To appear in the *Proceedings of the 23rd International Symposium on Computer Architecture (ISCA)*, 1996

Decoupled Hardware Support for Distributed Shared Memory

Steven K. Reinhardt, Robert W. Pfile, and David A. Wood

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

Abstract

This paper investigates hardware support for fine-grain distributed shared memory (DSM) in networks of workstations. To reduce design time and implementation cost relative to dedicated DSM systems, we decouple the functional hardware components of DSM support, allowing greater use of off-the-shelf devices.

We present two decoupled systems, Typhoon-0 and Typhoon-1. Typhoon-0 uses an off-the-shelf protocol processor and network interface; a custom access control device is the only DSM-specific hardware. To demonstrate the feasibility and simplicity of this access control device, we designed and built an FPGA-based version in under one year. Typhoon-1 also uses an off-the-shelf protocol processor, but integrates the network interface and access control devices for higher performance.

We compare the performance of the two decoupled systems with two integrated systems via simulation. For six benchmarks on 32 nodes, Typhoon-0 ranges from 30% to 309% slower than the best integrated system, while Typhoon-1 ranges from 13% to 132% slower. Four of the six benchmarks achieve speedups of 12 to 18 on Typhoon-0 and 15 to 26 on Typhoon-1, compared with 19 to 35 on the best integrated system. Two benchmarks are hampered by high communication overheads, but selectively replacing shared-memory operations with message passing provides speedups of at least 16 on both decoupled systems. These speedups indicate that decoupled designs can potentially provide a cost-effective alternative to complex high-end DSM systems.

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant F33615-94-1-1525 and ARPA order no. B550, NSF PYI Award CCR-9157366, NSF Grant MIP-9225097, an AT&T Graduate Fellowship, and donations from AT&T Bell Laboratories, Digital Equipment Corp., Sun Microsystems, Thinking Machines Corp., and Xerox Corp. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

1 Introduction

Technological and economic trends make it increasingly cost-effective to assemble large-scale parallel systems from off-the-shelf workstations and networks [2]. These "networks of workstations", or NOWs, primarily target message-passing applications. Although shared-memory applications can be run on these machines, the lack of explicit support for this model introduces significant overheads. This paper explores hardware support for cache-coherent distributed shared memory (DSM) on workstation-based systems. In keeping with the original motivation for networks of workstations, we focus on leveraging off-the-shelf components and minimizing the amount of custom, DSM-specific hardware.

In a DSM system, processors transparently access both local and remote memory using loads and stores. References to non-local data are completed by copying the data into a local cache and performing the reference on the copy. The functions required to implement DSM can be divided into three areas:

- *messaging* provides the inter-node communication required for parallel computation,
- *access control* detects memory references that require non-local action, e.g., a load of data for which there is no valid local copy, and
- *protocol processing* sequences messages and manipulates access control to satisfy processor memory references while maintaining a globally consistent image of memory.

Industry trends suggest that now, or in the near future, we can apply commodity components in two of these three areas. General-purpose microprocessors rapidly and flexibly sequence operations and manage data, making them suitable for protocol processing. Commercial network technologies such as switched Ethernet, ATM, and Myrinet [7], and research efforts such as SHRIMP [6], promise rapid advances in messaging performance. Access control is the only function for which there is no prospective off-the-shelf solution. We can leverage available components by decoupling DSM support—that is, by building simple access control hardware that co-exists with off-the-shelf processors and networks.

In contrast, dedicated DSM systems [1, 28, 13, 25, 33, 37] integrate all three functions in custom hardware. This hardware provides high performance, but requires complex custom chips that are costly both to design and to manufacture. At the other extreme, many systems use no DSM-specific hardware. These systems send messages over existing networks and do protocol processing in software on general-purpose CPUs—as do the decoupled systems we propose. However, they perform access control either entirely in software [4, 39, 23] or using standard virtual memory hardware [29, 10, 24]. The cost of avoiding custom hardware for access control is that the user must compromise on performance, programming model, or both.

This paper has two primary contributions: the design of two decoupled systems, Typhoon-0 and Typhoon-1, and the quantitative comparison of their (simulated) performance with two systems with integrated DSM support.

Both Typhoon-0 and Typhoon-1 rely on general-purpose CPUs for protocol processing. Typhoon-0 also uses a generic network; a custom access control module per node is its only DSM-specific hardware. To demonstrate the feasibility and relative simplicity of this access control device, two people implemented it in less than one year using two FPGAs and two SRAMs [34]. Typhoon-1 uses a similar access control device that also integrates the network interface. This integration improves performance by eliminating data movement through the protocol processor. Both devices use *cacheable control registers*, a novel technique that increases the efficiency of communication from bus devices to processors by leveraging the local bus coherence protocol.

To exploit the flexibility of software protocol processing, both decoupled designs run user-level protocol handlers and support the Tempest interface [35]. This allows the use of previously written application-specific protocols [14, 32], which selectively replace shared-memory operations with message passing operations to improve performance. Shadow spaces allow user-level handlers to directly convey protected addresses to the bus devices [6, 19, 43].

Using simulation, we quantify the performance impact of decoupling—that is, given subsystems with similar capabilities, what is the performance penalty for implementing those subsystems as separate components rather than integrating them into a single device? We compare the performance of Typhoon-0 and Typhoon-1 with two integrated systems. The first integrated system is an idealized Simple COMA [18] implementation that combines a network interface and access control with an infinitely fast hardwired protocol state machine. The second, modeled after Typhoon [37], replaces the state machine with a user-level protocol processor to allow execution of optimized application-specific protocols. All of these systems can be built from off-the-shelf workstations, unlike those that replace standard cache or memory controllers, such as Alewife [1], FLASH [13], and S3.mp [33].

In our evaluation, all of the systems use a single processor per node for computation, although small-scale (e.g., four-way) bus-based multiprocessor nodes may be more cost-effective. All of the designs we describe are compatible with multiprocessor nodes. The decoupled systems add a second general-purpose CPU dedicated to protocol processing. This configuration provides a direct comparison to the integrated systems, which also have dedicated protocol processing resources. However, in the long term, we expect that decoupled systems will dynamically schedule protocol processing along with computation across all of the processors in a multiprocessor node [15].

We find, not surprisingly, that the decoupled designs have significantly higher communication overheads. A simple remote miss takes roughly four times longer on Typhoon-0 than on either of the integrated systems. Typhoon-1's integration of access control and networking cuts this latency nearly in half, but is still twice as slow as the integrated designs. However, the net effect on overall performance is application dependent. Four of six benchmarks spend more than half of their time computing on the integrated systems, mitigating the impact of higher communication overheads. On 32 nodes, these benchmarks achieve speedups of 12 to 18 on Typhoon-0 and 15 to 26 on Typhoon-1, compared with 19 to 35 on the Simple COMA system. The other two benchmarks spend 30% or less of their time computing on the integrated systems, so higher overheads have a severe impact. However, application-specific protocols significantly improve their performance, providing speedups of 16 or more on both decoupled systems. These speedups, combined with the simplicity of the required hardware, indicate that decoupled designs have the potential to provide cost-effective parallelism [46] for shared-memory programs.

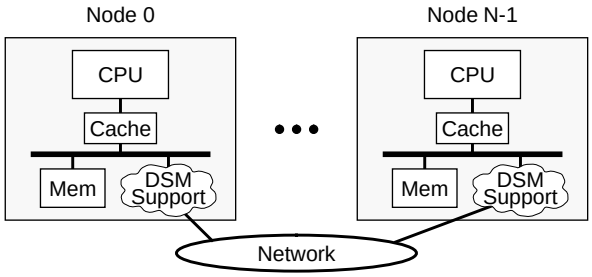

Figure 1. Common system organization.

In the next section, we describe the systems in more detail. Section 3 presents simulated performance results. Section 4 discusses related work, and Section 5 provides a summary and conclusions.

2 System descriptions

To study the performance impact of decoupling, we compare four systems that use the same base workstation and network technology, but differ in the level of integration of DSM support. The first part of this section discusses the common features of all four systems. The second subsection describes the integrated systems. The final subsection describes features common to both decoupled systems and then the systems themselves.

2.1 Common framework

Figure 1 depicts the common organization: a set of workstation-based nodes connected by a point-to-point network. The DSM support hardware (represented by the "cloud" in Figure 1) encompasses three components: access control, messaging, and protocol processing.

As in Simple COMA [18] and Typhoon [37], each node's local DRAM acts as a cache for remote data using a combination of virtual address translation and fine-grain access control. Virtual address translation directs remote data accesses to local physical memory. Accesses that require local memory allocation are detected and handled via page faults.

The systems maintain coherence at cache-block granularity (e.g., 32 to 128 bytes) using fine-grain access control. Each cache-block-sized piece of memory has an associated access tag, which may be *read-write*, *read-only*, or *invalid*. (These tags correspond, respectively, to the *exclusive*, *shared*, and *invalid* states found in most hardware caches.) Every processor memory access is checked against the access tag of the referenced block. A conflicting access (a read or write to an invalid block or a write to a read-only block) causes a *block access fault*, which suspends the access and invokes a coherence protocol action. These actions are performed by hardware on Simple COMA and by software handlers on the other systems. The protocol resumes the access once the conflict is resolved (e.g., after the remote data is fetched). Non-conflicting accesses proceed normally.

A snooping device on the memory bus enforces fine-grain access control using the signals intended for local bus-based coherence.[1] 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

1. We assume an ownership-based invalidation protocol with write-back caches. Other bus protocols do not necessarily preclude implementing access control via snooping, but they are less common and introduce some complications, so we do not address them here.

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 read-only 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 read-write to invalid) require a bus transaction to invalidate any copies that may be in the hardware caches.

Each node interfaces to the network through a pair of 64-bit–wide 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. In Typhoon, Typhoon-0, and Typhoon-1, the message queues are memory-mapped and directly accessible from user-level software via loads and stores, as in the CM-5. The queues are directly accessed by the Simple COMA system's hardware state machine. Though it is impossible to predict the commodity network interface of the future, we believe that it will provide a queue abstraction [8] and have roughly similar performance characteristics.

2.2 Integrated systems

To provide a reference point for the decoupled designs' performance, we study two systems that tightly integrate DSM support functions in a single device.

The first system is an idealized implementation of Simple COMA [18]. The network interface queues and access control snooping logic are tightly coupled with an infinitely fast hardwired state machine implementing a full-map invalidation-based coherence protocol. We assume this device has zero-cycle access to all protocol state information. Only network queue and memory bus interface delays are charged.

The second system, Typhoon [37], combines a network interface, access control logic, and a user-level protocol processor on a single device (see Figure 2). Dispatch hardware rapidly invokes user handlers in response to message arrivals and block access faults. By virtue of running on the integrated processor, these handlers have single-cycle access to the memory-mapped registers that manipulate access control and send and receive message data.

Typhoon implements access control using a "reverse translation lookaside buffer", or RTLB. The RTLB is an on-chip cache indexed by physical page number. Each entry contains the page's access tags, the corresponding virtual page number, and a pointer to the per-page protocol data structure. Each bus transaction is checked against the tags stored in the RTLB. On an access fault, the virtual page number is used to reverse translate the physical address from the bus to a virtual address for the user-level handler.

The Typhoon device also contains a block buffer and an independent block transfer unit to stage data between the network and memory. The buffer has address tags, like a cache, to accelerate the completion of remote misses. Arriving data is written into the buffer and the tag is set to match the memory address. Data requested by the compute processor (e.g., when it retries a faulting access) can be fetched from the buffer (as a cache-to-cache transfer) without waiting for the data to be written to memory. The block buffer also keeps data transfers from polluting the protocol processor's data cache.

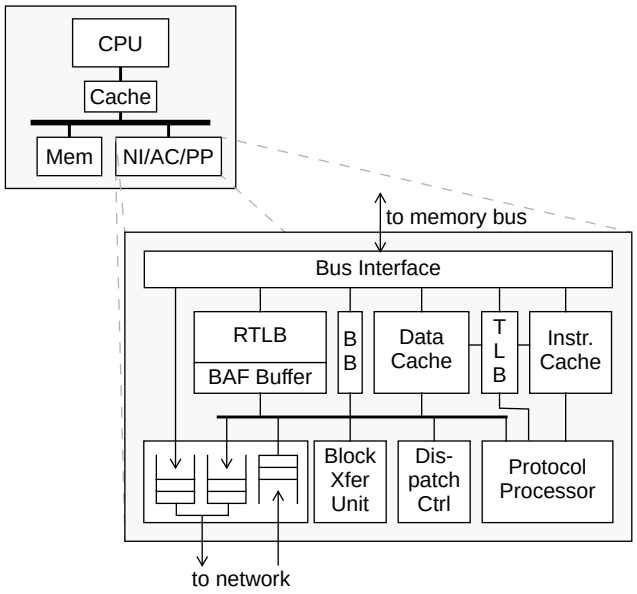

Figure 2. A Typhoon node, including a block diagram of the network interface/access control/protocol processor device. RTLB is the reverse TLB; BB is the block buffer; the BAF buffer holds information on block access faults. The second network send queue is for use by the compute processor(s).

2.3 Decoupled systems

Simple COMA and Typhoon rely on complex, highly integrated devices to provide high performance. This section presents two decoupled designs, Typhoon-0 and Typhoon-1, that sacrifice some of this performance to reduce design cost and complexity. Both perform protocol processing in software on general-purpose CPUs. Typhoon-0 also uses a generic network interface.

In this paper, we assume that Typhoon-0 and Typhoon-1 have a dedicated protocol processor per node, corresponding to the processor on the integrated Typhoon's network interface device. However, protocol processing can be dynamically scheduled along with computation across all of the CPUs in a node.

The down side of decoupling is its effect on performance. The components must communicate across the system bus, which is both slower than an on-chip interconnect and subject to contention. Both Typhoon-0 and Typhoon-1 use a novel technique, *cacheable control registers*, to efficiently transfer information across the bus. 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. A cacheable control register has two features:

- As long as the register's value does not change (and the block is not replaced), repeated accesses are satisfied in the processor's cache, reducing access latency and bus traffic. This allows a processor to efficiently poll the register. Because interrupts are expensive in modern CPUs, the protocol processor polls for events in both Typhoon-0 and Typhoon-1.
- An entire cache block of data is transferred in a single burst. If multiple words of data must be fetched from the device, a burst is much more efficient than a series of uncached loads, each requiring a separate bus transaction.

Another issue is the protected communication of addresses across the bus. Tempest's user-level protocol software needs to

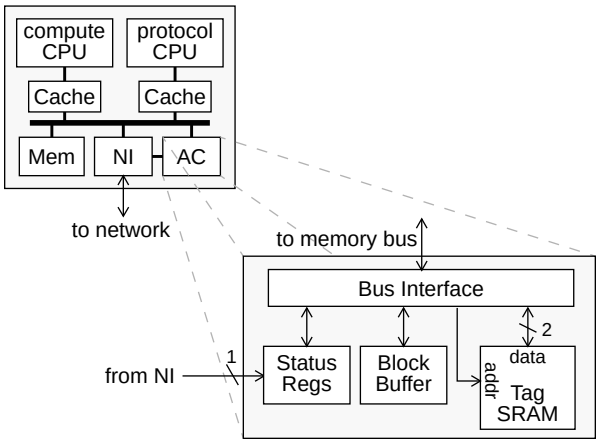

Figure 3. A Typhoon-0 node, including a block diagram of the access control device.

manipulate access control hardware using virtual addresses. However, passing virtual addresses as data requires a translation and a protection check in the receiving component. We avoid both of these using a *shadow space* [6, 19, 43]. The access control device supports a physical address range—the "shadow space"—as large as, and at a fixed offset from, the machine's physical memory. Accesses to the shadow space are interpreted as operations on the real memory space. A user process can directly access a shadow space page if and only if it has permission to manipulate the corresponding physical memory page.

2.3.1 Typhoon-0

Typhoon-0 separates DSM support functions across three components: a protocol processor, a network interface, and a fine-grain access control device. The first two components are off-the-shelf devices, while the third is custom hardware. To demonstrate the feasibility of the access control device, we implemented it for a cluster of SPARCstation 20 workstations [34]. The device is a standard-sized MBus module (approximately 3.3" by 5.8") containing two Altera 81188-2A FPGAs (each containing 12,000 "usable gates"), two 4M by 1 static RAMs, and a few miscellaneous parts. Each workstation has two ROSS HyperSPARC CPUs; by software convention, one acts as the compute processor and the other acts as the protocol processor. The nodes in our Typhoon-0 implementation are connected by a Myricom Myrinet network [7].

Figure 3 shows a block diagram of a Typhoon-0 node. Access tags are implemented as a two-bit directory with one tag for every block in physical memory. The bus logic observes every coherent bus transaction, and indexes the tag SRAM with the physical address. Depending on the tag value and the access type, the bus logic may intervene in the transaction, as described in Section 2.1. The Typhoon-0 card only provides fine-grain access control; it does not duplicate the full functionality of the Typhoon RTLB. Reverse translation and per-page state lookup are performed in software using an inverted page table structure.

Typhoon-0 provides protected user-level tag access using a shadow space, as described above. A read from the shadow space returns the corresponding tag value. A block's tag is modified by writing the new tag value to the shadow space. When access to a block is downgraded (e.g., from read-write to invalid), Typhoon-0 issues a read-invalidate bus operation to invalidate any cached copies and retrieve the current version, which may be in a processor's cache. The block buffer (shown in Figure 3) stores the data returned by the read-invalidate. The block buffer is a cacheable control register, so a processor can read its contents in a single burst.

Because data must be copied explicitly to and from the network interface, every shared-memory data transfer passes through the protocol processor twice—once on the sender and once on the receiver. Even for network interfaces that support DMA, like the Myrinet in our actual implementation, cache block transfers typically are too small to amortize setup overheads.

A protocol action that combines a data transfer and a block access change must be sequenced carefully. For example, to send a writable (potentially modified) block to another node, the protocol processor changes the tag—causing a read-invalidate from the access control device—then copies the contents of the block buffer to the network interface. The protocol processor cannot directly access the block after the tag has changed because Typhoon-0 must enforce access tags for both CPUs to avoid illegal bus protocol states. The system cannot send the block data before the tag is changed, or writes made by the compute processor after the send but before the tag change would be lost.

A similar situation arises when a message arrives containing data for a previously invalid block. The message handler cannot directly write the data without changing the block's access tag, but changing the tag first would create a race where another thread could access the old contents. Instead, the message handler writes the data via an uncached alias that bypasses the access tag check. Once memory is updated, the tag can be upgraded.

Typhoon-0 supports dispatch of access fault and message handlers using a cacheable control register called the *dispatch register.* The dispatch register combines a user-specified base address with status information to form a program counter [21]. The status portion of the PC forms an index into a code table, much like a processor trap vector table. The protocol processor polls for events by performing an indirect jump to the PC location. If no events are pending, the code table entry simply returns to the polling loop. The cacheable nature of the register allows efficient polling when the status changes infrequently. The burst transfer capability of the cacheable register is used on access faults to transfer the physical address of the block on which the fault occurred, the access type (read or write), and the block's tag value in a single bus operation. This data is formatted to accelerate fault handling. For example, the physical page number is in a separate word and is pre-shifted to form an index into the inverted page table. Only eight instructions are required to determine the virtual address of the faulting block and select and invoke a handler specific to the page and fault type.[1]

The Typhoon-0 dispatch register signals message arrivals as well as access faults. The message status is controlled by an external signal, shown in Figure 3, that is connected to the network interface. The network interface's message arrival interrupt signal (if any) can be routed to the Typhoon-0 card. (The SPARCstation implementation of Typhoon-0 uses a status LED signal from the Myrinet network interface.) The integration of access fault and message status allows a single poll on a (cacheable) value to check for all possible protocol events. Unlike Typhoon, Typhoon-0's dispatch logic only notifies the processor of the existence of a message; the NI must explicitly be accessed to determine the address of the message handler to invoke.

2.3.2 Typhoon-1

Typhoon-1 combines access control and the network interface on a single device, as shown in Figure 4. Typhoon-1 significantly improves performance over Typhoon-0 without a large increase in complexity. No major new control or datapath features are added

1. This total comprises two loads to the cacheable dispatch register, an indirect jump on the dispatch PC, two loads to read the 128-bit inverted page table entry, one load to read the protocol handler PC, an indirect call to the handler, and an OR to combine the virtual page number with the page offset. Assuming cache hits, this sequence takes ten cycles on the HyperSPARC, which has a one cycle load-use delay.

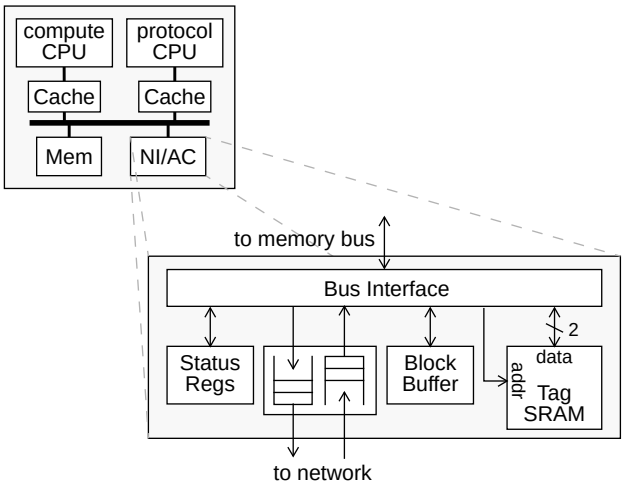

Figure 4. A Typhoon-1 node, including a block diagram of the network interface/access control device.

other than the network interface itself, which could be a separate single-chip device such as the Myrinet LANai [7], Dolphin/LSI Logic SCI NodeChip [30], or Cray SCX adapter [40]. Typhoon-1 incorporates three specific advances over Typhoon-0: user-level cache block DMA with combined access control, a tagged block buffer, and enhanced message dispatching.

First, Typhoon-1 uses Typhoon-0's shadow space to support user-level DMA at cache-block granularity so that data transfers need not pass through the protocol processor. In effect, Typhoon-0 interprets a write to the shadow space as a command to change a block's access tag; Typhoon-1 simply adds a few new commands. One of these commands copies the corresponding memory block to the message send queue; another effectively copies data from the message receive queue to the memory block. An additional set of commands atomically combine an access tag modification with block DMA to or from the network queues. These commands eliminate the awkward copies Typhoon-0 must perform when invalidating a modified block or receiving data for a previously invalid block. Since Typhoon-0 can become a bus master and issue burst transactions, these additional features do not significantly increase complexity.

The shadow space mapping solves the address translation and protection issues usually associated with user-level DMA [5]. Because the DMA size is limited to a single cache block, there is only a small window during which the virtual-to-physical translation must remain valid, so we can use a simple technique to prevent the DMA mechanism from using a stale translation [19]. Typhoon-1 provides a single status bit that indicates that a DMA is outstanding; the operating system temporarily disables DMA initiation and waits for this bit to clear before invalidating a translation that was potentially used for DMA.

A second enhancement, borrowed from the integrated Typhoon, is the addition of address tags and snooping to the block buffer. Copies from the network to memory are performed by moving the data into the block buffer and tagging it with the destination physical address; the new data will be provided on a future access to that block as a cache-to-cache transfer. The data is also written to memory in the background, but unlike Typhoon-0 this copy is not on the critical path of a remote miss.

Typhoon-1's third major improvement over Typhoon-0 is greater support for message handler dispatch. When a message arrives, the entire message header is transferred in the dispatch register block, saving several uncached loads per message receive.

3 Performance evaluation

In this section, we use simulation to compare the performance of our four designs. First we describe the simulation parameters and methodology, then we present results for a simple microbenchmark and a set of macrobenchmarks.

We simulate 32-node systems. Each node has a 200 MHz dual-issue SPARC processor. We assume a perfect instruction cache and a 1 MB direct-mapped data cache with 64-byte address blocks and 32-byte subblocks. The instruction latencies, issue rules, and memory hierarchy are modeled after the Ross HyperSPARC [38, 41].

The processor(s), memory, access control and/or network interface devices within each node are connected by a 50 MHz MBus. The MBus is a 64-bit, multiplexed address/data bus that maintains coherence on 32-byte blocks using a MOESI protocol [42]. On a cache miss, the critical word is returned from main memory 140 ns (seven bus cycles or 28 processor cycles) after the request is issued on the MBus. Miss detection, processor/bus clock synchronization, and bus arbitration add 11-14 processor cycles to the total miss latency. Bus occupancy, contention, and arbitration are fully simulated.

To isolate the effects of decoupling, Typhoon-0, Typhoon-1, and the integrated Typhoon are given the same protocol-processing resource—a dedicated CPU, identical to the compute CPU. Although this assumption results in a more controlled experiment, it diverges from expected practice in two ways. First, the symmetric dual-processor nodes of the decoupled designs may be more efficiently used by dynamically scheduling protocol handlers and computation across both processors. Second, the design effort required for an actual Typhoon implementation would likely result in an integrated protocol processor that is a generation or more behind the compute processor.

Timing parameters for the Typhoon-0 and Typhoon-1 access control devices are taken from our FPGA-based Typhoon-0 implementation. The devices are clocked at bus speed, 50 Mhz. Tag and control register accesses take three and four bus cycles, respectively. For reads to cacheable control registers, the first data word is returned in three bus cycles and additional words are returned on every second cycle.

Our simulated Typhoon-0's network interface is an independent MBus device, similar to the CM-5 NI, with a message arrival signal that feeds the access control device's dispatch register. Register access delays are set to match our measured results from the CM-5: seven bus cycles for reads and three for writes.

The integrated Typhoon's embedded protocol processor is identical to the primary CPU, as discussed above. This processor has single-cycle memory-mapped access to all on-chip control registers and the network interface queues. Up to 64 bits of data per cycle can be transferred between the network interface queues and the data cache or block buffer. We assume an infinite RTLB.

The Simple COMA controller processes each access fault or message 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 our simulator, messages observe an additional cycle of pipelined latency between arrival and processing.

To emphasize the performance impact of DSM support, we assume an aggressive network latency of 100 processor cycles (500 ns) from the injection of the tail at the sending network interface to the arrival of the head at the receiving interface. Although dedicated MPP interconnects may surpass this speed, current off-the-shelf networks are typically slower by an order of magnitude

or more. Contention occurs at network inputs and outputs, but not internally. We assume that the network is reliable.

End-to-end flow control is enforced by the network interface. Each node may have at most four messages outstanding to each other node. The NI generates an acknowledgment as each message is read by the protocol processor. We avoid deadlock by buffering blocked messages at the sending node and injecting them as acknowledgments arrive. The Simple COMA controller has an infinite send buffer and can inject a buffered message one cycle after an acknowledgment is received. On the Typhoon systems, the run-time library queues blocked messages in the sender's address space and sets an NI mode bit that causes acknowledgments to invoke a software handler in the same manner as protocol messages. This handler sends queued messages and clears the mode bit when the queue is empty.

Fine-grain access control is performed as described in Section 2.1. On a block access fault, the access control device inhibits the memory controller and gives the requesting processor a "relinquish and retry" response, forcing the processor to re-arbitrate for the bus. The access control device masks the arbiter to keep the processor off the bus until the access can be completed [28]. While this may be difficult to implement on existing systems, its performance is representative of emerging systems which support deferred responses, either explicitly (like the Intel P6 [17]) or using a split-transaction bus. (Unfortunately, our SPARCstation 20 implementation must generate a bus error; the kernel trap vector is modified to spin on a flag which is set when the data arrives.)

Results were obtained using a detailed execution-driven discrete-event simulator. The cache, MBus, and device simulation are detailed enough that they were used for initial design verification of the Typhoon-0 implementation. Actual SPARC binaries are rewritten (using a tool based on EEL [27]) to replace memory accesses with calls to the simulator and to add instrumentation to count instruction execution cycles. All software protocols were written in C and compiled and linked with the simulated benchmarks. Full application results were obtained by simulating the nodes of a system in parallel on a Thinking Machines CM-5 using a conservative, synchronous parallel simulation algorithm based on the Wisconsin Wind Tunnel [36].

3.1 Micro-evaluation

To gain insight into the overheads of these systems, we trace a single remote read miss and break down the latency into its components. We assume a cache page has been previously allocated on the caching node and the block is 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 read-write to read-only 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 read-only, and signals the compute processor to retry the access.

The results are presented in Table 1. Our common system assumptions lead to a minimum latency of 299 processor cycles. The home node latency includes two bus cycles (eight processor cycles) to request and acquire the bus and ten 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 seven bus cycles (28 processor cycles) to fetch the critical word and three 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 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 [20].[1]

	Step	S-COMA	Typhoon	Typhoon-1	Typhoon-0
Caching node	detect L1 cache miss, issue bus transaction	10	10	10	10
	detect access fault, dispatch handler	0	6	101	101
	get fault state	0	16	18	18
	send msg	0	13	45	45
	request msg latency	100	100	100	100
Home node	dispatch msg handler	1	6	78	159
	read msg	0	3	7	40
	directory lookup, branch	0	20	20	20
	send msg header	0	17	38	52
	fetch data, change tag, send	48	48	122	293
	response msg latency	100	100	100	100
Caching node	dispatch msg handler	1	6	78	159
	read msg header	0	3	7	40
	read msg data, change access tag	0	12	20	261
	unmask CPU, reissue bus transaction	10	10	32	32
	fetch data, resume	31	31	31	31
Total	200 MHz CPU cycles	301	401	807	1461
	50 MHz bus cycles	76	101	202	366
	bus transactions	3	3	16	36

Table 1. Breakdown of remote miss latency. Values are 200 MHz processor cycle counts except where noted.

Because these fundamental latencies dominate, Typhoon takes only 33% longer to satisfy the miss despite the cost of running software handlers. The decoupled 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.

We also timed this remote miss on our Typhoon-0 implementation. The results cannot be directly compared with the simulation because the current platform has slower processors (66 MHz rather than 200 MHz) and a much slower network (a Myricom Myrinet with the interface on the 25 MHz SBus I/O bus). However, we can determine the implementation's coherence overhead by subtracting the total miss latency—72 µs—from the round trip time for sending a short request and a receiving a 32-byte reply—67 µs on our messaging layer. This leaves 5 µs, or 330 processor cycles, for the block access fault detection and handler dispatch on the caching node, the tag downgrade on the home, and the tag upgrade and CPU restart back on the caching node—a value not out of line with those presented in Table 1.

3.2 Macro-evaluation

To determine how these overheads translate into application performance, we simulated the six shared-memory benchmarks listed in Table 2. *Appbt* is from the NAS parallel benchmark suite

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

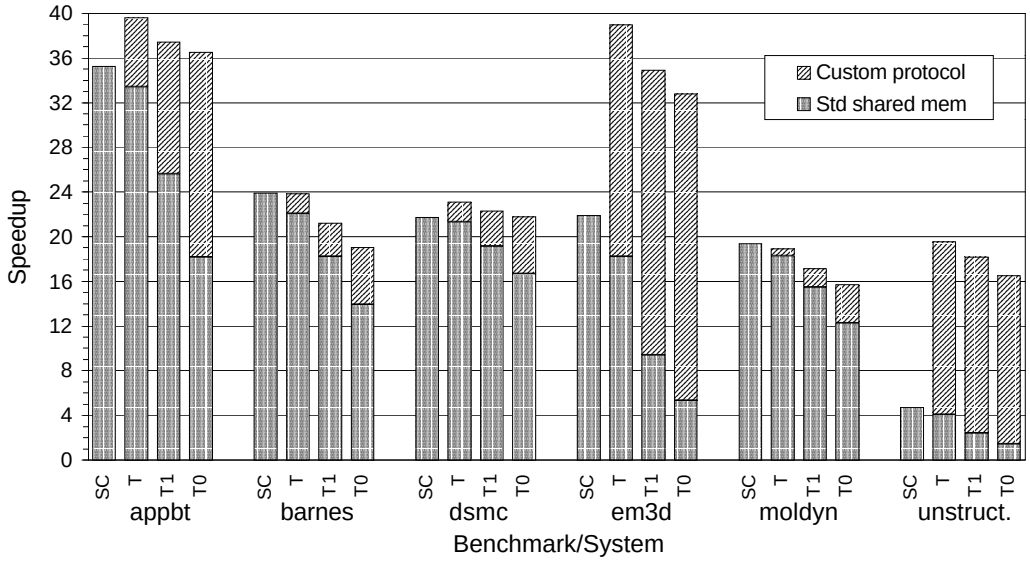

Figure 5. Speedups on 32-node systems.
SC=Simple COMA, T=Typhoon, T1=Typhoon-1, T0=Typhoon-0.

Benchmark	**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
unstructured	CFD	static mesh	9428 nodes, 59863 edges, 5864 faces, 5 iterations

Table 2. Benchmark descriptions and inputs.

[3], parallelized for shared memory [9]. *Barnes* is from the SPLASH-2 suite [44]. *Em3d* is a shared-memory version of an original Split-C program from Berkeley [12]. *Dsmc*, *moldyn*, and *unstructured* are irregular applications originally from Maryland [32]. All of the benchmarks are written in C and compiled with gcc version 2.6.3 at optimization level -O2.

All of the benchmarks except *em3d* use a first-touch migrate-once scheme [31] to improve the assignment of pages to home nodes. The first node to touch a page after the parallel phase of the program begins becomes the page's home for the remainder of the execution. This scheme is simple to implement and guarantees that every page is assigned to one of the nodes that references it. *Em3d* explicitly allocates the graph so that writes are always to local pages.

Although we simulate the full application, we focus on the portion of execution where a production version will spend most of its time by measuring only the second and following computation iterations. For most of the applications, iteration times are very regular, so we can get meaningful results with only a few iterations. There are two exceptions. In *moldyn*, the molecule interaction list is occasionally rebuilt, 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 at first the number of particles increases with each iteration. It is impractical to simulate far enough to reach steady state, so 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.

Figure 5 shows speedups for 32-node systems relative to our best sequential version on a workstation identical to one node of the parallel system. The shorter, shaded bars indicate the speedup for the unmodified shared-memory applications. For the Tempest systems, this merely requires linking with the standard protocol library, which implements the same sequentially consistent full-map invalidation protocol used in the Simple COMA system. The unmodified benchmarks achieve speedups of 19 or better on the Simple COMA system, with the exception of *unstructured* at under five. (The large speedups for *appbt* and *em3d* with the custom protocol are due to cache and TLB effects.)

We ran all of the benchmark/system combinations for block sizes of 32, 64, 128, and 256 bytes. For the larger block sizes, every inter-node coherence action involves multiple 32-byte MBus blocks. Due to space restrictions, we only present results for 64-byte coherence blocks. The 64-byte block size is within 5% of the best performance for most cases. The only exceptions are *em3d* on Typhoon-1, which does 20% better with a 256-byte block size, and *unstructured*, for which each of the systems does 10-20% better at 128 or 256 bytes.

For each of the Tempest systems, we also ran existing custom protocols that were hand-optimized for each application [14, 32]. In general, these protocols use the programmer's knowledge of sharing and synchronization patterns to send explicit update messages for critical data in the computation loop. Speedups for these versions are presented as the taller, hatched bars in Figure 5. These protocols were written and optimized for a very different system—Blizzard-E [39] on the CM-5—with much slower processors and even higher relative overheads. Although their impact is reduced by the lower overheads of these hardware-assisted systems, all of the protocols still provide some improvement over the default

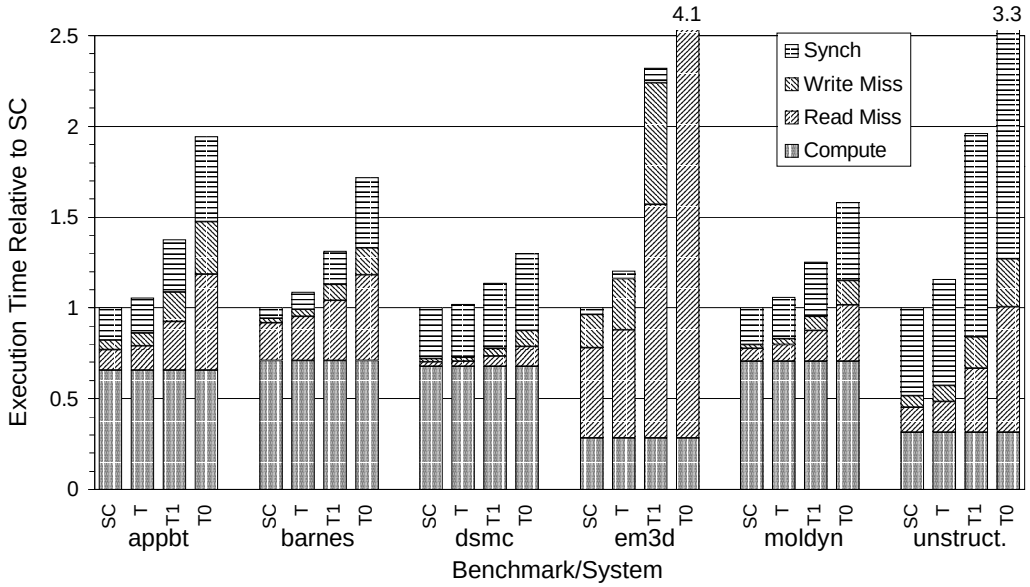

Figure 6. Execution time breakdown for standard shared memory applications.
SC=Simple COMA, T=Typhoon, T1=Typhoon-1, T0=Typhoon-0.

shared memory. Two show dramatic improvement even on Typhoon—a factor of two for *em3d* and almost five for *unstructured*—causing them to outperform the Simple COMA system by nearly the same margins as well. Moving to Typhoon-1 and Typhoon-0, the higher overheads leave greater room for improvement, so the more efficient protocols have a greater impact. Only for *moldyn* and *barnes* do the custom protocols on each of the Tempest platforms fail to outperform standard shared memory on any system—including Simple COMA. Of course, prefetching and weak consistency could improve performance on all of the platforms.

To facilitate a direct comparison of the systems, Figure 6 presents execution times for the standard shared-memory benchmarks normalized to the Simple COMA system for each benchmark. We break out the time spent on read, write, and synchronization stalls. Nearly all of the remaining time is computation, so we label that segment "compute", even though it includes some factors (such as TLB misses) that are negligible in all of these cases.

For every application, the total stall time increases significantly as we move from Simple COMA to the decoupled designs—by 92% to 525% for Typhoon-0 and by 41% to 181% for Typhoon-1. However, the effect of this increase on bottom-line performance varies according to the contribution of the stall times to the overall execution.

We can roughly divide the benchmarks into groups according to the fraction of time spent on computation in the Simple COMA system. *Appbt*, *barnes*, *dsmc*, and *moldyn* form the first group, spending 65% or more of the Simple COMA execution in computation. For these benchmarks, the effect of increased overheads is mitigated by their smaller overall contribution. Typhoon-0 is at most 94% slower than Simple COMA (71% excluding *appbt*), and Typhoon-1 is at most 38% slower. In the second group, *em3d* and *unstructured*, less than 32% of the Simple COMA execution is spent computing. Here, the decoupled designs show their weakness, turning in performance a factor of two or more slower than Simple COMA. *Unstructured* fails to produce much speedup for any platform. *Em3d*'s speedup stems primarily from cache and TLB effects: the data set is over 20 MB, so the uniprocessor execution spends over 90% of its time waiting for cache and TLB misses.

This grouping of the benchmarks also predicts the effectiveness of the application-specific protocols. Intuitively, the applications with higher overheads have more to gain by eliminating those overheads. As mentioned above, *em3d* and *unstructured* show impressive gains, while the improvements for *dsmc* and *moldyn* *barnes* are smaller. *Appbt* straddles the fence: the customized protocol gains a factor of 2 on Typhoon-0 but only 18% on Typhoon.

The application-specific protocols also serve to diminish the performance difference between the various Tempest implementations. Typhoon-0 is 22% to 71% slower than Typhoon for the standard shared memory benchmarks, but only 6% to 17% slower for the custom protocols. Similarly, Typhoon-1's worst-case performance disadvantage is reduced from 48% to 11%. 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 only improves performance on the receiving node.

4 Related work

Alewife [1] was the first hybrid hardware/software DSM system. As with later variants from other researchers [22, 45, 16], custom hardware generates requests and handles responses on the caching side and implements some basic directory functions. In all of these designs, a single device integrates this hardware control with the cache and/or memory controllers and the network interface. To avoid deadlock, the CPU must handle interrupts while memory accesses are outstanding, precluding some existing off-the-shelf microprocessors.

FLASH [13], StarT-NG [11], and Typhoon [37] perform all protocol processing—on both the directory and the caching nodes—in software. FLASH and Typhoon execute protocol soft-

ware on a custom processor integrated with the network interface; FLASH also incorporates the memory controller on this device. Like Typhoon-0 and Typhoon-1, StarT-NG uses a commodity CPU as a protocol processor, although StarT-NG places the network interface on the CPU's level 2 cache bus.

Typhoon-0 and Typhoon-1 use virtual address translation to map remote pages into local DRAM, a feature shared with Typhoon, Simple COMA [18], and page-based software DSM systems [29, 10, 24]. Because cached remote data is transparently accessed, protocol handlers are only executed when coherence action is required. In contrast, StarT-NG runs a software handler for every remote reference that misses in the hardware cache, even if the data is cached in local DRAM; FLASH must execute software on every local hardware cache miss.

Kontothanassis and Scott [26] propose using network interfaces such as SHRIMP [6] to implement weakly consistent page-based DSM. Rather than changing the coherence model to suit a specific style of network interface, we propose simple hardware—either separate from or integrated with the network interface—to support both traditional fine-grain coherence and application-specific protocols.

5 Summary and conclusions

This paper explores the performance impact of decoupling DSM support functions—usually integrated in high-performance designs—and implementing them using off-the-shelf components. Decoupling leads to simpler systems and a shorter design cycle.

We present two decoupled designs, Typhoon-0 and Typhoon-1. Typhoon-0 combines an off-the-shelf processor and network interface with a simple access-control device. A new technique, cacheable control registers, provides efficient polling and data transfer. A shadow space allows the processor to communicate addresses to bus devices in a protected manner. To demonstrate the feasibility and simplicity of the access control device, two people completed an FPGA-based implementation in less than one year. Typhoon-1 improves on Typhoon-0's performance by integrating the network interface with the access control device. Typhoon-1 leverages Typhoon-0's cacheable control registers and shadow space for messaging without greatly increasing complexity.

Although decoupling significantly increases communication overheads, the impact is mitigated on applications that spend most of their time computing. On 32 nodes, four of our six benchmarks achieve speedups of 12 to 18 on Typhoon-0 and 15 to 26 on Typhoon-1, compared with 19 to 35 on the Simple COMA system. At the other extreme, overheads dominate some applications, including two of our benchmarks. In these cases, application-specific protocols reduce the overheads resulting in speedups of at least 16 on the decoupled systems.

We demonstrate that decoupled designs greatly simplify DSM hardware by avoiding protocol state machines and integrated processors, yet provide significant speedups on unmodified shared-memory applications—within a factor of four, and usually within a factor of two, for our benchmarks. These results indicate that decoupled hardware support for DSM can potentially provide a cost-effective alternative to complex integrated systems.

Acknowledgments

The Typhoon-0 implementation was made possible through the generosity of Andreas Nowatzyk and Sun Microsystems and by a donation from the Altera Corporation. Babak Falsafi and Shubu Mukherjee contributed to the development of the simulator used in this paper. Mark Hill and Jim Larus provided valuable comments on drafts of this paper.

References

[1] Anant Agarwal, Ricardo Bianchini, David Chaiken, Kirk L. Johnson, David Kranz, John Kubiatowicz, Beng-Hong Lim, Ken Mackenzie, and Donald Yeung. The MIT Alewife Machine: Architecture and Performance. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 2–13, June 1995.

[2] Thomas E. Anderson, David E. Culler, David A. Patterson, and the NOW team. A Case For NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[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] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway Distributed Shared Memory System. In *COMPCON 1993*, 1993.

[5] Matthias A. Blumrich, Cesary Dubnicki, Edward W. Felten, and Kai Li. Protected, User-level DMA for the SHRIMP Network Interface. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture (HPCA)*, January 1996.

[6] Matthias A. Blumrich, Kai Li, Richard Alpert, Cezary Dubnicki, Edward W. Felten, and Jonathon Sandberg. Virtual Memory Mapped Network Interface for the SHRIMP Multicomputer. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 142–153, April 1994.

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

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

[9] Doug Burger and Sanjay Mehta. Parallelizing Appbt for a Shared-Memory Multiprocessor. Technical Report 1286, Computer Sciences Department, University of Wisconsin–Madison, September 1985.

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

[11] Derek Chiou, Boon S. Ang, Arvind, Michael J. Beckerle, Andy Boughton, Robert Greiner, James E. Hicks, and James C. Hoe. StarT-NG: Delivering Seamless Parallel Computing. Technical Report CSG Memo 371, MIT Laboratory for Computer Science, February 1995.

[12] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Steven Lumetta, Thorsten von Eicken, and Katherine Yelick. Parallel Programming in Split-C. In *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

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

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

[15] Babak Falsafi and David A. Wood. When does Dedicated Protocol Processing Make Sense? Technical Report 1302, Computer Sciences Department, University of Wisconsin–Madison, February 1996.

[16] Hakan Grahn and Per Stenstrom. Efficient Strategies for Software-Only Directory Protocols in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 38–47, June 1995.

[17] Linley Gwennap. Intel's P6 Bus Designed for Multiprocessing. *Microprocessor Report*, 9(7), May 30, 1995.

[18] Erik Hagersten, Ashley Saulsbury, and Anders Landin. Simple COMA Node Implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, January 1994.

[19] John Heinlein, Kourosh Gharachorloo, Scott A. Dresser, and Anoop 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, October 1994.

[20] Mark Heinrich, Jeffrey Kuskin, David Ofelt, John Heinlein, Joel Baxter, Jaswinder Pal Singh, Richard Simoni, Kourosh Gharachorloo, David Nakahira, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John 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.

[21] 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.

[22] 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. An earlier version appeared in ASPLOS V.

[23] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. Technical Report LCS-TM-517, MIT Laboratory for Computer Science, March 1995.

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

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

[26] Leonidas I. Kontothanassis and Michael L. Scott. Software Cache Coherence for Large Scale Multiprocessors. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture (HPCA)*, pages 286–295, January 1995.

[27] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, June 1995.

[28] 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.

[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] LSI Logic Inc. *L64601 SCI NodeChip Technical Manual*.

[31] Michael Marchetti, Leonidas Kontothanassis, Ricardo Bianchini, and Michael L. Scott. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. In *Ninth International Parallel Processing Symposium*, April 1995.

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

[33] 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.

[34] Robert W. Pfile. Typhoon-Zero Implementation: The Vortex Module. Technical Report 1290, Computer Sciences Department, University of Wisconsin–Madison, October 1995.

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

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

[38] ROSS Technology Inc. *SPARC RISC User's Guide*, September 1993.

[39] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 297–306, October 1994.

[40] Steve Scott. The SCX Channel: A New, Supercomputer-Class System Interconnect. Hot Interconnects III, August 1995.

[41] Doug Shore. Personal communication, November 1994.

[42] Sun Microsystems Inc. *SPARC MBus Interface Specification*, April 1991.

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

[44] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[45] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993. Also appeared in CMG Transactions, Spring 1994.

[46] David A. Wood and Mark D. Hill. Cost-Effective Parallel Computing. *IEEE Computer*, 28(2):69–72, February 1995.