even on Typhoon—86% for EM3D and 384% for unstructured—causing them to outperform the Simple COMA system by nearly the same margins as well. (The large absolute speedup for EM3D is due to the memory system effects mentioned above.) On Typhoon-1 and Typhoon-0, the higher overheads leave more room for improvement, so the more efficient protocols have a greater impact, increasing performance by on Typhoon-1 and on Typhoon-0. However, Typhoon executes the standard versions of three of our benchmarks nearly as quickly as the custom-protocol versions, and faster than the custom-protocol versions on the less aggressive systems. In these cases, the additional hardware acceleration avoids the need to develop more efficient custom protocols. Nevertheless, these results indicate significant potential for custom protocols.

Figure 7(b) breaks down the execution times for the Tempest-optimized applications. Because time that the main processor spends doing explicit message passing is counted as computation, most of the application-specific protocols increase the amount of computation over the original version; computation time varies on the different platforms due to varying message-passing overheads. Reductions in the stall times compensate for these increases. Unstructured is an exception to this pattern: both the computation and stall times decrease significantly. This effect is due to the optimized version's more efficient reduction phase. Where the original application performs a global reduction on an array of values, the optimized code sums only the non-zero contributions for each value (see Section 2.2). Although the primary intent of this optimization is to eliminate the communication of the zero values, it also eliminates the computation involved in summing them into the result.

Figure 7(b) also shows that the efficiency of each application on Simple COMA correlates inversely with the effectiveness of the application-specific protocols. Intuitively, the applications with higher overheads have more to gain by eliminating those overheads. The two benchmarks with very low efficiency, EM3D and unstructured, show impressive gains, while the improvements for DSMC, moldyn, and Barnes are smaller. Appbt straddles the fence: the custom protocol gains a factor of two on Typhoon-0 but only 18% on Typhoon.

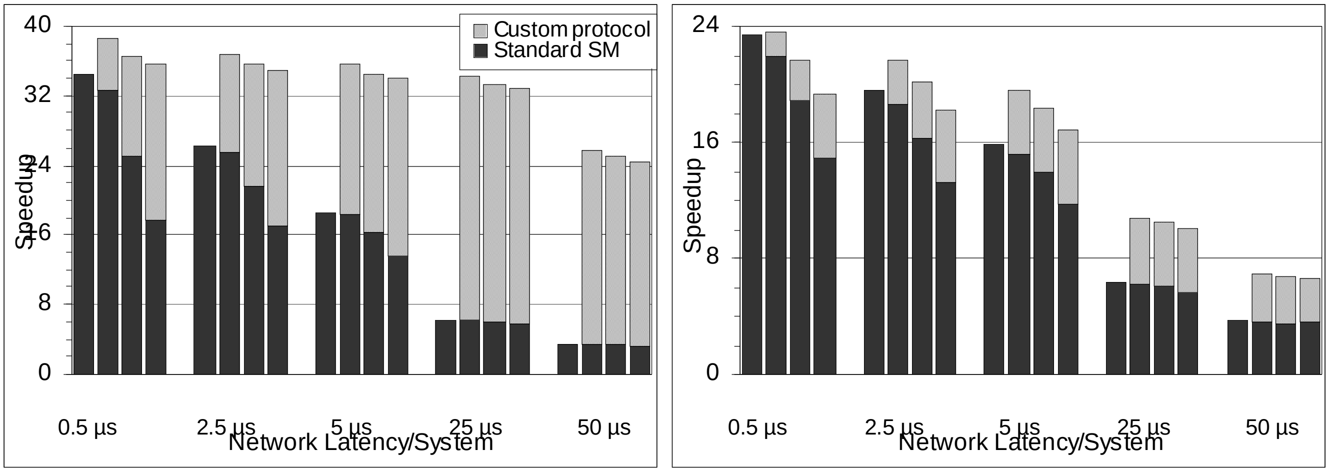
TLB misses, which are not a factor for any of the standard shared-memory benchmarks, are a noticeable component of execution time for two of the Tempest-optimized codes (roughly 6% for Barnes and 10% for EM3D). These costs merely reflect differences in reference locality between the standard and custom-protocol versions; they are not inherent to Tempest or the hardware platforms. The custom-protocol version of Barnes splits the body structure so that different protocols can be applied to different fields. As a result, the data for one body, which is contiguous in the original code, is spread across three structures on three different pages. The difference in TLB behavior for EM3D is due to data restructuring in the standard-protocol version intended primarily to improve locality for remote references. With some effort, this restructuring could be applied to the custom-protocol EM3D as well.

The application-specific protocols also serve to diminish the performance difference between the various Tempest implementations. Typhoon-0 is 28–327% slower than Typhoon for standard shared memory, but only 5–47% slower for the custom protocols. Similarly, Typhoon-1's worst-case performance disadvantage is reduced from 122% to 13%. There are two reasons for this trend. First, the custom protocols eliminate most of the demand fetches from the computation iterations. The access control mechanism is only lightly used, if at all, so its overheads are insignificant. Second, the optimized communication in the custom protocols usually takes the form of message sends from the compute processor. These sends must cross the bus on all three systems; the tight coupling of the network interface and protocol processor on Typhoon improves performance only on the receiving node.

4.3 Impact of network latency

To emphasize the impact of Tempest support, the previous section assumed a fairly aggressive network latency of 0.5 μs. Although dedicated MPP networks may match or surpass this speed, current high-performance off-the-shelf networks are more typically in the 10–30 μs range. This section examines the effect of these larger network latencies on system performance.

We simulated all our systems and benchmarks with one-way network latencies of 0.5, 2.5, 5,



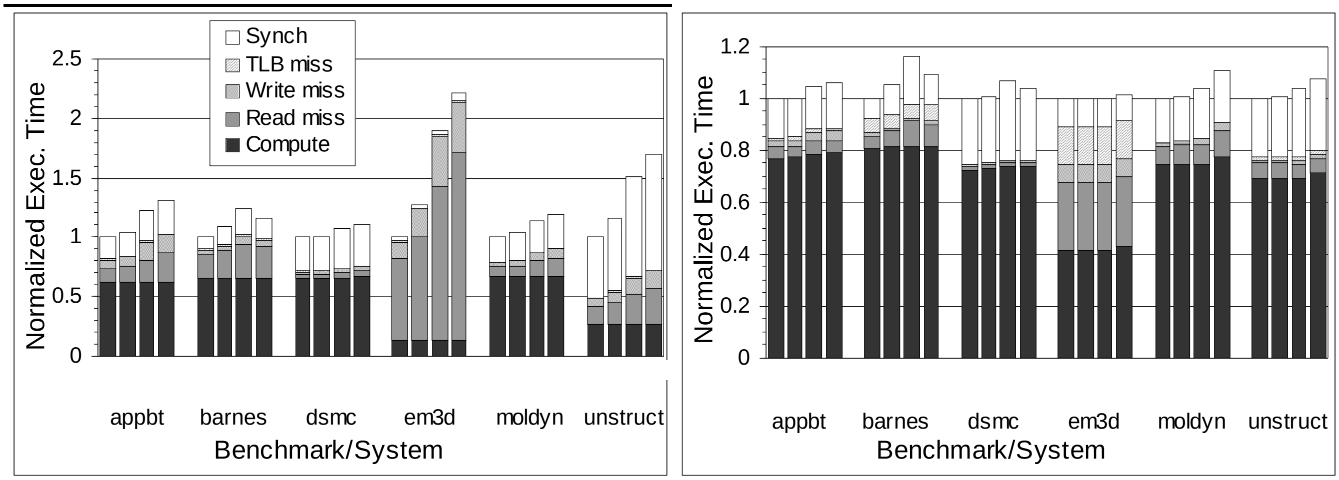
(a) Appbt (b) Barnes

Figure 8. Speedups at various network latencies.

25, and 50 μs (100, 500, 1,000, 5,000, and 10,000 processor cycles, respectively). Due to space limitations, Figure 8 shows speedups for only two of the benchmarks (Appbt and Barnes).

Results for most of the benchmarks are qualitatively similar to the Appbt graph. Because the application-specific protocols reduce or eliminate stalling for demand misses, they tolerate large network latencies better than standard shared memory. As the latency increases from 0.5 to 50 μs, the standard shared-memory benchmarks on the idealized Simple COMA system slow down by at least a factor of five (for DSMC) and as much as a factor of 33 (for unstructured). In contrast, the custom-protocol versions slow down by as little as 1% (for EM3D) and at worst a factor of three (for Barnes). As a result, the relative improvement provided by the application-specific protocols grows with increasing latency. For example, the Barnes application-specific protocol provides a speedup of only 8% over the original version on Typhoon at a 0.5 μs latency, but at a 50 μs latency this improvement grows to 94%.

Higher network latencies also reduce the performance difference between the systems, especially for standard shared memory. Increasing the fraction of execution time due to the network itself diminishes the relative impact of other overheads. Sufficiently high network overheads make any hardware DSM support superfluous, leaving software-only fine-grain [50, 48] or page-based [26] systems as the most cost-effective approaches to DSM in this domain.



(a) Standard shared memory

(b) Custom protocols

Figure 9. Typhoon performance at various protocol processor speeds.

4.4 Impact of protocol-processor performance

The results presented thus far assume that Typhoon's integrated protocol processor is as powerful as the off-the-shelf general-purpose CPU used by Typhoon-1 and Typhoon-0. As discussed earlier, this assumption is unrealistic; due to the additional design time required, it is probable that Typhoon's integrated processor will be slower than a contemporary off-the-shelf CPU.

To examine the effect of this assumption, we simulated two additional versions of Typhoon, changing the protocol processor to be two and four times slower than the compute CPU. Figure 9 shows the benchmark execution times for these systems, where T-*n*X indicates that the protocol CPU is *n* times slower than compute CPU. For the standard shared-memory benchmarks, slowing the protocol processor by a factor of two reduces application performance by at most 28%, while a factor of four processor slowdown reduces overall performance by up to 90%. Excluding EM3D and unstructured, application performance is reduced by at most 8% and 24%, respectively. For comparison, Figure 9 also includes the Typhoon-1 results from Figure 7. For most of the applications, Typhoon-1 is nearly as fast as the Typhoon version with the fourfold-slower protocol processor, and is faster for Barnes. These results suggest that Typhoon's protocol processor need not be as fast as the compute CPU, but at some point protocol processor speed does become a bottleneck. In general, the most effective protocol CPU speed is probably dependent on network, memory, and bus bandwidths as well as the speed of the compute CPU.

Figure 9 also shows results for the custom-protocol versions of the benchmarks. As with the less integrated systems, the application-specific protocols mitigate the impact of increased overheads; the twofold and fourfold decreases in protocol processing power reduce overall performance by at most 5% and 16%, respectively. Again, however, the protocol processor can become the bottleneck: the slowest Typhoon system is slower than Typhoon-1 for two of the applications (Barnes and DSMC) and nearly as slow for the others.

5 The Typhoon-0 prototype

To demonstrate the feasibility of combining custom fine-grain access control logic with user-level protocol software—the common theme of the designs presented in Section 3—we built a prototype of Typhoon-0. Measured speedups from the prototype agree with results from the simulator, strengthening the credibility of our other simulation results.

Typhoon-0's substantial reliance on off-the-shelf components makes it ideal for prototyping; in fact, the original motivation for the Typhoon-0 design was to enable a simple prototype of the concepts embodied in Typhoon. The relatively small effort required to construct the prototype—approximately two student-years—clearly demonstrates the advantage of this approach.

The prototype corresponds closely to the Typhoon-0 design outlined in Section 3.4 and simulated in Section 4: Sun SPARCStation-20 nodes, each with dual Ross HyperSPARC processors, an off-the-shelf network interface, and a fine-grain access control device. This device—the only custom component—is implemented as a FPGA-based plug-in board called *Vortex*. Due to space limitations, we describe the prototype only briefly, focusing on changes from the simulated Typhoon-0 of Section 4. We then report performance measurements and compare measured speedups with simulation results. Reinhardt's Ph.D. thesis [41] describes the prototype system more fully, including operating system support. Pfile's master's report [39] details the Vortex hardware implementation.

5.1 Differences between the simulated and constructed systems

Our simulated Typhoon-0 system of Section 4 reflects our original design for the prototype.

However, constraints encountered during construction induced three changes: the CPU configuration, the method for suspending bus operations, and the network used to interconnect the nodes.

First, due to cost and availability constraints, the prototype uses 66 Mhz processors with 256 Kbyte caches (vs. simulation parameters of 200 Mhz and 1 MByte, respectively).

Second, because of restrictions in the SPARCStation-20's bus arbiter, Vortex cannot simply suspend bus operations that conflict with the access tags as described in Section 4. Instead, Vortex returns an error acknowledgment for these operations, causing an exception on the memory reference instruction. After the block access fault is handled, software resumes the computation thread, reissuing the memory reference. Although standard Unix signals suffice for this scenario, the overhead of invoking the signal handler and resuming the process in Solaris 2.4 is roughly 100 μs—longer than servicing a simple remote miss. To avoid this penalty, we modified the kernel's trap vector code to invoke a user handler directly in under 5 μs [42, 53]. The suspended thread can be resumed without going through the kernel.

The third and most significant change involves the network used to connect the nodes. The prototype employs the Myricom Myrinet, the lowest-latency interconnect commercially available at that time. Although the Myrinet is far more efficient than a standard LAN, its overhead and latency are significantly higher than the network used in our simulations.

Each node connects to the Myrinet network via an interface card on the SBus (Sun's standard I/O bus). The interface card contains a simple processor, a DMA engine, and 128 KB of SRAM to store the processor's code and data. The host processors can access the interface SRAM directly using uncached memory operations; however, the interface processor can access host memory only via the DMA engine.

Our prototype's interface processors run custom software derived from Berkeley's Active Messages implementation [14]. Schoinas et al. [49] describe our modifications and enhancements for Tempest. As in the Berkeley implementation, our software allocates user-accessible send and receive queues in the shared SRAM. Each entry holds header information, a few words of message data, and an optional pointer to more message data in host memory. Short messages achieve low latency because they pass directly through the shared SRAM; larger messages enjoy the higher bandwidth of a DMA transfer. Due to the structure of the I/O MMU on the MBus/SBus bridge, DMA accesses are restricted to a dedicated, pinned kernel buffer. For performance, our device driver maps this buffer into the user process's address space as well.

The Myrinet interface does not provide an accessible message interrupt line to drive the message status bit in Vortex's dispatch register. We improvised this feature by co-opting a software-controlled status LED. As we installed each Myrinet interface, we removed the LED and replaced it with a connector. A short wire routes the signal to an input on the Vortex board. Our interface software toggles this signal on every message arrival.

5.2 Performance

Similar to Section 4, we first analyze a simple microbenchmark (Section 5.2.1), then report performance of our macrobenchmarks (Section 5.3). Section 5.4 compares the measured performance with results from the simulator used in Section 4.

5.2.1 Microbenchmark

This section breaks down the latency of simple remote misses on the prototype, similar to the breakdown for the simulated systems in Section 4.1. We used a logic analyzer to trace MBus activity for both cache- and home-node portions of simple read misses, write misses, and write upgrades (writes to read-only blocks). Because the analyzer observed only a single node, we derived network latencies by subtracting the home-node processing time from the observed latency of cache-node requests.

Table 3 presents timing breakdowns for a read miss and a write upgrade. Write miss timing differs from read miss timing in only one respect: because the home node does not keep a copy of the block, the "write data to memory" and "change tag to ReadOnly" steps are eliminated.

Table 3 shows that, for either transaction, nearly 70% of the latency is attributed to the network.

This latency—from the sending protocol processor's last write to the Myrinet interface until the receiving node's Vortex card invalidates the status cacheable control register (CCR)—is over 12 µs for a short message and over 23 µs for a message involving a 32-byte DMA. The nodes

Table 3. Remote miss latency breakdown for the Typhoon-0 prototype.

	Step	Read miss Inc.	Read miss Cum.	Write upgrade Inc.	Write upgrade Cum.
Caching node	detect HW cache miss, issue (aborted) bus transaction	0.24	0.24	0.16	0.16
	invalidate, refetch status CCR	0.52	0.76	0.52	0.68
	dispatch & execute handler	0.70	1.46	0.70	1.38
	send msg	1.14	2.60	1.16	2.54
	request msg latency	12.58	15.18	12.58	15.12
Home node	invalidate, refetch status CCR	0.52	15.70	0.52	15.64
	read msg, dispatch handler	2.12	17.82	2.12	17.76
	directory lookup, branch	0.72	18.54	0.72	18.48
	send msg header	1.84	20.38	1.14	19.62
	change tag to Invalid	0.94	21.32	0.94[a]	19.62
	fetch block buffer	0.52	21.84	n.a.	19.62
	write data to DMA region	0.64[b]	22.48	n.a.	19.62
	write data to memory	0.56	23.04	n.a.	19.62
	change tag to ReadOnly	0.14	23.18	n.a.	19.62
	finish msg send	0.62	23.80	n.a.	19.62
	response msg latency	23.22	47.02	12.58	32.20
Caching node	invalidate, refetch status CCR	0.52	47.54	0.52	32.72
	read msg head, dispatch handler	2.20	49.74	2.84	35.56
	copy msg data to memory	1.78	51.52	n.a.	35.56
	change tag	0.12	51.64	0.12	35.68
	resume flag handshake	1.30	52.94	1.30	36.98
	reexecute bus transaction	0.42	53.36	0.18	37.16
	Total		53.36		37.16

Latency (microseconds)

a. This operation is not on the critical path.

b. This time includes a TLB miss on the protocol processor (see text).

communicate through a single Myrinet switch with a worst-case latency of 550 ns [8], so virtually all of this time is consumed by the SBus bridge and the Myrinet interface. The primary culprit is control software running on the Myrinet processor—a 16-bit non-pipelined CISC processor clocked at 25 MHz. Newer versions of the Myrinet SBus interface card use a much faster processor; these devices would improve the prototype's performance noticeably.

The high cost of executing instructions on the Myrinet interface processor also contributes to the large additional latency for messages involving DMA. Software must parse the message header to detect the DMA request, then write the appropriate parameters to the DMA engine's control registers. On the sending node, 3.0 µs elapse from the final protocol processor write to DMA initiation. On the receiver, 2.2 µs pass by from the completion of the incoming DMA until Vortex signals the message arrival. The DMA itself consumes about 0.5 µs on the MBus, roughly

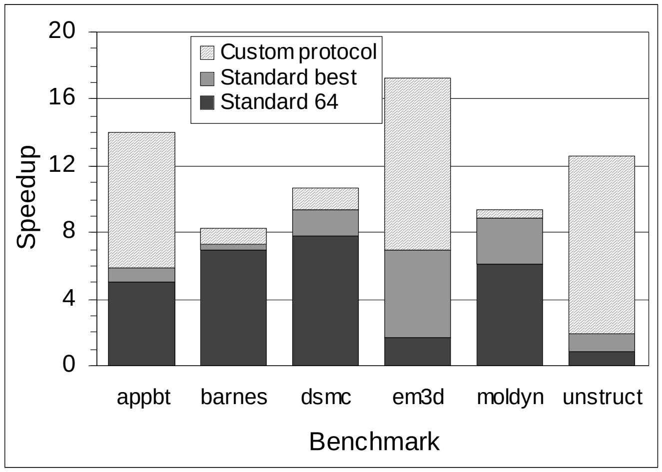


Figure 10. Application speedups on 16 nodes of the prototype .

half of which is due to a TLB miss in the Sbus bridge's I/O MMU. (For simplicity, we reserve a full page for each DMA message buffer, so accesses to the DMA region—from both the protocol processor and the SBus bridge—practically always involve a TLB miss.).

5.3 Macrobenchmarks

To measure the prototype's application-level performance, we ran the same six applications and data sets as in Section 4.2. Again, the reported speedups are relative to the best available sequential version and exclude initialization and the first parallel iteration. To reduce variation from external factors, we ran each experiment three times and selected the fastest run.

Figure 10 displays speedups obtained on 16 nodes of the prototype. The shortest bars indicate the speedup using standard shared memory with 64-byte blocks. The darker bars show additional speedup gained by selecting the best block size for the specific application (up to 2048 bytes)—but still using standard shared memory. For these results, appbt and Barnes use a block size of 128 bytes, DSMC uses 256 bytes, unstructured uses 1024 bytes, and EM3D and Moldyn use 2048 bytes. The hatched bars show speedups from the custom Tempest protocols described in Section 2.2.

In general, Figure 10 confirms the results of Section 4.2: appbt, EM3D, and unstructured see a significant gain from custom protocols, while the other benchmarks see smaller improvements. Unfortunately, it is difficult to compare Figure 10 with the results for larger network latencies in

Section 4.3 due to a number of differences, including system size (16 vs. 32 nodes) and processor speed (66 vs. 200 MHz). The next section correlates measured prototype performance with results from the simulator by eliminating these discrepancies in the simulated system.

5.4 Comparison with simulated results

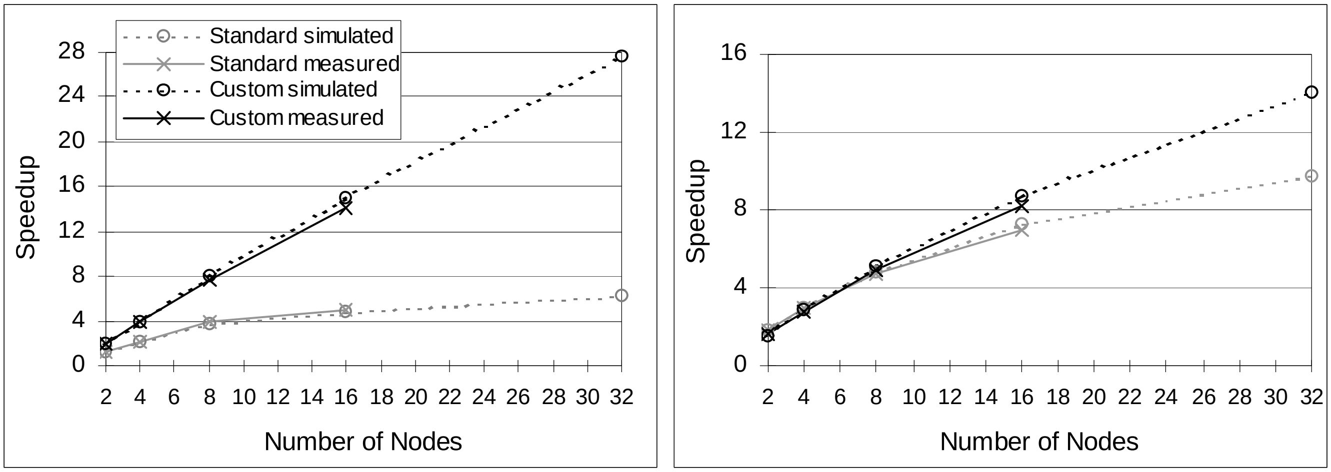
The Typhoon-0 prototype provides the opportunity to substantiate the accuracy of the simulation system used in the previous chapter. If the simulator's results for a configuration similar to the prototype correlate with measurements, it increases our confidence in its predictions for other configurations.

Most of the differences between our original simulated configuration and the prototype were simple to reflect in the simulator: processor speed (200 MHz vs. 66 MHz), cache size (1 MB vs. 256 KB), block access fault suspend/resume (arbiter control vs. error abort/software restart), and barrier synchronization (hardware vs. software).

Adapting the network model was more involved. The SBus bridge and Myrinet interface are the dominant contributors to message latency (as seen in Section 5.2.1) and are bottlenecks for message throughput [14]. Rather than develop detailed models of these complex components, we modified our existing network interface model—adjusting register access costs and adding an occupancy component—to approximate the Myrinet's performance characteristics. We tuned the new model until the simulator's performance matched the prototype's on a message-passing microbenchmark.

For historical reasons, the simulated Typhoon-0 and the prototype also differ in their strategy for message buffering on network overflow: the simulated system buffers at the sender but the prototype buffers at the receiver. Although this difference is moot if traffic is low enough that no buffering occurs, it results in a 20% performance discrepancy for the custom-protocol version of appbt. Rather than modifying either system—a significant effort—we modified the protocol to send data in fewer, longer messages to avoid triggering flow-control stalls.

The results are quite positive. Figure 11 plots the simulated and actual speedups for appbt and Barnes for both 64-byte block standard shared memory and custom protocol versions. The mean



(a) Appbt (b) Barnes

Figure 11. Comparison of simulated and measured speedups.

error is 3.0% and the worst case is 6.2% (for the appbt custom protocol on 16 nodes). The accuracy is higher for standard shared memory than for the custom protocols (2.5% vs. 3.5% mean error). This trend is reasonable, because custom protocols tend to have bursty communication patterns which place greater peak demand on the network—the least accurately modeled component.

6 Related Work

Alewife [2], S3.mp [38], and FLASH [28] implement their shared-memory coherence protocols at least partially in software. These systems forgo the economies of workstation-based nodes for higher performance, incorporating custom memory controllers and, in the case of Alewife, a custom CPU as well. None of these systems provides a protected, user-level interface to its memory coherence mechanisms, which limits users to a predefined set of system-provided policies. The Alewife and FLASH projects also explored combined message-passing and shared-memory programming models [27, 22]. However, both systems primarily viewed user-level message passing as an alternative to, rather than a building block for, shared memory. In particular, memory-to-memory message-based data transfers are semantically equivalent to memory-to-memory copies, meaning that users cannot explicitly transfer data without renaming it.

Research in workstation-based DSM has focused on all-software methods, using page-based coherence [32] or software-only fine-grain coherence [50, 48]. Munin [10] was a page-based DSM system that provided a fixed menu of protocols; programmer annotations guided protocol selection. Orca used compiler analysis with run-time feedback to control object replication policies and update protocols [5]. Bianchini et al. [6] propose a hardware accelerator for page-based DSM. As our results in Section 4.3 indicate, hardware support for fine-grain coherence is unlikely to be cost effective when faced with the high latencies (and overheads) of traditional local-area networks. Recent work shows that, on the Typhoon-0 prototype, fine-grain sequentially consistent DSM performs comparably to a coarse-grain DSM using lazy release consistency on the same hardware [58]. We expect that fine-grain DSM will show a performance advantage on systems with lower overheads and lower network latencies, such as those simulated in Section 4.

Other approaches to optimizing communication in DSM systems include relaxed memory models [25], prefetching [35], and special writes that deliver data to other nodes [31, 47]. The Tempest interface can support all of these optimizations; however, we have focused on exploring the limits of optimization by using fully custom protocols. It would be interesting to determine to what extent these more restricted models can deliver the performance improvements afforded by custom protocols.

7 Summary and Conclusions

This paper explores the interaction between two approaches to efficient distributed shared memory on networks of workstations: application-specific coherence protocols and hardware acceleration. We have designed and simulated three systems—Typhoon, Typhoon-1, and Typhoon-0—that add varying levels of hardware DSM acceleration to a cluster of workstations. All three systems support the Tempest interface, which lets programmers optimize communication by developing arbitrary, application-specific coherence protocols in unprivileged software. To demonstrate the feasibility of our hardware approach, we constructed a prototype of Typhoon-0. Measured speedups from the prototype match simulation results within 6%.

Our simulations indicate that the most aggressive system—Typhoon—runs standard shared-memory benchmarks within 10-25% of the speed of an idealized hardwired-protocol system, indicating that protocol flexibility need not significantly compromise effectiveness for unmodified code. Even with Typhoon's competitive base performance, custom protocols still show dramatic gains for two of our benchmarks, significantly outperforming the hardwired-protocol system as well. Custom protocols are also effective at hiding the higher overheads of less aggressive hardware; they decrease the performance gap between Typhoon-1 and Typhoon from 222% to 11% and the gap between Typhoon-0 and Typhoon from 427% to 22%. Custom protocols are also generally effective at reducing performance sensitivity to network latency and protocol-processor speed as well. However, Typhoon executes the standard versions of three of our benchmarks nearly as quickly as the custom-protocol versions, and faster than the custom-protocol versions on the less aggressive systems. In these cases, the additional hardware acceleration avoids the need to develop more efficient custom protocols.

References

[1] S. V. Adve and M. D. Hill. “Weak ordering - a new definition.” In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. “The MIT Alewife machine: Architecture and performance.” In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[3] T. E. Anderson, D. E. Culler, and D. A. Patterson. “A case for NOW (networks of workstations).” *IEEE Micro*, 15(1):54–64, Feb. 1995.

[4] D. Bailey, J. Barton, T. Lasinski, and H. Simon. “The NAS parallel benchmarks.” Technical Report RNR-91-002 Revision 2, Ames Research Center, Aug. 1991.

[5] H. E. Bal and M. F. Kaashoek. “Object distribution in Orca using compile-time and run-time techniques.” In *Proceedings of the Eigth Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA ’93)*, pages 162–177, Oct. 1993.

[6] R. Bianchini, L. I. Kontothanassis, R. Pinto, M. D. Maria, M. Abud, and C. L. Amorim. “Hiding communication latency and coherence overhead in software DSMs.” In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 198–209, Oct. 1996.

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

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

[9] D. Burger and S. Mehta. “Parallelizing appbt for a shared-memory multiprocessor.” Technical Report 1286, Computer Sciences Department, University of Wisconsin–Madison, Sept. 1995.

[10] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. “Implementation and performance of Munin.” In *Proceedings of the Thirteenth ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, Oct. 1991.

[11] S. Chandra and J. R. Larus. “Optimizing communication in HPF programs on fine-grain distributed shared memory.” In *Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 100–111, June 1997.

[12] S. Chandra, B. Richards, and J. R. Larus. “Teapot: Language support for writing memory coherence protocols.” In *Proceedings of the SIGPLAN ’96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[13] 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, Nov. 1993.

[14] D. E. Culler, L. T. Liu, R. P. Martin, and C. O. Yoshikawa. “Assessing fast network interfaces.” *IEEE Micro*, pages 35–43, Feb. 1996.

[15] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. “An integrated compile-time/run-time software distributed shared memory system.” In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 186–197, Oct. 1996.

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

[17] B. Falsafi and D. A. Wood. “Scheduling communication on an SMP node parallel machine.” In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture (HPCA)*, pages 128–138, Feb. 1997.

[18] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. “Memory consistency and event ordering in scalable shared-memory multiprocessors.” In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[19] R. B. Gillett. “Memory channel network for PCI.” *IEEE Micro*, 16(1):12–18, Feb. 1996.

[20] L. Gwennap. “Intel’s P6 bus designed for multiprocessing.” *Microprocessor Report*, 9(7), May 30, 1995.

[21] E. Hagersten, A. Saulsbury, and A. Landin. “Simple COMA node implementations.” In *Proceedings of the 27th Hawaii International Conference on System Sciences*, Jan. 1994.

[22] J. Heinlein, K. Gharachorloo, S. A. Dresser, and A. Gupta. “Integration of message passing and shared memory in the Stanford FLASH multiprocessor.” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 38–50, Oct. 1994.

[23] M. Heinrich, J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. “The performance impact of flexibility in the Stanford FLASH multiprocessor.” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 274–285, 1994.

[24] M. D. Hill. “Multiprocessors should support simple memory consistency models.” *IEEE Computer*, vol. 31. no. 8, pp. 28–34, Aug. 1998.

[25] P. Keleher, A. L. Cox, and W. Zwanepoel. “Lazy release consistency for software distributed shared memory.” In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[26] P. Keleher, S. Dwarkadas, A. Cox, and W. Zwaenepoel. “TreadMarks: Distributed shared memory on standard workstations and operating systems.” In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, Jan. 1994.

[27] D. Kranz, K. Johnson, A. Agarwal, J. Kubiatowicz, and B.-H. Lim. “Integrating message-passing and shared-memory: Early experience.” In *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 54–63, May 1993.

[28] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. “The Stanford FLASH multiprocessor.” In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, Apr. 1994.

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

[30] J. Laudon and D. Lenoski. "The SGI Origin: A ccNUMA highly scalable server." In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 241–251, June 1997.

[31] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. "The Stanford DASH multiprocessor." *IEEE Computer*, 25(3):63–79, Mar. 1992.

[32] K. Li and P. Hudak. "Memory coherence in shared virtual memory systems." *ACM Transactions on Computer Systems*, 7(4):321–359, Nov. 1989.

[33] M. Marchetti, L. Kontothanassis, R. Bianchini, and M. L. Scott. "Using simple page placement policies to reduce the cost of cache fills in coherent shared-memory systems." In *Proceedings of the Ninth International Parallel Processing Symposium*, Apr. 1995.

[34] M. Martonosi, D. Ofelt, and M. Heinrich. "Integrating performance monitoring and communication in parallel computers." In *Proceedings of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 138–147, May 1996.

[35] T. Mowry and A. Gupta. "Tolerating latency through software-controlled prefetching in shared-memory multiprocessors." *Journal of Parallel and Distributed Computing*, 12:87–106, June 1991.

[36] S. S. Mukherjee, S. K. Reinhardt, B. Falsafi, M. Litzkow, S. Huss-Lederman, M. D. Hill, J. R. Larus, and D. A. Wood. "Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator." In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[37] S. S. Mukherjee, S. D. Sharma, M. D. Hill, J. R. Larus, A. Rogers, and J. Saltz. "Efficient support for irregular applications on distributed-memory machines." In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, July 1995.

[38] A. Nowatzyk, M. Monger, M. Parkin, E. Kelly, M. Browne, G. Aybay, and D. Lee. "S3.mp: A multiprocessor in a matchbox." In *Proc. PASA*, 1993.

[39] R. W. Pfile. "Typhoon-Zero implementation: The Vortex module." Technical Report 1290, Computer Sciences Department, University of Wisconsin–Madison, Oct. 1995.

[40] S. K. Reinhardt. "Tempest interface specification (revision 1.2.1)." Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, Feb. 1995.

[41] S. K. Reinhardt. *Mechanisms for Distributed Shared Memory*. PhD thesis, Computer Sciences Department, University of Wisconsin–Madison, Dec. 1996.

[42] S. K. Reinhardt, B. Falsafi, and D. A. Wood. "Kernel support for the Wisconsin Wind Tunnel." In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 73–89, Sept. 1993.

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

[44] S. K. Reinhardt, J. R. Larus, and D. A. Wood. "Tempest and Typhoon: User-level shared memory." In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 325–337, Apr. 1994.

[45] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. "Decoupled hardware support for distributed shared memory." In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 34–43, May 1996.

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

[47] E. Rosti, E. Smirni, T. Wagner, A. Apon, and L. Dowdy. "The KSR1: Experimentation and modeling of poststore." In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, May 1993.

[48] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. "Shasta: A low overhead, software-only approach for supporting fine-grain shared memory." In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 174–185, Oct. 1996.

[49] I. Schoinas, B. Falsafi, M. D. Hill, J. R. Larus, C. E. Lucas, S. S. Mukherjee, S. K. Reinhardt, E. Schnarr, and D. A. Wood. “Implementing fine-grain distributed shared memory on commodity SMP workstations.” Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, Mar. 1996.

[50] I. Schoinas, B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. “Fine-grain access control for distributed shared memory.” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–306, Oct. 1994.

[51] J. P. Singh, W.-D. Weber, and A. Gupta. “SPLASH: Stanford parallel applications for shared memory.” *Computer Architecture News*, 20(1):5–44, Mar. 1992.

[52] Sun Microsystems Inc. *SPARC MBus Interface Specification*, Apr. 1991.

[53] C. A. Thekkath and H. M. Levy. “Hardware and software support for efficient exception handling.” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 110–119, Oct. 1994.

[54] Thinking Machines Corporation. “The Connection Machine CM-5 technical summary,” 1991.

[55] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. 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.

[56] S. C. Woo, J. P. Singh, and J. L. Hennessy. “The performance advantages of integrating block data transfer in cache-coherent multiprocessors.” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 219–229, Oct. 1996.

[57] Z. Xu, J. R. Larus, and B. P. Miller. “Shared-memory performance profiling.” In *Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, June 1997.

[58] Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. “Relaxed consistency and coherence granularity in DSM systems: A performance evaluation.” In *Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, June 1997.