Appears in: *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

# Tempest and Typhoon: User-Level Shared Memory

Steven K. Reinhardt, James R. Larus, and David A. Wood

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

## Abstract

Future parallel computers must efficiently execute not only hand-coded applications but also programs written in high-level, parallel programming languages. Today's machines limit these programs to a single communication paradigm, either message-passing or shared-memory, which results in uneven performance. This paper addresses this problem by defining an interface, *Tempest,* that exposes low-level communication and memory-system mechanisms so programmers and compilers can customize policies for a given application. *Typhoon* is a proposed hardware platform that implements these mechanisms with a fully-programmable, user-level processor in the network interface. We demonstrate the utility of Tempest with two examples. First, the *Stache* protocol uses Tempest's fine-grain access control mechanisms to manage part of a processor's local memory as a large, fully-associative cache for remote data. We simulated Typhoon on the Wisconsin Wind Tunnel and found that Stache running on Typhoon performs comparably (±30%) to an all-hardware $Dir_NNB$ cache-coherence protocol for five shared-memory programs. Second, we illustrate how programmers or compilers can use Tempest's flexibility to exploit an application's sharing patterns with a custom protocol. For the EM3D application, the custom protocol improves performance up to 35% over the all-hardware protocol.

## 1 Introduction

Consensus is emerging on two aspects of massively-parallel supercomputing. At the application level, these systems increasingly will be programmed in high-level parallel languages—such as HPF [17]—that support a shared address space in which processes uniformly reference data. At the lowest level, the machines are converging on workstation-like nodes connected by a point-to-point network. Unfortunately, no consensus has emerged on the communication model—shared memory or message passing—for parallel languages.

Current parallel machines take an all-or-nothing approach to providing a shared address space. Message-passing machines, such as the Thinking Machines CM-5 [44] and Intel Paragon [20], have no hardware support, so compilers for these machines synthesize a shared address space by generating code that copies values between processors in messages. In the best case, this approach performs well and efficiently uses a machine's memory and communications network. Unfortunately, the approach relies on static program analysis and performance degrades dramatically when a compiler (or programmer) cannot fully analyze a program.

On the other hand, shared-memory machines, such as the Kendall Square KSR-1 [21] and Stanford DASH [27], implement cache-coherent shared-memory policies and mechanisms entirely in hardware. Although these machines share a common hardware base with message-passing machines (workstation-like nodes and point-to-point message passing), compilers for shared-memory machines have been constrained to use memory loads and stores for communication, even when static analysis could identify better approaches [24].

This paper describes *Tempest* and *Typhoon*. Tempest is an interface that permits programmers and compilers to use hardware communication facilities directly and to modify the semantics and performance of shared-memory operations. It enables an application's user-level code to support shared memory and message passing efficiently, along with hybrid combinations of the two. Typhoon is a proposed hardware implementation of this interface.

At one extreme, programs with coarse-grain, static communication can send messages. Tempest does not impose shared-memory overhead on these message-passing programs. At the other extreme, programs with unanalyzable,

This work is supported in part by NSF PYI/NYI Awards CCR-9157366 and CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, a Univ. of Wisconsin Graduate School Grant, a Wisconsin Alumni Research Foundation Fellowship, an AT&T Ph.D. Fellowship, and donations from Digital Equipment Corporation, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

dynamic behavior can rely entirely on transparent shared-memory using the *Stache* protocol constructed on Tempest (see Section 3). Stache employs part of each processor's local DRAM memory as a large, fully-associative "level three cache," similar to the caches in cache-only memory architecture (COMA) machines. Unlike the extensive, custom hardware in a COMA machine, Stache runs in user-level software using the Tempest interface.

However, the real benefits of Tempest lie between these extremes, where programmers and compilers implement hybrid protocols that exploit an application's semantics to improve performance. For example, computing on dynamic, irregular grids requires run-time support, since a grid's structure is unknown at compile time. Software inspector-executor schemes incur large overheads to determine communication patterns. Transparent shared memory incurs large overheads to communicate modified values by invalidating and re-requesting them. In a hybrid protocol, a programmer or compiler could use Tempest's mechanisms to detect remote accesses dynamically, thereby eliminating most of the inspector. Furthermore, a delayed update protocol, similar to the one implemented for EM3D (Section 4), can eliminate non-essential communication by transmitting only modified values.

The Tempest interface defines an efficient set of *mechanisms* that compilers and run-time systems can use to implement shared-memory *policies*. User-level access to these building blocks is essential because the range of future applications, algorithms, data structures, and optimizations is impossible to anticipate. With the user-level facilities provided by Tempest, a programmer or compiler can tailor memory semantics to fit a particular program or data structure, much as RISC processors enable compilers to tailor instruction sequences for a particular function call or data reference [46].

Tempest contains the following four types of user-level mechanisms:

- Low-overhead messaging, which permits the fast communication fundamental to the performance of many parallel programs.
- Bulk data transfer, which allows large data transfers to overlap computation.
- Virtual memory management, which enables a compiler or run-time system to manage a program's address space efficiently and to migrate and replicate data without renaming.
- Fine-grained memory access control, which permits efficient run-time detection of memory access patterns and enforcement of memory consistency.

Typhoon is a proposed implementation of Tempest that provides hardware support for these mechanisms using a network interface device (dubbed the NP), which contains a fully-programmable, user-level processor. This processor is invoked either upon receipt of a message or by a local, fine-grain memory access fault. In either case, the NP processor uses a hardware-assisted dispatch mechanism to invoke a user-level procedure to handle the event. Protection is maintained by running the network interface processor in user mode and translating all addresses through a standard translation lookaside buffer (TLB).

We have implemented a virtual prototype of Typhoon using a modified version of the Wisconsin Wind Tunnel [36]. Existing shared-memory programs only need to be linked with the Stache library to run on Typhoon. Measurements of five benchmarks indicate that Stache performs comparably (±30%) to a conventional, all-hardware $Dir_NNB$ shared-memory system, despite Typhoon's greater flexibility. Furthermore, we show how customizing shared memory semantics to exploit sharing patterns can improve performance significantly. The EM3D application [7] runs up to 35% faster with a customized user-level protocol than on the all-hardware shared-memory system.

In the next section, we present the user-level communication and memory management mechanisms that comprise the Tempest interface. Section 3 shows how these mechanisms support transparent shared memory in the Stache protocol. Section 4 uses the EM3D application to illustrate the potential of user-customized memory semantics. Section 5 presents the detailed design of Typhoon, which illustrates the hardware necessary to implement the user-level mechanisms efficiently. Section 6 presents simulation results comparing Typhoon, using both Stache and a customized protocol, to an all-hardware shared-memory implementation. Finally, Section 7 surveys related work and Section 8 presents our conclusions.

## 2 Tempest: An Interface for User-Level Shared Memory

This section describes Tempest, a parallel machine interface that consists of four types of user-level mechanisms—low-overhead messages, bulk node-to-node data transfers, virtual memory management, and fine-grain access control—that we believe are both necessary and sufficient to implement the full range of shared-memory semantics in user-level software.[1] Fine-grain access control is an unusual mechanism, but is essential for transparent shared memory. All mechanisms are accessible from a user-level program.

Tempest can be realized in a variety of ways. Typhoon implements these mechanisms with a custom network interface processor. Other hardware implementations are,

1. We are investigating adding a set of synchronization primitives, to allow aggressive hardware implementations of common operations.

however, possible. Tempest can also be implemented in software for existing machines. We are currently investigating a "native" version for the CM-5. By abstracting from the implementation details, the Tempest interface provides portability between these different systems.

Section 3 shows how software can use these mechanisms to support transparent shared-memory semantics using the Stache protocol. Section 4 presents an example hybrid protocol, which exploits the sharing patterns of the EM3D application to improve performance. Section 5 describes Typhoon, an implementation of these mechanisms under current technical and economic constraints.

### 2.1 Low-Overhead Messages

Parallel machines typically communicate with point-to-point messages. Low overhead messages are fundamental to the performance of most programming models. The Active Messages model, where a message specifies a user-level handler to be invoked on its reception, provides an efficient building block for many paradigms, including shared memory [45]. With user-level access to fast messages, compilers can exploit the statically-determinable properties of data structures and program communication by explicitly communicating values. In addition, low-latency message handling is critical for transparent shared memory performance.

In Tempest, a processor sends a message by specifying the destination node, handler address, and data. The arrival of the message at its destination creates a thread that executes the handler, using the remainder of the message as arguments. Each handler executes atomically with respect to other message handlers, reducing synchronization requirements.

Our message model differs from similar systems [8,9,34] in that our message threads logically run concurrently with the primary computation thread. As in systems in which message handlers interrupt the main thread, shared resources must be protected; however, critical sections are sufficient since truly concurrent threads do not suffer from a "priority inversion" problem [41].

### 2.2 Bulk Node-to-Node Data Transfers

When compilers can fully analyze a program's communication pattern, they can improve performance by exploiting hardware mechanisms to overlap communication with computation. Furthermore, transferring bulk data via explicit messages is more efficient than using shared memory [23]. In Tempest, a processor initiates a bulk data transfer much like it would start a conventional DMA transaction, by specifying virtual addresses on both source and destination nodes. The transfer executes asynchronously with the computation thread. Completion of a transfer can be detected either by polling or with an interrupt.

### 2.3 Virtual Memory Management

Data replication and migration is required to eliminate unnecessary remote memory accesses. For dynamic and pointer-based data structures—for example, the tree used in the Barnes-Hut N-body simulations [4]—this replication and migration must be managed transparently at runtime. User-level memory management has two components: virtual address space management and access control. The mechanisms in this section enable user-level code to manage its address space. The next section describes mechanisms for access control.

Our memory model is a conventional flat, paged address space for each processing node. The operating system can reserve regions for conventional logical segments. A parallel process consists of a single address space per node, each with a private copy of the text segment[1] and private stack and heap segments (a single-program multiple-data model). A separate shared heap segment consists of a large user-reserved address range. Tempest relies on user-level code to provide semantics for accesses to this segment. The compiler or run-time library explicitly allocates physical memory pages at specified virtual addresses in this segment. Once allocated, these pages can be remapped or unmapped and freed. An access to an unmapped page or a write to a read-only page suspends the current computational thread and invokes a user-level handler. This mechanism provides a coarse-grain method for managing large pieces of the shared address space.

### 2.4 Fine-Grain Access Control

Memory access control is fundamental to transparent data replication. A runtime system must be able to track multiple copies of a datum to prevent unintentional incoherence. The access-control mechanisms must permit reads and writes to a local datum, permit reads but not writes, prevent both reads and writes, and transfer control to user-level code on an access violation. Virtual memory systems typically provide this form of access control [2]. The coarse granularity of their page-based mechanisms, however, is a poor match for many applications. In addition, access to page tables is typically an operating system privilege, so user-level changes incur a system call. User-level shared memory requires access control that is both fine grained and fast.

In our model, fine-grain access control is provided by tagged memory blocks. Every memory block—an aligned, power-of-two-sized region of memory, typically 32–128 bytes long—has an *access tag* of `ReadWrite`, `ReadOnly`, or `Invalid` that specifies which types of accesses are permitted. Tempest defines nine operations on memory blocks, listed in Table 1.

1. The system could provide transparent shared-memory semantics for text, but we ignore that here.

**TABLE 1. Operations on tagged memory blocks.**

| Operation | Description |
|---|---|
| read | Load with tag check; if access fault, suspend thread and invoke handler |
| write | Store with tag check; if access fault, suspend thread and invoke handler |
| force-read | Load without tag check |
| force-write | Store without tag check |
| read-tag | Return value of tag |
| set-RW | Set tag value to `ReadWrite` |
| set-RO | Set tag value to `ReadOnly` |
| invalidate | Set tag value to `Invalid` and invalidate any local copies |
| resume | Resume suspended thread(s) |

A processor's loads and stores translate to `read` and `write` operations on the corresponding memory block. A read or write on a `ReadWrite` block or a read on a `ReadOnly` block completes normally. However, an access to an `Invalid` block or a write on a `ReadOnly` block causes a *block access fault*, which is similar to a page fault. The faulting thread is suspended and a user-level handler invoked. The handler takes whatever actions are necessary to make the access permissible; it then updates the tag and restarts the access using `resume`.

## 3 User-Level Transparent Shared Memory

Efficient transparent shared memory involves replicating remote data to ensure that subsequent accesses are local and maintaining coherence between the multiple copies. Traditionally, the replication and coherence policies are both implemented in hardware, sometimes assisted by system software. This section describes a user-level transparent shared memory implementation that uses a new replication policy called *Stache* together with a conventional invalidation coherence protocol. The next section uses an application-specific coherence protocol, in conjunction with Stache, to improve the performance of an application. Section 6 presents performance results for these protocols running on the Typhoon system (described in Section 5).

Stache uses part of each processing node's local memory to replicate remote data. In effect, Stache uses this local memory as a large second- (or third-) level, fully-associative data cache, which eliminates much of the network traffic caused by capacity and conflict misses in smaller hardware caches [19]. For applications in which a processor manipulates data too large to fit in the hardware cache, but small enough to fit in local memory, Stache offers a large advantage over conventional directory-based shared-memory machines, which return a cache block to its home node on cache replacement [42]. COMA systems share this advantage, but require complex hardware support.

Stache is a user-level library that exploits the Tempest mechanisms. This library contains a page-fault handler, message handlers, block-access-fault handlers, and shared-memory allocation functions. It maps virtual addresses of shared data to local physical memory at page granularity, but maintains coherence at the block level.

To create a shared page, the home node processor allocates per-block directory structures (described below) and maps the physical page to the desired virtual address. It also initializes the block access tags to `ReadWrite` and associates the home node's ID with the virtual page in a distributed mapping table. As long as data on this page is not cached by another node, the home node can access it (and cache it in its hardware cache(s)) without software intervention.

When a node first accesses a shared page on a remote (non-home) node, the reference invokes a user-level page fault handler. This handler allocates a new physical page (a *stache page)*, maps it at the shared virtual address, and initializes the block access tags to `Invalid`. The home node's ID is found in the distributed table and cached in a local table. The handler then restarts the application at the faulting access. The restarted instruction now causes a block access fault because of the referenced block's `Invalid` tag. The thread is again suspended and a block access fault handler runs. The handler retrieves the home node's ID from the local table, sends a request for the block, and terminates.[1] At the home node, the request message invokes a handler that performs the appropriate coherence actions and returns the data. (If invalidations are required, the handler for the final invalidation acknowledgment actually sends the data.) When the response arrives from the home node, the message handler writes the data into the allocated page (with a `force-write` to bypass the tag check), changes the block's access tag (to `ReadOnly` or `ReadWrite`), and restarts the suspended thread. This time, the access completes and fetches the data into the CPU's cache.

Future accesses to the "stached" block complete at hardware speed. Since the page is mapped, but all other blocks are tagged `Invalid`, an access to another block on the page directly invokes a block access fault handler.

The scenario is similar on the home node, except that all blocks are initially `ReadWrite` and are downgraded to `ReadOnly` or `Invalid` as remote nodes request read-only or exclusive copies. Home block access fault handlers bypass sending requests and directly access directory data.

1. The remote request could be sent from the page fault handler, but this would require duplicating code from the block access fault handler. The performance gain does not justify the software maintenance overhead.

The Stache replication policy is independent of the coherence protocol. Our default coherence protocol is similar to the LimitLESS protocol [5], except that it is implemented entirely in software rather than partially in hardware. Specifically, the protocol preallocates 64 bits per cache block—to minimize bitfield operations, it allocates two bytes for state and six one-byte pointers. If more than six pointers are required, the current implementation uses the first four pointers as a bit vector. For systems larger than 32 nodes, the four node pointers contain the address of a larger auxiliary data structure.

When the Stache page fault handler cannot allocate a page, it must replace an existing stache page. In this case the handler invalidates all blocks within the page, sending modified data back to its home, and remaps the page at the new virtual address. Stache currently implements a simple FIFO replacement policy, since replacements are rare.

## 4 Custom User-level Shared Memory

To illustrate the benefits of a user-level, application-specific memory system, we implemented a new coherence protocol for the irregularly-structured EM3D application. EM3D models electromagnetic wave propagation through three-dimensional objects [7]. This program's principle data structure is a bipartite graph, in which *E nodes* represent electric field values and *H nodes* represent magnetic field values. Each iteration consists of two steps: first, new values are computed for the E nodes as a weighted sum of their neighboring H nodes; then the H nodes are updated based on the new E node values.

Program 1 illustrates the code to update the E nodes. For load balancing and to minimize communication, nodes are allocated evenly across the processors and each processor updates its local nodes (i.e., owners compute rule). For each local `e_node`, a processor fetches the values of neighboring `h_nodes`, which may be either local or remote.

Under transparent shared memory, this program incurs unnecessary communication. Each time an `e_node` is updated, the coherence protocol invalidates outstanding copies. But in the next step, each processor refetches the invalidated `e_nodes` to compute values for its local `h_nodes`. Thus in each iteration, a remote `e_node` (or `h_node`) will be fetched, cached, and invalidated, which requires at least four messages (i.e., request, response, invalidate, and acknowledge). Prefetching can hide communication latency, but does not reduce the message traffic. `Check_in` operations, which flush a block from a processor's cache [18], cut communication and latency by replacing the invalidation/acknowledgment with an asynchronous notification, but cannot attain the minimum of one message. In addition, these operations introduce additional computation that increases program overhead.

```
typedef struct e_node {
    double          value;
    int             edge_count;
    double          *weights;
    double          *(*h_nodes);
    struct e_node *next;
} e_node_t;

void compute_E()
{
    e_node_t        *n;
    int             i;

    for (n = e_nodes; n != NULL; n = n->next)
        for (i = 0; i < n->edge_count; i++)
            n->value -= n->h_nodes[i]->value
                        * n->weights[i];
    barrier();
}
```

**Program 1: Shared Memory EM3D.**

Ideally, at the end of a step, each processor would send its updated `e_nodes` (`h_nodes`) to the processors that need them. Using Tempest, we customized the Stache coherence protocol for EM3D. We use a delayed update protocol in which cache blocks become inconsistent within a step and are explicitly updated at the step's end. We introduce two new page types—a custom home page and a custom Stache page—and allocate graph nodes on the custom home pages. The customized Stache handlers are similar to the default handlers, except that they keep count of the number of stached `e_nodes` (`h_nodes`). The new home node handlers maintain a list of all outstanding `e_node` (`h_node`) copies. Because the program employs the owners compute rule, we can replace the barrier in Program 1 with a function that traverses the `e_node` (`h_node`) list and sends modified values. Because the handler is specific to EM3D, only the `value` field is sent, rather than the entire cache block.

In addition, this protocol does not require acknowledgments of update messages. Every processor knows how many remote graph nodes it has stached, and simply counts the updates and waits until they all arrive. The processors still must synchronize to ensure that graph nodes are not updated early, but this constraint is easily implemented as a fuzzy barrier in the handlers. By eliminating most synchronization and all invalidation traffic, the user-level coherence code attains near-minimum communication. In effect, this approach combines the communication efficiency of message passing with the low overhead and programming simplicity of shared memory [24].

Of course, the simple EM3D application could also be implemented efficiently with pure message passing, by a software inspection step that explicitly allocates space for remote nodes and builds an update list [7]. This approach is feasible because the graph is static and the inspector overhead can amortized over many iterations. However, in other codes the inspector cannot be moved out of the main

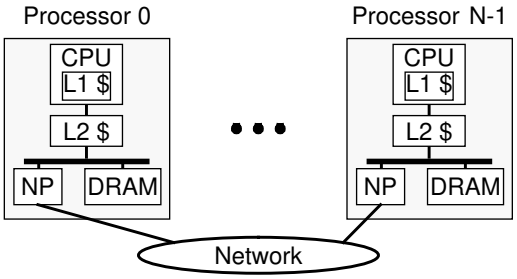


**FIGURE 1. Typhoon system architecture.** Level 2 caches are optional and not present in simulation.

loop. Furthermore, inspector-executor schemes are complex to implement [22,38,37]. Finally, a custom coherence protocol does not require extensive program modifications, unlike the software caching and updating in the message-passing version [7].

Although any protocol could be implemented in hardware (or system software [23]), system designers cannot anticipate the full range of protocols that programmers and compilers will devise. System-level protocols face a difficult choice between generality and specificity: a protocol general enough for many sharing patterns may not be optimal for any of them, but a protocol tailored to a specific pattern may not support others. Instruction set designers have learned to implement primitives, rather than solutions [46]. Memory systems should also provide mechanisms that compilers can compose into efficient solutions.

# 5 Typhoon: A User-Level Shared-Memory System

This section describes Typhoon, a system designed to implement Tempest's user-level memory mechanisms. This implementation has the following goals:

- to demonstrate the feasibility of implementing the mechanisms within current technological and practical constraints;
- to illustrate the hardware support needed to implement user-level mechanisms efficiently and to accelerate the common cases; and
- to provide a detailed design that can be simulated to obtain concrete performance results.

Typhoon consists of homogeneous, workstation-like processor/memory nodes connected by a high-bandwidth, low-latency point-to-point network (see Figure 1). For economic reasons, commodity components are used for the processor, bus, memory controller, and DRAM. Specifically, each Typhoon node has a SuperSPARC processor connected to a level-2 MBus [31].[1] The one custom component is the network interface device—the *network interface processor* (NP)—that connects to the shared bus to the network, as illustrated in Figure 2.

1. However, the basic design should work with any coherent bus using an ownership protocol and cache-to-cache transfers.

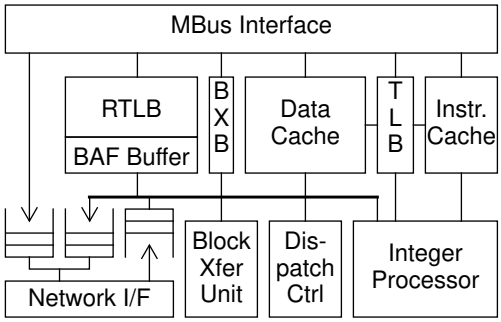


**FIGURE 2. NP Block Diagram.** RTLB denotes the reverse TLB; BXB is the block transfer buffer; the BAF buffer holds information on block access faults.

Typhoon's network architecture is based on that of the Thinking Machines CM-5 [26], but with a larger maximum packet payload (twenty 32-bit words rather than the CM-5's five). The only aspects of the network that are significant in this context are that it provides two independent virtual networks for deadlock avoidance and that it can be context-switched between user processes.

The following subsections describe how Typhoon implements the message support, bulk data transfer, memory management, and fine-grain access control of the Tempest interface described in Section 2.

## 5.1 Low-Overhead Messages

The network interface processor (NP) is not simply a passive network device, as illustrated in Figure 2. It contains a full SPARC integer processor, taken from a previous-generation SPARC design, with instruction and data caches to enhance performance and a TLB to allow virtual addressing. The processor is closely coupled to the network interface, which enables rapid handling of incoming messages.

Although coupling a network interface tightly to the primary processor can decrease end-to-end message latencies [16,9], any advantage is outweighed by the cost of changing the processor, which precludes leveraging off the tremendous investments in commodity components. On the other hand, decreasing message latency and reducing primary-processor interrupts justifies the investment in something larger and more complex than a passive network device. As commodity microprocessors become more aggressively superscalar, the overhead of interrupts or polling increases to unacceptable levels. Separating message handling on a distinct processor frees the primary processor for computation. Because the NP does not require state-of-the-art computation, a previous-generation integer core with a network interface on its cache bus is sufficient. A custom processor with tightly integrated messaging would provide even better performance, but it is not clear that the benefit justifies the additional design time and complexity.

Typhoon's send and receive queues are memory-mapped on the cache bus, so single-cycle loads and stores can transfer 32 bits between a queue and a general purpose NP register. A block transfer unit, also on the cache bus, can asynchronously transfer 32 bytes between a queue and an aligned address. The processor initiates a block transfer by storing the address to a control register, where the register address also encodes the transfer direction. A block transfer that misses in the data cache uses a separate 32-byte block transfer buffer to take advantage of MBus block transfers and maintain coherence with the local CPU cache without polluting the NP data cache.

The NP initiates a message send by storing the destination node ID to a memory-mapped register. Data words are moved to the send queue using stores or block transfers. The end of the message is signaled by a low-order bit in the register address. On the receiver, the first data word is interpreted as the receive handler PC, as in Active Messages [45]. The receive handler must pull the remainder of the message from the receive queue.

Scheduling on the NP is performed by a hardware-assisted dispatch loop [16]. The dispatch hardware constructs a handler PC in a dedicated register either by taking the first word of an incoming message or by using status bits as an offset from a user-specified base. Handlers can be prioritized or disabled via a user-accessible control register. The software dispatch loop simply reads the value in this register and jumps to it. If no action is required, the destination PC is the top of the loop itself. A handler terminates by jumping to the dispatch loop. To eliminate the need for synchronization between different handlers, scheduling among user handlers is *not* preemptive. Once a message or block access fault handler begins, it is run to completion.

The primary CPU can also send messages with memory-mapped stores across the MBus to a separate send queue. As on the NP, the destination node and final data word are distinguished by stores to distinct addresses. The CPU and NP send queues share a single network port. The primary CPU can send messages directly to its local NP, short-circuiting the network.

The ability to run user code on the NP processor is critical to providing performance and flexibility, but it does not come for free. The primary hardware cost is in the NP TLB (and RTLB, discussed below). However, the real cost is the design complexity of providing protected user-level access. For example, an NP handler could encounter a page fault. We avoid this problem by providing operating system calls that permit an application to specify part or all of its memory as swapped, rather than paged, so it is guaranteed to be in memory whenever the process is running. Thus an NP page fault is a user programming error that causes program termination. An alternate solution, taken in the Meiko CS-2 [30], NACKs the message that caused the fault, and brings in the page. However, a higher-level protocol must resend the message after the page-in completes. We expect to experiment with a combination of these schemes.

User-level handlers also complicate deadlock avoidance. There are two separate issues. First, the system cannot guarantee that every user protocol is deadlock-free, but must ensure that deadlock in one user application does not impact other users or the operating system. This is dealt with by context-switching the buffers in the network, as in the CM-5 [26]. Second, the system must provide sufficient support that users can write efficient deadlock-free protocols. Typhoon addresses this through a combination of mechanisms. First, the two independent virtual networks allow a pure request/response protocol to be deadlock-free if requests are sent only on one net and responses can always be processed. The scheduler gives lower priority to one of the networks, so using this net for requests guarantees that request handlers cannot starve response handlers. Second, the user must provide a buffer large enough to hold the message output of any single handler. If a send queue fills, the hardware will redirect further stores to this buffer transparently. This guarantees that any handler, once started, can run to completion without waiting for a send queue to empty. The user buffer is drained into the network by software as queue space becomes available. Finally, when either send queue is full, the scheduler invokes a user-level status handler instead of directly scheduling message handlers. The status handler implements a second-level dispatch, and can examine the PCs of incoming messages to decide whether they should be nacked, buffered, or processed (buffering any resulting sends).

### 5.2 Bulk Node-to-Node Data Transfers

Bulk data transfers are performed on the NP, asynchronously with respect to the primary CPU. Data must be packetized before being injected into the network. A maximum-size twenty-word packet holds a receive handler PC, a 32-bit address, and 64 bytes of data with two words to spare for status or extended addressing. To avoid tying up the NP, the data transfer thread suspends itself at regular intervals or when a message is received. The scheduler may be configured to invoke a status handler when either of the send queues is empty and this handler will reschedule any waiting data transfer threads. The primary CPU initiates a transfer by sending a message to its own NP containing either the transfer parameters or a pointer to the parameters. Because both the send and receive handlers are user code, they can be customized to implement arbitrary, application-dependent scatter-gather operations.

### 5.3 Virtual Memory Management

Conventional paged virtual memory hardware is sufficient to provide the needed user-level functions. The NP and primary CPU both implement versions of the SPARC

reference MMU [31]. While the primary processor and the NP may use separate page tables, they share a single table in our current implementation. The operating system interface is similar to that of [35].

### 5.4 Fine-Grain Access Control

As described in Section 2.4, the fine-grain access-control model provides access tags on memory blocks and defines nine operations on these blocks. In Typhoon, the `read` and `write` operations (the tag-checked accesses) correspond to primary CPU cacheable loads and stores. The NP enforces the tag semantics on these accesses and implements the remaining operations.

Tag semantics are enforced by monitoring the CPU's MBus transactions. Read and write misses are seen as read and read-invalidate transactions, respectively. A write to a cached but unowned block results in an invalidate transaction. Transactions involving `ReadWrite` blocks require no NP intervention since the memory controller responds with the data and the CPU acquires an owned cached copy. A read on a `ReadOnly` block is similar, except that the NP asserts the "shared" line to prevent the CPU from owning its cached copy. All other accesses are block access faults. The access is suspended by asserting the "inhibit" line to prevent the memory controller from responding, terminating the transaction with a "relinquish and retry" nack, and masking the CPU's bus request line to keep it from retrying the access. Information about the fault is placed in a buffer, where it is used by the NP dispatch hardware to schedule the appropriate block access fault handler and is accessible from the executing handler as well.

Because the NP monitors the node bus, it only observes the physical addresses of primary CPU references. To determine which, if any, action to take, the NP uses a reverse TLB (RTLB), indexed by physical page number, to determine the accessed block's tag state quickly. Each RTLB entry contains two bits for each 32-byte block in the page which encode four states: `ReadWrite`, `ReadOnly`, `Invalid`, and `Busy`. The first three correspond to Tempest's tag values. `Busy` has the same semantics as `Invalid`, but is useful for higher levels to distinguish blocks that require special handling, e.g. because they have been prefetched. The RTLB entry also contains several fields used to accelerate the invocation of a user-level block access fault handler: the virtual page number, the page mode, and 48 bits of uninterpreted state. The page mode is a four-bit value that is used, in conjunction with the access type (read or write) and access tag, to select the fault handler PC. The additional state bits are typically used for a 16-bit home node ID and a 32-bit pointer to an arbitrary user data structure (e.g., for Stache home pages, a vector of per-block directory structures).

All NP memory accesses bypass RTLB tag checking, implementing the Tempest model's `force-read` and `force-write` operations. Tag reads and writes are performed in the RTLB using memory-mapped operations. The `invalidate` operation invalidates any CPU cached copy via the MBus in addition to changing the access tag value. The `resume` operation merely unmasks the CPU's bus request line so that it can retry any suspended transaction.

Transactions that miss in the RTLB cause a "relinquish and retry" nack and are retried after the appropriate entry is fetched from memory. To improve the miss rate, an RTLB entry can either tag a 4 KB physical page or indicate a large region of physical memory that does not require block tags, e.g. text or kernel areas.

## 6 Performance of User-Level Shared Memory

In this section we compare the performance of Typhoon running the default invalidation-based Stache protocol (denoted *Typhoon/Stache*) against a conventional, all-hardware, directory-based $Dir_NNB$ cache-coherence protocol. Both systems are modeled using the Wisconsin Wind Tunnel, a parallel simulation system that runs on a Thinking Machines CM-5 [36].l Both target systems have 32 pro-

**TABLE 2. Simulation parameters.**

| | |
|---|---|
| **Common** | |
| CPU cache | 4-way assoc., random repl. |
| Block size | 32 bytes |
| CPU TLB | 64 ent., fully assoc., FIFO repl. |
| Page size | 4 Kbytes |
| Local cache miss | 29 cycles |
| Local writeback | 0 (assume perfect write buffer) |
| TLB miss | 25 cycles |
| Network latency | 11 cycles |
| Barrier latency | 11 cycles |
| **$Dir_NNB$ Only** | |
| Remote cache miss (cycles) | 23 + 5-16 if replacement[a] + network/directory cost + 34 |
| Remote cache invalidate (cycles) | 8 + 5-16 if replacement |
| Directory op (cycles) | 16 + 11 if block rcvd + 5 per msg sent + 11 if block sent |
| **Typhoon Only** | |
| NP TLB, RTLB | 64 ent., fully assoc., FIFO repl. |
| (R)TLB miss | 25 cycles |
| NP D-cache | 16 Kbytes, 2-way assoc |
| NP I-cache | 8 Kbytes, direct-mapped |

a. A replacement costs 5 or 16 cycles for shared or exclusive blocks, respectively.

cessing nodes and use latency parameters, listed in Table 2, loosely based on the DASH prototype [28]. The network latency is probably optimistic for future systems, but the low value will tend to favor $Dir_NNB$ by making Typhoon's overhead relatively larger.

Our simulation of Typhoon is accurate enough to run SPARC binaries for both the primary CPU and the NP. The Stache message and fault handlers are all written in C++ and compiled using *gcc*. Unaltered shared-memory programs are simply re-linked with the Stache runtime library. We use a version of Fast-Cache [25] to rewrite executables with instrumentation code that calculates instruction times, implements the NP special operations, and simulates the data caches and TLBs on both the primary CPU and the NP.

The major limitations of the simulations are that they do not accurately model network and bus contention, instruction cache behavior, stack references, and the difference in the execution rate of the superscalar primary processor and the simpler NP processor. Not modeling the NP instruction cache has no impact since the sum of the current handlers requires less than its 8 Kbyte capacity. We approximated the last difference by charging a single cycle for each instruction (plus memory system delays). This is nearly correct for the simple integer core of the NP, but gives the primary CPU a big boost since it executes many floating point operations in the applications.

We evaluated these two systems using five benchmarks: Appbt, a locally-parallelized version of the NAS benchmark [3]; Barnes, MP3D, and Ocean from the SPLASH suite [40]; and a transparent shared-memory version of EM3D (discussed in Section 4). Appbt is a computational fluid dynamics program, which solves multiple independent systems of non-diagonally dominant, block tridiagonal equations with a 5x5 block size. Barnes performs a gravitational N-body simulation using the Barnes-Hut algorithm. MP3D solves a rarefied fluid flow simulation (e.g., a wind tunnel). Ocean is a hydrodynamic simulation of a two-dimensional cross-section of a cuboidal ocean basin. Each application was simulated for two data sets, one significantly larger than the other (see Table 3). The smaller data sets are scaled for a 4 Kbyte cache, as advocated by Gupta, et al. [13], and fit entirely in the larger caches.

Figure 3 summarizes the simulation results. It shows the relative execution time of Typhoon/Stache versus $Dir_NNB$ (application execution time on Typhoon/Stache over its time on $Dir_NNB$). The results show that Typhoon/Stache outperforms the conventional protocol by as much as 25% for data sets that do not fit in the CPU's primary hardware cache. Typhoon/Stache can satisfy the resulting capacity and conflict misses from local memory, while $Dir_NNB$ must incur the overhead of additional remote accesses.

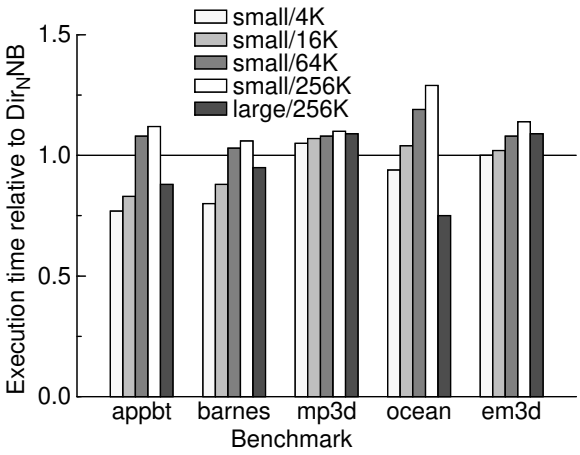


**FIGURE 3. Performance of Typhoon/Stache.** Shorter bars indicate better Typhoon/Stache performance. Legend indicates data set size/CPU cache size.

**TABLE 3. Application Data Sets.**

| Application | Small Data Set | Large Data Set |
|---|---|---|
| Appbt | 12x12x12 | 24x24x24 |
| Barnes | 2048 bodies | 8192 bodies |
| MP3D | 10,000 mols | 50,000 mols |
| Ocean | 98x98 grid | 386x386 grid |
| EM3D | 64,000 nodes, degree 10 | 192,000 nodes, degree 15 |

Of course, the $Dir_NNB$ results can be significantly improved using careful data placement to ensure that most misses are satisfied locally. Stenstrom, et al., show that a "first touch" page placement strategy eliminates much of the difference by allocating pages on the node that accesses them first [42]. Page migration algorithms also help, as does restructuring an algorithm to enhance locality [39]. However, most of these $Dir_NNB$ improvements require additional hardware, additional run-time overhead, or significant effort by the applications programmer. The Typhoon/Stache simulations required no modifications to the existing applications.

The most important result in Figure 3 is that the generality of Typhoon does not significantly degrade transparent shared memory performance, even in the worst case for these benchmarks. Typhoon/Stache performs within 30% of $Dir_NNB$ (excluding Ocean, within 15%) even when the data sets fit in the primary CPU's hardware cache. This is possible because, in the best case, the NP executes only 14 instructions to request a missing block, 30 instructions for the remote node to respond with the data, and 20 instructions when the data arrives at the requesting node. The critical path is even shorter, since most bookkeeping is performed after a message is sent.

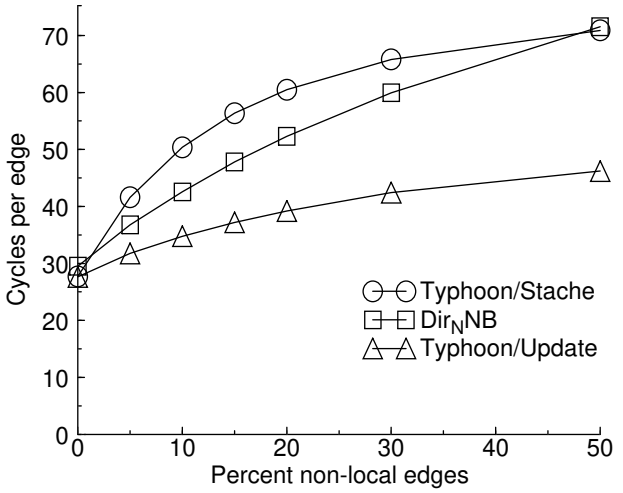


**FIGURE 4. EM3D Update Protocol Performance**
using the large data set (192,000 nodes, degree 15).

The real advantage for Typhoon comes from its flexibility, which enables compilers (and application programmers) to exploit knowledge of program behavior. As discussed in Section 4, communication in the EM3D application can be reduced to near-minimum using a customized delayed-update protocol. The performance of this protocol on Typhoon is displayed in Figure 3, where it is compared to Dir$_N$NB and the default invalidation-based Stache protocol. The performance advantage of the custom protocol increases as the fraction of edges that connect remote nodes is increased so that at 50% remote edges, the custom protocol outperforms Dir$_N$NB by 35%.

## 7 Related Work

The Tempest interface and Typhoon implementation evolved from our previous work on the Wisconsin Wind Tunnel [36], a parallel simulation system that runs on a Thinking Machines CM-5. The Wisconsin Wind Tunnel models cache-coherent shared-memory systems with a Stache-like caching scheme that synthesizes fine-grain access control from the CM-5's error-correcting code (ECC) bits. The Tempest interface generalizes these mechanisms and Typhoon provides first-class hardware support for them.

Typhoon's low-overhead messaging draws heavily on message-driven systems for fine-grain computations [8,9,33,34]. The NP's message sending interface closely follows the J-Machine's [10], but uses memory-mapped loads and stores instead of integrated instructions, and provides optimized block transfers. The NP's receiving interface is an "active" message model [8], in which the sender explicitly specifies the address of the handler. Rather than queuing the message in memory [10] or requiring polling by the primary processor [8], our handlers are directly invoked on a separate message processor, much like *T [33]. However, Typhoon provides a single, general-purpose message processor per node, not a pair of specialized co-processors like *T.

While the message models of these machines are similar, the proposed implementations differ. The fine-grain research machines tightly integrated their network interfaces into the primary processor [34,9]. *T multiplexes its logically separate coprocessors on a single RISC-like processor [33]. Commercial machines typically implement their network interfaces as a passive memory-mapped device [26,20]. The Thinking Machines CM-5 reduces message latency by mapping its interface at user-level, thereby eliminating the need for system calls. The Meiko CS-2 is similar to Typhoon since its tightly integrated network processor is separate from the primary CPU [30]. However, the CS-2 is optimized for relatively long messages and provides no fine-grain access control support. The Intel Paragon also provides a message processor, but uses a standard i860 CPU and a passive network device rather than tightly integrating the two [20].

Tagged memory, which Typhoon uses for fine-grain access control, has been implemented in many earlier machines. Machines for symbolic languages, such as Lisp, use word-granularity tags to support run-time typing [32,43]. Some parallel machines provide tags for fine-grain synchronization [1,5]. The word-granularity tags in the J-machine support shared-memory semantics [11], as well as other functions, but do not provide the `ReadOnly` tag necessary for replication. The IBM 801 and RS/6000 support fine-grain access control by providing a "lock bit" per 128 bytes in their TLB entries. However, the single bit limits them to two states, much like the Wisconsin Wind Tunnel [6]. Typhoon is, we believe, unique in using a reverse-TLB to provide tags for a commodity processor.

Tempest's user-level memory management interface is similar to Appel and Li's user-level primitives [2]. Both provide mechanisms to support distributed shared memory [29]. The differences arise from Tempest's fine-grain access control.

The Stache memory-allocation policy bears strong similarities to distributed shared memory systems [29]. Stache's default page location algorithm is similar to IVY's *fixed distributed manager algorithm*, where pages are assigned round-robin and the home nodes never change. However, Stache also allows pages to be allocated on specific nodes and provides support to allow explicit page migration. Stache differs from distributed shared memory systems because its maintains coherence on a much finer granularity.

Stache is also similar to Cache-Only Memory Architectures (COMA). Both use the local main memory to cache remote data [42]. However, COMA machines use complex hardware and make all of local memory into a cache

[14,21], while Stache uses much simpler hardware and only as much of the local memory as an application chooses to use. The Swedish Institute of Computer Sciences's Data Diffusion Machine (DDM) [14] and Kendall Square Research's KSR-1 [21] use coherence protocols fixed in hardware. Stache uses Typhoon's tag mechanisms to accelerate user-level software coherence algorithms. DDM and KSR-1 are both hierarchical machines, with hierarchical directory structures. Stache is flat, like Stenstrom, et al.'s proposed COMA-F machine [42].

Recent research has begun focusing on supporting multiple paradigms in a single computer—by integrating shared memory and message passing—and allowing a compiler to select the model appropriate for a program or data structure [24]. Frank and Vernon proposed message-passing mechanisms for shared memory systems [12]. The MIT Alewife system handles some cache coherence events in software and allows shared-memory programs to send explicit messages. Their preliminary results show that some run-time operations, for example, task creation, are more efficiently implemented with explicit messages than shared memory [23]. The Stanford FLASH goes further and replaces a hard-wired directory controller with a programmable controller that implements shared memory by explicitly sending messages [15]. However, neither Alewife nor FLASH provides protected, user-level interfaces for these mechanisms, which limits programmers and compilers to a predefined set of system-provided policies.

## 8 Conclusions

This paper describes a new approach to designing parallel computers that is based on user-level software control of the shared address space. Previous systems have implemented shared address space policies in hardware, sometimes assisted by system-level software, which limits flexibility and performance. The Tempest interface provides the four primitives—fast messages, bulk data transfer, memory management, and fine-grain access control—that enable a compiler or run-time library to implement or customize communication and shared-memory operations efficiently.

To make these ideas concrete, we described Typhoon, an initial design of a system that implements the Tempest primitives at low hardware cost. Processing nodes in Typhoon are similar to those in existing message-passing computers, except for an additional Network Interface Processor (NP). The NP provides mechanisms that enable user-level code to respond quickly to incoming messages from the network and to cache misses or explicit requests from the node processor.

We demonstrated this approach with a detailed simulation of a new transparent shared-memory protocol based on a new memory allocation policy called Stache. Although this system runs at user-level, it outperforms a complex hardware directory protocol in the common case in which a node's working set exceeds its cache size and performs competitively in other cases. In addition, we demonstrated the benefits of user-level protocols by customizing the coherence protocol for the EM3D application, thereby improving its performance by 35%.

## Acknowledgments

This work is part of the Wisconsin Wind Tunnel project, which is co-led by Mark Hill, James Larus, and David Wood and funded by the National Science Foundation. We especially would like to thank Mark Hill for numerous discussions and suggestions and Alvy Lebeck for several key extensions to Fast-Cache. We would also like to thank David Culler for providing the EM3D application and J.P. Singh for supplying the SPLASH benchmarks. We would also like to thank Doug Burger, Satish Chandra, Trishul Chilimbi, Rahmat Hyder, Alvy Lebeck, Shubu Mukherjee, Brad Richards, Anne Rogers, and Guri Sohi for their suggestions on drafts of this paper.

## References

[1] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera Computer System. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 1–6, June 1990.

[2] Andrew W. Appel and Kai Li. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, April 1991.

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

[4] J.E. Barnes and P. Hut. A Hierarchical O(N log N) Force Calculation Algorithm. *Nature*, 324(4):446–449, December 1986.

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

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

[7] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing 93*, pages 262–273, November 1993.

[8] David E. Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain Parallelism with Minimal Hardware Support: A Compiler-Controlled Thread Abstract Machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 164–175, April 1991.

[9] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Nuth, Scott Wills, Paul Carrick, and Greg Flyer. The J-Machine: A Fine-Grain Concurrent Computer. In G. X. Ritter, editor, *Proc. Information Processing 89*. Elsevier North-Holland, Inc., 1989.

[10] William J. Dally, J. A. Stuart Fiske, John S. Keen, Richard A. Lethin, Michael D. Noakes, Peter R. Nuth, Roy E. Davison, and Gregory A. Flyer. The Message-Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro*, 12(2):23–39, April

1992.

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

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

[13] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative Evaluation of Latency Reducing and Tolerating Techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, June 1991.

[14] Erik Hagersten. Toward Scalable Cache Only Memory Architectures. Technical report, The Royal Institute of Technology Swedish Institute of Computer Science, October 1992. Stockholm, SwedenPh.D. Thesis, Swedish Institute of Computer Science Dissertation Series 08.

[15] John Hennesey. FLASH. ARPA HPC Software Fall PI Meeting, September 1993.

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

[17] High Performance Fortran Forum. High Performance Fortran Language Specification. Version 1.0, May 1993.

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

[19] Mark D. Hill and Alan Jay Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, C-38(12):1612–1630, December 1989.

[20] Intel Corporation. Paragon Technical Summary. Intel Supercomputer Systems Division, 1993.

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

[22] Charles Koelbel and Piyush Mehrotra. Compiling Global Name-Space Parallel Loops for Distributed Execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, October 1991.

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

[24] James R. Larus. Compiling for Shared-Memory and Message-Passing Computers. *ACM Letters on Programming Languages and Systems*, 1(4):?, December 1993. To appear.

[25] Alvin R. Lebeck and David A. Wood. Fast-Cache: A New Abstraction for Memory System Simulation. Univ. of Wisconsin Technical Report.

[26] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1992.

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

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

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

[30] Meiko World Inc. Computing Surface 2: Overview Documentation Set, 1993.

[31] Sun Microsystems. The SPARC Architecture Manual (Version 8), December 1990.

[32] David A. Moon. Symbolics Architecture. *IEEE Computer*, 20(1):43–52, January 1987.

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

[34] Gregory M. Papadopoulos and David E. Culler. Monsoon: An Explicit Token Store Architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 82–91, May 1990.

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

[36] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[37] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-Time Scheduling and Execution of Loops on Message Passing Machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[38] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-Time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, 40(5):603–612, May 1991.

[39] Jaswinder Pal Singh, Truman Joe, Anoop Gupta, and John L. Hennessy. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessor. In *Proceedings of Supercomputing 93*, pages 214–225, November 1993.

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

[41] Ellen Spertus, Seth Copen Goldstein, Klaus Erik Schauser, Thorsten von Eicken, David E. Culler, and William J. Dally. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 302–313, May 1993.

[42] Per Stenstrom, Truman Joe, and Anoop Gupta. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 80–91, 1992.

[43] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR Lisp Architecture. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 444–452, June 1986.

[44] Thinking Machines Corporation. The Connection Machine CM-5 Technical Summary, 1991.

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

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