Mechanisms for Distributed Shared Memory

by

Steven K. Reinhardt

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN—MADISON

1996

Abstract

Distributed shared memory (DSM) systems simplify the task of writing distributed-memory parallel programs by automating data distribution and communication. Unfortunately, DSM systems control memory and communication using fixed policies, even when programmers or compilers could manage these resources more efficiently.

This thesis proposes a new approach that lets users efficiently manage communication and memory on DSM systems. Systems provide primitive DSM mechanisms without binding them to fixed protocols (policies). Standard shared-memory programs use default protocols similar to those found in current DSM machines. Unlike current systems, these protocols are implemented in unprivileged software. Programmers and compilers are free to modify or replace them with optimized custom protocols that manage memory and communication directly and efficiently.

To explore this new approach, this thesis:

- identifies a set of mechanisms for distributed shared memory,
- develops *Tempest*, a portable programming interface for mechanism-based DSM systems,
- describes *Stache*, a protocol that uses Tempest to implement a standard shared-memory model,
- summarizes custom protocols developed for six shared-memory applications,
- designs and simulates three systems—*Typhoon, Typhoon-1,* and *Typhoon-0*—that support Tempest, and
- describes a working hardware prototype of Typhoon-0, the simplest of those designs.

Tempest combines fine-grain coherence support, an active message model, and virtual-memory–based page allocation to provide portability across a range of platforms.

Typhoon, Typhoon-1, and Typhoon-0 support Tempest using different levels of custom hardware integration. Typhoon achieves high performance by integrating key components on one device. Typhoon-1 and Typhoon-0 use off-the-shelf parts for some of these components, trading some performance for simpler designs.

Typhoon demonstrates that mechanism-based DSM systems can compete with hardwired-protocol systems on unmodified shared-memory applications (within 25% across six benchmarks). Despite Typhoon's low overheads, custom protocols improve performance significantly for some applications—by 384% for one benchmark. Results for Typhoon-1 and Typhoon-0 on unmodified applications are varied, but custom protocols bring them within 13% and 47% of Typhoon, respectively.

A working Typhoon-0 prototype demonstrates the feasibility of these designs. Measurements of the prototype's performance substantiate simulator projections.

Acknowledgments

I cannot possibly thank all who have contributed to making my time in Madison so enjoyable and successful. Nonetheless, I will try.

David Wood, my advisor, provided guidance, encouragement, and raw material in just the right amounts. Mark Hill and Jim Larus, the other members of the Wisconsin Wind Tunnel triumvirate, provided ideas, opinions, support, and invaluable advice on both the technical and non-technical aspects of research. It has been a privilege to start on the ground floor and watch the WWT group grow in size and stature under their leadership.

Guri Sohi did his best to keep me honest in his role as the loyal opposition. Thanks to David, Mark, Jim, Guri, and Jim Smith for serving on my thesis committee. Jim Smith and Guri provided comments from an external perspective that greatly improved the presentation of this thesis. Thanks also to Jim Goodman for his interest in my work.

Thanks to all the other students in the Wisconsin Wind Tunnel group and in the larger UW computer architecture community for their support and camaraderie. In particular, Alvy Lebeck provided friendship, counsel and occasional commiseration over countless lunches and coffee breaks. Babak Falsafi had the dubious pleasure of hacking on a lot of my code; I thank him for not complaining any more than he did. Rob Pfile designed Vortex, the key component of the Typhoon-0 prototype. That system is a reality due to his dedication. Alvy, Babak, Shubu Mukherjee, and Yannis Schoinas helped to develop and maintain a lot of critical software including simulators, Tempest support components, and the custom protocols for the benchmarks used in this thesis.

I also thank AT&T Bell Labs and the Wisconsin Alumni Research Foundation for their generous fellowship awards.

Diana, my wife, set aside her personal goals to follow me to Wisconsin, and ended up raising three children while I completed my degree. I cannot thank her enough for her love, support, patience, and friendship. Thanks also to our children—Ziggy, Anastasia, and Sophia—for giving me extra motivation to come home and (almost) always being glad to see me when I arrived. I also thank our parents for their generosity and support.

Neither Diana nor I could have survived without lots of help and prayers from our church family, especially Danean & Dan Kirchner and Kelly & Ken Jensen.

Most of all, I must give thanks and praise to Jesus Christ. Through faith in him, I am reconciled with the one true God of the universe—a fact that makes earthly trials and achievements pale in comparison.

Oh, the depth of the riches of the wisdom and knowledge of God!
How unsearchable his judgments, and his paths beyond tracing out!
"Who has known the mind of the Lord? Or who has been his counselor?"
"Who has ever given to God, that God should repay him?"
For from him and through him and to him are all things.
To him be the glory forever! Amen.
Romans 11:33–36 (NIV)

Contents

Abstract i

Acknowledgments iii

List of Figures viii

List of Tables x

Chapter 1. A New Approach to Distributed Shared Memory 1

1.1 Mechanisms for distributed shared memory 5

1.2 The Tempest interface 9

1.3 Hardware support for Tempest 9

1.4 Thesis organization 11

Chapter 2. Mechanisms for Distributed Shared Memory 12

2.1 Fundamental mechanisms 13

2.2 The Tempest interface 15

2.2.1 Messaging 16

2.2.2 Local storage management 18

2.2.3 Access control 19

2.2.4 Mechanism interactions 21

2.3 Transparent shared memory using Tempest 22

2.4 Optimizing applications using Tempest 27

2.4.1 EM3D 29

2.4.2 Appbt 32

2.4.3 Barnes 33

2.4.4 DSMC 34
2.4.5 Moldyn 35
2.4.6 Unstructured 36
2.5 Related work 38
2.6 Summary 41

Chapter 3. Hardware Support for Tempest 43
3.1 Common features 47
3.1.1 Messaging 47
3.1.2 Local storage management 48
3.1.3 Fine-grain access control 49
3.2 Typhoon: integrated hardware support for Tempest 50
3.2.1 Protocol processor 50
3.2.2 Access control (RTLB) 51
3.2.3 Block transfer unit and block buffer 53
3.2.4 Handler dispatch 54
3.2.5 Primary CPU access 54
3.3 Decoupled hardware support for Tempest 55
3.3.1 Cacheable control registers 56
3.3.2 Handler dispatch 57
3.3.3 Access control 58
3.3.4 Typhoon-1 60
3.3.5 Typhoon-0 63
3.4 Performance 65
3.4.1 Microbenchmark 69
3.4.2 Macrobenchmarks 71
3.4.3 Impact of network latency 78
3.4.4 Impact of protocol-processing performance 79
3.5 Related work 83

3.6 Summary 86

Chapter 4. The Typhoon-0 Prototype 89

4.1 Hardware 90

4.2 Software 92

4.2.1 Vortex configuration 93

4.2.2 Process setup 93

4.2.3 Other Solaris modifications 95

4.2.4 System issues 96

4.2.5 Messaging 98

4.3 Performance 99

4.3.1 Microbenchmarks 99

4.3.2 Macrobenchmarks 102

4.4 Comparison of measured and simulated results 103

4.5 Summary 106

Chapter 5. Conclusion 107

5.1 Summary 108

5.2 Future directions 110

References 113

Appendix A. Tempest Interface Specification 124

Appendix B. Access Control Via Bus Snooping 141

List of Figures

Figure 1-1. Distributed-memory system organization. 2
Figure 2-1. DSM memory access flowchart. 14
Figure 2-2. Stache example, step 1 24
Figure 2-3. Stache example, step 2 25
Figure 2-4. Stache example, step 3 26
Figure 2-5. Example EM3D bipartite graph. 29
Figure 2-6. EM3D program fragment. 30
Figure 3-1. Common system organization. 45
Figure 3-2. Logical components of Tempest support. 46
Figure 3-3. Component integration diagram. 46
Figure 3-4. Shadow tag space example. 50
Figure 3-5. Typhoon node, including a block diagram of the network interface/ access control/protocol processor device. 51
Figure 3-6. Typhoon RTLB diagram. 52
Figure 3-7. Dispatch program counter format. 57
Figure 3-8. Dispatch register layout for block access faults. 61
Figure 3-9. Dispatch code for block access faults. 61
Figure 3-10. Typhoon-1 node, ...62
Figure 3-11. Typhoon-0 node, ...64
Figure 3-12. Application speedups for transparent shared memory on 32-node systems. 73
Figure 3-13. Execution time breakdown for transparent shared memory. 74
Figure 3-14. Application speedups on 32-node systems, including application-specific protocols. 75

Figure 3-15. Execution time breakdown for the application-specific protocols, normalized to transparent shared memory on Simple COMA. 77
Figure 3-16. Speedups for appbt, Barnes, and DSMC on 32-node systems at various network latencies. 80
Figure 3-17. Speedups for EM3D, moldyn, and unstructured on 32-node systems at various network latencies. 81
Figure 3-18. Execution time for transparent shared memory on Typhoon, varying the protocol processor speed. 82
Figure 3-19. Execution time for the application-specific protocols on Typhoon, varying the protocol processor speed. 83
Figure 4-1. Block diagram of a Typhoon-0/COW node. 90
Figure 4-2. Application speedups on 16 nodes of the prototype system. 102
Figure 4-3. Simulated and measured speedups for Appbt 105
Figure 4-4. Simulated and measured speedups for Barnes 105

List of Tables

Table 3.1: Remote miss latency breakdown for simulated systems. 70
Table 3.2: Benchmark applications and data sets. 71
Table 4.1: Remote miss latency breakdown for the Typhoon-0 prototype. 101
Table A.1: Access tag semantics. 125
Table A.2: Block tag change enumeration values (type `TPPI_BlkTagChange`). . . .133
Table B.1: Bus monitor snooping behavior. 142

Chapter 1

A New Approach to Distributed Shared Memory

Modern microprocessors provide a tremendous amount of computing power for a relatively small price. Currently, the fastest microprocessors execute up to four instructions every two nanoseconds—a peak rate of two billion instructions per second—yet sell for $3,000 or less [Mic96] and can be had in complete systems for under $10,000 [Eno96]. It is difficult and costly to reach higher performance levels by building faster processors: the cheapest non-microprocessor–based system that is faster by any measure is the half-million–dollar Cray J90 [McC96, Cra96]. With few exceptions, the most cost-effective approach to higher performance is to gang multiple microprocessors together.

Nearly all current multi-microprocessor systems organize their processors and memory using one of two methods. The simplest approach uses a bus—a common set of wires—to join the processors and memory [Bel85]. However, competition for use of the shared bus limits these *bus-based* systems to at most a few tens of processors. Larger systems avoid the bus bottleneck by grouping processors and memory into nodes—each one essentially a bus-based uni- or multiprocessor system itself—which communicate via a point-to-point messaging network (see Figure 1-1). The size of these *distributed-memory* systems is lim-

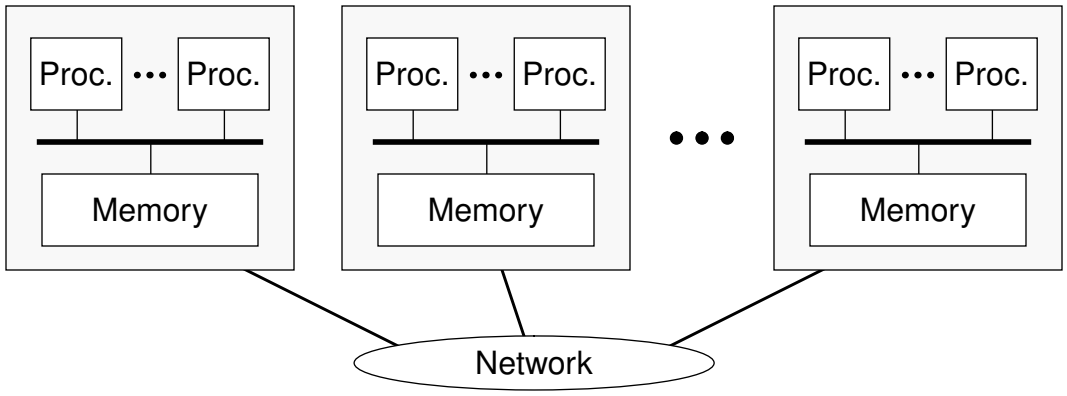

Figure 1-1. Distributed-memory system organization.

ited primarily by cost: Intel is scheduled to deliver a machine with 9,000 processors and 262 gigabytes of memory by the end of this year (1996)—at a cost to the U.S. government of $46 million [Int95].

Although distributed memory allows the construction of larger machines, it introduces two new tasks: distributing data among the nodes and communicating across the network when processors need data from other nodes. To achieve high performance—the primary motivation for building a large machine—these tasks must be performed well. Efficient data distribution and communication—storing data on the node or nodes where it is used and overlapping network delays with useful work—improves performance by reducing or eliminating the time processors wait for data to cross the network. Two prevalent types of distributed-memory machines—*message passing* and *distributed shared memory*—handle these tasks in markedly different ways.

Message-passing systems leave data distribution and communication entirely to software. Programs specify explicitly on which node each datum is located and when each message is sent. If programmers and compilers can predict precisely where and when each datum is used, they can exploit this control to use memory and network resources efficiently. However, less predictable programs require potentially costly run-time software to

locate data and process messages dynamically. In addition, a message-passing program's data distribution and communication patterns affect its structure and correctness substantially. Programmers who use message passing directly must deal with these issues up front, and decisions made early in development often are not easily reversible.

In contrast, distributed shared memory (DSM) systems distribute and communicate data automatically. These systems provide the abstraction of one global, uniformly fast memory. Programs access data by referencing locations in this global memory space. Systems transparently fetch data, regardless of its physical location, to satisfy these references. At the same time, these systems replicate and migrate data dynamically across the nodes to keep values near the processors that reference them. Because the shared-memory abstraction matches the model that arises naturally on bus-based multiprocessors, DSM systems take advantage of programs written for—and programmers trained on—the much more common bus-based machines. Programmers can develop correct, working programs without considering data distribution or communication.

Unfortunately, DSM machines hide the underlying distributed-memory organization too well at times. Although programs that are naive about data distribution and communication will function correctly on a DSM system, they may not perform well. Programmers must reason about (or compilers must deduce) a program's memory and communication behavior to improve its performance. However, once a program's behavior is understood, DSM systems provide no standard, direct techniques for implementing efficient data distributions or communication protocols. Instead, programmers and compilers must attempt to effect these optimizations indirectly by modifying the program's reference pattern.

This thesis proposes a new approach that lets users efficiently manage communication and memory on DSM systems. This approach is based on separating DSM *mechanisms*—the low-level operations that make DSM feasible—from the *policies* that govern their use [Bri70, LCC+75, Wul81]. Systems implement primitive memory and communication mechanisms sufficient to support distributed shared memory—mechanisms present in

some form in most DSM machines. Unlike existing systems, which bury these mechanisms underneath fixed policies, these new mechanism-based systems let ordinary software access these mechanisms directly. For standard shared-memory programs, software libraries provide universal policies, similar to those implemented in current DSM systems. However, because these policies are ordinary software—just like any other program component—programmers and compilers may modify or replace them, exploiting program-specific knowledge to optimize data distribution and communication. This mechanism-based approach maintains DSM's convenient model and dynamic data management while enabling the potential efficiencies of message passing's explicit resource control.

To make this proposal concrete, I describe a programming interface for mechanism-based DSM systems called *Tempest*. Tempest strikes a balance between portability and performance by combining virtual-memory page mapping—as used by page-based software DSM systems [LH89]—with the fine-grain coherence found in hardware DSM machines. This balance makes Tempest a suitable interface for all-software implementations on generic message-passing machines [SFL+94, SFH+96] as well as higher-performance hardware-accelerated systems [RLW94, RPW96]. Applications and programming tools written to the Tempest interface run unmodified across this wide range of systems. I describe *Stache*, a software library that implements application-transparent shared memory—similar to that provided by a typical DSM system—using Tempest. I also summarize six shared-memory applications which programmers have optimized by accessing Tempest's mechanisms directly [FLR+94, MSH+95].

An interface is a contract between the users and the implementors of a system, so it must carefully consider both perspectives. On the implementation side, this thesis examines Tempest systems that can be constructed by adding custom hardware to networks of off-the-shelf workstations. I describe three designs that explore the relationship between the level of custom hardware integration and system performance. The systems proposed here demonstrate that Tempest is amenable to a range of custom hardware acceleration techniques. Simulations show that the design with the most highly integrated Tempest sup-

port—called *Typhoon*—performs at a level comparable to a hardware-controlled DSM system. I also describe a prototype implementation of the simplest custom-hardware design—known as *Typhoon-0*—that demonstrates the feasibility of this approach and provides a real-world system for benchmarking.

The following section discusses the mechanism-based approach to distributed shared memory further. The remaining sections describe the two main contributions of this thesis—the Tempest interface and the design and analysis of hardware support for Tempest—in more detail.

1.1 Mechanisms for distributed shared memory

To understand the role of mechanisms in distributed shared memory, consider the operation of typical DSM systems. These systems approximate the behavior of a uniform shared memory by caching data stored on other nodes (*remote* data). When a processor in such a system accesses remote memory, the system copies the requested data, plus some neighboring values—a data *block*—to a portion of local memory called a *cache*. The system then satisfies future accesses to that remote data block using the local cache copy, avoiding further network traversals. Because most programs exhibit locality—that is, once a processor accesses a memory location, it is likely to access that location or others near it shortly—the local cache handles most memory accesses, regardless of the data's original location. As memory values change, *coherence protocols* ensure that processors do not access out-of-date cache copies. These protocols track the nodes that copy each data block and send messages to invalidate or update those copies when a block is modified.

Because programmers communicate with the system only through memory accesses (loads and stores), they must relinquish message passing's direct control over memory and communication to gain shared memory's ease of use. This lack of control forces them to forgo potential optimizations [KJA+93, Lar94]. For example, a programmer may know (or a compiler may deduce) that a value written on node A will be read next on node B. In this case, sending the value directly from A to B is almost certainly more efficient than relying

on a DSM system's coherence protocol. Even without exact information, a programmer or compiler may be able to supply useful hints. For example, if a processor is not likely to access a memory region again, the system can avoid coherence overhead by not caching the data—while still behaving correctly if the hint is wrong. Finally, some operations, such as synchronization, are expressed more naturally and efficiently in terms of messages.

This thesis proposes that systems give users direct control over memory and communication by providing access to the same primitive mechanisms employed by the coherence protocol. Systems that provide primitive mechanisms can avoid the needless run-time generality of a one-size-fits-all solution [Wul81]. For DSM systems, this one-size-fits-all solution is the coherence protocol. By providing access to a complete set of mechanisms, programmers and compilers are free to modify or replace protocols arbitrarily. Libraries provide standard, universal protocols, such as those found in current DSM systems.

Mechanisms exist at numerous levels; for example, fetching remote data is a mechanism that relies on lower-level mechanisms, including sending a request message and receiving a reply message. To maximize generality and flexibility, this thesis seeks to identify the lowest-level mechanisms that make efficient distributed shared memory feasible. I identify three mechanisms that underlie nearly all DSM systems: *messaging*, *local storage management*, and *memory access control*. Explicit messaging provides direct control over network communication. Local storage management lets users control each node's memory contents by associating global addresses with local memory locations. Memory access control detects accesses that require coherence action by marking locations invalid, read-only, or writable.

Although these mechanisms are very general, this thesis focuses on their use in optimizing shared-memory application performance. Specifically, programmers tune performance-critical data structures by replacing the standard coherence protocol with application-specific *custom protocols*. These programmer-developed protocols rely on knowledge of the application's communication patterns to improve performance, typically

by sending data directly to its consumer. Performance improvements of an order of magnitude have been observed from custom protocols [FLR+94]. Although this thesis reports on manual optimizations only, automatic optimizations based on static program analysis [Lar94] or programmer annotations [RAK89, CBZ91, HLRW93] are a promising approach to achieving similar efficiency with reduced programmer effort. Other researchers are investigating tools to aid custom protocol development [CRL96] as well as other applications of these mechanisms—for example, language-specific shared-memory support [LRV94].

In contrast with the mechanism-based approach proposed here, other DSM system designers have dealt with this issue through a combination of two techniques: extending the shared-memory interface and implementing low-overhead, low-latency DSM hardware. Mechanism-based DSM is less restrictive than specific interface extensions—enabling a large variety of higher-level programming interfaces—and provides an alternative to building aggressive DSM hardware.

Many systems extend the shared-memory interface—adding operations beyond simple loads and stores—to provide a channel for software to communicate additional information. These operations include uncached accesses, prefetch instructions [CKP91, MG91], writes that update outstanding data copies [RSW+93, LLG+92], atomic read-modify-write operations [GGK+83], and explicit synchronization operations [GVW89]. Weak consistency models [AH90, GLL+90] modify the semantics of ordinary loads and stores to allow more reordering—and hence more overlap—among memory accesses. These models require additional operations that force a partial order between accesses to achieve meaningful memory semantics. At a higher level, the Munin software DSM system [CBZ91] lets programmers annotate data structures with expected reference patterns. The system uses these annotations to select from a set of coherence protocols.

In theory, a DSM system that provides a complete set of universal mechanisms can support any or all of these extensions. Ordinary software composes the available mechanisms

to support the desired operation(s). Simple extensions such as prefetch may be implemented as a library or inline function; more complex interfaces may involve compiler support as well. In practice, the specific mechanisms provided by a DSM system, and their run-time cost, determine the feasibility and efficiency of a particular operation on that machine.

A second approach to dealing with shared-memory inefficiencies—complementary to extending the interface—is to minimize the impact of these inefficiencies by building aggressive hardware. If communication overhead is low, many applications can achieve good performance within a standard shared-memory model. Simple interface extensions such as prefetching enable good performance for more communication-intensive programs. Systems such as MIT's Alewife [ABC+95] and Stanford's DASH [LLG+92] and FLASH [KOH+94] employ significant amounts of custom hardware to reduce coherence protocol overheads. Although this hardware may increase the number of bus-based multiprocessor applications that perform well without modification, the expense of designing and manufacturing these systems may limit their affordability.

Due to the overhead of sequencing mechanisms in software, mechanism-based systems have a performance disadvantage relative to these hardware-protocol DSM systems on demanding, unmodified shared-memory applications. However, simulations of the Typhoon design (see Section 1.3) indicate that high-end mechanism-based systems can compete with hardware DSM systems on moderately demanding standard shared-memory applications—and have the potential to outperform them significantly when custom protocols are used to optimize communication-intensive applications. The mechanism-based approach also enables effective shared-memory performance on less expensive hardware platforms, as demonstrated by software-only implementations described elsewhere [SFL+94, SFH+96] and the less integrated hardware designs described in Section 1.3.

1.2 The Tempest interface

Tempest provides a concrete, portable interface to the three DSM mechanisms identified in the previous section: messaging, local storage management, and memory access control. Tempest's messaging borrows from von Eicken's Active Messages [vECGS92]. Standard virtual address translation mechanisms are used for local storage management, as in software DSM systems [LH89]. The most innovative aspect of Tempest is its specification of *fine-grain* access control, a feature that enables fine-grain coherence and provides scalability to high-performance systems.

Tempest allows a range of implementations so that optimized applications and optimizing compilers need not be re-implemented for every platform. Platform-specific mechanisms are less likely to inspire developers to invest the effort to exploit them, since their effort will be wasted if they move to a different platform. Tempest implementations also span a wide cost/performance range. Low-cost implementations are key to making an interface widely available, which in turn generates a supply of trained programmers, a body of applications, and a market for development tools such as compilers. At the same time, higher-cost, higher-performance implementations must also exist both to support the most demanding applications and to reassure low-end users that their software investment is not wasted if their computational needs increase.

1.3 Hardware support for Tempest

The second focus of this thesis is the design and evaluation of hardware support for Tempest. Existing software-only Tempest implementations [SFL+94, SFH+96]—not covered in this thesis—demonstrate the potential of Tempest for low-end systems. The goals of studying hardware support are:

- to demonstrate that Tempest scales up incrementally, through a range of higher-cost, higher-performance solutions, to the point where it is competitive with high-end, dedicated distributed shared memory systems;

- to identify techniques that can be used to accelerate Tempest at various cost/performance points; and
- to quantify the performance of these alternatives for shared-memory applications, with and without Tempest optimizations.

I describe three distributed-memory Tempest designs—*Typhoon*, *Typhoon-1*, and *Typhoon-0*—and compare their performance using simulation. I also describe a prototype hardware implementation of the Typhoon-0 system that demonstrates the feasibility of the hardware fine-grain access control approach used in all three systems.

All three designs use cost-effective off-the-shelf workstations for the processor–memory nodes. Each design adds three logical components to every node: a snooping access control device, a network interface, and a protocol processor. The systems differ in the level of custom hardware integration used to implement these components.

- Typhoon integrates all three components on a single device. This integration lets protocol software efficiently interact with the access control and messaging mechanisms, resulting in high performance—within 25% of a comparable hardwired-protocol system on a set of six unmodified shared-memory benchmarks.
- Typhoon-1 integrates the network interface with access control, but leaves protocol processing to an off-the-shelf processor. Relative to Typhoon, this lower level of integration reduces design complexity, and (potentially) manufacturing cost, at the expense of some performance—11% to 222% on the unmodified benchmarks.
- Typhoon-0 uses a custom access control device with an off-the-shelf protocol processor and an off-the-shelf network interface. Typhoon-0 has the least complexity and lowest performance of the three systems. On the unmodified benchmarks, Typhoon-0 ranges from 28% to 427% slower than Typhoon.

The simulation results also show the effectiveness of Tempest-based custom protocols. Even on Typhoon, with its low-overhead support for standard shared memory, custom protocols sped up one of the benchmarks by 384% and another by 86%. With the custom pro-

tocols, Typhoon's performance ranges from 2% worse to four times better than the hardwired-protocol system.

Custom protocols provide even greater benefits on systems with higher overheads. Although Typhoon-1 and Typhoon-0 can be significantly slower than Typhoon for standard shared memory—a factor of two and four, respectively, on the most demanding benchmark—custom protocols reduce the differences to 13% and 47%. Custom protocols also provide more robust application performance in the face of other increased overheads, such as larger network latencies and, for Typhoon, slower embedded processors.

A prototype hardware implementation of Typhoon-0 demonstrates the feasibility of these designs and provides a real-world system for benchmarking. Despite a high-overhead commercial messaging network, five of the six benchmarks achieve better than 58% efficiency on sixteen nodes of the prototype. Application-specific protocols are critical to achieving this efficiency in three of the five cases. The prototype's measured performance agrees substantially with simulator projections.

1.4 Thesis organization

Chapter 2 motivates the fundamental shared-memory mechanisms, then describes the Tempest interface and its use for both application-transparent and custom shared-memory protocols. Chapter 3 discusses hardware support for Tempest, with a detailed descriptions and a simulation-based performance comparison of Typhoon, Typhoon-1, and Typhoon-0. Chapter 4 focuses on the Typhoon-0 hardware prototype, including measured performance results. Chapter 5 concludes with a summary and future directions for this work.

Chapter 2

Mechanisms for Distributed Shared Memory

This thesis proposes a new, flexible approach to distributed shared memory. Systems provide primitive shared-memory mechanisms without prescribing specific policies (e.g., cache coherence protocols). Programmers and compilers can combine these mechanisms arbitrarily to optimize existing protocols and to implement new ones.

This chapter begins by examining the abstract mechanisms required for distributed shared memory (DSM). Three mechanisms—messaging, local storage management, and access control—underlie nearly all DSM systems. Section 2.2 describes Tempest, a concrete, portable interface to these mechanisms. Tempest uses a variant of Active Messages [vECGS92] for messaging. Virtual address translation provides local storage management. Tempest's most innovative feature is *fine-grain* access control. The next two sections show how Tempest can be used, first to provide application-transparent shared memory (Section 2.3), then to optimize the performance of shared-memory applications using custom protocols (Section 2.4). The chapter closes with a discussion of related work (Section 2.5) and a summary (Section 2.6).

2.1 Fundamental mechanisms

This section identifies three abstract mechanisms—messaging, local storage management, and access control—required to support shared-memory programming models on distributed-memory machines. Mechanisms are the separable components of a process or system. Policies are the rules that govern how mechanisms are used to achieve a desired result. The division of a process into mechanisms can be done at several levels; a mechanism at one level may be built from a lower-level set of mechanisms and policies. To maximize generality and flexibility, this work seeks to identify the lowest-level mechanisms that make efficient distributed shared memory feasible.

These mechanisms assume a typical distributed-memory parallel machine, comprising a set of nodes connected by a messaging network (see Figure 1-1 on page 2). Each node contains one or more processors that share a memory module and an interface to the network. Hardware maintains coherence among processor caches within a node. Memory accesses are performed only on the node where they are issued—that is, remote data references are performed on a copy of the data in local main memory or a hardware cache.[1]

Every distributed-memory machine incorporates a **messaging** mechanism—that is, a way for nodes can communicate through the network—even if messaging is not directly available to users. To understand the remaining DSM mechanisms, consider the process of performing a shared-memory access. The flowchart in Figure 2-1 summarizes this process. In the figure, unshaded objects represent mechanisms and shaded objects represent policies.

As the first step in performing an access, a **local storage management** mechanism finds the local memory, if any, allocated to the shared-memory address. If there is no local

1. This model excludes systems which may perform individual loads and stores on a node other than the one on which they are issued. Examples include systems that use write-through or write-update caching of remote memory [BR90, WHL92, WJI+94, IDFL96] and those that migrate threads to data on misses [CR95]. Support for these models can conflict with some common implementation features—for example, writeback processor caches.

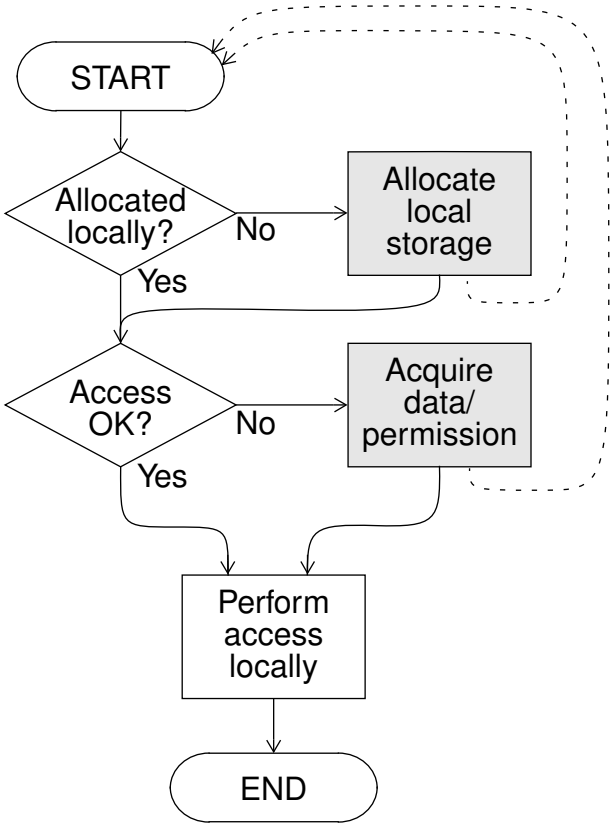

Figure 2-1. DSM memory access flowchart. If either the allocation check or the access check fails, the system must take action before the reference can be performed. The solid arrows indicate that the reference is suspended and later resumed without repeating the allocation and access checks. An alternate approach is to abort and retry the reference, as indicated by the dotted arrows.

memory for the referenced address, the mechanism invokes an allocation policy. This policy selects the local memory to allocate, possibly evicting previously cached data to make room. The policy interacts with local storage management to set up and tear down mappings from addresses to local memory. It also uses messaging to send evicted data to other nodes when necessary.

Once local memory is allocated, an **access control** mechanism determines whether that memory contains a valid copy of the data. If not, the mechanism invokes an access policy to acquire up-to-date data or additional access permissions. (A copy's validity may depend on the type of the access; in most protocols, replicated copies are valid for reading but not

for writing.) The access policy includes both the global allocation policy, which selects the node to which a particular request is sent, and the coherence protocol, which determines how requests are handled. This policy uses both messaging and access control. For example, the policy may send messages telling other nodes to mark their copies invalid before it lets the local node write its copy.

In contrast to earlier work [Cha94, WCF+93], I do not consider operations internal to the coherence protocol (e.g., recording pointers to the nodes sharing a block) as mechanisms. This thesis seeks to identify *fundamental* mechanisms—mechanisms that cannot be synthesized efficiently. Stanford's FLASH system [KOH+94, HKO+94] and the Typhoon design described in Chapter 3 demonstrate that software protocols are not inconsistent with high-performance systems.

After the allocation and access control checks succeed, the system performs the access on the local copy.

Messaging, local storage management, and access control are three mechanisms that are fundamental to the operation of practically any DSM system. To allow flexible, efficient policies, systems must present these mechanisms to users in a practical fashion. The next section describes an interface designed to meet that need.

2.2 The Tempest interface

Tempest is a concrete interface to the three abstract mechanisms identified in the previous section. One of Tempest's primary goals is portability across a wide cost/performance range. At the low-cost end, Tempest allows all-software implementations on generic message-passing hardware [SFL+94]. At the same time, the interface is amenable to hardware support. Chapter 3 describes a range of designs that provide hardware acceleration for Tempest; the most aggressive (Typhoon) is competitive in performance with dedicated distributed shared memory systems.

The following sections describe the significant features of Tempest. (See Appendix A for a more complete interface specification.) Each of the abstract mechanisms is examined in turn. For messaging, Tempest borrows from the Active Messages model [vECGS92]. Standard virtual memory hardware supports local storage management. The most innovative aspect of Tempest is *fine-grain* access control, which is provided using a tagged memory model. The final section discusses mechanism interactions.

2.2.1 Messaging

Most systems designed for explicit user message passing perform best on large, statically scheduled transfers. The implicit messaging behavior of DSM systems differs in the following ways:

- Messages are short: some contain only control information (e.g., a request for data), while others consist of control information plus a small amount of data (e.g., the response to a request). Assuming 64-bit addresses and 128-byte data transfers, we can estimate the former at 16-24 bytes, whereas the latter may be around 150 bytes.
- Most messages are handled asynchronously at the receiver. That is, requests for data and coherence operations are driven by the dynamic execution of other nodes, and are unrelated to the execution on the receiving node.
- The control information in messages originates in processor registers on the sender and is consumed in registers on the receiver.
- Low latency is important for high performance. Several techniques exist to tolerate memory latency—such as multithreading, prefetching, and non-blocking loads—but none of these is capable of hiding the latencies of most modern networks.

Tempest bases its messaging model on Active Messages [vECGS92], a flexible, low-overhead messaging model designed for fine-grain message-based systems. The header of each message contains a pointer to a function that is invoked at the receiving node to handle the message arrival. These message handlers are invoked asynchronously with respect to the main thread of computation at the receiver. Each handler is responsible for dispos-

ing of the remainder of the message. To reduce scheduling and synchronization overheads, each handler is run to completion before another message handler is started.

Deadlock is an issue in any messaging system. The network buffers message data as it is routed to its destination. A node cannot send a message if there is no buffer space available in the network. A node may not simply wait for space to become available without removing any of the messages destined for it, because a cycle of nodes waiting for each other to remove their messages will deadlock. Because each Tempest (or Active Messages) message handler must complete before another handler is allowed to run, there is the potential for deadlock whenever a message is sent from within a message handler. Active Messages avoids deadlock by restricting handlers to sending at most one reply, and guaranteeing buffer space for that one message. However, this solution is awkward for DSM cache coherence protocols, in which a handler may send out multiple messages to update or invalidate all the sharers of a particular memory block. To avoid placing unnecessary burden on the user, Tempest places no restrictions on sending messages from within message handlers. In cases where these sends overflow the network, the runtime system buffers message data in the user's virtual memory. A truly errant process will be terminated when it exhausts the memory available to it. A similar approach is taken in the Fugu system [MKAK94].

In addition to the modified Active Messages model, Tempest includes a separate messaging interface for bulk data transfer. This alternate interface complements Tempest's Active Messages by providing asynchronous, high-bandwidth memory-to-memory communication for large amounts of data. Users set up *virtual channels* that bind a sending node with a buffer on the receiver.[1] The sender can transmit directly to the receiver's buffer without explicitly invoking a handler on the receiver. The sender can use the channel repeatedly without reestablishing it. Although this bulk interface can be implemented

1. This interface is modeled after the virtual channels facility in the CMMD library on the Thinking Machines CM-5, which similarly complements CMMD's CM Active Messages.

on top of Active Messages, it is included in the specification to give system developers the freedom to optimize the implementation—for example, by employing DMA hardware.

2.2.2 Local storage management

The second mechanism, local storage management, maps shared-memory accesses to local storage locations and invokes a software handler to allocate and map local storage if an unmapped address is referenced. The virtual address translation hardware found in most processors is suitable for this task, a fact that has been exploited in numerous DSM systems [LH89, CBZ91, BZS93, KDCZ94, HSL94]. Shared virtual addresses are mapped to local physical memory; a reference to an unallocated location results in a page fault.

Using virtual address translation for the local storage management mechanism has several advantages. Support is ubiquitous in modern processors, so portability is guaranteed. Capacity is bounded only by the available local memory. Associativity is practically unlimited. As long as the application's working set fits in the processor's TLB, the per-reference overhead is very low. There are two potential drawbacks. First, the cost of taking a page fault is typically large [ALBL91]. However, most of this overhead is due to operating system structure, so it can be greatly reduced at the cost of modifying the OS [RFW93, TL94]. Second, the virtual memory page size is large enough (typically four to eight kilobytes) to make fragmentation a concern. In cases where this occurs, the only real solution is to modify the application to improve its data layout or its reference pattern. Applications that suffer from fragmentation are also likely to suffer from poor TLB locality, so this type of optimization will pay off in both of these areas. (Table 3.2 on page 71 reports the observed fragmentation overhead for six applications.) The drawbacks are usually outweighed by the advantages: because allocation should be necessary for only a small fraction of references, the per-reference overhead typically has the greatest impact on performance.

The Tempest interface specifies functions to allocate and map, unmap, and remap pages, and to install a user function as the page fault handler.

2.2.3 Access control

Access control is the mechanism that filters loads and stores, determining for each one whether action is required to obtain access rights to the referenced data. This mechanism can be viewed as a function that returns a single bit (action vs. no action) based on two inputs: (1) an access tag associated with the referenced address and (2) whether the access is a load or a store. Tempest provides three access tags: Invalid (action on both loads and stores), ReadOnly (action on stores but not loads), and Writable (no action on either loads or stores).[1] All three tags are necessary to support common multiple-reader, single-writer protocols. To reduce tag space overhead, a single access tag is associated with a memory *block* (an aligned, contiguous group of bytes). A reference that invokes an action is said to cause a *block access fault*. The thread that issued the reference is suspended, and a user function—the *block access fault handler*—is invoked. The block access fault handler initiates a protocol sequence that should culminate in acquiring access rights for the block, updating the access tag to allow the reference, and resuming the suspended thread.

Although only three access tags are required, implementations are unlikely to allocate fewer than two bits per block to encode these states. To exploit this extra capacity, Tempest specifies a fourth tag, Busy, which has the same access semantics as Invalid. For example, software can use the Busy tag to identify blocks which are inaccessible but for which there are outstanding requests.

Access control granularity can have a significant effect on performance due to *false sharing*: when different processors write different locations in the same block, a standard single-writer protocol will unnecessarily serialize the writes, causing potentially tremendous performance degradation as the block "ping-pongs" among the writing nodes [EK89]. A larger granularity makes false sharing both more likely and more difficult to remedy by rearranging or padding data.

1. There is a fourth possible tag with distinct semantics—one that invokes an action on loads but not stores—but such a "write only" tag is of no practical value.

Page-based DSM systems use virtual address translation hardware for access control as well as local storage management. As a result, the access control granularity is the virtual memory page size—four to eight kilobytes, typically—which easily leads to false sharing. To avoid ping-ponging these large pages, the most efficient page-based protocols allow multiple simultaneous writers for each page [CBZ91, KCZ92]. Unfortunately, these protocols trade significant computation and memory overheads for this reduction in communication. To merge updates from multiple nodes, each node isolates its contribution by comparing its modified copy with a second, unmodified local copy. Creating the unmodified copy and performing the comparison may take hundreds of microseconds [ACD+96].[1] As a result, these protocols are effective only when communication overheads are comparably large. In addition, these protocols support only weak consistency models.

Higher-performance hardware-based systems reduce false sharing by providing access control at cache-block granularity, typically 16 to 128 bytes. At these granularities, false sharing occurs less frequently and can usually be avoided by rearranging or padding application data structures. The protocol is free to focus on providing efficient coherence for true sharing patterns.

To provide portability to high-performance systems, Tempest specifies *fine-grain* access control. The number of bytes per block is implementation-specific, but must be a power of two no greater than 128. If Tempest allowed a coarser granularity, users would be forced to implement a multiple-writer protocol that is unsuitable for a high-performance system. Flexibility would also be impaired, because coherence policies would have to be tuned to dealing with false sharing rather than supporting actual application sharing patterns. The only potential drawback to specifying fine granularity is that it prevents the use of standard virtual memory; however, we have demonstrated an efficient, portable, software-only fine-grain access control technique [SFL+94].

1. These operations are memory intensive, so their performance is limited by memory bandwidths rather than computation rates.

While Tempest allows for low-cost, software implementations of fine-grain access control, it is specifically designed to support higher-performance hardware-assisted techniques as well. In [SFL+94], my colleagues and I enumerate five methods: in software, in the TLB, in the cache, in the memory controller, and in a bus snooping device. Chapter 3 describes three system designs that perform fine-grain access control in hardware via bus snooping. A detailed discussion of this technique appears in Appendix B.

The Tempest interface provides functions to read and change access tags and to register user functions as block access fault handlers. To allow implementations to optimize handler dispatch, users register separate handlers for each of the five possible fault types (read–Invalid, write–Invalid, read–Busy, write–Busy, and write–ReadOnly). Each handler invocation is passed the virtual address of the faulting access and two additional values associated with the referenced virtual memory page. These values, provided by the user when the page is allocated, are uninterpreted by Tempest but are intended to be used as a pointer to a per-page protocol data structure and to identify the page's home node.

For additional flexibility, users can register multiple sets of handlers. All block access faults on a particular page use the same set of handlers; the particular set is specified when the page is allocated. As described in Section 2.3, a typical protocol uses two sets of handlers, one for the page that contains the primary data copy and another for the cached copies on other nodes. The interface supports a larger number of handler sets to allow multiple protocols in the same application.

2.2.4 Mechanism interactions

To support distributed shared memory effectively, the mechanisms described above must be designed to work together in an appropriate manner. This section describes two aspects of Tempest that cross mechanism boundaries: handler execution and atomic access control/messaging functions.

Tempest systems run each message, allocation, and block access fault handler to completion before another handler of any type is invoked. In effect, Tempest extends the Active Messages handler execution model to include allocation (page) fault and block access fault handlers. Handlers enjoy mutually exclusive access to coherence protocol state without the overhead of locking. Unfortunately, this model precludes concurrent handler execution on multiprocessor nodes, which can increase throughput on communication-intensive benchmarks [FW96a].

Tempest also specifies functions that atomically change a block's access tag and send or receive the contents of the block. To understand why these calls are necessary, consider the common situation where a node must relinquish write access to a block and transmit the block's data to another node. Software must perform two actions: (1) change the block's access tag from Writable to Invalid and (2) send the block's contents. The tag change cannot be done first, because a block tagged Invalid cannot be accessed directly, even from inside a handler. (Because some implementations cannot easily disable access control for handlers, Tempest forbids handlers from performing accesses that could result in block access faults.) However, sending the block's contents first leads to a race: handlers are (possibly) concurrent with respect to computation threads (e.g., on multiprocessor nodes), so the block could be modified by another thread after the contents are sent but before the tag change occurs. Because the permanent copy of the block is the one in the message, this modification will be lost. A similar situation arises when a message arrives containing data for a previously Invalid block: the message handler is unable to write the data directly without first changing the tag, but changing the tag first creates a window where another thread could access incorrect data before the message data is written to memory.

2.3 Transparent shared memory using Tempest

This section illustrates how Tempest can provide a standard shared-memory model in a manner transparent to the application—that is, the application uses only shared-memory

loads and stores; it does not observe the operation of the distributed shared memory layer or the underlying distributed-memory system. In contrast, the next section (Section 2.4) illustrates application-specific shared memory using Tempest, where applications customize the programming model and coherence protocol to improve performance.

The standard implementation of transparent shared memory (TSM) on Tempest is called *Stache*.[1] Stache is a user-level library that exploits the Tempest mechanisms. This library contains an allocation (page) fault handler, message handlers, block access fault handlers, and shared-memory allocation functions. Stache maps virtual addresses of shared data to local physical memory at page granularity, as do page-based software DSM systems [LH89]. However, Stache takes advantage of Tempest's fine-grain access control to maintain coherence at the block level. Because false sharing is not a major concern, Stache provides a strong consistency model using a standard single-writer, multiple-reader invalidation-based coherence protocol. The proposed Simple COMA design [HSL94] also combines page-granularity allocation with fine-grained invalidation-based coherence. However, Simple COMA uses a fixed hardware-implemented coherence protocol. In contrast, Stache is just one possible software protocol for Tempest systems.

In Stache, each shared page has a unique home node. Currently, Stache provides two home-node placement algorithms. The first assigns pages to nodes round-robin as they are allocated. The second algorithm—a simple first-touch migrate-once scheme [MKBS95]—attempts to reduce communication by placing each page on a node that references it. In this algorithm, the first node to access a page is the initial home. Unfortunately, all shared data written during the sequential initialization phase ends up on one node. To redistribute this data, the protocol also traps the first access to each page in the parallel portion of the code. If this access occurs on a node other than the initial home, that node will become the page's new, permanent home.

1. The name "Stache" is due to James Larus.

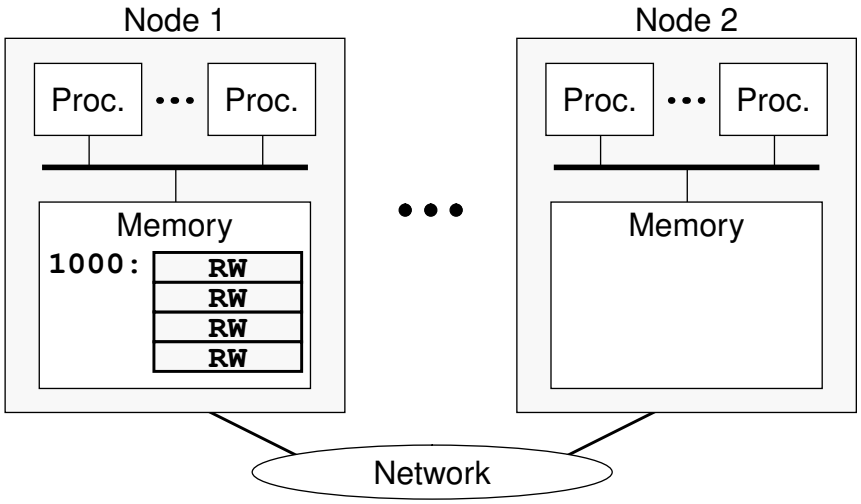

Figure 2-2. Stache example, step 1. Node 1 is the home node for the page at virtual address 1000. Node 1's processors have read-write access to all the blocks on the page (indicated by the **RW** label on each block). Node 2 has no mapping for virtual address 1000.

In either case, the home node uses a Tempest function to allocate a physical memory page and map it at the desired virtual address. It also allocates a vector of per-block directory structures (described later) on the program's heap and associates the home node's ID with the virtual page in a distributed mapping table. The page's block access tags are initialized to ReadWrite; as long as data on this page is not cached by another node, the home node can access it without software intervention. Figure 2-2 illustrates this situation, with node 1 serving as the home node for a shared data page.

When another node first accesses a shared page, the reference causes an allocation (page) fault. The user-level allocation fault handler (part of the Stache library) allocates a physical memory page and maps it at the shared virtual address. To prevent concurrent computation threads from accessing the new page, the allocation function atomically sets the page's block access tags to Invalid. The handler also looks up the home node's ID in the distributed mapping table and stores it in a per-page data structure allocated on the heap. Finally, the handler restarts the thread at the faulting access. Figure 2-3 continues the example of Figure 2-2 to this point, with node 2 preparing to satisfy a reference to the shared data on node 1.

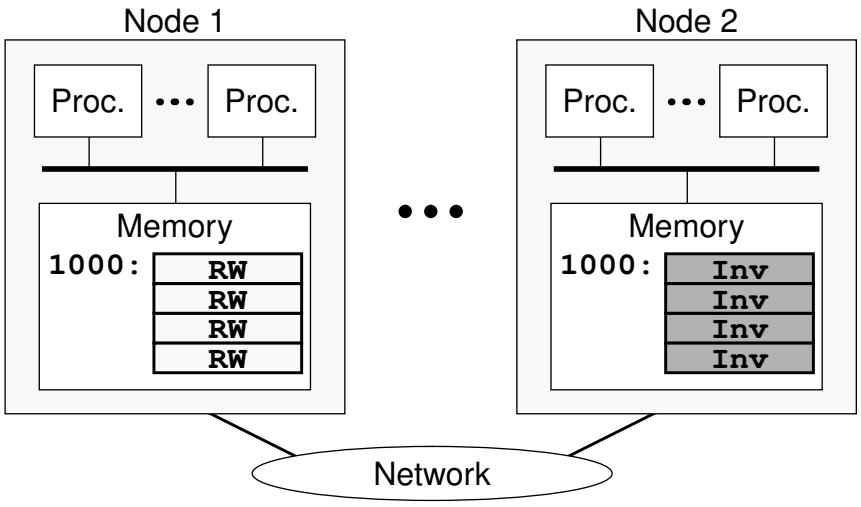

Figure 2-3. Stache example, step 2. Node 2 has mapped a memory page at address 1000 in response to an access from one of its processors. Because node 2 has no valid data for the page, the blocks on the page are marked invalid using the fine-grain access control mechanism (indicated by the **`Inv`** label on each block).

The restarted instruction now causes a block access fault because of the referenced block's Invalid tag. The block access fault handler retrieves the home node's ID from the page's local data structure, 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 replies with the data. (If invalidations are required, the handler for the final invalidation acknowledgment message sends the data.) When the reply message arrives from the home node, the message handler writes the data into the allocated page, changes the block's access tag to ReadOnly or ReadWrite, and restarts the access. This time, the access completes and the program continues. Figure 2-4 illustrates the final stage of the example, where node 2 has obtained a shared data block from node 1.

Once a block is loaded into local memory, processors may access it repeatedly without protocol action. If the initial access was a read, the protocol tags the block ReadOnly; additional reads will complete locally, but the first write will invoke a block access fault

1. This initial request could be sent from the page fault handler, but the protocol software organization makes it simpler to keep a single copy of the request code in a block access fault handler.

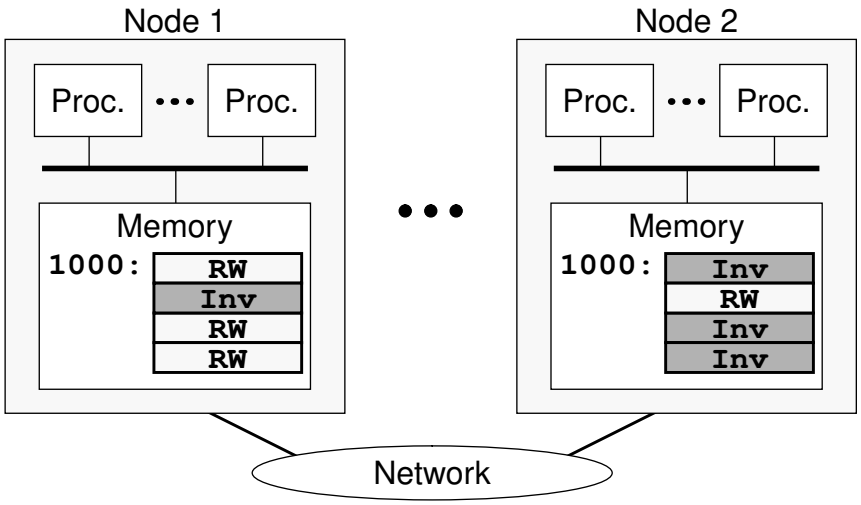

Figure 2-4. Stache example, step 3. Node 2 has fetched a block of data from node 1 to satisfy the store access. In this example, node 2 obtains exclusive access to that block and node 1's copy is marked invalid. To satisfy a read access, node 2 would obtain a read-only copy, leaving node 1 also with a read-only copy.

handler to obtain an exclusive copy. Accesses to other blocks on the page, which are still tagged Invalid, avoid the page fault and directly invoke a block access fault handler.

When a node runs out of unused physical memory pages, accesses to additional shared data pages must reuse pages containing other cached data. In this case, the page fault handler selects a page to reuse, sends any modified data on the page back to its home, marks the page's blocks Invalid, and remaps the page at a new virtual address.

Blocks on the home page are initially ReadWrite, but are downgraded to ReadOnly or Invalid as remote nodes request read-only or exclusive copies. Thus references on the home node may require protocol action to reacquire a valid or exclusive copy. Stache registers a distinct set of block access fault handlers for home pages. These handlers directly access protocol data structures and perform needed coherence actions—for example, sending invalidation requests to caching nodes.

The Stache coherence protocol is similar to LimitLESS [CKA91], except that it is implemented entirely in software. The protocol allocates eight bytes of directory state per

cache block. Two bytes store the protocol state. The other six are typically used as six one-byte node pointers. (Full bytes are used, rather than a denser packing, to avoid bitfield operations.) If more than six pointers are required, four of the six bytes are used as a bit vector.

The protocol source code is independent of the coherence block size. The protocol can be compiled for any power-of-two block size—from the Tempest access-control granularity to the page size, inclusive—by defining a C preprocessor constant. Tempest aids this flexibility by including a block size parameter for access-control functions. Tempest implementations transparently support operations on any power-of-two multiple of the system's fine-grain access control granularity. (Non–power-of-two multiples are not supported directly because their use is rare and they complicate pointer alignment checks.)

Tempest allows the Stache implementation to emphasize speed and simplicity rather than generality. Although only one implementation is currently available, the library can provide several variations. Initialization code can install a version optimized for the number of nodes and block size—and perhaps other parameters—specific to that execution. For example, block sizes smaller than 128 bytes may use a protocol that allocates less than eight bytes per block. Systems larger than 32 nodes may use dynamic pointer allocation [SH91] to avoid multi-word bit vectors. Systems larger than 256 nodes will require multi-byte node pointers. However, small systems need not pay at runtime to support these alternatives.

2.4 Optimizing applications using Tempest

The Stache protocol effectively supports applications written to a standard shared-memory programming model. However, the real power of Tempest lies in the opportunity it presents to optimize performance by tailoring the coherence protocol to the specific application. Because the protocol is simply user-level software—a set of functions linked in with the application—a programmer or compiler can customize protocols for specific data structures and specific phases of the application. Tempest allows users to bind a different

set of block access fault handlers to each memory page, so multiple protocols—including Stache—can coexist peacefully, each managing a distinct portion of the shared data. Tempest gives programmers several paths to improve the performance of applications. They can:

- *Select from available transparent–shared-memory protocols.* Protocols can implement transparent shared memory (TSM) with a variety of policies (e.g., update vs. invalidate) and parameters (e.g., block size). Changing the coherence protocol for the entire application is as simple as linking with a different protocol library; using different protocols for different data strctures requires only modest effort. Although some knowledge of the application is useful for selecting appropriate protocols, these changes cannot break a correct program. The range of available protocols should be expanded by the availability of high-level languages and compilers targetted for this domain [CRL96].
- *Develop a protocol that exploits application-specific knowledge.* Programmers or compilers can use precise knowledge of an application's synchronization and sharing patterns to optimize the coherence protocol aggressively. The program still executes in a shared address space, but an algorithmic change may require modifications to the coherence protocol as well. Most of the optimizations described in the examples below are of this type. In this work, these changes are performed manually, but the process can be automated using existing compiler techniques to extract access patterns from source code [VL96, DCZ96].
- *Use message passing.* Some operations, such as synchronization, fit the message-passing model naturally; in these cases, programmers can dispense with shared memory entirely. For example, the default lock and barrier implementations in Tempest's utility library use Tempest's Active Messages directly. Active Messages are also useful for building higher-level operations such as fetch-and-op and remote write.

This section illustrates some of these optimizations using six scientific applications. In each case, the programmer started with an optimized transparent–shared-memory parallel

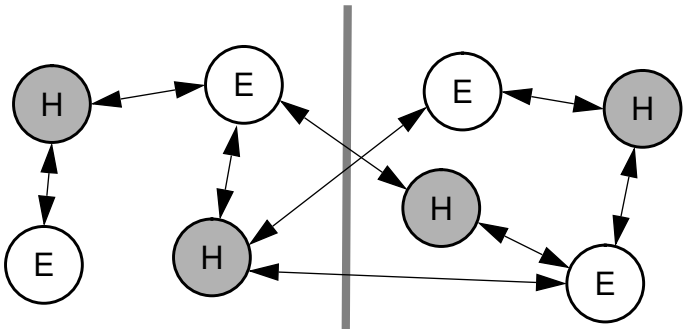

Figure 2-5. Example EM3D bipartite graph. The vertical line represents a division of the graph across two nodes of a distributed-memory system.

program and further improved its communication behavior by developing custom protocols for critical data structures. I describe one of the applications—EM3D—in detail. Five other applications—Appbt, Barnes, DSMC, moldyn, and unstructured—were optimized by other members of the Wisconsin Wind Tunnel research group. I describe them here briefly to suggest the variety of optimizations that can be performed, and to provide background for the following chapters where all six of these applications are used as benchmarks. The original papers reporting on these optimizations (Falsafi et al. [FLR+94] for Appbt, Barnes, and EM3D,[1] and Mukherjee et al. [MSH+95] for DSMC, moldyn, and unstructured) detail the applications and the evolutionary optimization process.

2.4.1 EM3D

EM3D models electromagnetic wave propagation through three-dimensional objects [CDG+93]. The principle data structure is a bipartite graph, in which *E nodes* represent electric field values and *H nodes* represent magnetic field values (see Figure 2-5). The main computation iteratively updates the field values to model the passage of time. Each iteration has two phases. The first computes new values for the E nodes using a weighted sum of their neighboring H node values. The second updates the H node values similarly based on the new values of their neighboring E nodes.

1. I wrote the original Tempest application-specific protocol for EM3D as part of the original paper on Tempest and Typhoon [RLW94]. This protocol was further analyzed and improved by Falsafi and Rogers [FLR+94].

```
typedef struct e_node {
   double        value;        /* field value at this node */
   int           edge_count;   /* number of edges/neighbors */
   double        *weights;     /* vector of edge weights */
   struct h_node *(*h_nodes);  /* vector of adjacent H nodes */
   struct e_node *next;        /* next local E node */
};

void compute_E()
{
   struct e_node *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];
}

main()
{
   int iter;

   for (iter = 0; iter < num_iters; iter++)
   {
      compute_E();
      barrier();
      compute_H();
      barrier();
   }
}
```

Figure 2-6. EM3D program fragment. The `struct h_node` type and `compute_H()` function are symmetric with `struct e_node` and `compute_E()`, respectively.

Figure 2-6 lists part of the shared-memory EM3D code. The parallel version assigns a set of E nodes and a set of H nodes to each processor. During each iteration, every processor updates its own nodes. Because only one processor updates each node, no locking is required. A barrier after each phase guarantees that all the new values for one type of node have been written before any processor starts reading the values to update the other nodes.

On a distributed-memory system, the program reduces traffic by allocating the graph nodes—that is, by placing their Stache home pages—in memory local to the processor that owns them. Communication occurs only when one processor reads a value that is

written by another, which happens when a graph edge connects nodes owned by different processors. Unfortunately, transparent shared memory does not support this producer–consumer communication pattern efficiently. When a processor reads a remote graph node value, the coherence protocol fetches the value and caches it. In the next phase, the owning processor updates the node value, causing the protocol to invalidate the cached copy. Thus in each iteration, each communicated value is fetched, cached, and invalidated, which requires at least four messages (request, response, invalidation, and acknowledgment). I optimized the transparent shared memory version to amortize this overhead by placing multiple values in a cache block. This optimization modifies the graph node data structure, replacing the embedded value with a pointer into a packed value array.

Tempest lets us optimize communication much more directly by using a custom protocol for the graph nodes. In the first iteration, this protocol behaves very much like the Stache protocol: references to remote values invoke block access fault handlers, which fetch and cache the data. Unlike Stache, the protocol records information regarding the addresses and nodes that are involved in each request. Because the graph does not change during execution, each requester needs the same set of addresses in every iteration. This set cannot be determined statically because the graph structure is input dependent.

After the first iteration, the protocol switches to update mode. After each phase, each processor uses the sharing information obtained in the first iteration to send updated values to the processors that need them. Because the updates are simply user-level messages, the protocol sends only the modified values, not full cache blocks. To further reduce overheads, the protocol packs all the values destined for a particular processor in a single message, and transmits that message using Tempest's virtual channels (see page 17). Each node also determines, from the information collected in the first iteration, how many update messages it should receive; the full barrier is replaced by a loop that waits until all of the expected messages arrive.

Using this custom Tempest protocol, iterations after the first reduce communication to one message between each producer–consumer pair, the same level of communication efficiency—and the same level of performance [FLR+94]—as a message-passing implementation. However, the program retains its original shared-memory structure. The only changes to the code in Figure 2-6 are a function call to initialize the custom protocol, a call to notify it of the end of the first iteration, and calls to the new function that combines the updates and synchronization (replacing the barriers). The graph data structure is still pointer based; neither the `struct e_node` type nor the `compute_E()` function changes. The initialization code (not shown in Figure 2-6) uses a new memory allocation function that places the graph nodes on pages serviced by the custom protocol, but is otherwise unchanged. In contrast to the few simple changes in the original source, the custom protocol comprises over one thousand lines of C. This value overstates the protocol's complexity somewhat; this code represents one of the earliest custom protocols, and was written without the aid of experience or protocol development tools.

Although other programming models may achieve similar levels of efficiency, they generally require more drastic modifications to the program's code and data structures. For example, the optimized Split-C version modifies the graph itself, explicitly inserting local "ghost nodes" to cache remote values [CDG+93]. The optimized TSM version described earlier adds a level of indirection from the graph nodes to their values to increase locality. Even with this non-intuitive change, the TSM version falls well short of the Tempest version's efficiency. In contrast, the optimized Tempest program still enjoys the advantages of a shared-memory model: pointer-based data structures in a uniform, global address space.

2.4.2 Appbt

Appbt is a three-dimensional computation fluid dynamics code from the NAS Parallel Benchmarks collection [BBLS91], parallelized for transparent shared memory by Burger and Mehta [BM95]. The primary data structure is a three-dimensional dense matrix. The parallel version assigns a subcube of the matrix to each processor. All communication

occurs along the faces of these subcubes. The TSM version synchronizes by having processors spin on shared counters. After a processor updates a column on a subcube face, it advances the corresponding counter; the neighbor waiting for this column then performs its computation which depends on the new values. The values are transferred on demand as the processor references them. Assigning counters to columns, rather than faces, allows processors to pipeline their dependent computations.

As in EM3D, the Tempest-optimized version of Appbt replaces all of the communication and synchronization in the main iteration with a single message between each producer–consumer pair. This message notifies the receiver that the updated face is available and carries the new values for that face. Unlike EM3D, the sharing pattern is not input dependent, so no special first iteration is needed to determine it. Again, even though the custom protocol achieves a message-passing level of efficiency, the bulk of the program does not change from the original shared-memory version. Approximately one hundred lines were added or modified, a small fraction of the roughly seven thousand lines in the original program. Most of these changes were repetitive replacements of synchronization statements. The custom protocol itself required about 750 lines of C.

2.4.3 Barnes

Barnes is an N-body gravitational force computation application from the SPLASH benchmark suite [SWG92]. Barnes avoids calculating all N^2 interactions by approximating the force from a distant cluster of bodies as that from a single body at the cluster's center of mass. The primary data structure is an oct-tree. The interior nodes of the tree represent regions of three-dimensional space; the nodes at a given level of the tree bisect their parent's region in all three dimensions. The leaves of the tree are the bodies located in the region represented by their parent node. The application iterates to simulate the evolution of the system through discrete time steps. Each iteration has two phases. The first builds the tree from scratch based on the current position of the bodies. The second calculates the new accelerations, velocities, and positions of the bodies.

The SPLASH transparent shared memory version parallelizes both phases of the iteration. Each processor is responsible for a group of bodies. In the tree-build phase, the processors add their bodies to the tree in parallel. Locks protect the internal tree nodes as new levels are added to the tree. In the calculation phase, each processor updates its bodies, referencing other parts of the tree to calculate the applicable forces. Because a processor writes only the bodies it owns, this phase needs no mutual exclusion.

We applied three Tempest optimizations to Barnes. The first and most significant optimization uses a custom update protocol to propagate the bodies' new positions to potential consumers. This protocol is more complex than those used by EM3D and Appbt because there is no static sharing pattern. Instead, the processor sends position updates to the memory location's home node, which forwards the data to the nodes that are caching the location at the time. The second optimization replaces the shared-memory MCS locks [MCS91] used for the internal tree nodes in the build phase with message-passing locks. Both types of locks build a queue of waiting processors which spin locally until they are granted the lock; by using Active Messages instead of shared-memory reads and writes, the message-passing version generates half as much traffic as the MCS locks under contention. All the locks are defined using the PARMACS macros [BBD+87], so this optimization can be applied simply by recompiling. For the third optimization, each processor invalidates its cached tree nodes explicitly at the end of each iteration, eliminating the implicit invalidation and acknowledgment messages that would otherwise occur as the node structures are reused during the next tree build phase. Because the first optimization modified the body data structure to isolate the position fields, source changes for the optimized version of Barnes were more widespread than for the other applications, making it difficult to quantify the scope of the changes.

2.4.4 DSMC

DSMC simulates gas particles moving and colliding in a three-dimensional space. The space is divided into fixed cells. The first phase of each iteration randomly selects pairs of

particles in the same cell and simulates their collision. The second phase updates the particles' positions, and the third and final phase reassigns particles to cells based on their new positions.

The transparent–shared-memory implementation assigns a contiguous group of cells to each processor. Interprocessor communication occurs only during the final phase of each iteration, where a particle may be reassigned to a cell belonging to a different processor. Rather than allowing one processor to manipulate another's cell data structures directly, each processor has a buffer that collects the particles that move into its cells. To avoid locking, barriers divide the reassignment step into phases, and in each phase a processor has exclusive access to the buffer of its neighbor in a particular direction. Most moves occur between adjacent cells, so these phases capture most of the communication. A separate buffer, protected by a lock, receives the few particles that move between non-adjacent cells.

Because the TSM version of the program performs well, only one simple Tempest optimization was applied. Instead of writing to the particle buffers using shared-memory writes, processors send Tempest Active Messages to write the buffers on the destination node. In addition to eliminating coherence overhead for the buffers, the atomic message handlers serialize the writes, so all the nodes can issue their writes concurrently without barriers or locking. This optimization modified only two lines in the original program. The Tempest-optimized write protocol involves approximately 150 lines of code.

2.4.5 Moldyn

Moldyn is a molecular dynamics application that models force interactions between molecules. To speed the calculation, moldyn computes interactions only between molecules within a cut-off radius. The interaction list tracks pairs of molecules that are near enough to interact. There are two main steps in the computation: one that examines every pair of molecules to build the interaction list, and another that iterates over the interaction

list to compute the forces between interacting molecules. Moldyn rebuilds the interaction list after every 20 iterations of the force computation.

The TSM version parallelizes both steps. In the first step, each processor examines a fixed subset of the molecule pairs to generate a portion of the interaction list. In the second step, each processor computes the forces due to the interactions on its portion of the list. Because a molecule typically participates in many interactions handled on different processors, the updates to the molecule's state must be synchronized. Instead of locking the molecule every time it participates in an interaction, each processor accumulates the updates it generates for each molecule in a local array. After all the interactions have been processed, the processors merge their updates using a synchronized reduction: in each of N phases, separated by barriers, each of the N processors adds its updates to one Nth of the shared molecule array. The communication in this phase resembles a pipeline, where each Nth of the array migrates from node to node.

As with DSMC, the TSM version is heavily optimized to reduce communication. Tempest allows programmers to take these optimizations to their logical conclusion. In the custom-protocol version of moldyn, Tempest's virtual channels used to move array sections from node to node in the molecule update reduction phase. This optimization is straightforward to apply—changing only one line of the original code—because the messages simply implement the communication pattern the programmer was trying to achieve with the original synchronous reduction algorithm. The virtual-channel protocol itself involves less than two hundred lines of C.

2.4.6 Unstructured

Unstructured is a computational fluid dynamics application that uses a mesh to model forces on a three-dimensional solid structure. The mesh is static but input dependent. The computation consists of iterations over the nodes, edges, and faces of the mesh that update values associated with the nodes. The parallel version partitions the mesh nodes across the

processors, then the edges and faces. An edge or face that connects nodes not all on the same processor is assigned to one of the involved processors.

Loops that iterate over the mesh nodes require no synchronization, because a node is only updated by the processor that owns it. Edge and face loops are more difficult: the computation centered around an edge or face updates all the involved mesh nodes, which may be owned by different processors. The naive solution—locking the nodes—leads to poor performance due to the overhead of locking and the frequent migration of the mesh node data structure. Instead, as in moldyn, each processor accumulates its updates locally, then merges its updates into the global mesh data during a synchronous reduction phase.

This synchronous reduction is less effective in unstructured than it is in moldyn for two reasons. First, the *N*-phase reduction (described on page 36) circulates the entire shared node array, piece by piece, to every processor, so that each processor has the opportunity to update each node. If each processor updates only a fraction of the nodes, as is typically the case for unstructured, then most of this communication is unnecessary. Second, each update involves only a small amount of computation, so efficient communication is particularly critical for good performance. For these reasons, the Tempest-optimized version replaces the *N*-phase reduction with a far more efficient custom protocol. This protocol examines the mesh before the iterations begin to set up virtual channels connecting the processors that share edges. Updates are sent across these channels directly to the node's owner, which then performs the reduction locally. This optimization added five lines in the body of the program. The reduction protocol involves just over six hundred lines of C.

Although the reduction step efficiently updates the values for each node at the node's owner, these new values must be propagated to the other processors that reference them in the edge and face loops. The TSM version fetches the new values on demand as the processors reference them. The Tempest version optimizes this communication using a second custom protocol, an update protocol very similar to the one used in EM3D (see

Section 2.4.1). This optimization modified twelve lines in the body of the program. The protocol itself is less than six hundred lines of C.

2.5 Related work

This thesis proposes flexible distributed-memory systems that provide only fundamental mechanisms, allowing users to construct efficient, customized shared-memory policies. The separation of mechanisms from policies as a technique for building flexible systems first arose in the operating systems domain [Bri70, WCC+74, LCC+75]. Wulf rephrases the concept as "primitives, not solutions" and argues for its application to processor instruction sets, emphasizing the use of compiler technology to generate situation-specific policies automatically [Wul81].

Dally and Wills [DW89] identify three universal primitive mechanisms for distributed-memory systems: communication, naming, and synchronization. We agree on communication (messaging) as a fundamental mechanism. Their naming mechanism binds logical to physical names—a generalization of local storage management, which performs the same task on the restricted domain of memory addresses. They include synchronization because of their focus on fine-grain concurrent programming models (e.g., dataflow) that complement their fine-grain research platform, the J-Machine [DCF+89]. In contrast, I focus on the medium- to coarse-grained shared-memory and message-passing models prevalent on commercial hardware, where synchronization performance is less critical. Although Dally and Wills describe a shared-memory model, they do not consider caching. As a result, they omit access control. In particular, their mechanisms lack support for read-only data replication.

This thesis differs from previous work on mechanisms for distributed shared memory in its goal and scope. Johnson et al. [JKW95] decompose DSM systems into three high-level mechanisms for the purpose of classifying implementations. Chaiken [Cha94] and Wood et al. [WCF+93] focus solely on the coherence directory. In contrast, I propose mechanisms as a tool to provide flexibility in both the caching and directory aspects of DSM.

Tempest is the first interface to provide flexible shared memory by allowing users to go underneath the shared-memory abstraction, using fundamental mechanisms to construct arbitrary protocols. Others have added features to the shared-memory model to achieve the same goal of higher performance. Prefetching [MG91] and the checkin–checkout (CICO) annotations [HLRW93] direct the local cache to fetch and replace blocks to anticipate usage patterns. Poststore or deliver operations [LLG+92, RSW+93, KCPT95] let the user select an update policy for individual writes. Munin [CBZ91] allows programmers to annotate variable declarations to indicate the expected sharing pattern, so that the run-time system can choose an appropriate protocol from the set that it implements. Clouds [RAK89] provides operations that modify the synchronization and coherence semantics of shared-memory operations. Labeling synchronization operations to implement weak consistency models [AH90, BZS93, GLL+90] also falls into this category. Any of these higher-level features could be implemented using Tempest's mechanisms.

The integration of shared memory and message passing has been the subject of recent work [KJA+93, HGDG94]. Tempest cleanly achieves this by exposing message passing as one of the mechanisms used to implement shared memory. Other systems such as Alewife [ABC+95], FLASH [KOH+94], and Start-NG [CAA+95] support both models, but do not allow the user to combine their features arbitrarily. For example, none of these systems allows data to be sent in a message without being renamed at the receiver. Frank and Vernon [FV93] propose extensions for a shared-memory machine that send data without renaming. However, their message-passing and shared-memory primitives interact in a fixed way and do not allow arbitrary coherence policies.

The Tempest mechanisms may be implemented in hardware, but Tempest protocols are by definition software-based. Hybrid hardware–software support for shared memory was first employed in VMP [CSB86, CGBG88], a bus-based shared memory system that uses software at the processor caches to handle misses and support bus snooping. A follow-on system, Paradigm [CGB91] (originally called VMP-MC [CGB89]) adds a simple hardware directory at main memory to efficiently support a hierarchical-bus organization.

Hybrid protocols for distributed shared memory were first proposed for Alewife [ABC+95], whose LimitLESS protocol [CKA91] implements a few pointers in hardware and traps to software to handle blocks with many sharers. Dir_1SW [HLRW93] and Dir_1SW+ [WCF+93] implement a single pointer in hardware, shifting even more complexity into software. Alewife and Dir_1SW use software only at the directory; caches are hardware-controlled. FLASH [KOH+94] and Start-NG [CAA+95] implement shared memory entirely in software. FLASH executes this software on a processor integrated in the memory controller and specially designed for this purpose. Start-NG uses a commodity processor with special external hardware for capturing and generating bus transactions. The one perspective that all of these systems share, and that differentiates them from Tempest, is that the software part of the protocol is protected system software, beyond the reach of user processes. This severely restricts the flexibility of these approaches, since it implies a protected interface and thus limited coupling between the application and the protocol. In fact, the original motivation for the hybrid protocols in LimitLESS and Dir_1SW was to reduce hardware complexity, not to add flexibility. (Chaiken later explored modifying the software part of LimitLESS to do profiling and adaptive optimization [Cha94].) Though flexibility is an explicit goal of both FLASH and Start-NG, these systems lack a general, portable interface for exporting that flexibility to user programs.

Tempest's use of virtual address translation for local storage management is borrowed from page-based software DSM systems, of which Ivy [Li86, LH89] was the first. Unlike Tempest, these systems also use virtual address translation for access control, which forces them to maintain coherence at the granularity of virtual memory pages. More recent page-based DSM systems combat the resulting false sharing using weak memory models and multiple-writer protocols [CBZ91, BZS93, KDCZ94]. Recently developed memory-mapped network interfaces can merge updates from multiple writers at the home node [IDFL96, kontothanassis:mmni-dsm], eliminating the overhead of copying and comparing pages at each writer. Page-based allocation implies the use of local DRAM to cache remote data, which gives all of these systems (including Tempest) some of the characteris-

tics of "cache-only" (COMA) machines like the KSR-1 [Ken92] and the Data Diffusion Machine (DDM) [HLH92].

A number of systems combine page-based storage management with hardware fine-grain coherence. PLUS [BR90], Sesame [WHL92], and Galactica Net [WJI+94] maintain coherence at word granularity using a write-through hardware update protocol. Simple COMA [HSL94] combines page-based allocation with an invalidation-based hardware protocol to implement a COMA architecture.

Several software systems implement object-based shared memory on message-passing machines. All of these systems amortize software overheads by performing access control once for a group of accesses to an object. Emerald, Amber [CAL+89], and Orca [BKT92] and maintain coherence on language objects and perform access control once per method invocation. Midway [BZS93], CRL [JKW95], and SAM [SL94] rely on programmer annotations both to identify objects and to group object accesses. Amber and Midway use virtual address translation to map objects into local memory. CRL and SAM use a hash table to map objects, but combine the translation with the coherence annotations so that the lookup only occurs once per group of accesses.

2.6 Summary

Tempest is a flexible, portable interface for shared-memory programming on distributed-memory machines. Tempest provides three fundamental mechanisms—messaging, local storage management, and access control—that software can compose to construct cache-coherent distributed shared memory. Direct, unprivileged access to these primitives gives programs unprecedented control over their memory and communication. Programmers and compilers can optimize performance by customizing coherence protocols for specific data structures in specific phases of an application.

The Tempest interface is carefully designed to balance the portability, utility, and performance of each mechanism it provides. Two complementary interfaces support messaging:

a variant of Active Messages [vECGS92] and asynchronous, bandwidth-oriented virtual channels. The local storage management interface targets standard virtual address translation hardware. Tempest's most innovative aspect is its specification of *fine-grain* access control, which avoids severe false sharing and allows protocols to scale to high-performance systems.

Stache is a user-level library that implements application-transparent cache-coherent distributed shared memory using the Tempest interface. Stache provides a large, fully associative remote data cache on each node by mapping remote virtual pages into local memory. Tempest's fine-grain access control allows coherence on a much smaller granularity. Stache implements a single-writer invalidation-based coherence protocol in software.

Any DSM system with a single coherence protocol—including Stache—incurs unnecessary communication due to its fixed policy. Tempest gives programmers a practically unlimited set of options for customizing communications policies to improve the performance of shared-memory applications. Section 2.4 described the optimizations applied to six shared-memory programs.

Of course, Tempest is not useful unless it can be implemented portably and efficiently. We have demonstrated all-software Tempest implementations on generic message-passing platforms elsewhere [SFL+94, SFH+96]. The next chapter describes and analyzes three system designs that provide hardware support to accelerate Tempest. This range of implementation alternatives demonstrates Tempest's portability. I establish Tempest's efficiency by showing (via simulation) that the fastest of these systems—Typhoon—rivals all-hardware distributed shared memory systems in performance.

Chapter 3

Hardware Support for Tempest

The Tempest interface is a contract between a system's users and its developers. The previous chapter focused primarily on the former group, discussing how Tempest can be used to implement and optimize distributed shared memory. This chapter addresses the concerns of the latter group: the practicality and performance potential of systems that implement Tempest.

This chapter describes and analyzes designs for three systems—called *Typhoon, Typhoon-1,* and *Typhoon-0*—that use custom hardware to support the Tempest interface. These designs have the following goals:

- to demonstrate that Tempest is portable across a range of implementations, including high-performance systems;
- to illustrate techniques that can be used to support Tempest in hardware; and
- to provide the basis for obtaining quantitative performance results via simulation.

To further demonstrate the feasibility of these designs, I led the development of a prototype implementation of Typhoon-0. This prototype is described in Chapter 4.

Rather than attempt to cover the full design space of Tempest systems, I focus on a single access-control technique—bus-based snooping hardware. Bus-based hardware access control provides higher performance than software techniques, and complements our other work in the latter area [SFL+94, SFH+96]. On the other hand, more aggressive hardware implementations require modified processors or caches, making them economically infeasible in the near term. See Section 3.5 for further discussion of access control implementation alternatives.

Within this restricted domain, I develop three different systems by varying the level of integration, or the coupling, of the logical components used for Tempest support. Greater integration reduces communication overhead between components, leading to higher performance. However, using a general-purpose off-the-shelf part reduces design time and complexity compared to implementing that logic as part of a custom integrated device. For rapidly advancing mass-market technologies such as processors, the off-the-shelf part is likely to have better raw performance as well. Decoupling (separating) the functional components lets designers plug in these off-the-shelf parts when available.

The overall cost difference between integrated and decoupled systems depends on design complexity and manufacturing volumes. The integrated solution increases design cost by an amount proportional to the custom device's complexity. Even if the component circuits are available, as is the case with embedded processor cores, it is costly to integrate the circuits and verify the complete design. However, integration typically lowers manufacturing costs by reducing board space and physical component count. The integrated solution is more expensive if the additional design cost, per unit, is greater than the reduction in manufacturing cost. For a complex component such as a processor in the (currently) low-volume market for distributed-memory machines, it is likely that design costs dominate—and thus decoupled systems are cheaper. Of course, integration still is justified when the need for additional performance outweighs the additional cost.

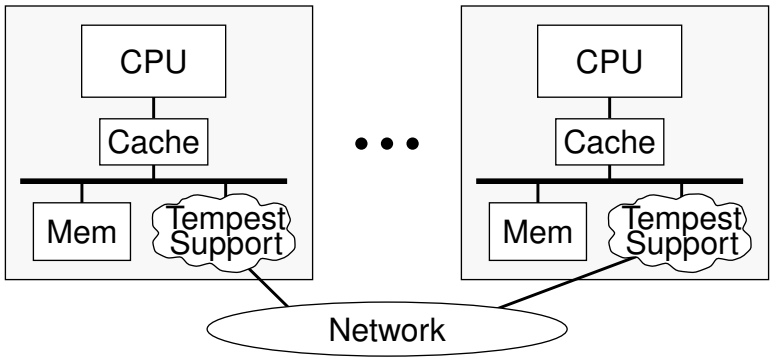

Figure 3-1. Common system organization.

All three systems combine off-the-shelf workstations with a generic point-to-point data network, leading to the common organization shown in Figure 3-1. This "network of workstations" approach [ACP95] leverages the rapid technological advances and the economies of scale of the workstation and personal computer markets. However, builders of these systems are unable to modify or replace existing workstation components. Instead, they add separate Tempest support hardware to each workstation node, interfacing with both the node's processor–memory bus and the network.

In each system, the Tempest support hardware comprises three logical components, plus glue that connects them, as shown in Figure 3-2. The first component, a processor, is dedicated to executing protocol software (message and block access fault handlers). The other two components directly support two of the three Tempest mechanisms. A network interface, which connects the node to the data network, serves as the foundation for messaging. Custom bus-snooping logic enforces Tempest's fine-grain access control. The workstation's existing virtual address translation facilities are sufficient for Tempest's third mechanism, local storage management.

As discussed above, the three systems differ primarily in the extent to which they integrate these components. Figure 3-3 depicts each system's level of integration graphically. Of the three systems, Typhoon provides the highest performance—at the highest cost—by integrating all three components on a single custom device. Typhoon-1 decouples the pro-

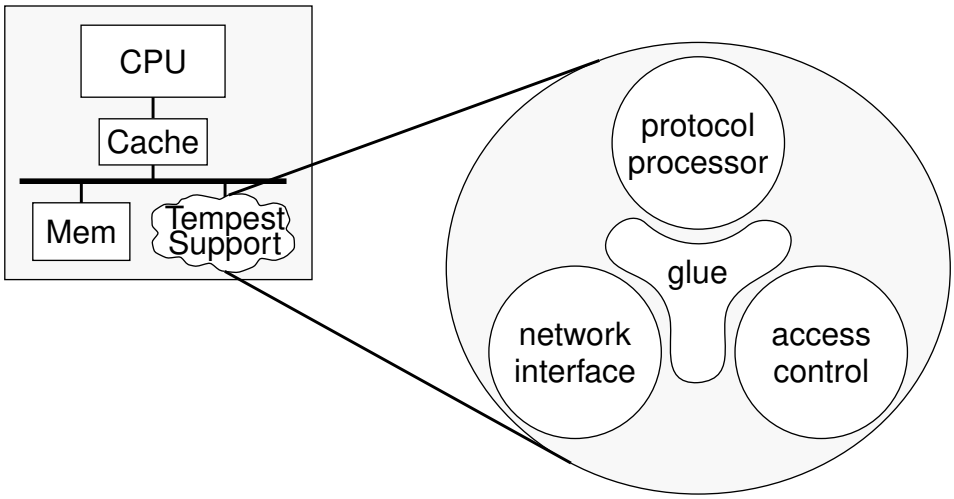

Figure 3-2. Logical components of Tempest support.

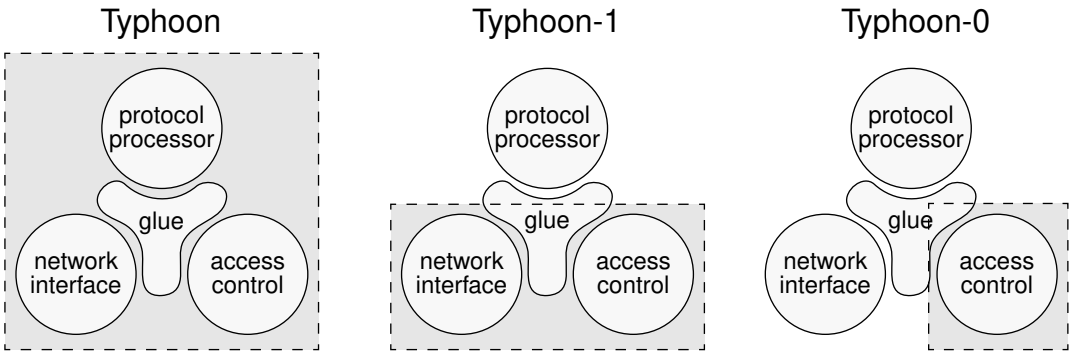

Figure 3-3. Component integration diagram. For each system, a shaded rectangle indicates the components integrated into custom hardware.

tocol processor, replacing Typhoon's integrated processor with an off-the-shelf CPU. Typhoon-0 achieves the lowest cost by splitting all three components across separate devices, two of which—the protocol processor and the network interface—can be purchased off the shelf. A relatively simple access control device is Typhoon-0's only custom component.

The first section of this chapter describes the common features of the three systems. The following section (Section 3.2) discusses the integrated Typhoon system. Section 3.3 cov-

ers the two decoupled systems, Typhoon-1 and Typhoon-0. Section 3.4 analyzes the performance of all three systems using simulation. Section 3.5 describes related work and Section 3.6 summarizes the chapter.

3.1 Common features

To focus on the integration–decoupling trade-off, all three systems use the same basic techniques to support Tempest. This section describes these common features for each Tempest mechanism—messaging, local storage management, and fine-grain access control.

3.1.1 Messaging

Each node interfaces to a reliable point-to-point network through a pair of 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. The message queues are memory mapped and directly accessible from user-level software via loads and stores, as in the Thinking Machines CM-5. Although future network interfaces are likely to differ in some details, I believe they will provide a similar queue abstraction [BCL+95, MFHW96].

Network buffer management uses a simple acknowledgment-based scheme. Each network interface reserves space in its receive queue for a small number of messages from each sender. When a message is consumed by software, the NI generates an acknowledgment to notify the sender that the buffer space is free. The sending network interface counts the number of unacknowledged messages outstanding to each destination, and rejects messages that would cause this count to exceed the number of reserved spaces at the destination NI.

The run-time library avoids deadlocking for buffer space by queueing blocked messages in the sending node's memory and injecting them as acknowledgments arrive.[1] Library send functions may either query the NI's outstanding message counter before attempting to send and, if the counter indicates that no destination buffers are available, build the message directly in the software queue; or they may attempt to send the message, check a status bit afterward, and, if the message was rejected, copy the message out of the hardware send queue. When the library puts a message in the software queue, it sets a hardware mode bit that causes acknowledgments to invoke a software handler in the same manner as protocol messages. This acknowledgment handler sends queued messages and clears the mode bit when the software queue is empty.

Software implements Tempest's Active Messages directly on top of the hardware queues: the first word of each message is used for the receive handler's program counter. Tempest's bulk data transfer functions are implemented in a straightforward fashion on top of the Active Message layer.

3.1.2 Local storage management

The main processor's virtual address translation hardware supports Tempest's local storage management. A special device driver manages a segment of the virtual address space reserved for shared data, separate from the typical text, stack, and (private) data segments. Page faults within this segment invoke a per-process user-level handler. The device driver provides operations that bind and unbind addresses in the segment to physical memory. Optimized operating system exception paths provide higher performance than the standard Unix signal interface [RFW93, TL94].

---

1. Because Tempest does not prevent users from consuming unbounded amounts of buffer space, a system cannot guarantee that deadlock will never occur (see Section 2.2.1). An ill-behaved program may terminate if the software queue overflows the sending process' virtual memory.

3.1.3 Fine-grain access control

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

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

For efficiency, the snooping device allows direct manipulation of access control tags from Tempest's user-level protocol software. This software manipulates tags based on virtual addresses, but—because the hardware sees only physical addresses on the bus—access control is performed on physical memory locations. If software were to send virtual addresses directly to the hardware, the device would be required to perform a translation and a protection check. A *shadow space* [BLA+94, HGDG94, Thi91] avoids both of these, as illustrated in Figure 3-4. 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 address range. Accesses to a location in the shadow space are interpreted as operations on

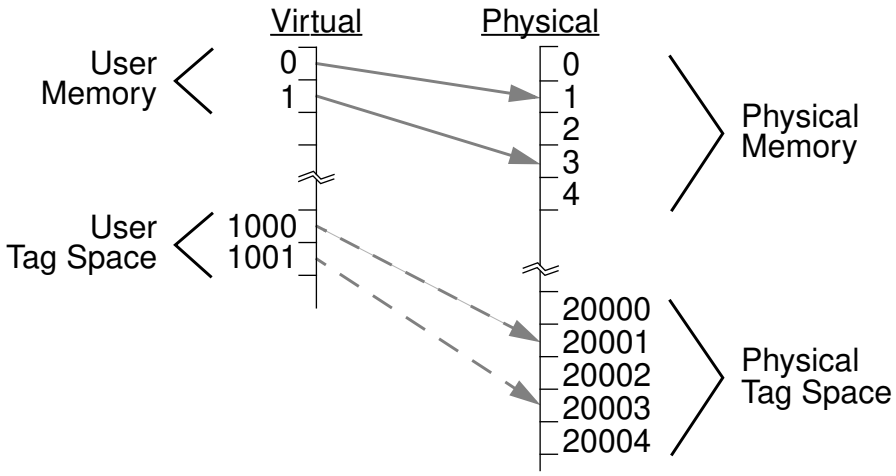

Figure 3-4. Shadow tag space example. The hardware device interprets accesses to physical tag space page `2000x` as tag operations on physical memory page `x`. The user program requests operations on its virtual page `y` by accessing its virtual page `100y`. The shadow mappings (dashed arrows) dual the memory mappings (solid arrows), implicitly translating the address component of the user's requests.

the corresponding real memory location. When a user process allocates a physical memory page, the device driver described in Section 3.1.2 provides a mapping to the corresponding shadow space page as well.

3.2 Typhoon: integrated hardware support for Tempest

Typhoon [RLW94] combines the network interface, access control logic, and a user-level protocol processor on a single device (see Figure 3-5). To enable potential reuse of existing VLSI design components, Typhoon's protocol device is structured as a standard processor with closely coupled on-chip peripherals. In contrast, the FLASH system's MAGIC chip [KOH+94] implements a specialized data path and control architecture tuned for software-driven coherence protocol processing.

3.2.1 Protocol processor

The protocol processor is a standard pipelined integer unit, using the same instruction set as the main CPU. To run user-level code efficiently, the protocol processor includes a

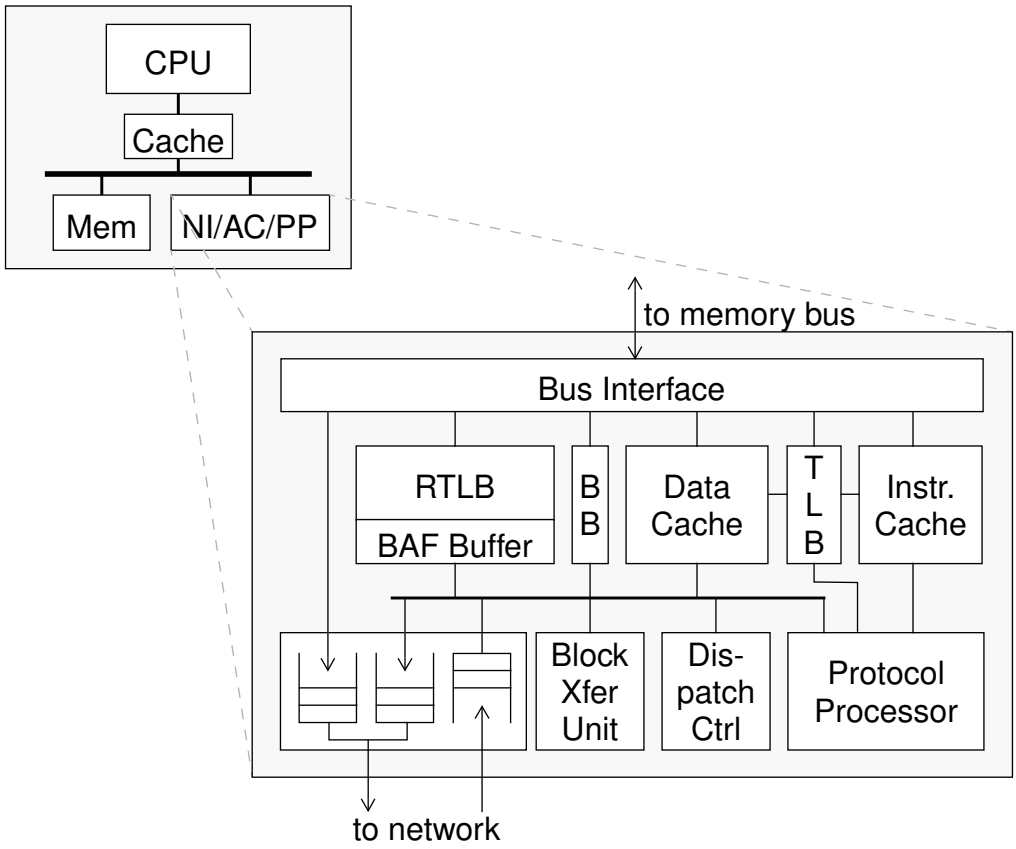

Figure 3-5. Typhoon node, including a block diagram of the network interface/access control/ protocol processor device. RTLB is the reverse TLB; BB is the block buffer; the BAF buffer holds information on block access faults. The second network send queue is for use by the compute processor(s).

standard TLB and instruction and data caches. The processor, TLB, and cache designs may be taken from a previous-generation CPU or an embedded ASIC core design.

The other components of the Typhoon device connect to the protocol processor's cache bus. Software running on the protocol processor accesses these components via memory-mapped registers with single-cycle latency.

3.2.2 Access control (RTLB)

Typhoon implements access control using a component called the *reverse translation lookaside buffer*, or RTLB (see Figure 3-6). The RTLB is a cache with per-page entries

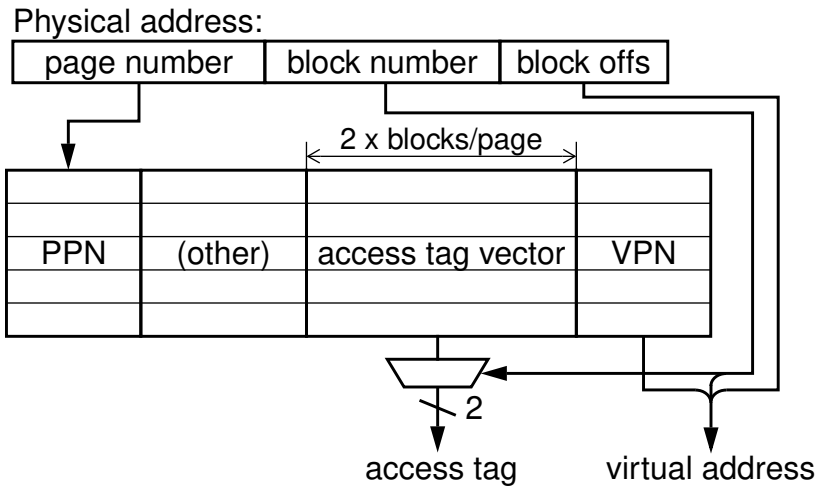

Figure 3-6. Typhoon RTLB diagram. Arrows indicate how a physical bus address is processed by the RTLB to generate the block's access tag and virtual address. The handler table pointer, user data pointer, and home node ID RTLB fields are not shown. The PPN field holds the physical page number; VPN is the virtual page number.

indexed by physical page number. Each entry contains the page's access tags, the corresponding virtual page number, a pointer to the page's block access fault handler address table, a per-page protocol data pointer, and the home node identifier. Each bus transaction is checked against the tags stored in the RTLB. On a block access fault, the RTLB latches the access type (load or store), the block's access tag, the block's virtual address (formed from the stored virtual page number and the physical page offset), and the remaining fields from the RTLB entry in a buffer. (The RTLB's name comes from this ability to generate the block's virtual address from its physical address.) Dispatch code on the protocol processor uses the access type, access tag, and handler address table pointer to identify and invoke the appropriate block access fault handler function registered by the user. The virtual address, protocol data pointer, and home node identifier are passed as arguments to the handler.

A bus transaction may reference a page whose entry is not resident in the RTLB. In this case, the RTLB defers the transaction and retrieves the entry from a memory-resident page

table. To increase the RTLB's coverage, an entry may indicate a region larger than a page that does not require block access checks, such as text or kernel areas.

Protocol software manipulates the access tag values by reading and writing an RTLB-supported shadow space. Tags are modified in the RTLB entry and written back to memory on replacement. An update to a non-resident entry first loads the entry from memory. Values written to the shadow space may specify that, in addition to the tag change, the memory block data should be transferred atomically to or from the network queues, as described in Section 2.2.4. By specifying a null tag change, software can use these commands to perform simple block-sized network DMA transfers. These operations, as well as other tag changes requiring a bus transaction (see Section 3.1.3), are handled in conjunction with the block transfer unit.

3.2.3 Block transfer unit and block buffer

The block transfer unit and block buffer help move data blocks efficiently into and out of the Typhoon device. Their services play a role in supporting Tempest's access control and messaging functions. The block transfer unit copies data blocks directly between main memory, the block buffer, and the network send and receive queues. Copies involving main memory take advantage of burst transfers on the memory bus. In coordination with the RTLB, the unit uses coherent memory operations to implement access tag downgrades and the atomic block transfer and access tag change operations.

The block buffer functions as a flexible write buffer for data being copied to main memory. It is a 256-byte direct-mapped coherent data cache, organized as four cache blocks with two 32-byte subblocks each, matching the compute processor's cache block structure. The block transfer unit writes data destined for main memory into the block buffer, where it is tagged with the destination address and marked as modified. The buffer may perform a bus invalidation to guarantee that it has an exclusive copy with respect to the local coherence protocol. If the block's access tag is Invalid at the time of the write, as is typical when data is received from the network, the access control logic guarantees that no

copies are cached, so the bus invalidation can be skipped. Because only entire blocks are written, the buffer never fetches the previous version as a standard cache would. To keep replacement-induced writebacks off the critical path, the buffer autonomously flushes modified blocks to memory in the background.

Efficient data transfer on the completion of remote misses is an important benefit of the block buffer. Data from the response message is copied into the block buffer without generating any bus traffic. When the main CPU retries the faulting access, the block is fetched directly from the buffer (as a cache-to-cache transfer) without waiting for the data to be written to memory.

3.2.4 Handler dispatch

Dispatch hardware accelerates the invocation of user handlers in response to message arrivals and block access faults. Four memory-mapped registers provide a program counter and three handler arguments. The scheduling loop simply loads these four values into the processor's register file, placing the arguments in the conventional argument-passing registers, and jumps to the PC. Hardware populates the dispatch registers from the BAF buffer, if a block access fault has occurred, or from the network interface receive queue, if a message is waiting there. If no events are pending, the dispatch hardware stalls the register loads until an event occurs.

3.2.5 Primary CPU access

Most Tempest functions can be invoked both inside and outside of protocol handlers. Computation threads executing on the primary CPU may send messages and manipulate access tags. Typhoon provides a second send queue in the network interface to allow the primary CPU to construct messages concurrently with the protocol processor. The two send queues arbitrate for a single network port. To support tag manipulations from the primary CPU, the RTLB's shadow space is accessible via the bus interface as well.

3.3 Decoupled hardware support for Tempest

Typhoon provides competitive high-end performance for Tempest, as will be shown by the simulation results in Section 3.4, but it requires a complex piece of custom hardware that is expensive to design and manufacture. Designers can reduce the cost and complexity of Tempest support—at the expense of some performance—by decoupling Typhoon's integrated components and replacing them with off-the-shelf parts, when available. This section describes two decoupled designs: Typhoon-1 and Typhoon-0. Both designs replace Typhoon's integrated protocol processor with a general-purpose off-the-shelf CPU. Typhoon-1 uses a custom device which integrates access control logic and a network interface. Typhoon-0 also replaces the network interface with an off-the-shelf device, leaving the access control logic as its only custom component.

A significant portion of Typhoon's complexity is due to the integrated protocol processor. Even if the basic design for the processor and its caches is recycled from another source, the cache bus and memory bus interface must be extended or redesigned to accommodate the additional components, and these components must deal with concurrent accesses from the internal and external busses. Also, the die space consumed by the processor and caches contributes significantly to the manufacturing cost. In contrast, an off-the-shelf CPU is a mass-produced part that simply plugs into the memory bus. Because Typhoon's protocol processor is constrained by die size and faces the additional design delay of component integration, the off-the-shelf processor is likely to have higher raw performance as well.

Decoupling the protocol processor from the other Tempest-specific hardware creates an opportunity as well. Instead of dedicating a processor, protocol processing tasks can be shared among all of the processors. In effect, a virtual protocol processor is bound dynamically, as needed, to a physical CPU. When there are no protocol events to handle, all of a node's processors can work on the application's primary computation. Falsafi and Wood [FW96b] show that this dynamic model is often more efficient than dedicating a protocol

processor, particularly when there are only two processors per node. Both the Typhoon-1 and Typhoon-0 hardware designs are capable of supporting either model. However, to provide a direct performance comparison with Typhoon's dedicated, integrated protocol processor, this dissertation assumes that the decoupled systems dedicate a protocol processor on each node.

This section begins with three topics that apply to both decoupled designs. Section 3.3.1 describes *cacheable control registers*, a novel technique that reduces the overhead of communicating between an off-the-shelf CPU and a bus-based hardware device. Section 3.3.2 describes how the systems use hardware-generated program counters to accelerate handler dispatch. Section 3.3.3 discusses the access control implementation shared by Typhoon-1 and Typhoon-0, including the specifics of dispatching block access fault handlers. Like Typhoon, these systems use the snooping approach described in Section 3.1.3; however, the decoupling of the protocol processor indicates an implementation other than Typhoon's RTLB. Finally, Sections 3.3.4 and 3.3.5 discuss the specifics of Typhoon-1 and Typhoon-0, respectively.

3.3.1 Cacheable control registers

The drawback to decoupling the protocol processor is its effect on performance: the processor must communicate with the other components across the memory bus, which is both slower than an on-chip interconnect and subject to contention. Both decoupled systems 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 poll the register efficiently.

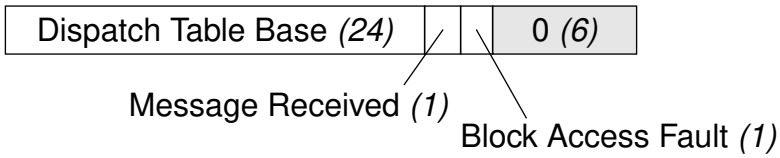

Figure 3-7. Dispatch program counter format. Parenthesized numbers are field widths in bits.

- 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.

Both Typhoon-1 and Typhoon-0 use a cacheable control register, the *dispatch register*, to accelerate the invocation of protocol handlers. The dispatch register leverages both cacheable control register features, allowing the protocol processor to poll efficiently for events and transferring several words of event information in one cache block (see Section 3.3.3).

The Typhoon-0 design includes a second cacheable register—its block buffer, described in Section 3.3.5. This cacheable register lets the protocol processor fetch the buffer's contents—a full block of data—in a single burst transfer.

3.3.2 Handler dispatch

Unlike Typhoon, the decoupled systems do not map events directly to user handlers in hardware. As described in the following section, their simplified access control hardware does not store enough information to generate either the fault handler's address or its arguments. Also, Typhoon-0's independent network interface prevents that system's custom hardware from dispatching message handlers directly. Each event requires a small amount of dispatch software to determine the appropriate user handler address and set up the handler's arguments.

In both decoupled systems, custom hardware helps the protocol processor efficiently check for events and invoke the appropriate dispatch software. The user arranges event dispatch code in a table, much like a processor trap vector table. Hardware forms a pro-

gram counter within this table by concatenating an offset to the user-specified base address [HJ92] (see Figure 3-7). The offset is a bit vector indicating the set of pending events. Four offsets encode the two possible events (message arrivals and block access faults).[1] The offset is shifted up six bits, allowing up to sixty-four bytes (sixteen SPARC instructions) for each table entry.

Instead of testing device status bits explicitly, the protocol processor polls by simply loading and jumping to the program counter address. The initial code table entry contains the polling loop; when no events are pending, the offset is zero and the indirect jump returns to the top of the loop.

This dispatch hardware also supports systems without dedicated protocol processors, as discussed earlier. In these systems, the initial code table entry contains a return instruction rather than the polling loop. A thread polls for events in the midst of other computation by performing an indirect call on the dispatch PC. Control returns to the computation immediately if no events are pending. Each poll expands code size by as few as two instructions—a load and an indirect call—assuming that a CPU register contains the device register's address and that the call's delay slot can be filled with useful work. (On SPARC version 8 systems, this sequence has the additional advantage that it does not modify the processor's condition codes, which are costly to save and restore.) The run-time overhead of the poll includes the cost of the control transfer and the execution of the return instruction in the dispatch table.

3.3.3 Access control

Typhoon's RTLB is not necessarily the best access control structure for a decoupled system. The RTLB both enforces fine-grain access control and caches the per-page data needed by a block access fault handler. The latter feature is useful primarily because the RTLB is tightly coupled with the protocol processor. Decoupling also reduces the benefit

1. The Typhoon-0 prototype adds two software-controlled event types, for a total of 16 offsets.

of integrating these two functions—i.e., combining the access tag and per-page data look-ups, and using the stored virtual page number to perform the full reverse translation in hardware—because the overhead of sending the event notification across the memory bus dominates the handler invocation latency. Instead, the decoupled systems separate these functions, enforcing access control in custom hardware but caching handler data in the protocol processor's data cache. The larger size of the off-the-shelf processor's data cache, relative to the one afforded by Typhoon's integrated device, mitigates the performance impact of storing this additional data.

In both decoupled systems, the access control device maintains only the two-bit access tags for each memory block. The device stores the tags for all of physical memory in an on-board SRAM array indexed directly by physical address. (For a 32-byte block size, one Mbyte of SRAM holds the tags for 128 Mbytes of physical memory.) As in Typhoon, a shadow space provides protected user-level tag access.

An inverted page table in cacheable main memory stores the same per-page information cached by each Typhoon RTLB entry—the virtual page number, a pointer to the page's block access fault handler address table, a protocol data pointer, and the home node identifier. Although this table occupies virtual memory proportional to the amount of installed physical memory, only the portions containing valid entries are allocated and mapped.

On a block access fault, software obtains the physical page number from the access control hardware and uses it to index the appropriate inverted page table entry. From there, it combines the virtual page number with the hardware-supplied page offset to form the virtual address, then selects and invokes the appropriate handler. Two features accelerate this process. First, the cacheable dispatch register transfers all the needed information from the hardware device in a single burst: the physical address of the block on which the fault occurred, the access type (read or write), and the block's tag value. Second, the hardware formats this data to accelerate fault handling, as shown in Figure 3-8. For example, the physical page number is in a separate word and is pre-shifted to form an index into the

inverted page table. From the detection of a block access fault to the invocation of the appropriate Tempest user handler requires only eight SPARC instructions (detailed in Figure 3-9).

To avoid expensive switches between processor privilege levels, the dispatch software is unprivileged, user-level code, just like the Tempest handlers it invokes. Exposing physical addresses to the user does not compromise security, because the addresses are only meaningful as indices for the inverted page table. If a virtual page is remapped to a different physical page, the operating system transparently relocates the corresponding table entry. However, after the user obtains a physical address from the hardware, the operating system should not remap the physical page until the dispatch software has retrieved the information from the inverted page table. By restricting the valid operations on and the lifetime of addresses retrieved from the hardware, the operating system can detect that the physical address is in use and either avoid remapping the page in that interval or restart the translation sequence after completing the remap [BRE92].

3.3.4 Typhoon-1

The Typhoon-1 design divides its Tempest support into two parts: a general-purpose off-the-shelf CPU and a separate device that combines the network interface and access control logic, as shown in Figure 3-10. The network interface queues, block transfer unit, and block buffer are identical to the corresponding Typhoon components, except that they connect only to the memory bus interface. Unlike Typhoon, the device provides only one network send queue; software must guarantee mutual exclusion between conflicting accesses from the protocol and computation threads.

The access control logic operates as described in Section 3.3.3. It implements a shadow space functionally identical to that supported by Typhoon's RTLB. As in Typhoon, the integrated block transfer unit and network interface cooperate to support atomic block transfer and access tag change operations. The block buffer holds data to be written to memory and accelerates the completion of remote misses.

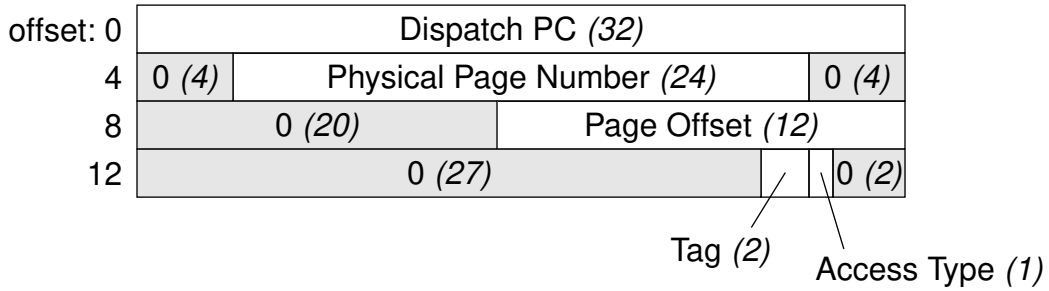

Figure 3-8. Dispatch register layout for block access faults. Parenthesized numbers are field widths in bits. The access control device aligns fault information to accelerate the handler dispatch code (see Figure 3-9). Because this register is cacheable, these four words are transferred in a single bus transaction.

Local registers are initialized as follows:
`%l5` – inverted page table base
`%l6` – inverted page table base + 8
`%l7` – device dispatch register base

Event polling code:

```
ldd  [%l7+0], %l0 .... load from dispatch register:
                           dispatch PC → %l0
                           fault physical page number (shifted) → %l1
jmp  %l0 ............. jump to event-specific dispatch code
```

Block access fault dispatch code:

```
ldd  [%l7+8], %l2 .... load from dispatch register:
                           fault page offset → %l2
                           fault tag/access type (shifted) → %l3
ldd  [%l5+%l1], %o0.. load from inverted page table:
                           virtual page number → %o0
                           user protocol data pointer → %o1
ldd  [%l6+%l1], %o2.. load from inverted page table:
                           home node ID → %o2
                           handler table ptr → %o3
ld   [%o3+%l3], %o3.. load from handler table:
                           user handler function ptr → %o3
or   %o0, %l2, %o0... fault virtual address (virtual page number | offset) → %o0
call %o3 ............. call user handler; arguments are:
                           %o0 – fault virtual address
                           %o1 – user protocol data pointer
                           %o2 – home node ID
```

Figure 3-9. Dispatch code for block access faults. The SPARC `ldd` instruction loads a 64-bit doubleword into two adjacent 32-bit registers. In practice, the instructions are reordered to fill delay slots and minimize load-use stalls. Assuming cache hits, these eight instructions take ten cycles on the Ross HyperSPARC, a dual-issue processor with a one-cycle load-use delay.

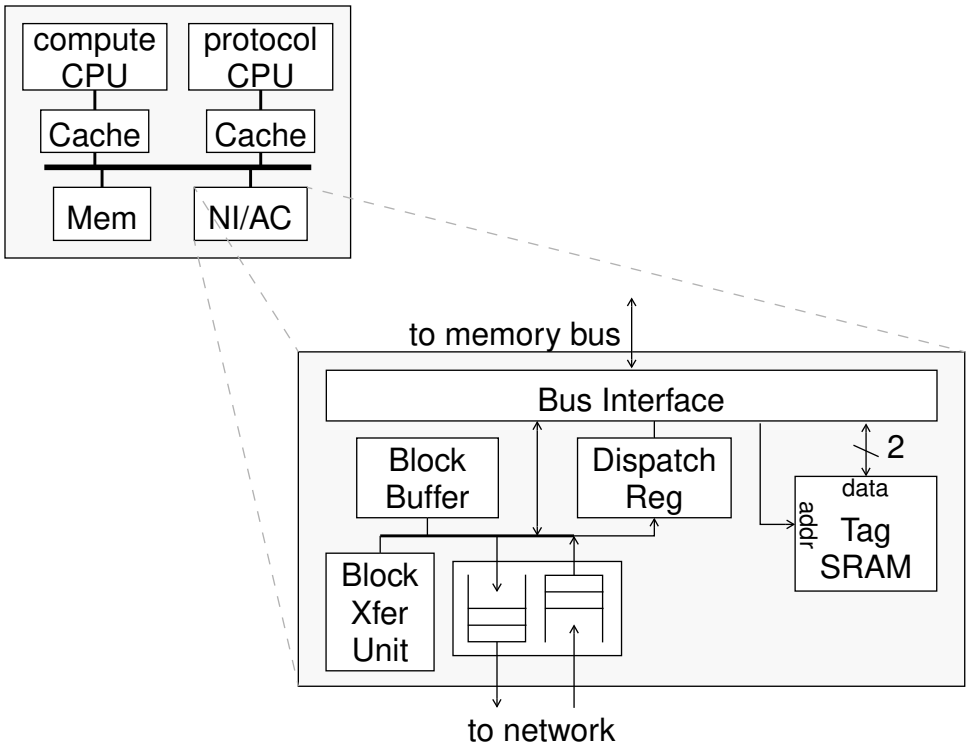

Figure 3-10. Typhoon-1 node, including a block diagram of the network interface/access control device. Control logic, other than the block transfer unit, is not shown.

The Typhoon-1 device's cacheable dispatch register integrates dispatch of access fault and message handlers. For either event type, the device supplies enough information in the dispatch register block to invoke the appropriate user handler with arguments. When a block access fault occurs, the device provides fault information as described in Section 3.3.3. If a received message is indicated, the other words in the block contain the message's source, its length, the sender-specified handler PC, and the first word of the body (used for the cache block virtual address in a typical coherence protocol). Because the register block can supply fault or message information, but not both, the hardware must select one event when both types are outstanding. To avoid starving local misses under heavy load, block access faults take precedence over received messages.

The dispatch mechanism must guarantee that exactly one handler is invoked for each event. The cacheable dispatch register cannot use clear-on-read semantics—the standard solution for this situation—because the processor must read multiple words from the dispatch block; these reads may result in multiple fetches from the device if the block is replaced from the cache, which could occur at any time due to conflicts or interrupts. Instead, Typhoon-1 uses a circular queue of four dispatch registers. The protocol processor polls a register, starting with the first, until an event occurs. After the event is handled, it continues polling using the next register in the queue. The access to the new dispatch register informs the device that the previous event has been handled. The device invalidates the old register so it is ready for use the next time around the queue. In the unlikely case that the device is unable to invalidate old register copies as quickly as the protocol processor consumes new ones, the device may stall dispatch register requests until it catches up.

3.3.5 Typhoon-0

The protocol processor is not the only Tempest component that can potentially be purchased off the shelf. Low-latency commercial networks such as Myricom's Myrinet [BCF+95] and DEC's Memory Channel [Gil96] are emerging on the market. These proprietary interconnects interface to standard I/O busses such as PCI. To take advantage of these off-the-shelf networks, Typhoon-0 separates Tempest support functions across three devices: a protocol processor, a network interface, and a fine-grain access control device. The first two components are off-the-shelf parts, while only the third is custom hardware. Figure 3-11 shows a block diagram of a Typhoon-0 node. Chapter 4 describes a prototype implementation of Typhoon-0, including a custom FPGA-based access control device, which demonstrates the feasibility and relative simplicity of this design.

Typhoon-0's access control device implements the two-bit directory described in Section 3.3.3. It also provides a shadow space for user-level tag manipulation. A read from the shadow space returns the corresponding tag value; a write modifies the tag. When access to a block is downgraded (e.g., from Writable to Invalid), the device issues a read-

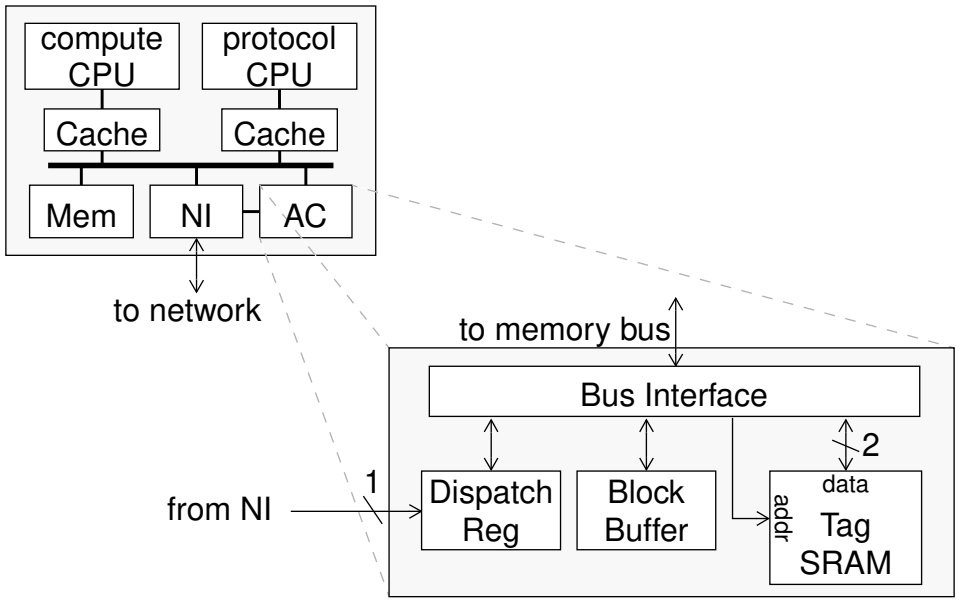

Figure 3-11. Typhoon-0 node, including a block diagram of the access control device. The device's control logic is not shown. The network interface is shown on the memory bus, but may be attached to an I/O bus instead.

invalidate bus operation to invalidate any cached copies and retrieve the current version, which may reside only in a processor's cache.

The block buffer (shown in Figure 3-11) holds the data returned by the latest read-invalidate. The block buffer is a cacheable register, so a processor can read its contents in a single burst. Because Typhoon-0's block buffer is not involved in transfers from the network to main memory, it is much simpler than that of Typhoon or Typhoon-1: it holds only one block, resides at a fixed address and does not have an address tag.

Atomic block transfer and access tag change operations—implemented in hardware in Typhoon and Typhoon-1—require software-controlled sequences of suboperations in Typhoon-0. These sequences must be designed carefully to avoid the race conditions described in Section 2.2.4. For example, to combine a block send with a tag change from Writable to Invalid, the processor first changes the tag, initiating a read-invalidate from the access control device. The processor then synchronizes with the device, using an uncached

load, to guarantee that the read-invalidate has completed. Finally, it copies the contents of the block buffer to the network interface. Similarly, when combining a block receive with a tag change from Invalid to Writable or ReadOnly, the processor writes the data via an uncached alias that bypasses the access tag check, then upgrades the tag.

The Typhoon-0 access control device provides a dispatch register similar to that of Typhoon-1. Of course, unlike Typhoon-1, this single-purpose device cannot fully integrate the dispatch of message and block access fault handlers. However, to avoid having the protocol processor poll the access control device and the network interface separately, the dispatch register can indicate message arrivals as well as access faults. If the network interface provides an accessible message arrival interrupt signal, it can drive an input on the access control device, as shown in Figure 3-11. This input signal controls a status bit in the dispatch register. To remain independent of the semantics of the NI's interrupt line, the dispatch register sets the message status bit on any transition of the input signal; software clears the bit after it has drained the NI queue. Even with this optimization, the dispatch register indicates only the existence of a message; the NI must be accessed explicitly to determine the message handler to invoke.

Unlike Typhoon-1, Typhoon-0's hardware does not prioritize events. When both block access fault and message events are pending, the dispatch register sets both status bits. Event prioritization is left to software, which handles block access faults first. After an event is handled, a write to an uncached register clears the corresponding status bit. Because the access control device supports only a single dispatch register, the processor must perform an uncached load to guarantee that the stale copy has been invalidated before it resumes polling.

3.4 Performance

This section compares the performance of these three designs via simulation. I first describe the simulation parameters and methodology, then present results for a simple microbenchmark and a set of application macrobenchmarks. Section 3.4.3 examines the

impact of network latency on application performance, and Section 3.4.4 examines the impact of integrating a lower-performance protocol processor in Typhoon.

The nodes of the simulated systems are based on the technology used for the Typhoon-0 prototype. Each node has a 200 MHz dual-issue SPARC processor with a 1 Mbyte direct-mapped data cache with 64-byte address blocks and 32-byte subblocks.[1] The instruction cache is not modeled; all instruction references are treated as hits. The instruction latencies, issue rules, and memory hierarchy are modeled after the Ross HyperSPARC [Ros93, Sho94].

A 50 MHz MBus, as used in the Sun SPARCStation 20, connects the processor(s), memory, access control and network interface devices within each node. The MBus is a 64-bit, multiplexed address/data bus that maintains coherence on 32-byte blocks using a MOESI protocol [Sun91]. On a cache miss, main memory returns the critical doubleword 140 ns (seven bus cycles or 28 processor cycles) after the MBus request is issued, followed by the remaining doublewords in consecutive bus cycles. Miss detection, processor/bus clock synchronization, and bus arbitration add 11-14 processor cycles to the total miss latency. The bus simulation accounts fully for occupancy, contention, and arbitration delays; the model is sufficiently detailed and accurate that the same simulator was used for initial functional design of the Typhoon-0 prototype's access control device.

On a block access fault, the access control logic inhibits the memory controller and gives the requesting processor an MBus *relinquish and retry* response, forcing the processor to rearbitrate for the bus. The access control device masks the arbiter to keep the processor off the bus until the access can be completed [LLG+92]. Although this technique cannot be implemented on an unmodified SPARCstation 20, its performance is representative of more recent systems which support deferred responses, either explicitly (like the Intel P6 [Gwe95]) or using a split-transaction bus.

1. These processor parameters reflect announced technology that could be installed in the prototype system; because the prototype was assembled at an earlier date, and under cost constraints, it uses 66 MHz processors with 256 Kbyte caches.

Timing parameters for the Typhoon-0 and Typhoon-1 access control devices are taken from the 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.

To equalize the comparison with Typhoon-0 and Typhoon-1, which store all the access tags on the access control device, the Typhoon RTLB is assumed to be large enough to map all the active shared pages on each node. Because simulation limits the size of the data sets, it is unlikely that replacement overheads due to a finite RTLB would affect the results significantly.

The network interface queues transfer up to 64 bits per cycle, at the bus clock rate in Typhoon-1 and Typhoon-0 and at the protocol processor clock rate in Typhoon. Each node may have at most four messages outstanding to each other node. The simulated Typhoon-0's network interface queues are located in 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 measured results from the CM-5: seven bus cycles for reads and three for writes.

Network contention is modeled at the interfaces, but not internally. As a result, the network wire latency, measured from the injection of the tail at the sending network interface to the arrival of the head at the receiving interface, is constant. To emphasize the performance impact of DSM support, the default latency is a fairly aggressive 0.5 µs (100 processor cycles). Although dedicated MPP interconnects may surpass this speed, current off-the-shelf networks are typically slower by an order of magnitude or more. Section 3.4.3 examines the overall performance impact of higher latency networks.

The systems have identical hardware support for barrier synchronization. As each node arrives at a barrier, it sets a bit in a bus device register and spins on a second bit, which the

hardware sets after all nodes have arrived. The latency from the last arrival to notification matches the one-way network latency. The notification bit resides in a cacheable control register so that processors can spin-wait without consuming bus bandwidth. Although this barrier hardware is inspired by MPPs such as the CM-5 [LAD+92] and Cray T3D [KS93], PAPERS [DCMM94] demonstrates how it can be added inexpensively to a cluster of PCs or workstations. None of the benchmarks in Section 3.4.2 performs a significant amount of barrier synchronization, so the absence of this feature would not noticeably affect the results.

To isolate the effects of decoupling, Typhoon-0, Typhoon-1, and the integrated Typhoon all use a dedicated protocol 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 used more efficiently by dynamically scheduling protocol handlers and computation across both processors (see page 55). 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. Section 3.4.4 examines the effect of using a slower processor in the integrated Typhoon device.

To quantify the performance impact of software protocols, a fourth system—an idealized implementation of Simple COMA [HSL94]—is included as a baseline. This system is similar to Typhoon, but replaces the protocol processor with a hardwired engine implementing a full-map invalidation-based coherence protocol. The idealized protocol engine processes each access fault or message event with zero overhead, including manipulation of protocol state and the injection of an arbitrary number of messages. Events are processed at a maximum rate of 200 MHz. Messages observe latency due to network transport, potential queueing at the controller, and fetching data over the MBus. Due to the structure of the simulator, messages observe an additional cycle of pipelined latency between arrival and processing.

To obtain results, application codes are compiled and linked with portable software protocols (written in C using the Tempest interface) and platform-specific Tempest runtime software, exactly as they would be for an actual implementation. A rewriting tool (based on EEL [LS95]) processes the resulting SPARC binaries, replacing memory accesses with calls to the simulator and adding instrumentation to count instruction execution cycles. Direct execution of the modified binaries drives the detailed discrete-event simulator. To enable larger systems and data sets, the system nodes are simulated in parallel on a Thinking Machines CM-5 using a conservative, synchronous parallel simulation algorithm based on the Wisconsin Wind Tunnel [RHL+93].

3.4.1 Microbenchmark

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

The results are presented in Table 3.1. The 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 Stanford

FLASH designers report remote read miss latencies of 1.11 and 1.45 µs, depending on whether the data is dirty in the remote processor's cache [HKO+94].[1]

Table 3.1: Remote miss latency breakdown for simulated systems.

Location	Step	Latency (200 MHz cycles)			
		S-COMA	Typhoon	Typhoon-1	Typhoon-0
Caching node	detect HW 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
Network	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 from memory, change tag, send	48	48	122	293
Network	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 tag	0	12	20	261
	unmask CPU, reissue bus transaction	10	10	32	32
	fetch data, resume	31	31	31	31
Totals	200 MHz CPU cycles	301	401	807	1461
	50 MHz bus cycles	76	101	202	366
	microseconds	1.5	2.0	4.0	7.3
	bus transactions	3	3	16	36

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

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.

3.4.2 Macrobenchmarks

To determine how these overheads translate into application performance, I simulated the six shared-memory applications described in Section 2.4. All benchmarks are written in C and were compiled with gcc version 2.6.3 at optimization level -O2. Table 3.2 summarizes the applications and indicates the data sets used.

Table 3.2: Benchmark applications and data sets.

Benchmark	Application domain	Primary data structure(s)	Data set	Frag. overhead
appbt	CFD	3D array	32x32x32 array, 5 iterations	58%
Barnes	hierarchical N-body	oct-tree	16,384 bodies, dtime=0.025, tstop=0.075 (4 iterations)	373%
DSMC	Monte Carlo particle-in-cell	cell array, particle list	48,000 particles in 9720 cells, increasing to 72,000 particles, 400 iterations	68%
EM3D	electro-magnetics	static bipartite graph	192,000 nodes, degree 5, 5% remote edges, 20 iterations	175%
moldyn	molecular dynamics	molecule list, interaction list	8788 particles, 30 iterations, interaction list rebuilt once	16%
unstructured	CFD	static mesh	9428 nodes, 59863 edges, 5864 faces, 5 iterations	129%

Table 3.2 also reports each application's fragmentation overhead—the amount of additional physical memory consumed because of Tempest's page-granularity allocation. Specifically, the table reports the number of allocated but unused physical memory blocks, expressed as a percentage of the number of allocated and used blocks. A block is consid-

ered used if it contained valid data at any point in the program. These values were obtained from simulations using 64-byte blocks, 4096-byte pages, and unlimited physical memory at each node. Restricting the available physical memory could force page replacements, reducing the actual physical memory overhead.

Although I simulated the full applications, I report results for only the second and following computation iterations to focus on the portion of execution where a production version will spend most of its time. For most of the applications, iteration times are very regular, so meaningful results require only a few iterations. There are two exceptions. In moldyn, the molecule interaction list is occasionally rebuilt, resulting in an iteration that is an order of magnitude longer than the others; I 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 I arbitrarily chose to run for 400 iterations. As the number of particles increases, the speedup also increases, but very slowly; I do not expect results for a longer run to be qualitatively different.

Although parallel simulation on the CM-5 allows larger benchmark runs than sequential simulation, working set sizes are still too small to cause replacements in each node's main-memory–based remote data cache. Because no page replacement and very little initial page allocation occurs during the measured interval, the Simple COMA results provide an approximate upper bound for the performance of hardwired CC-NUMA systems similar to DASH [LLG+92].

All of the benchmarks except EM3D use the first-touch migrate-once scheme described on page 23. EM3D explicitly allocates the graph so that writes are always to local pages.

Figure 3-12 shows speedups for the transparent–shared-memory (TSM) applications on 32-node systems. For the Tempest systems, this requires simply linking with the standard protocol library, which implements the same sequentially consistent full-map invalidation

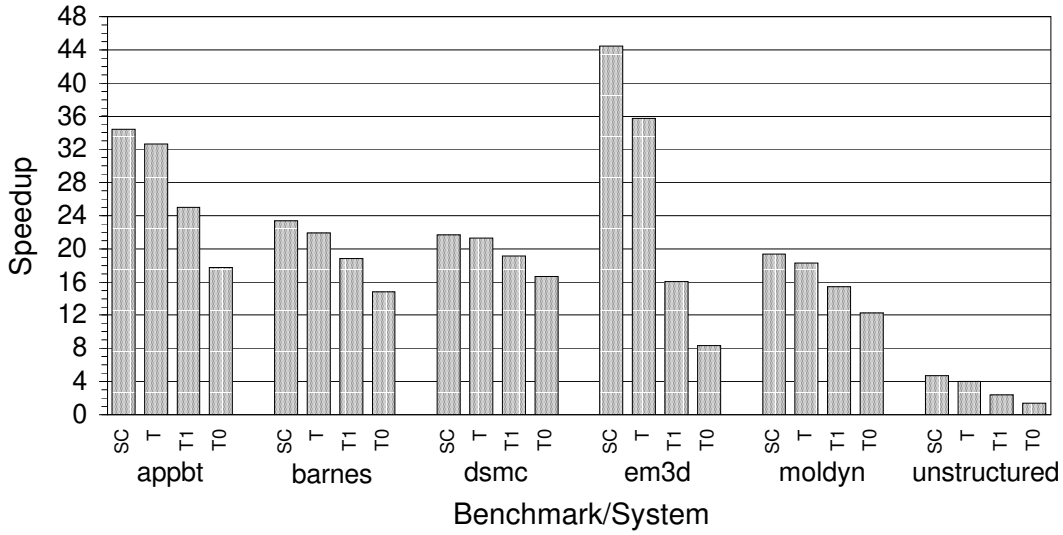

Figure 3-12. Application speedups for transparent shared memory on 32-node systems. SC=Simple COMA, T=Typhoon, T1=Typhoon-1, T0=Typhoon-0.

protocol used in the Simple COMA system. These speedups are relative to the best available sequential version on a workstation identical to one node of the parallel system. Although these applications do not take advantage of the Tempest interface, they have been tuned for performance within the TSM model; see Falsafi et al. [FLR+94] and Mukherjee et al. [MSH+95] for details. These benchmarks achieve speedups of 19 or better on the Simple COMA system, with the exception of unstructured at under five. (The large speedup for appbt is due to cache effects; one third of the sequential execution is spent waiting for cache misses.)

I 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, I only present results for 64-byte coherence blocks. The 64-byte block size is within 10% of the best performance for most cases. The only exceptions are EM3D, which is 10–40% faster with 512 byte blocks, and unstructured, for which Typhoon and Typhoon-1 are 12–14% faster at 256 bytes.

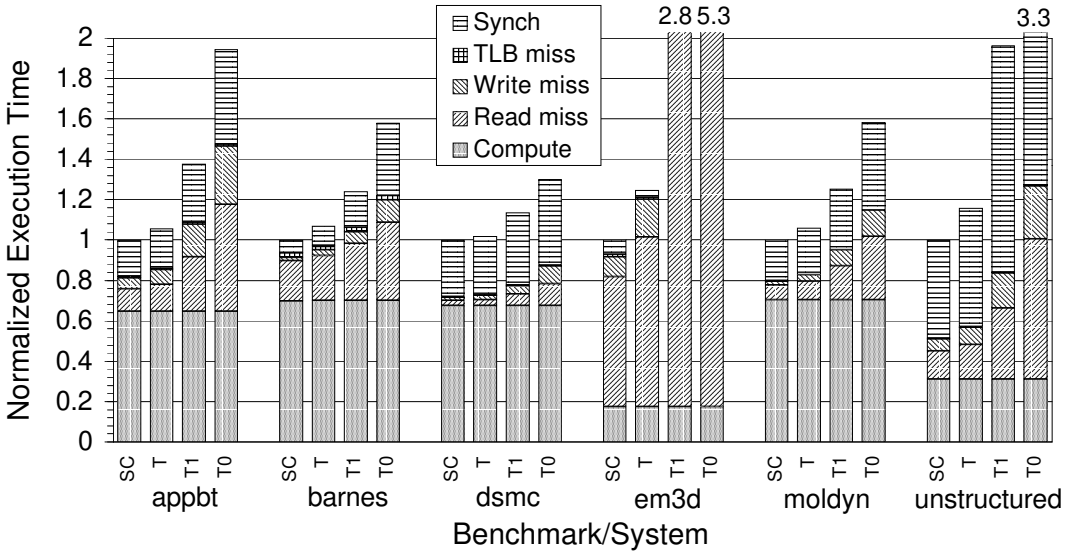

Figure 3-13. Execution time breakdown for transparent shared memory. SC=Simple COMA, T=Typhoon, T1=Typhoon-1, T0=Typhoon-0.

To facilitate system comparisons, Figure 3-13 presents execution times for the TSM benchmarks normalized to the Simple COMA system, breaking out the time spent by the main processor on computation and on read, write, TLB, and synchronization stalls. Two of the benchmarks—EM3D and unstructured—have poor processor efficiency, spending less than a third of the time computing even on the Simple COMA system. Unstructured fails to produce much speedup for any platform. EM3D achieves speedup on the integrated systems because its large (20 MB) data set thrashes the uniprocessor's cache and TLB, but fits in the caches and TLBs of the parallel system. The uniprocessor execution spends over 87% of its time waiting for the memory system—45% on cache misses and 42% on TLB misses.

Even on these unmodified applications, Typhoon's performance is at most 25% less than the idealized Simple COMA's, and is under 10% for the four benchmarks with higher efficiencies. Typhoon demonstrates that the flexibility of Tempest's user-level software protocol processing is compatible with high performance.

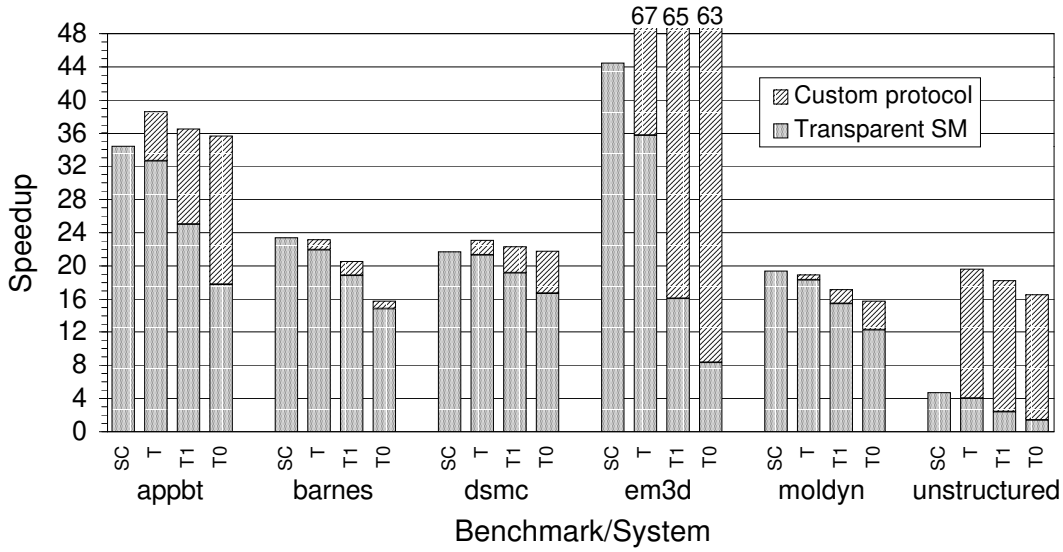

Figure 3-14. Application speedups on 32-node systems, including application-specific protocols. SC=Simple COMA, T=Typhoon, T1=Typhoon-1, T0=Typhoon-0.

Moving from the integrated to the decoupled designs increases the total stall time significantly—by 41–213% for Typhoon-1 and by 92–522% for Typhoon-0 relative to Simple COMA. However, the effect of this increase on bottom-line performance varies according to the contribution of the stall times to the overall execution. For the benchmarks with relatively high efficiency—appbt, Barnes, DSMC, and moldyn—the effect of increased overheads is mitigated by their smaller overall contribution. On these applications, Typhoon-1 is 11–30% slower than Typhoon, and Typhoon-0 is 28–84% slower than Typhoon (13–38% and 30–94% slower than Simple COMA, respectively). On the low-efficiency benchmarks—EM3D and unstructured—the decoupled designs show their weakness, turning in performance a factor of two or more slower than the integrated systems.

Figure 3-14 repeats Figure 3-12, adding the speedups for the Tempest-optimized versions of the benchmarks described in Section 2.4. Because the Simple COMA system uses a hardwired protocol engine, it is incapable of supporting the Tempest-specific optimizations. These application-specific protocols were written and optimized for a very different system—Blizzard-E [SFL+94] 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 custom protocols still provide some improvement over transparent shared memory. Two show dramatic improvement even on Typhoon—86% for EM3D and 384% for unstructured—causing them to outperform the Simple COMA system by nearly the same margins as well. (The large absolute speedup for EM3D is due to the memory system effects mentioned above.)

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 transparent shared memory on any system, including Simple COMA. Of course, there are other methods to improve the performance of the TSM programs without resorting to custom protocols—for example, adding prefetch instructions or using a weaker consistency model. Nevertheless, these results indicate significant potential for custom protocols in some situations.

Figure 3-15 breaks down the execution times for the Tempest-optimized applications, normalized to the TSM version on Simple COMA as in Figure 3-13. Because time that the main processor spends doing explicit message passing is counted as computation, most of the application-specific protocols actually increase the amount of computation over the original version; computation time varies on the different platforms due to varying message-passing overheads. Reductions in the stall times compensate for these increases. Unstructured is an exception to this pattern: both the computation and stall times decrease significantly. This effect is due to the optimized version's more efficient reduction phase. Where the original application performs a global reduction on an array of values, the optimized code sums only the non-zero contributions for each value (see Section 2.4.6). Although the primary intent of this optimization is to eliminate the communication of the zero values, it also eliminates the computation involved in summing them into the result.

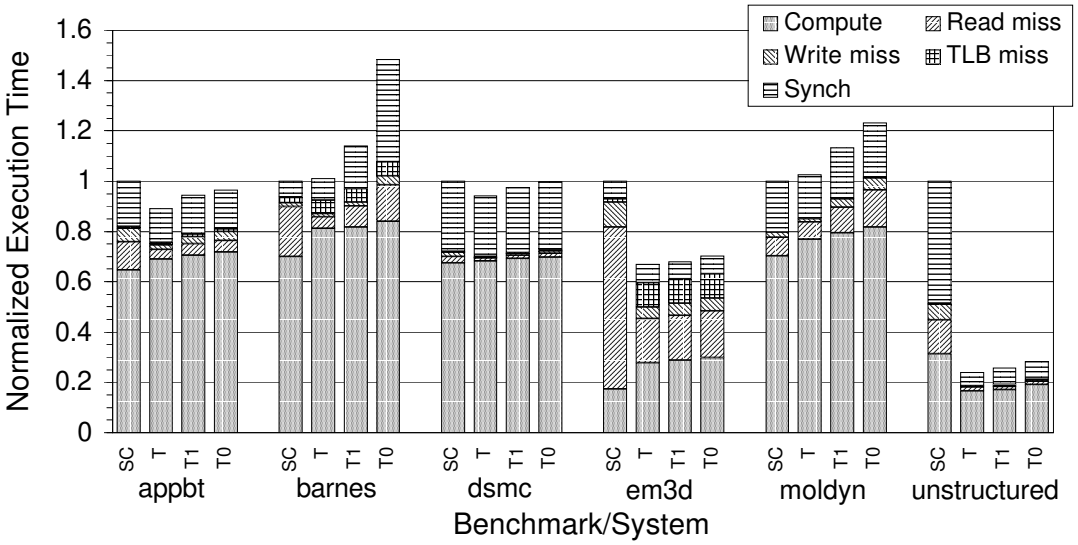

Figure 3-15. Execution time breakdown for the application-specific protocols, normalized to transparent shared memory on Simple COMA. SC=Simple COMA, T=Typhoon, T1=Typhoon-1, T0=Typhoon-0.

TLB misses, which are not a factor for any of the transparent–shared-memory benchmarks, are a noticeable component of execution time for two of the Tempest-optimized codes (roughly 6% for Barnes and 10% for EM3D). These costs merely reflect differences in reference locality between the TSM and custom-protocol versions; it is not inherent to Tempest or the hardware platforms. The custom-protocol version of Barnes splits the body structure so that different protocols can be applied to different fields. As a result, the data for one body, which is contiguous in the original code, is spread across three structures on three different pages. The difference in TLB behavior for EM3D is due to a locality optimization in the TSM version that was not applied to the application-specific protocol version. This optimization, which pulls the writable data values out of the otherwise read-only graph structure and allocates them in a separate array, improves cache locality for remote references in the TSM version. The custom-protocol version leaves the writable values embedded in the graph nodes because (1) packing values from multiple graph nodes in a single cache block complicates dynamic sharing pattern detection and (2) the values are updated using a custom protocol, so spatial locality for remote misses is not a

performance issue. With some effort, this data restructuring could be applied to the custom-protocol EM3D, bringing its TLB performance in line with the TSM version.

Figure 3-15 also shows that the efficiency of each application on Simple COMA correlates inversely with the effectiveness of the application-specific protocols. Intuitively, the applications with higher overheads have more to gain by eliminating those overheads. The two benchmarks with very low efficiency, EM3D and unstructured, show impressive gains, while the improvements for DSMC, moldyn, and Barnes are smaller. Appbt straddles the fence: the custom protocol gains a factor of two on Typhoon-0 but only 18% on Typhoon.

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

3.4.3 Impact of network latency

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

Figures 3-16 and 3-17 show speedups for both versions of each application for one-way network latencies of 0.5, 2.5, 5, 25, and 50 µs (100, 500, 1,000, 5,000, and 10,000 processor cycles, respectively). The first group of columns in each graph (at 0.5 µs) corresponds to Figure 3-14.

Because the application-specific protocols reduce or eliminate stalling for demand misses, they generally tolerate large network latencies better than transparent shared memory. As the latency increases from 0.5 to 50 µs, the TSM benchmarks on the idealized Simple COMA system slow down by at least a factor of five and as much as a factor of 33. In contrast, the custom-protocol versions slow down by as little as 1% (for EM3D) and at worst a factor of 3.6 (for Barnes). As a result, the relative improvement provided by the application-specific protocols grows with increasing latency. For example, the Barnes application-specific protocol provides a speedup of only 6% over the TSM version on Typhoon at a 0.5 µs latency, but at a 50 µs latency this improvement grows to 80%.

Higher network latencies also reduce the performance difference between the systems, especially for transparent shared memory. Increasing the fraction of execution time due to the network itself diminishes the relative impact of other overheads.

3.4.4 Impact of protocol-processing performance

As discussed on page 68, the results presented thus far assume that Typhoon's integrated protocol processor is as powerful as the off-the-shelf general-purpose CPU used by Typhoon-1 and Typhoon-0. This assumption allows us to attribute all of the performance differences between these designs to the effects of decoupling. However, it is unrealistic; due to the additional design time required, it is probable that Typhoon's integrated processor will be slower than a contemporary off-the-shelf CPU.

To examine the effect of this assumption, I simulated two additional versions of Typhoon, changing the protocol processor to be two and four times slower than the compute CPU. Figure 3-18 shows the execution times for each of the TSM benchmarks on

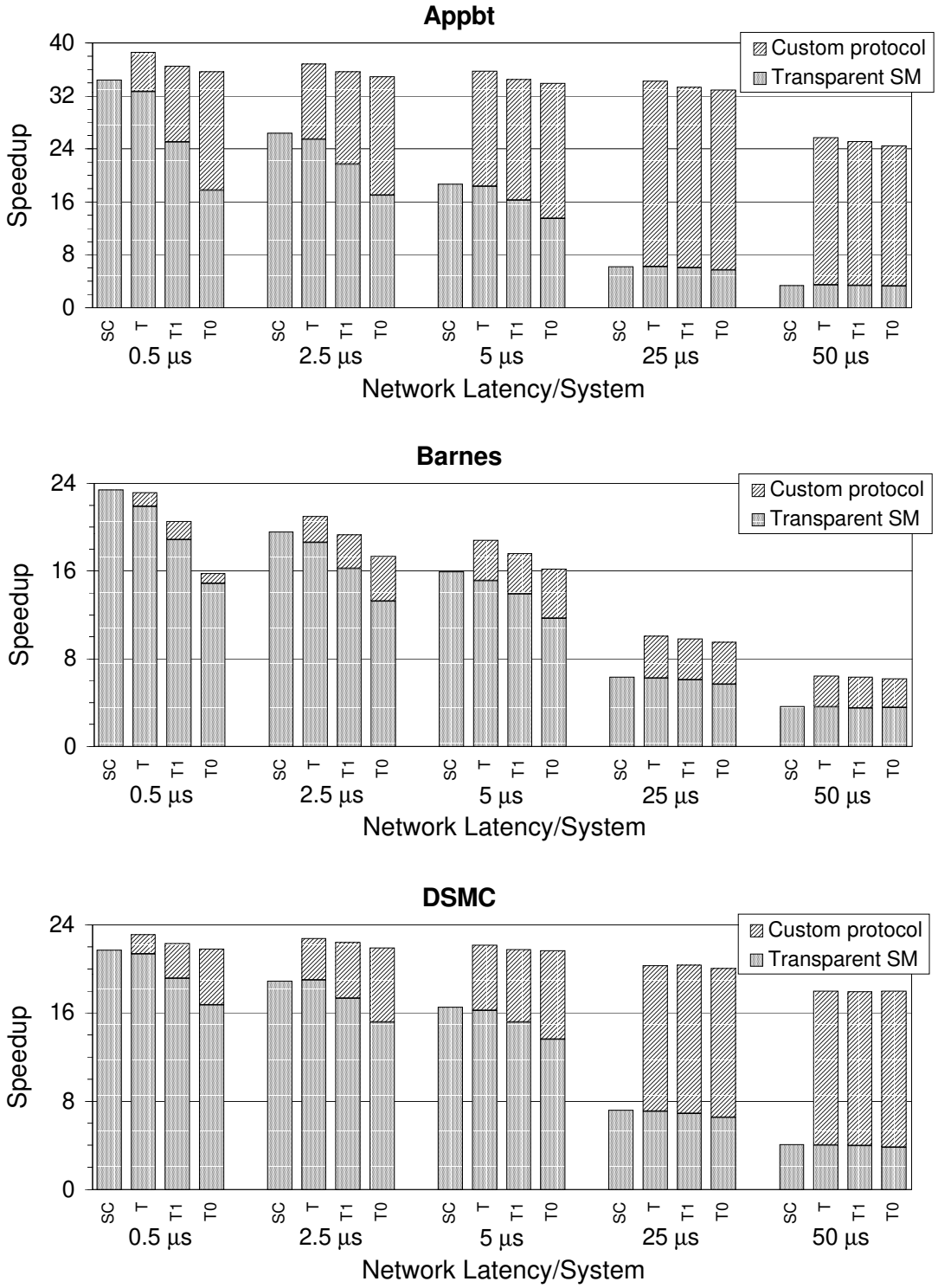

Figure 3-16. Speedups for appbt, Barnes, and DSMC on 32-node systems at various network latencies. SC=Simple COMA, T=Typhoon, T1=Typhoon-1, T0=Typhoon-0.

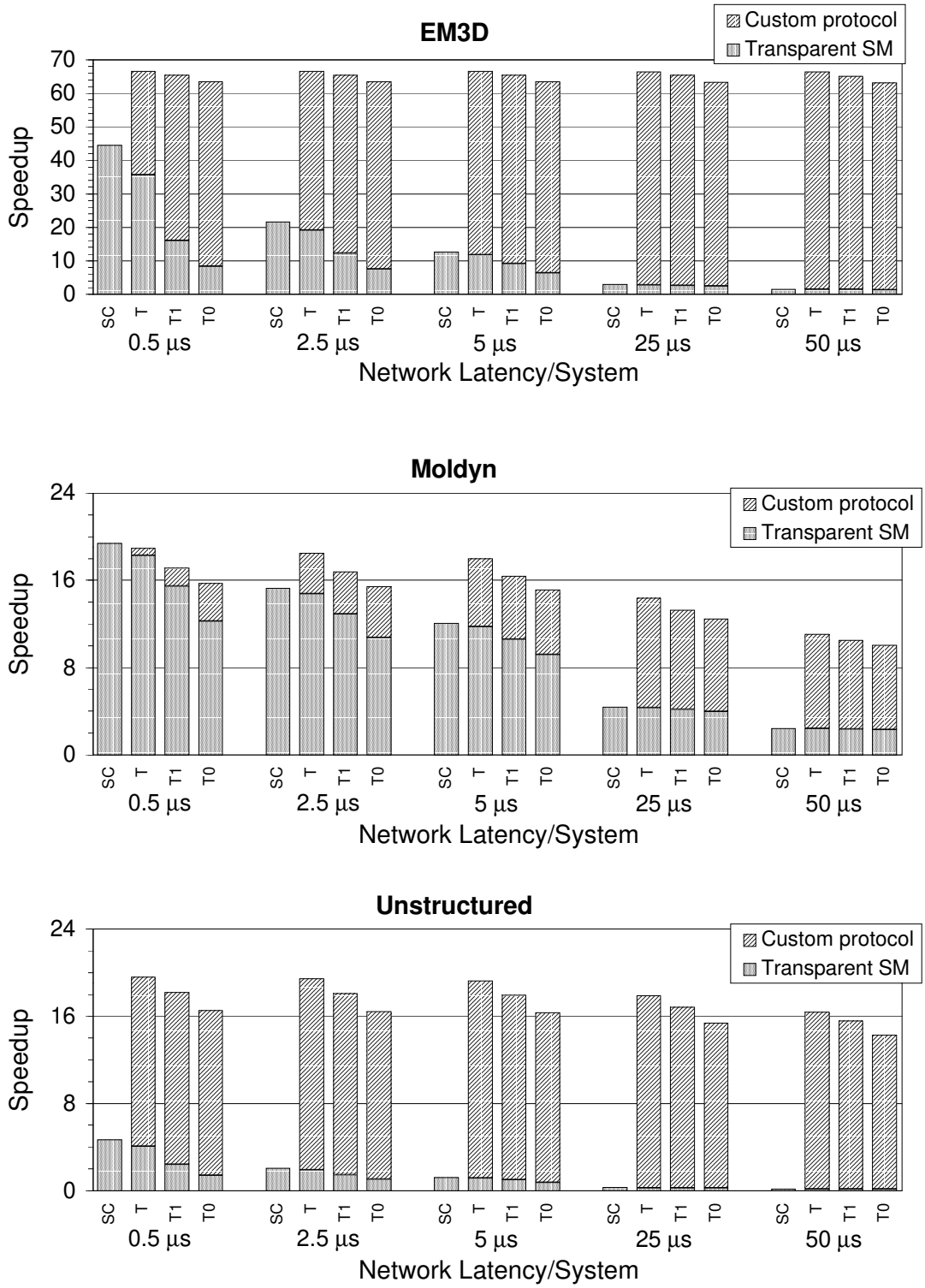

Figure 3-17. Speedups for EM3D, moldyn, and unstructured on 32-node systems at various network latencies. SC=Simple COMA, T=Typhoon, T1=Typhoon-1, T0=Typhoon-0.

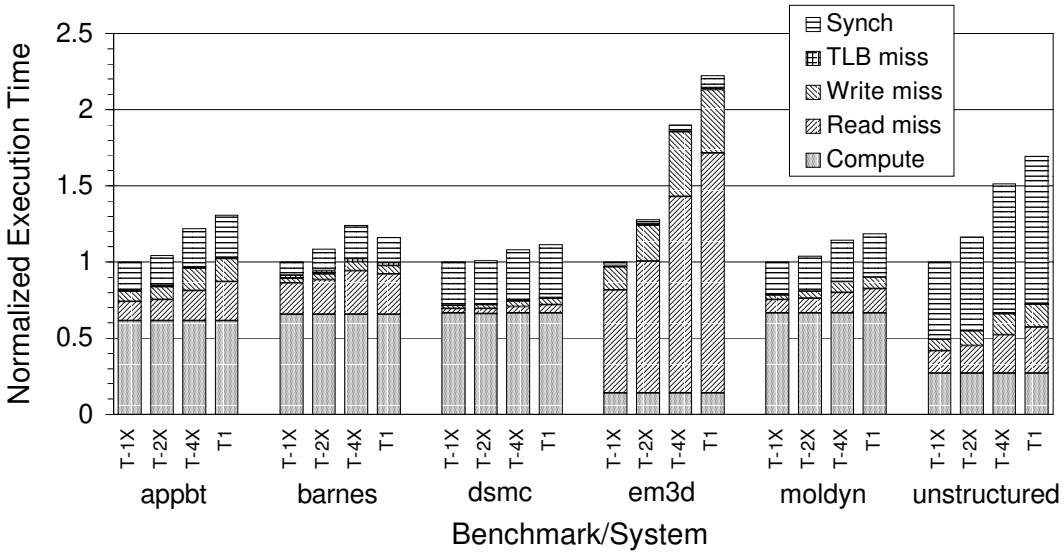

Figure 3-18. Execution time for transparent shared memory on Typhoon, varying the protocol processor speed. T-*n*X=Typhoon with protocol CPU *n* times slower than compute CPU, T1=Typhoon-1. Network latency is 0.5 µs.

these systems, normalized to the benchmark's execution time assuming a protocol processor identical to the compute processor (as reported in Figure 3-13 on page 74). While slowing the protocol processor by a factor of two reduces application performance by at most 28%, a factor of four processor slowdown reduces overall performance by up to 90%. Among the more efficient benchmarks—excluding EM3D and unstructured—application performance is reduced by at most 8% and 24%, respectively. For comparison, Figure 3-18 also includes the Typhoon-1 results from Figure 3-13. For most of the applications, Typhoon-1 is nearly as fast as the Typhoon version with the fourfold-slower protocol processor, and is faster for Barnes. These results suggest that Typhoon's protocol processor need not be as fast as the compute CPU, but at some point protocol processor speed does become a bottleneck. For the parameters used here, this point occurs when the protocol CPU is between two and four times slower than the compute CPU; in general, the required protocol CPU speed is probably dependent on network, memory, and bus bandwidths as well as the speed of the compute CPU.

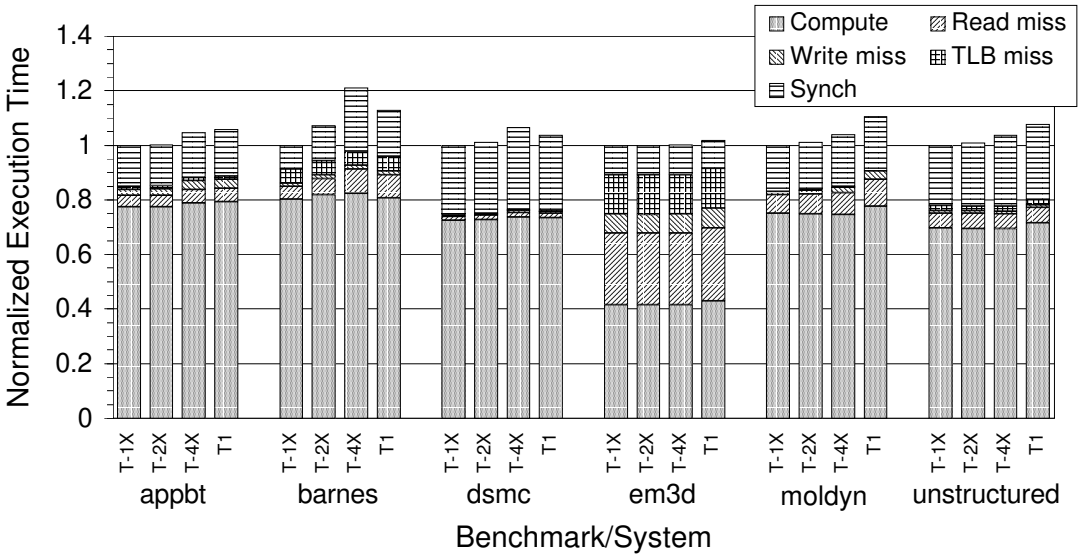

Figure 3-19. Execution time for the application-specific protocols on Typhoon, varying the protocol processor speed. T-*n*X=Typhoon with protocol CPU *n* times slower than compute CPU, T1=Typhoon-1. Network latency is 0.5 µs.

Figure 3-19 repeats Figure 3-18 using the application-specific protocol versions of the benchmarks. As we saw with the decoupled systems, the application-specific protocols mitigate the impact of increased overheads; the twofold and fourfold decreases in protocol processing power reduce overall performance by at most 7% and 21%, respectively; excluding the outlying Barnes, the reductions are at most 1% and 7%, respectively. Again, however, the protocol processor can become the bottleneck: the slowest Typhoon system is slower than Typhoon-1 for two of the applications (Barnes and DSMC) and nearly as slow for the others.

3.5 Related work

The designs in this chapter are related both to systems that implement Tempest using different techniques and to non-Tempest systems that use similar techniques. This section restricts its focus to implementation issues; Section 2.5 discusses related work in the context of the Tempest interface itself.

These systems do not represent the entire design space for Tempest support. In another publication [SFL+94], we list several alternate approaches to fine-grain access control, and describe two software implementations that run on an unmodified Thinking Machines CM-5 (and were later ported to a cluster of workstations [SFH+96]). The Blizzard-E system manipulates the memory controller's error correction codes (ECC) to force bus errors—inducing processor traps—on accesses to invalid blocks.[1] Read-only blocks are synthesized using page-level protection. Custom operating system extensions manipulate ECC via a memory controller diagnostic mode [RFW93, SFH+96]. I was involved in the design of an enhanced memory controller that extended ECC semantics to include a read-only state and provided limited user-level access [Nim93].

Another implementation we report, Blizzard-S [SFL+94], uses an executable editor to insert software access checks before each load and store. This approach is very portable, requiring no special hardware or operating system support. A variety of optimizations can lower the overhead of executing the access checks [SFH+96, SGT96]. Horowitz et al. [HMMS96] propose a processor feature that can be used to execute the access check code only on hardware cache misses.

A modified TLB on the compute processor can also support fine-grain access control. The IBM 801's lock bits [CM88] implement a form of TLB-based fine-grain access control, but with only two access states. MIT's M-Machine [FKD+95] provides all three of Tempest's states specifically to support DSM. Although this approach provides low overhead for both access checks and handler invocations, it introduces a coherence problem in multiprocessor nodes [ENCH96].

Tempest's protocol processing does not require a dedicated CPU. In both Blizzard systems on the CM-5, the single processor on each node executes both application and protocol code. To handle asynchronous message arrivals, the systems either use hardware

1. This technique, and the code that implements it, was taken from the Wisconsin Wind Tunnel [RHL+93], where it was used for direct-execution cache simulation.

interrupts or poll from the application code. In the latter case, the polling code is added automatically via binary editing. Even when multiple processors are available on each node, Falsafi and Wood [FW96b] show that dynamically scheduling application and protocol tasks is typically more efficient than dedicating a protocol processor (see page 55).

The designs in this chapter share several features with non-Tempest systems. DASH [LLG+92] also implements fine-grain access control for distributed shared memory using custom snooping logic that participates in an off-the-shelf multiprocessor's bus-based coherence protocol. Unlike the Tempest systems, which provide direct access to fine-grain access control from user-level software, DASH's snooping logic interfaces directly to the hardware state machines that implement the global coherence protocol. DASH also modified the multiprocessor node to gain control over bus arbitration. The simulations of Section 3.4 use this feature as well, but more recent bus designs—with split or deferred transactions—enable similar performance with no hardware changes to a commodity workstation node.

Typhoon's integrated device is similar to FLASH's MAGIC chip [KOH+94]. Both integrate a processor, a network interface, and access control. MAGIC integrates the memory controller as well, which provides greater control over bus transaction handling and the opportunity for higher bandwidth, but precludes off-the-shelf workstation nodes. Instead of using hardware access control to filter memory references, MAGIC runs a software handler, which incorporates the access control check, on every main memory request. MAGIC incorporates a number of hardware optimizations, including a message handling macropipeline and a separate datapath for memory and message data. These features reduce the latency and occupancy of software handlers [HHS+95], enabling high performance on transparent shared memory [HKO+94].

Although both FLASH and Tempest espouse software protocols, they embody different approaches to flexibility and performance. FLASH minimizes the performance impact on transparent shared memory by using aggressive hardware and restricting access to the sys-

tem's flexibility. In contrast, Tempest seeks primarily to maximize user-level flexibility. Although this flexibility may impact performance on demanding shared-memory applications (such as EM3D and unstructured), it enables custom protocols which can provide good performance on a wider range of implementations.

Like Typhoon-0 and Typhoon-1, Start-NG [CAA+95] uses an off-the-shelf protocol processor. This processor has two tightly coupled peripherals on the level-2 cache bus: a network interface and an address capture device (ACD). The ACD latches bus transactions in software-accessible registers, allowing software-based access control similar to FLASH. As discussed in Section 2.5, all protocol software is privileged in both FLASH and Start-NG, making them unable to directly support Tempest.

SHRIMP [BDFL96] and FLASH [HGDG94] use shadow space mappings to specify source and destination addresses for network data transfers, as do Typhoon-1 and Typhoon.[1] In SHRIMP's user-level DMA, the transfer forms a complete message and the sender specifies both the source and destination addresses. In the Typhoon systems, the transfer is one part of a larger message and the destination address is specified by the receiver's message handler (as in Alewife [KA93]). Because FLASH's shadow space is simply a technique for communicating addresses between the compute CPU and the embedded protocol processor, FLASH can implement a variety of DMA protocols. All these systems must guarantee that stale translations are never used for DMA operations. Because only local addresses are involved and the transfer size is limited to a single cache block, Typhoon and Typhoon-1 can use simple techniques such as FLASH's hold-off approach or SHRIMP's address register check.

3.6 Summary

This chapter describes three system designs—Typhoon, Typhoon-1, and Typhoon-0—that provide hardware support for the Tempest interface. All three use off-the-shelf work-

1. The shadow space was not part of the original Typhoon design [RLW94], but was retrofitted after the design of Typhoon-1 to support operations from the compute processor.

station nodes and enforce Tempest's fine-grain access control with custom bus-snooping logic. They differ in the extent to which they integrate this logic with two other Tempest support components, a network interface and a protocol processor. These different levels of integration place the systems at different points on the cost–performance spectrum.

Typhoon achieves high performance by integrating all three components on a single device. Simulation results show that Typhoon's performance for unmodified shared-memory applications is competitive with an idealized hardwired protocol engine—within 25% across six benchmarks on 32-node systems. The results also provide strong evidence for Tempest's flexibility: in spite of Typhoon's effectiveness for transparent shared memory, application-specific protocols sped up one benchmark by 86% and another by 384%—almost a factor of five.

Typhoon-1 simplifies Typhoon's integrated device by decoupling its most complex subcomponent, the protocol processor. Instead, an off-the-shelf general-purpose CPU executes protocol software and controls a custom network interface/access control device. A novel invention, a *cacheable control register*, lets the device and the processor communicate across the memory bus more efficiently than traditional uncached device registers. Typhoon-1 is no more than 30% slower than Typhoon for the four benchmarks with good processor efficiency (65% or better on the idealized hardwired system), but is roughly a factor of two slower for the two benchmarks with low efficiency. Using the custom-protocol versions of the benchmarks, which avoid demand misses and communicate more efficiently, reduces Typhoon-1's worst-case performance penalty to 13%.

Typhoon-0 decouples the Tempest support components further. Like Typhoon-1, it uses an off-the-shelf CPU for protocol processing; however, it also uses a generic off-the-shelf network interface. A standalone access control device is the only custom component. While this separation of components simplifies the design and reduces costs, some common operations require a large number of bus transactions to coordinate the devices. For transparent shared memory, Typhoon-0 is 28–84% slower than Typhoon for the four

benchmarks with higher processor efficiency, and three to four times slower for the low-efficiency benchmarks. Because Typhoon-0's overheads are higher, the custom protocols produce a larger performance improvement; using these benchmarks, Typhoon-0 is at most 47% slower than Typhoon.

The simulation results mentioned thus far assume a fairly low network latency (0.5 μs). If the network latency is increased, it dominates other overheads, deemphasizing the performance difference between the system designs. Because they generally replace demand fetches with direct updates, the custom-protocol benchmarks tolerate high network latencies much better than the transparent–shared-memory versions.

In addition to serving as a contrast to Typhoon and Typhoon-1, Typhoon-0 is simple enough to implement in an academic environment. In the next chapter, I describe a working hardware prototype of the Typhoon-0 system that demonstrates the feasibility of the designs discussed here.

Chapter 4

The Typhoon-0 Prototype

This chapter describes a prototype implementation of the Typhoon-0 system. This prototype demonstrates the feasibility of combining custom fine-grain access control logic with user-level protocol software—the common theme of the three designs presented in the previous chapter. In addition, the prototype's performance correlates with results from the simulator, strengthening the credibility of the results reported in the previous chapter.

Typhoon-0's decoupled design is ideal for prototyping: other than the fine-grain access control logic—implemented on a FPGA-based board called *Vortex*—the system is built entirely from off-the-shelf components. The relatively small effort required to construct the prototype—approximately two student-years—clearly demonstrates the advantage of this approach.

The first two sections of the chapter describe the hardware and software components of the Typhoon-0 prototype, respectively. Section 4.3 reports performance measurements from the prototype, and Section 4.4 compares some of these measurements with results obtained from the simulator. Section 4.5 summarizes the chapter.

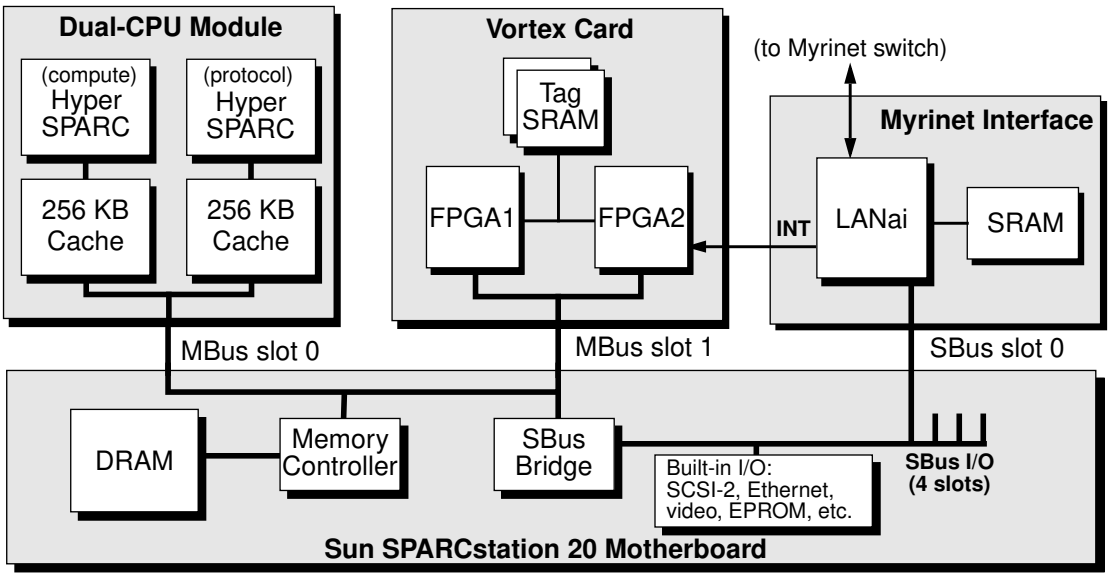

Figure 4-1. Block diagram of a Typhoon-0/COW node. The Myrinet LANai chip integrates a simple processor, a DMA engine, network transceivers, and most of the bus interface logic.

4.1 Hardware

The Typhoon-0 prototype builds on Wisconsin's Cluster Of Workstations (COW), infrastructure shared by several UW-Madison investigators researching cluster-based systems. The COW comprises forty Sun SPARCStation 20s (SS-20s) connected with a Myricom Myrinet [BCF+95] network (in addition to standard 10- and 100-Mbps Ethernet). Myrinet is a commercial local-area network derived from the high-speed, low-latency interconnect developed for the Caltech Mosaic machine, a research MPP.

Figure 4-1 shows a block diagram of a Typhoon-0/COW node. The SS-20 uses processor daughtercards that plug into either of two slots, to allow adding or upgrading processors in the field. Because these slots are intended for cache-coherent processors, they provide full access to the coherent processor–memory bus (the MBus [Sun91]). The prototype uses one of these slots for the Vortex fine-grain access control board. The other MBus slot holds a module with two CPUs, one for computation and one for protocol software. The COW's processor modules have two 66-MHz Ross HyperSPARC processors, each with a 256-KB unified cache.

Our original design for the prototype interconnected the nodes with a Thinking Machines CM-5E network. (The CM-5E was an enhanced version of the CM-5; from our perspective, the only significant difference was a greater maximum message length.) The CM-5E design included a single-chip MBus network interface connecting each node's SPARC processor to the custom interconnect. The only effort required for the prototype was to obtain a network in a stand-alone package and physically integrate the network interface chip and connectors onto the Vortex card. Unfortunately, at the time we began the prototype effort, Thinking Machines encountered financial problems that led to their exit from the hardware business. After this avenue closed, we selected the Myricom Myrinet as the fastest commercially available interconnect at the time.

Each node connects to the Myrinet network using a standard Myrinet host interface card. This card plugs into the SBus, Sun's standard I/O Bus [Mas94]. Most of the interface's functionality—including a simple processor and a DMA engine—is contained on a single chip, the LANai. The card also has 128 KB of SRAM to store the LANai processor's code and data. The host processors can access the LANai's SRAM directly using uncached memory operations. However, the LANai can access host memory only via the DMA engine.

The Vortex card implements the Typhoon-0 access control device as described in Section 3.3.5, with one exception. Because the SS-20's bus arbiter does not allow masking of all processor bus requests,[1] Vortex cannot simply suspend a bus operation that conflicts with the access tags as was assumed in the previous chapter. Instead, it aborts these operations by returning an error acknowledgment, causing the memory reference instruction to fault. After the block access fault is handled, software reissues the instruction to resume computation.

1. The arbiter has a control register that masks requests from bus devices, but—presumably to prevent inadvertent deadlocks—processor 0's requests cannot be masked.

To keep design time short and costs low, Vortex uses FPGA technology rather than a custom ASIC or gate array. No contemporary FPGA was both dense enough to implement all the logic and capable of running at the 50 MHz MBus clock rate, so we partitioned the logic across two devices. Two standard 4Mx1 SRAMs store 4M two-bit tags; given MBus's 32-byte block size, this configuration supports up to 128 MB of physical memory. Rob Pfile's master's report [Pfi95] details the implementation of Vortex.

The Myrinet interface does not provide an accessible message interrupt line to drive the message status bit in Typhoon-0's dispatch register (see page 65). We improvised this feature by co-opting a software-programmable status LED. As we installed each Myrinet interface, we removed the LED and replaced it with a connector. A short wire routes the signal to the input on the Vortex board.

4.2 Software

Typhoon-0/COW's software starts with Solaris 2.4 and adds drivers and other system code to support the Vortex and Myrinet devices. A user-level run-time library builds on these system components and exports the Tempest interface. Job control software derived from DJM [Min95] lets users initiate and control parallel jobs across multiple COW nodes.

As with the hardware, Typhoon-0/COW leverages existing or shared components for much of its software support. The off-the-shelf Solaris 2.4 system is of course the most significant piece. The Myrinet interface software is based on Berkeley's LANai Active Messages [CLMY96]; modifications to make it more suitable for Tempest are shared with other software-only COW Tempest implementations [SFH+96]. The parallel job management software is part of the larger COW infrastructure.

This section focuses on software specific to the Typhoon-0 prototype. The first four subsections cover software issues related to Vortex: the first describes the steps required to configure a Typhoon-0/COW node each time it boots, the second relates how a user pro-

cess works with the device driver to access Vortex, the third reports other needed Solaris modifications, and the fourth discusses system issues that have not yet been fully addressed in the prototype. The final subsection briefly describes the Myrinet interface software.

4.2.1 Vortex configuration

To enforce access control, "coherent invalidate" (CI) transactions on ReadOnly blocks must fail, invoking a block access fault handler. (CIs result from writes to blocks that are in a shared state in a processor's hardware cache.) The MBus specifies that a single device (typically the memory controller) acknowledges all CIs; in Typhoon-0/COW, Vortex takes over this task so that it can provide the proper response based on the tag state. Switching responsibility from the memory controller to Vortex in a running system is nearly impossible: an unacknowledged CI—or one with two acknowledgments—will crash the system. Instead, we disable the memory controller's CI acking, and enable Vortex's, when the system is powered up or reset—before the processors enable their caches. Using the SS-20's OpenBoot features [Sun94], we store a command sequence in nonvolatile memory and set it to execute on each hard reset.

The Vortex device driver performs the remaining configuration tasks when it is loaded. First, the driver checks whether the installed memory configuration is compatible with Vortex and, if so, sets up Vortex's address mapping appropriately.[1] It then maps the supervisor (privileged) device registers into the kernel address space, initializes the SRAM tags to ReadWrite, and enables Vortex's tag checking.

4.2.2 Process setup

To use Vortex for fine-grain access control, a process first opens the Vortex device driver. The process then maps the user device registers into its address space via an `mmap()` call.

1. In the SS-20, the set of populated physical addresses is not necessarily contiguous and depends on the size of the installed SIMMs. To simplify the logic that maps physical addresses to tag array indices, Vortex supports only systems where all SIMMs are the same size.

In addition to installing the mapping, the driver informs Vortex of the virtual page number where the cacheable control registers (the dispatch register and the block buffer) are mapped. Because the HyperSPARCs use virtually indexed caches larger than the page size, Vortex's cacheable control register invalidations must provide the low-order bits of this virtual page number. (The MBus provides eight bits in the transaction specifier for this purpose.)

To limit the impact of fine-grain access control on the kernel, the driver allows modification of access control tags only on a special memory segment which it manages directly [RFW93]. Although the standard text, static data, and stack segments cannot be tagged, this model supports the PARMACS macros [BBD+87] used by our benchmark applications, which assume that shared memory must be dynamically allocated via a special version of `malloc()`.

The process initializes the taggable memory segment via a Vortex driver `ioctl()`. The driver creates three same-sized segments in the process's virtual address space: one that contains the primary (cached) memory mapping, one that maps the corresponding Vortex shadow space pages for tag access and manipulation (see page 49), and a third—an alias of the first—that provides uncached memory access for bypassing tag checks (see page 65). These segments are initially empty; the process allocates physical pages as needed via another `ioctl()`. The process specifies the initial value for the new page's access tags and the primary virtual address. The driver acquires a physical page frame, initializes its tags, maps it at the specified address in the primary segment, and creates shadow space and uncached mappings at the same offset within the other virtual segments.

As described in Section 3.3.3, dispatch software—part of the user-level runtime library—uses an inverted page table to translate the physical addresses provided by Vortex on a block access fault to virtual addresses for the user-level protocol handlers. The current implementation pins taggable pages in physical memory, so we avoid the complexities of page remapping discussed in Section 3.3.3; for simplicity, the user-level runtime

library maintains the inverted page table. The table is a sparse demand-allocated array, set up using an `mmap()` on `/dev/zero`. When allocating a taggable page, the library uses the physical page number—returned by the device driver—to populate the appropriate entry. To provide access to the same per-page information given only a virtual address—as when a remote protocol request is received—a similar sparse array maps virtual page numbers to their inverted page table entries.

4.2.3 Other Solaris modifications

Ideally, all OS support for Vortex should be encapsulated cleanly in a device driver written to published kernel interfaces. Unfortunately, Solaris—like any other commercial Unix variant—emphasizes monolithic featurism over flexibility and efficiency, so three modifications outside the purview of a standard device driver were required. Despite the unusual nature of these changes, we were able to implement them in a dynamically loadable module; however, this module is not portable to other Solaris versions and cannot be unloaded.[1]

First, a new memory segment driver supports the three segments provided by the Vortex driver. In Solaris, a memory segment driver is a piece of code that manages a region of a virtual address space in a particular fashion. For example, one driver manages mapped files while another manages mapped devices. Although Solaris provides a variety of memory segment drivers, none have the flexibility needed to manage three related segments in a coherent fashion.

Second, modified trap vector code accelerates the invocation of user-level synchronous trap handlers. As described earlier, Vortex responds with an error acknowledgment on block access faults, causing a synchronous exception on the issuing (compute) processor. The compute processor waits for the protocol processor to indicate that the miss handling is complete, then restarts the aborted memory access instruction. The standard Unix signal

1. These changes—and the module that implements them—are shared with Blizzard-E/COW, a Tempest implementation that uses ECC for fine-grain access control as described on page 84.

interface supports this scenario, albeit slowly: in Solaris 2.4, the trip from the exception through the signal handler and back to the faulting instruction takes roughly 100 µs—longer than servicing a simple remote miss over the Myrinet. In contrast, using simple, known techniques [RFW93, TL94], our modified kernel delivers a synchronous exception to a user handler in under 5 µs. The suspended thread can be resumed without going through the kernel. (In the version 8 SPARC architecture, user code cannot resume a faulted instruction in a control transfer delay slot. The prototype avoids this scenario by using an EEL-based tool [LS95] to move loads and stores out of delay slots. The version 9 SPARC architecture corrects this oversight.)

Finally, an unsavory workaround gives the device driver control over the cacheability of mappings. The Solaris kernel controls page cacheability internally, assuming that all device registers are uncacheable and that all main memory is cacheable. Both Vortex's cacheable registers and the uncached tag-bypass memory alias violate this assumption. By manipulating internal kernel function pointers, the driver intercepts calls to the function that writes entries into the hardware page tables. At each call, the driver examines the mapping represented by the page table entry; if it corresponds to either of our special cases, it adjusts the cacheability attribute before performing the write.

4.2.4 System issues

This section discusses four system issues that have not yet been fully addressed in the prototype: paging tagged memory, kernel and DMA accesses to tagged memory, and multiprogramming the Vortex device. Although the solutions here have not been implemented fully, this discussion should provide a framework for future work in this area.

Conceptually, tagged memory can be paged simply by moving each page's tag values to and from disk along with the memory data. To better fit in the kernel's page-oriented virtual memory framework, the driver can allocate additional memory—*backing tag* memory—to serve as backing store for tag values. This memory can be charged to the process allocating tagged memory, but it need not be mapped into the process's address space.

When a tagged page is written to disk, its tags are copied to a backing tag page; after all the pages supported by a backing tag page have been written to disk, the backing tag page can be written out as well. Similarly, when a tagged page is read back in, the tags are restored from the backing tag page (after reading it from disk also, if necessary). For Typhoon-0/COW, one backing tag page could support 128 tagged memory pages. Unfortunately, Solaris does not provide hooks to notify a segment or device driver when a page is written out, so our current implementation pins tagged pages in physical memory.

The handling of kernel accesses to tagged memory is a more difficult issue. For the kernel to have the same view of shared memory as the user, kernel references that conflict with access tags should cause block access faults handled by the user-level protocol. However, this solution would leave a suspended kernel thread waiting on an unreliable user-level protocol. Instead, we can provide a library of wrapper functions for system calls that pass pointers into the kernel. If the pointer arguments reference shared memory, the wrappers copy the data into local-memory buffers; any needed block access faults will occur during this user-level copy operation. To protect against inadvertent or malicious use of tagged-memory pointer arguments, the internal kernel routines used to access user memory can detect access tag conflicts and return an error code, as they would for any other invalid pointer.

DMA accesses to tagged memory raise a similar issue, although with two significant differences. First, the desired semantics are unclear; for example, it is inappropriate to invoke the coherence protocol when paging tagged memory to disk. Second, the DMA hardware restricts the available options: a DMA access that receives a bus error acknowledgment typically crashes the machine. Because paging is unimplemented, the first issue is unresolved. Vortex addresses the second problem by suppressing the access tag check on bus transactions initiated by the MBus DMA controller.

Although Vortex was designed to work with one user process at a time, software could virtualize the device to enable multiprogramming. The kernel can read and write all Vor-

tex's state via the supervisor register space, so the user registers can be timeshared among multiple processes (as done with the CM-5 network interface). A more serious obstacle is that the user-accessible dispatch register signals every block access fault, regardless of the processor (or thread) that caused it. To avoid having one process erroneously handle another's block access faults, either a process's threads must be strictly gang scheduled (within each node) or direct access to Vortex's user registers must be disallowed. In the latter case, the kernel could notify a process of block access faults via the bus error exception handler. Because the block buffer—a shared resource—is modified on tag downgrades, the kernel must deny users direct write access to the tag space as well. The kernel would mediate use of the block buffer, most likely requiring a kernel call for every tag change.

4.2.5 Messaging

This section briefly describes the Myrinet software used in Typhoon-0/COW. This software is derived from Berkeley's LANai Active Messages [CLMY96]. Modifications and enhancements for Tempest are described by Schoinas et al. [SFH+96].

As with most high-speed networks, the Myrinet's primary bottleneck is the transfer from the application through the interface and onto the switching fabric. From the hardware perspective, the SBus-based Myrinet achieves the lowest latency when the main processor accesses the shared SRAM on the Myrinet interface card; the highest bandwidth comes when the LANai (the embedded processor on the interface card) accesses main memory using its DMA engine. As in the Berkeley implementation, our software allocates send and receive queues in the shared SRAM. Each fixed-size queue entry holds header information, a small amount of message data, and an optional pointer to a larger message in main memory. Short messages achieve low latency because they pass entirely through the shared SRAM; larger messages enjoy the higher bandwidth of a DMA transfer.

To eliminate kernel intervention, the Myrinet device driver maps the shared SRAM directly into the process's address space. Due to the structure of the I/O MMU on the MBus/SBus bridge, the DMA engine can access only kernel virtual addresses. The driver

allocates a dedicated kernel buffer for the Myrinet and maps it into the process's address space as well. As with Vortex, this Myrinet configuration supports only one process at a time, although it could be timeshared in a gang-scheduled environment.

4.3 Performance

This section discusses the performance of the Typhoon-0 prototype. As in Section 3.4, I use simple microbenchmarks to illustrate system overheads, then examine a set of application macrobenchmarks. Finally, I compare the prototype's measured performance with simulation results.

4.3.1 Microbenchmarks

This section shows where the prototype spends its time when servicing a remote miss. To measure the individual operations, I collected traces of MBus activity using a logic analyzer attached to one node of the prototype system. The prototype ran a simple test program that generated three types of misses on 32-byte blocks: read misses, write misses, and write upgrades (writes to shared blocks). For each type of miss, I ignored the initial miss that involved allocation of the cache page, then used the fastest of at least a dozen samples to eliminate the effects of unnecessary cache and TLB misses.

The MBus traces allow a fairly accurate accounting of the activity on one node. (The HyperSPARC's write buffers introduce some approximation when attributing time to a specific task.) However, because only a single logic analyzer was available, I could not observe both the caching and home nodes for any one transaction. Instead, I ran the test program twice, reversing the role of the node attached to the analyzer. For a given type of transaction, subtracting the time spent on the home node from the latency observed on the caching node gives a value for the total network latency. For the write upgrade transaction, which involves two short messages, I estimated the one-way latency by taking half of the total network latency. Read and write misses involve a short request and a bulk (DMA)

response. For these, I assumed the request message had the same latency as in the write upgrade case, and attributed the remaining network latency to the response.

Table 4.1 presents timing breakdowns for a read miss and a write upgrade. Write miss timing differs from read miss timing in only one respect: because the home node does not keep a read-only copy on a write miss, the "write data to memory" and "change tag to ReadOnly" steps are eliminated, making write misses almost one microsecond faster than read misses. The write upgrade is significantly faster, primarily because the response message does not transfer data. The write upgrade also enjoys a slight (80 ns) advantage on the initial bus transaction; Vortex can abort an MBus invalidate faster than a read because it need not suppress the memory controller's response.

Table 4.1 shows that, for either transaction, nearly 70% of the latency is attributed to the network. This latency—spanning from the acknowledgment of the sending protocol processor's last write to the LANai memory until the receiving node's Vortex card starts to invalidate the status cacheable control register (CCR)—is over 12 µs for a short message and over 23 µs for a message involving a 32-byte DMA. The nodes communicate through a single Myrinet switch with a worst-case latency of 550 ns [BCF+95], so virtually all of this time is consumed by the SBus bridge and the LANai device. The primary culprit is control software running on the LANai processor—a 16-bit non-pipelined CISC processor clocked at 25 MHz. Newer versions of the Myrinet SBus interface card use a 32-bit processor with a lower CPI clocked at 50 MHz; these devices would significantly improve the prototype's performance.

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

Table 4.1: Remote miss latency breakdown for the Typhoon-0 prototype.

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

a. This operation is not on the critical path.
b. This time includes a TLB miss on the protocol processor (see text).

nals the message arrival by invalidating the status CCR. The DMA itself consumes about 0.5 µs on the MBus, roughly half of which is due to a TLB miss in the I/O MMU on the

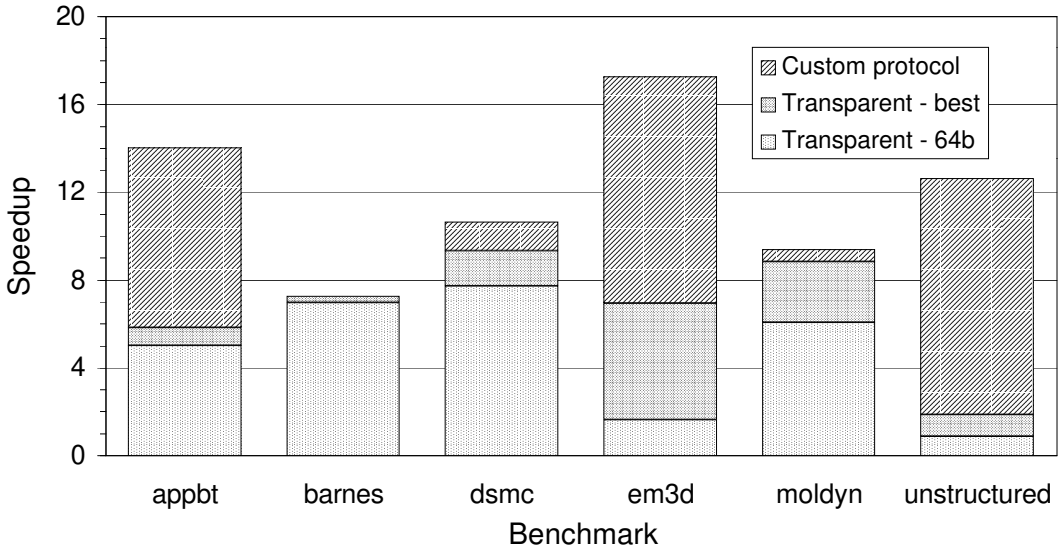

Figure 4-2. Application speedups on 16 nodes of the prototype system.

Sbus bridge. (The LANai interface treats the DMA region as a circular queue of pages, reserving a full page for each message, so accesses to the DMA region—from both the protocol processor and the SBus bridge—practically always involve a TLB miss.)

4.3.2 Macrobenchmarks

To measure the prototype's application-level performance, I ran the six applications from Section 2.4 using the same data sets as in the previous chapter. (Table 3.2 on page 71 summarizes the applications and data sets.) As in Section 3.4.2, the reported speedups are relative to the best available sequential version and exclude initialization and the first parallel iteration. To reduce variation from external factors, I ran each experiment three times and selected the fastest run.

Figure 4-2 displays speedups obtained on 16 nodes of the prototype. The shortest bars indicate the speedup using transparent shared memory with 64-byte blocks. The darker bars show additional speedup gained by selecting the best block size for the specific application (up to 2048 bytes)— but still using transparent shared memory. For these results,

appbt and Barnes use a block size of 128 bytes, DSMC uses 256 bytes, unstructured uses 1024 bytes, and EM3D and Moldyn use 2048 bytes. The hatched bars show speedups from the custom Tempest protocols described in Section 2.4—except for Barnes, for which the custom protocol is slightly slower than the transparent shared-memory version.

In general, Figure 4-2 confirms the results of Section 3.4.2: appbt, EM3D, and unstructured see a significant gain from custom protocols, while the other benchmarks do not. Unfortunately, it is difficult to compare Figure 4-2 with the results for larger network latencies in Section 3.4.3 due to a number of differences, including system size (16 vs. 32 nodes) and processor speed (66 vs. 200 MHz). The next section correlates measured prototype performance with results from the simulator by eliminating these discrepancies in the simulated system.

4.4 Comparison of measured and simulated results

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

To simulate the prototype, I made several modifications to the simulated Typhoon-0 system from the previous chapter. Most were simple:

- processor speed (200 MHz vs. 66 MHz),
- cache size (1 MB vs. 256 KB),
- block access fault suspend/resume (arbiter control vs. error abort/software restart), and
- barrier synchronization (hardware vs. software).

Simulating the Myrinet's performance was more involved. The simulations in the previous chapter assume a minimal, CM-5–like network interface; the host processor copies data in and out of hardware queues directly attached to the network. In contrast, the Myrinet interface is the dominant contributor to latency (as seen in the Section 4.3.1) and the

bottleneck for message throughput [CLMY96]. Rather than introduce detailed models of the SBus bridge and LANai device—with its processor and DMA engine—I approximated the effect by adding an occupancy component to the existing network interface model. Every message passing through the interface, in either direction, occupies a single server for a specified interval. Messages queue at the server in FIFO order. To simulate the overhead of DMA, messages longer than eight words incur a larger base occupancy plus a small additional per-word occupancy at both the sending and receiving interfaces. To mimic Myrinet's higher host CPU overhead—including copying data into and out of the DMA region, which is not explicitly simulated—I also increased the cost of reads and writes to the simulated NI's registers. I ran a message-passing microbenchmark on the prototype to select network parameter values—occupancies, register access costs, and mean node-to-node latency.

The simulated Typhoon-0 and the prototype also differ in their strategy for message buffering. Tempest implementations must be prepared to buffer message data in main memory to avoid deadlock (see Section 2.2.1). For historical reasons, the simulated system buffers at the sender but the prototype buffers messages at the receiver. Although this difference may seem subtle—and is moot if traffic is low enough that no buffering occurs—it results in a 20% discrepancy for the custom-protocol version of appbt. In this program, as each node produces a new column of values, it sends them to its neighbor in a series of short messages—enough to trigger the credit-based flow-control mechanism—before going on to compute the next column. The simulated system's sender buffering allows the producer node's compute processor to buffer its outgoing messages and continue computing, with the protocol processor injecting the buffered messages as flow-control credits become available. In contrast, the prototype's receiver buffering forces the producer to block while the consumer processes its messages. Rather than modifying either system—a significant effort—I modified the application to send data in fewer, longer messages to avoid triggering flow-control stalls.

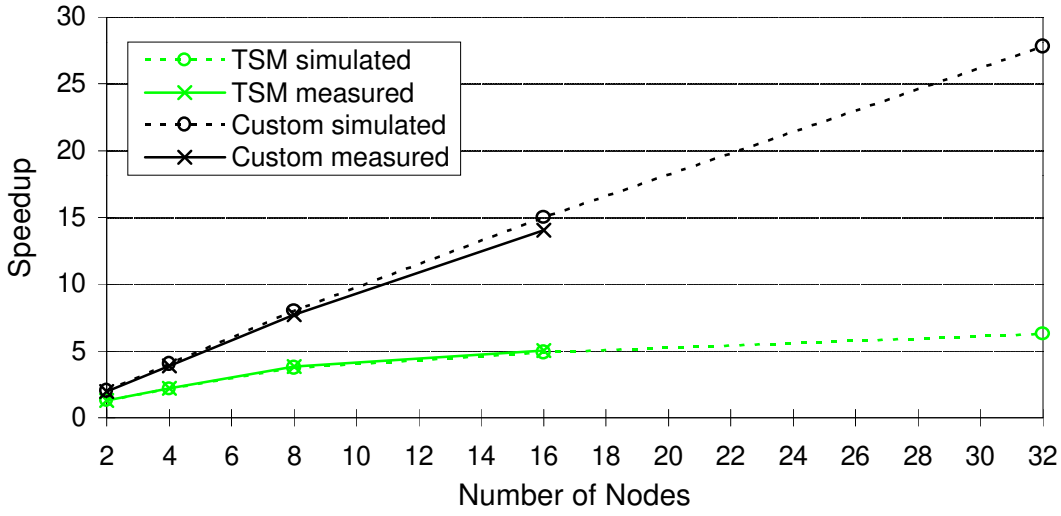

Figure 4-3. Simulated and measured speedups for Appbt.

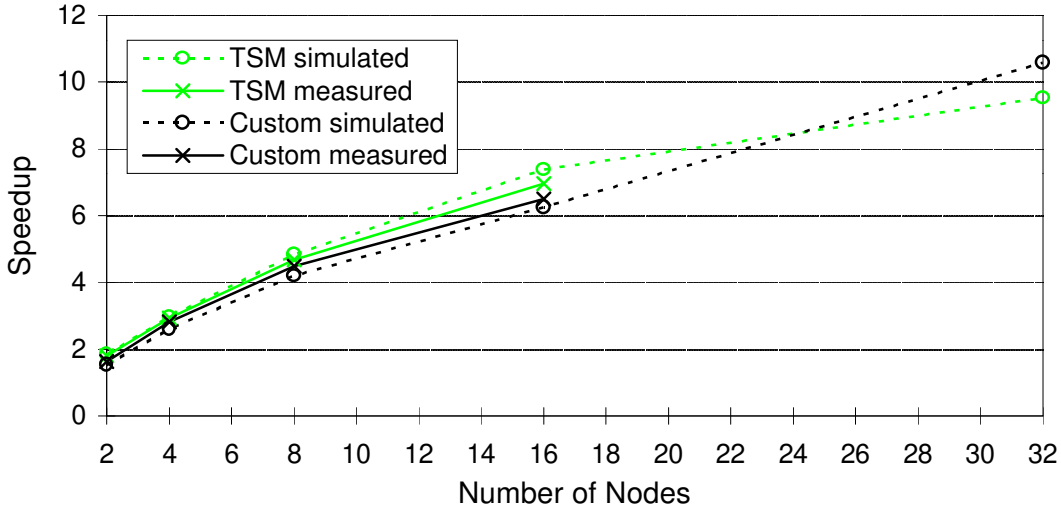

Figure 4-4. Simulated and measured speedups for Barnes.

The results are quite positive. Figures 4-3 and 4-4 plot the simulated and actual speedups for appbt and Barnes for both the transparent shared memory (64-byte blocks) and the custom protocol versions. The mean error is 3.8% and the worst case is 7.4% (for the Barnes custom protocol on four nodes). The accuracy is higher for transparent shared memory

than for the custom protocols (2.6% vs. 5.0% mean error). This trend is reasonable, because the custom protocols, with their bursty communication patterns, place more stress on the network—probably the least accurately modeled component.

4.5 Summary

A working prototype of the Typhoon-0 system serves several purposes. First, it demonstrates the feasibility of the bus-based fine-grain access control that underlies all of the proposed implementations in Chapter 3.

Second, the relatively small amount of effort required to construct the prototype—approximately two student-years from concept to the first running application—validates the contention that architects can leverage off-the-shelf components for faster, cheaper designs.

Third, the correlation of simulator results with the prototype's measured performance—with a worst-case error of under 8%—strengthens the credibility of the simulation results in the previous chapter.

Finally, the prototype provides a production-scale platform for others to continue experimentation with applications, languages, and operating systems for fine-grain distributed shared memory.

Chapter 5

Conclusion

Distributed shared memory (DSM) systems provide a programmer-friendly shared-memory abstraction on top of a scalable distributed-memory physical organization. By partitioning processors and memory into network-connected nodes, the distributed-memory organization eliminates central bottlenecks and enables hundred- and thousand-processor systems. Distributed shared memory automates inter-node data distribution and communication, giving programmers the simple, familiar abstraction of a single, global shared memory. Unfortunately, DSM systems control memory and communication by rote, even when programmers or compilers have the potential to manage these resources more efficiently.

This thesis proposes a new approach to distributed shared memory. Systems implement primitive DSM mechanisms without restricting the protocols (policies) governing their use. Standard software libraries provide universal protocols—similar to those found in current DSM machines. However, programmers and compilers can also manage memory and communication directly using the same primitive mechanisms. When appropriate,

they can optimize performance using custom protocols that rely on application- or domain-specific knowledge.

This chapter summarizes the detailed contributions of this thesis, then suggests directions for future work in this area.

5.1 Summary

This thesis:

- identifies a set of mechanisms that support distributed shared memory;
- describes *Tempest*, an interface to these mechanisms;
- describes three system designs—*Typhoon, Typhoon-1,* and *Typhoon-0*—that support Tempest, and evaluates their performance through simulation; and
- reports on a hardware prototype of Typhoon-0.

Three mechanisms, in combination, are sufficient for distributed shared memory: messaging, local storage management, and memory access control. *Tempest* is a concrete, portable interface to these mechanisms. Tempest's messaging borrows from von Eicken's Active Messages [vECGS92]. Standard virtual address translation mechanisms are used for local storage management, as in software DSM systems [LH89]. The most innovative aspect of Tempest is its specification of *fine-grain* access control, a feature that enables fine-grain coherence and provides scalability to high-performance systems.

The *Stache* protocol uses Tempest to provide standard, application-transparent distributed shared memory. This thesis also describes custom, application-specific protocols hand-written for six shared-memory benchmarks [FLR+94, MSH+95]. These user-level custom protocols can achieve message-passing levels of efficiency while supporting a uniform global address space.

Three system designs explore the implementation side of the Tempest interface. Although Tempest requires no special hardware, these designs use different levels of cus-

tom hardware support to address different cost/performance points. *Typhoon* achieves high performance by closely integrating three components—an access control device, a network interface, and a protocol processor—on a single device. *Typhoon-1* and *Typhoon-0* employ the same logical components, but trade some performance for simpler, potentially less costly designs. Specifically, both Typhoon-1 and Typhoon-0 use off-the-shelf processors for protocol software execution; Typhoon-0 uses an off-the-shelf network interface as well. Both systems use *cacheable control registers* to communicate events efficiently from a bus-based hardware device to the protocol processor.

Simulation provides performance results for the three designs using both transparent (standard) and custom shared memory protocols. As a baseline for comparison, I also simulated a comparable, idealized hardwired-protocol system similar to Simple COMA [HSL94]. These results indicate several conclusions:

- *Systems based on primitive mechanisms can provide high performance.* On six unmodified shared-memory benchmarks, Typhoon's performance is within 25% of the hardwired-protocol system—within 7% for the four with higher computational efficiencies. Although supporting user-level software protocols involves some opportunity cost, these systems are not inherently confined to lower echelons of performance.
- *Even high-performance, low-overhead DSM systems can benefit from custom protocols.* For two of the benchmarks, custom-protocol versions on Typhoon execute 50% and four times faster than optimized standard shared-memory versions on the hardwired-protocol system.
- *Custom protocols can be the key to good performance on systems with high overheads.* Custom protocols are generally more capable of coping with high overheads and large network latencies than standard shared-memory models. On the most demanding transparent shared memory benchmark, Typhoon-1 and Typhoon-0 are a factor of two and four slower than Typhoon, respectively; custom protocols reduce the differences to 13% and 47%. Custom protocols also provide more robust application performance in

the face of other increased overheads, such as larger network latencies and, for Typhoon, slower embedded protocol processors.

A prototype hardware implementation of Typhoon-0 demonstrates the feasibility of these designs and provides a real-world system for benchmarking. Despite a high-overhead commercial messaging network, five of the six benchmarks achieve better than 58% efficiency on sixteen nodes of the prototype. Application-specific protocols are critical to achieving this efficiency in three of the five cases. The prototype's measured performance agrees substantially with simulator projections.

5.2 Future directions

Future work stemming from this thesis falls into three areas: modifications and enhancements to Tempest, applications, and implementation techniques.

Experience with Tempest has exposed a few compromises in the current incarnation. First, users cannot implement software write buffers or forward writes to other nodes. To support these operations, systems must be capable of completing writes by delivering the write data to a software handler. (Currently, writes complete only by modifying a valid, local copy of the referenced block.) Although this feature is feasible in a software system such as Blizzard-S, it is practically impossible given bus-based hardware access control and writeback caches.

Tempest's single-threaded handler execution model (see Section 2.2.4) is another potential limitation. Although this model avoids locking protocol data structures, it prevents concurrent execution of protocol handlers on multiprocessor nodes, which can increase throughput on communication-intensive benchmarks [FW96a].

An important field unexplored in this thesis is the development of software tools that manage and exploit Tempest's flexibility. There are already several examples of work in this area. Teapot [CRL96] is a language/compiler system for developing Tempest proto-

cols. A compiler for the data-parallel language C** [VL96] automatically exploits Tempest-based custom protocols [LRV94] to improve performance. FlashPoint [MOH96] uses a custom protocol for performance monitoring on Stanford's FLASH system.

On the implementation front, researchers should continue to explore the design space for fine-grain access control. Although the snooping hardware described in this thesis provides zero-overhead access checks, it requires either restartable bus error exceptions or a dedicated protocol processor to avoid deadlock. Section 3.5 describes other implementation possibilities, including software-only techniques, integration of access control tags with memory error correcting codes, and integrated processor support. The tradeoffs are only partially understood [SFL+94], and will change as technology advances. For example, hardware techniques for recent processors must cope with non-blocking and speculative memory accesses. The interaction between access control techniques and operating system features such as multiprogramming and I/O also deserves further investigation (see Section 4.2.4).

Handling simple critical-path operations in hardware could increase the performance of decoupled designs similar to Typhoon-1. In current Tempest implementations, the system invokes a software handler to send a request message when it detects a miss; similarly, software handles the response message and signals that the miss has been satisfied. Although software control of these paths provides important flexibility, invocation and execution of this code is on the critical path for remote data requests.

A hypothetical enhanced version of Typhoon-1's access control/network interface device could automatically generate requests in response to block access faults and handle the data transfer portion of simple requests and responses. Software handlers run in the background to update protocol data structures. In complex or unusual cases, the system falls back on software control and behaves like the current Typhoon-1. By removing software from the critical path, this design's performance may approach or even exceed that of Typhoon, while still avoiding an embedded protocol processor. Several issues must be

addressed before we can realize this hypothetical system. How flexible is the device's state machine, and how do we control it? How do we synchronize between the device's state machine and software handlers? How do we compile Tempest handlers (or modify Tempest) to take advantage of hardware acceleration?

In the longer term, trends point to the integration of processors and main memory on a single chip [BGK96, SPN96]. Several likely characteristics of this future environment make it a ripe target for mechanism-based distributed shared memory:

- Any system containing more than one processor–memory chip has a distributed-memory organization.
- On-chip processing is much faster than off-chip communication, making it worthwhile to optimize communication using software protocols.
- Integration allows tight coupling of access control and protocol processing resources (as in Typhoon).
- An abundance of processing resources, perhaps coupled with novel execution models [KD92, TEL95], may allow low-overhead invocation and execution of protocol software without dedicating a processor to the task.

Even if distributed-memory systems do not become commodities, they will be common wherever users need more performance than just a few processors can provide. In either case, the concepts proposed in this thesis point the way to more flexible and efficient systems in the future.

References

[AAWC94] Michael S. Allen, Michael Alexander, Chuck Wright, and Joe Chang. Designing the PowerPC 60X bus. *IEEE Micro*, 14(5):42–51, October 1994.

[ABC+95] 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.

[ACD+96] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwanepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[ACP95] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering - a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[AKK+93] A. Agarwal, J. Kubiatowicz, D. Kranz, B.-H. Lim, D. Yeung, G. D'Souza, and M. Parkin. SPARCLE: An evolutionary processor design for multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[ALBL91] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating systems. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 108–120, April 1991.

[BBD+87] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfield, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston Inc., 1987.

[BBLS91] 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.

[BCF+95] 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.

[BCL+95] 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.

[BDFL96] 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)*, pages 154–165, February 1996.

[Bel85] C. Gordon Bell. Multis: A new class of multiprocessor computers. *Science*, 228:462–466, 1985.

[BGK96] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 78–89, May 1996.

[BJ81] Özalp Babaoglu and William Joy. Converting a swap-based system to do paging in an architecture lacking page-referenced bits. In *Proceedings of the Eighth ACM Symposium on Operating System Principles (SOSP)*, pages 78–86, December 1981.

[BKT92] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3), March 1992.

[BLA+94] 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.

[BM95] Doug Burger and Sanjay Mehta. Parallelizing appbt for a shared-memory multiprocessor. Technical Report 1286, Computer Sciences Department, University of Wisconsin–Madison, September 1995.

[BR90] Roberto Bisiani and Mosur Ravishankar. PLUS: A distributed shared-memory system. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 115–124, June 1990.

[BRE92] Brian N. Bershad, David D. Redell, and John R. Ellis. Fast mutual exclusion for uniprocessors. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 223–237, Boston, Massachusetts, 1992.

[Bri70] Per Brinch Hansen. The nucleus of a multiprogramming system. *Communications of the ACM*, 13(4):238–250, April 1970.

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The Midway distributed shared memory system. In *COMPCON 1993*, 1993.

[CAA+95] 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.

[CAL+89] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating System Principles (SOSP)*, pages 147–158, December 1989.

[CBZ91] 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.

[CDG+93] 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.

[CGB89] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Multi-level shared caching techniques for scalability in VMP-MC. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 16–24, June 1989.

[CGB91] David R. Cheriton, Hendrik A. Goosen, and Patrick D. Boyle. Paradigm: A highly scalable shared-memory multiprocessor. *IEEE Computer*, 24(2):33–46, February 1991.

[CGBG88] David R. Cheriton, Anoop Gupta, Patrick D. Boyle, and Hendrik A. Goosen. The VMP multiprocessor: Initial experience, refinements and performance evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 410–421, June 1988.

[Cha94] David L. Chaiken. *Mechanisms and Interfaces for Software-Extended Coherent Shared Memory.* PhD thesis, Massachusetts Institute of Technology, September 1994. Available as MIT/LCS/TR-644.

[CKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 224–234, April 1991.

[CKP91] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 40–52, April 1991.

[CLMY96] David E. Culler, Lok Tin Liu, Richard P. Martin, and Chad O. Yoshikawa. Assessing fast network interfaces. *IEEE Micro*, pages 35–43, February 1996.

[CM88] Albert Chang and Mark F. Mergen. 801 storage: Architecture and programming. *ACM Transactions on Computer Systems*, 6(1):28–50, February 1988.

[CR95] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in Olden. In *Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, 1995.

[Cra96] Cray Research, Inc. The CRAY J90se computer systems. World Wide Web document, 1996. http://www.cray.com/PUBLIC/product-info/J90/J90se.html.

[CRL96] Satish Chandra, Brad Richards, and James R. Larus. Teapot: Language support for writing memory coherence protocols. In *Proceedings of the SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, May 1996.

[CSB86] David R. Cheriton, Gert A. Slavenburg, and Patrick D. Boyle. Software-controlled caches in the VMP multiprocessor. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 366–374, June 1986.

[DCF+89] William J. Dally, Andrew Chien, Stuart Fiske, Waldemar Horwat, John Keen, Michael Larivee, Rich Lethin, Peter Nuth, Scott Wills, Paul Carrick, and Greg Fyler. The J-Machine: A fine-grain concurrent computer. In G. X. Ritter, editor, *Information Processing 89*, pages 1147–1153. Elsevier North-Holland, Inc., 1989.

[DCMM94] H. G. Dietz, W. E. Cohen, T. Muhammad, and T. I. Mattox. Compiler techniques for fine-grain execution on workstation clusters using papers. In *Proceedings of the Seventh Annual Workshop on Languages and Compilers for Parallel Computing*, August 1994.

[DCZ96] Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. An integrated compile-time/run-time software distributed shared memory system. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 186–197, October 1996.

[DL92] Czarek Dubnicki and Thomas J. LeBlanc. Adjustable block size coherence caches. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 170–179, June 1992.

[DW89] William J. Dally and D. Scott Wills. Universal mechanisms for concurrency. In *PARLE '89: Parallel Architectures and Languages Europe*. Springer-Verlag, June 1989.

[EK89] Susan J. Eggers and Randy H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 257–270, 1989.

[ENCH96] Andrew Erlichson, Neal Nuckolls, Greg Chesson, and John Hennessy. SoftFLASH: Analyzing the performance of clustered distributed virtual shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, October 1996.

[Eno96] Enorex Microsystems, Inc. Enorex opens direct-mail channel for superfast Alpha desktop PCs. World Wide Web document, November 1996. http://www.enorex.com/pr111896.htm.

[FKD+95] Marco Fillo, Steven W. Keckler, William J. Dally, Nicholas P. Carter, Andrew Chang, Yevgeny Gurevich, and Whay S. Lee. The M-Machine multicomputer. In *28th Annual International Symposium on Microarchitecture (MICRO-28)*, pages 146–156, December 1995.

[FLR+94] 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.

[FV93] 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.

[FW96a] Babak Falsafi and David A. Wood. Parallel network interfaces for smp nodes. Submitted for publication, December 1996.

[FW96b] 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.

[GGK+83] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. The nyu ultracomputer—designing an mimd shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, February 1983.

[Gil96] Richard B. Gillett. Memory channel network for PCI. *IEEE Micro*, 16(1):12–18, February 1996.

[GLL+90] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Philip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, June 1990.

[GVW89] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 64–77, April 1989.

[Gwe95] Linley Gwennap. Intel's P6 bus designed for multiprocessing. *Microprocessor Report*, 9(7), May 30, 1995.

[HGDG94] 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.

[HHS+95] Chris Holt, Mark Heinrich, Jaswinder Pal Singh, Edward Rothberg, and John Hennessy. The effects of latency, occupancy, and bandwidth in distributed shared memory multiprocessors. Technical Report CSL-TR-95-660, Computer Systems Laboratory, Stanford University, January 1995.

[HJ92] 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.

[HKO+94] 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.

[HLH92] Erik Hagersten, Anders Landin, and Seif Haridi. DDM — a cache-only memory architecture. *IEEE Computer*, 25(9):44–54, September 1992.

[HLRW93] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in ASPLOS V.

[HMMS96] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 260–270, May 1996.

[HSL94] Erik Hagersten, Ashley Saulsbury, and Anders Landin. Simple COMA node implementations. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, January 1994.

[IDFL96] Liviu Iftode, Cesary Dubnicki, Edward W. Felten, and Kai Li. Improving release-consistent shared virtual memory using automatic update. In *Proceedings of the 2nd International Symposium on High-Performance Computer Architecture (HPCA)*, pages 14–25, February 1996.

[Int95] Intel Corporation. Intel supercomputer teraflops program. World Wide Web document, 1995. http://www.ssd.intel.com/tflop.html.

[JKW95] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-performance all-software distributed shared memory. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles (SOSP)*, pages 213–228, December 1995.

[KA93] John Kubiatowicz and Anant Agarwal. Anatomy of a message in the Alewife multiprocessor. In *Proceedings of the 7th ACM International Conference on Supercomputing*, July 1993.

[KCPT95] D.A. Koufaty, X. Chen, D.K. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multprocessors. In *Proceedings of the 1995 International Conference on Supercomputing*, 1995.

[KCZ92] Pete Keleher, Alan L. Cox, and Willy Zwanepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[KD92] Stephen W. Keckler and William J. Dally. Processor coupling: Integrating compile time and runtime scheduling for parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 202–213, May 1992.

[KDCZ94] Pete Keleher, Sandhya Dwarkadas, Alan Cox, and Willy Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *Proceedings of the Winter 94 Usenix Conference*, pages 115–131, January 1994.

[Ken92] Kendall Square Research. Kendall Square Research technical summary, 1992.

[KJA+93] David Kranz, Kirk Johnson, Anant Agarwal, John Kubiatowicz, and Beng-Hong Lim. Integrating message-passing and shared-memory: Early experience. In *Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 54–63, May 1993.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 302–313, April 1994.

[KS93] R. E. Kessler and J. L. Schwarzmeier. Cray t3d: A new dimension for cray research. In *Proceedings of COMPCON 93*, pages 176–182, San Francisco, California, Spring 1993.

[LAD+92] 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 Fourth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1992.

[Lar94] James R. Larus. Compiling for shared-memory and message-passing computers. *ACM Letters on Programming Languages and Systems*, 2(1-4):165–180, March–December 1994.

[LCC+75] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in Hydra. In *Proceedings of the Fifth ACM Symposium on Operating System Principles (SOSP)*, pages 132–140, November 1975.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[Li86] Kai Li. *Shared Virtual Memory on Loosely-coupled Multiprocessors*. PhD thesis, Yale University, October 1986.

[LLG+92] 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.

[LRV94] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory system support for parallel language implementation. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 208–218, October 1994.

[LS95] 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.

[Mas94] Susan A. Mason. *SBus Handbook*. SunSoft Press, 1994.

[McC96] John D. McCalpin. Stream benchmark results - standard set. World Wide Web document, November 22 1996. http://www.cs.virginia.edu/stream/standard/Bandwidth.html.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[MFHW96] Shubhendu S. Mukherjee, Babak Falsafi, Mark D. Hill, and David A. Wood. Coherent network interfaces for fine-grain communication. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 247–258, May 1996.

[MG91] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12:87–106, June 1991.

[Mic96] MicroDesign Resources. Chart watch: Workstation processors. *Microprocessor Report*, 10(9):19, July 8, 1996.

[Min95] Minnesota Supercomputer Center, Inc. Distributed job manager. World Wide Web document, 1995. http://www.msc.edu/msc/docs/djm.

[MKAK94] Kenneth Mackenzie, John Kubiatowicz, Anant Agarwal, and Frans Kaashoek. Fugu: Implementing translation and protection in a multiuser, multimodel multiprocessor. Technical Report TM-503, MIT LCS, October 1994.

[MKBS95] 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.

[MOH96] Margaret Martonosi, David Ofelt, and Mark Heinrich. Integrating performance monitoring and communication in parallel computers. In *Proceedings of the 1996 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 138–147, May 1996.

[MSH+95] 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.

[Nim93] Nimbus Technology. Nim 6133 memory controller specification. Technical report, Nimbus Technology, 1993.

[Pfi95] Robert W. Pfile. Typhoon-Zero implementation: The Vortex module. Technical Report 1290, Computer Sciences Department, University of Wisconsin–Madison, October 1995.

[RAK89] Umakishore Ramachandran, Mustaque Ahamad, and M. Yousef A. Khalidi. Coherence of distributed shared memory: Unifying synchronization and data transfer. In *Proceedings of the 1989 International Conference on Parallel Processing (Vol. II Software)*, pages 160–169, August 1989.

[Rei95] Steven K. Reinhardt. Tempest interface specification (revision 1.2.1). Technical Report 1267, Computer Sciences Department, University of Wisconsin–Madison, February 1995.

[RFW93] Steven K. Reinhardt, Babak Falsafi, and David A. Wood. Kernel support for the Wisconsin Wind Tunnel. In *Proceedings of the USENIX Symposium on Microkernels and Other Kernel Architectures*, pages 73–89, September 1993.

[RHL+93] 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.

[RLW94] 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.

[Ros93] Ross Technology, Inc. *SPARC RISC User's Guide: hyperSPARC Edition*, September 1993.

[RPW96] Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Decoupled hardware support for distributed shared memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 34–43, May 1996.

[RSW+93] E. Rosti, E. Smirni, T.D. Wagner, A.W. Apon, and L.W. Dowdy. The ksr1: Experimentation and modeling of poststore. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 74–85, May 1993.

[SFH+96] Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, Christopher E. Lucas, Shubhendu S. Mukherjee, Steven K. Reinhardt, Eric Schnarr, and David A. Wood. Implementing fine-grain distributed shared memory on commodity smp workstations. Technical Report 1307, Computer Sciences Department, University of Wisconsin–Madison, March 1996.

[SFL+94] 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.

[SGT96] Daniel J. Scales, Kourosh Gharachorloo, and Chandramohan A. Thekkath. Shasta: A low overhead, software-only approach for supporting fine-grain shared memory. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VII)*, pages 174–185, October 1996.

[SH91] Richard Simoni and Mark Horowitz. Dynamic pointer allocation for scalable cache coherence directories. In *International Symposium on Shared Memory Multiprocessing*, pages 72–81, April 1991.

[Sho94] Doug Shore. Personal communication, November 1994.

[Sil96] Silicon Graphics, Inc. Origin servers: Technical overview of the origin family. World Wide Web document, 1996. http://www.sgi.com/Products/hardware/servers/technology/overview.html.

[SL94] Daniel J. Scales and Monica S. Lam. The design and evaluation of a shared object system for distributed memory machines. In *First USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 101–114, November 1994.

[SPN96] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 90–101, May 1996.

[Sun91] Sun Microsystems Inc. *SPARC M{Bus Interface Specification*, April 1991.

[Sun94] Sun Microsystems Computer Corporation. *Writing F{Code Programs*. SunSoft Press, 1994.

[SWG92] 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.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 392–403, June 1995.

[Thi91] Thinking Machines Corporation. The Connection Machine CM-5 technical summary, 1991.

[TL94] Chandramohan A. Thekkath and Henry M. Levy. Hardware and software support for efficient exception handling. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pages 110–119, October 1994.

[vECGS92] 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.

[VL96] Guhan Viswanathan and James R. Larus. Compiler-directed shared-memory communication for iterative parallel applications. In *Proceedings of Supercomputing '96*, November 1996.

[WCC+74] W. Wulf, E. Cohen, W. Corwin, A. Jones, R. Levin, C. C. Pierson, and F. Pollack. Hydra: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[WCF+93] 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.

[WHL92] Larry D. Wittie, Gudjon Hermannsson, and Ai Li. Eager sharing for efficient massive parallelism. In *Proceedings of the 1992 International Conference on Parallel Processing*, pages 251–255, August 1992.

[WJI+94] Andrew W. Wilson Jr., Richard P. LaRowe Jr., Robert J. Ionta, Ralph P. Valentino, Beeching Hu, Peter R. Breton, and Pocheong Lau. Update propagation in the Galactica Net distributed shared memory architecture. In *Proceedings of the International Workshop on Support for Large Scale Shared Memory Architecture*, 1994.

[Wul81] William A. Wulf. Compilers and computer architecture. *IEEE Computer*, 14(7):41–48, July 1981.

Appendix A

Tempest Interface Specification

This appendix is a formal specification of the Tempest interface taken from [Rei95]. The specification is divided into two parts: Section A.1 describes the general execution model of a Tempest application and Section A.2 documents the specific interface functions for the C programming language. In addition to the interface specification, comments labeled "**Rationale**" and "**Implementation note**" are provided to give insight into the motivation for certain features and to suggest implementation approaches, respectively.

A.1 Execution model

The Tempest architecture assumes a host that consists of many processing nodes, where each node contains one or more processors connected to a single memory module (that is, sharing a single physical address space). Communication between nodes occurs only through message-passing and the (as yet undefined) global operations (e.g., barriers). A Tempest application has a distinct virtual address space on each processing node. The SPMD ("single program, multiple data") model is used, i.e., the same program text is loaded at the same location in every address space, though each processor executes that text independently. Each address space may also contain other per-node private segments

for data and stacks. In addition, a contiguous segment at the same location in each address space is designated the *user-managed virtual segment*. Within this segment, the user maps virtual pages to physical memory, handles accesses to unmapped pages, and controls the accessibility of mapped memory at a fine granularity. Providing user-level control over the same virtual address region on every node is the basis of constructing a transparent single-address-space execution environment with user-defined semantics.

Tempest provides fine-grain memory access control by associating a *block access tag* with every aligned 2^k-byte region of memory (a *memory block*). The value of k is implementation-dependent, but is typically five, six, or seven; that is, memory blocks are typically 32, 64, or 128 bytes. The Tempest interface is designed so that code written assuming some block size b will be portable across all implementations whose block size is smaller than or equal to b. The actual block size supported by an implementation is referred to as the implementation's *minimum block size*, and blocks of the minimum block size are called *minimal blocks*.

Table A.1: Access tag semantics.

Access	Tag			
	Invalid	Busy	ReadOnly	Writable
load	fault	fault	return data	return data
store	fault	fault	fault	write data

Each block access tag has one of four values: `Invalid`, `Busy`, `ReadOnly`, and `Writable`. A block whose tag is `Invalid` or `Busy` is referred to as *invalid*, while blocks tagged `ReadOnly` or `Writable` are called *valid*. (`Invalid` and `Busy` have the same access semantics, but can be used by software to encode protocol information; e.g., whether or not a request is pending.) Loads and stores are checked against the value of the referenced block's access tag; conflicting accesses cause a *block access fault* (see Table A.1). The fault suspends the accessing thread and invokes a user-level handler to

process the fault. The handler typically performs some protocol actions to make the access permissible. Once access is allowed, the block's tag is changed and the faulting thread is resumed.

Each address space (and thus each node) can support multiple threads of execution. These threads may execute concurrently on multiprocessor nodes. One of the threads is distinguished as the *protocol thread*. All other threads are referred to as *computation threads*. The management of computation threads (creation, scheduling, etc.) is outside the domain of Tempest. The protocol thread is scheduled by Tempest and exists solely to execute user-defined *handler functions* to process *protocol events*: network message arrivals, timer expirations, and the page faults and block access faults of computation threads. Handler functions are executed sequentially, i.e., if a protocol event occurs while the protocol thread is executing a handler function, the event will be queued until the current handler function completes. If multiple events are queued, block access faults and page faults are given first priority, followed by timer expirations, and then message arrivals.

> **Rationale:** By dedicating a single thread for protocol processing and running handler functions to completion, the need for locking on protocol data structures is reduced or eliminated. If concurrency is desired in processing protocol events, the protocol thread's handler function can hand off tasks to computation threads.
>
> **Rationale:** Event prioritization is based on the following observations:
>
> - Message arrivals are beyond the control of the local node, so any event with lower priority may be subject to starvation. This is avoided by giving message arrivals lowest priority.
> - Servicing of faulting accesses should be given high priority to maximize application throughput. The number of concurrent page and block access faults is limited by the number of concurrently-executing computation threads, so starvation of other event types is not a problem.

Only the loads and stores of computation threads are guaranteed to be checked against the access tags. Normally, the protocol thread only references tagged memory indirectly via Tempest functions; the behavior of a direct protocol thread access that conflicts with the block's tag is undefined.

The type of protocol event determines the user-level handler function that is executed by the protocol thread. The handler for a message arrival is chosen by the sender and encoded

in the message header (as in Active Messages [vECGS92]). The handler for a timer expiration is specified when the timer is initialized. Handlers for page faults and block access faults are registered locally by the application. All page faults are serviced by a single handler. The handler invoked for a block access fault is determined by the combination of the access type (load vs. store), the tag value, and the memory page on which the referenced block is located. Specifically, the user associates a small integer (the *page mode*) with every page, and registers a set of five handlers (one for each of the "fault" cases in Table A.1) for each page mode.

> **Rationale:** The ability to associate different sets of handlers with different memory blocks facilitates the use of multiple protocols within a single application. Associating handlers with pages rather than individual blocks trades a small loss of flexibility for a large decrease in storage overhead. (Actions can still be specialized at block granularity in software.) Use of the page mode value rather than allowing a separate set of handlers for each page further reduces storage overheads.

A.2 The Tempest interface for C

This section defines a standard C user interface for Tempest to provide source-level application compatibility across all Tempest implementations. Implementations will typically provide a user library that bridges the gap between Tempest and the native operating system. A Tempest implementation may involve more than a library with which the application is linked. For example, the application source may be preprocessed to convert Tempest function calls to some intermediate form, or the compiled application may be postprocessed to insert code that provides fine-grain access control [SFL+94]. The key characteristic is that an automated process is provided that converts interface-compliant source into a functioning program.

The following subsections specify the operations provided by the interface, grouped by function: virtual memory management, fine-grain memory access control, fine-grain messaging, bulk data transfer, timers, and thread management.

A.2.1 Virtual memory management

As described in Section A.1, Tempest gives user-level control over a region of the virtual address space on each node. This region, known as the *user-managed virtual segment*, is located at the same address in every address space and is at least 1 Gbyte in size. Physical memory allocation and address translation are performed on the basis of pages. Pages mapped in the user-managed virtual segment are referred to as *user-managed* pages. Only user-managed pages support block access tags. Each user-managed page must be assigned a *page mode number*, which determines the set of block access fault handlers that are invoked for block access faults on that page.

A.2.1.1 Page size

#define TPPI_PAGE_SHIFT *implementation-specific*

#define TPPI_PAGE_SIZE (1 << TPPI_PAGE_SHIFT)

The page size in bytes is exported in the constant `TPPI_PAGE_SIZE`. `TPPI_PAGE_SHIFT` is the width in bits of a page offset (i.e. $\log_2$(`TPPI_PAGE_SIZE`)).

> **Implementation note:** The Tempest page size should be as small as possible to avoid memory fragmentation. Typically it is the same as the platform's MMU page size; e.g., the SPARC MMU page size is 4K so all existing implementations (which are all SPARC-based) have a 4K page size.

A.2.1.2 Page modes

typedef *implementation-specific* TPPI_PageMode;

#define TPPI_NUM_PAGE_MODES *implementation-specific*

#define TPPI_MAX_PAGE_MODE (TPPI_NUM_PAGE_MODES - 1)

Page mode numbers are small consecutive integers starting at 0. `TPPI_PageMode` is an unsigned integer type of implementation-defined size used to hold page mode numbers. The constant `TPPI_NUM_PAGE_MODES` indicates the number of page modes supported by the implementation, and `TPPI_MAX_PAGE_MODE` indicates the largest page mode number.

A.2.1.3 Page allocation and deallocation

int TPPI_alloc_and_map(void *pg, TPPI_PageMode mode, TPPI_BlkAccTag acc, TPPI_NodeId home, void *user_ptr);

Allocates a page of physical memory and maps it in the user-managed virtual segment at the page-aligned address `pg`. By definition, the allocated page is a user-managed page. The page mode (for block access fault handler selection) is set according to `mode`. The block access tags for all blocks on the page are initialized to `acc` (see Section A.2.2.1). The `home` and `user_ptr` fields are not interpreted by the system, but are intended to hold the node identifier (see Section A.2.3.1) of the page's directory node and a pointer to a per-page protocol data

structure, respectively. The `home` and `user_ptr` values can be retrieved via function calls (using the virtual address as a key) and are automatically provided to block access fault handlers for the page.

The return value is 1 if successful, 0 if unsuccessful. The call will fail if there is insufficient physical memory, `pg` is not aligned to the page size, a mapping already exists for the virtual address `pg`, or `pg` is not within the user-managed virtual segment.

> **Implementation note:** Page-level protection may be used in the place of a true fine-grain access control mechanism if, whenever the user invokes a tag-modifying block operation, the specified block length is always equal to or greater than the page size. Because pages are always initialized with the same tag on every block, this optimization can be performed optimistically; that is, the fine-grain access control mechanism need not be used for a given page until the first time the user invokes a tag-modifying block operation with a block length smaller than the page size.

int TPPI_unmap_and_free(void *pg);

Removes the mapping for user-managed page pointed to by the aligned address `pg`.

int TPPI_remap(void *old_pg, void *new_pg);

Removes the mapping for user-managed page `old_pg` and remaps the physical page to the address `new_pg`, which must also be in the user-managed virtual segment. The block access tags, page mode, home node ID, and user pointer are unchanged. Both `old_pg` and `new_pg` must be page-aligned.

> **Rationale:** While `TPPI_unmap_and_free` followed by `TPPI_alloc_and_map` has a similar effect, `TPPI_remap` differs in two significant ways. First, the same physical page is kept, so the data is not lost. Second, `TPPI_remap` keeps the page continuously under the ownership of the application; the `TPPI_unmap_and_free`/ `TPPI_alloc_and_map` sequence is non-atomic so it is possible that another application could perform an allocation in the middle causing the `TPPI_alloc_and_map` to fail due to insufficient memory.

A.2.1.4 Page fault handlers

typedef void (*TPPI_PageFaultHandlerPtr)(void *va, int pc, int is_write);

void TPPI_register_page_fault_handler(TPPI_PageFaultHandlerPtr fn);

Registers `fn` as the user's page fault handler. When a computation thread accesses an unmapped virtual address in the user-managed virtual segment, the Tempest implementation will notify the user by causing the protocol thread to invoke this function. The handler will be invoked as

```
void (*fn)(void *va, int pc, int is_write)
```

where `va` is the unmapped address that was accessed and `pc` is the program counter of the load or store that caused the fault. The `is_write` parameter is non-zero if the access was a store, or zero if it was a load. The faulting thread is suspended until `TPPI_resume_va` is called (see Section A.2.6), either by the page fault handler, a future message handler, or a different thread.

Before resuming the faulting access, the page fault handler may directly request needed data

from a remote node or it may simply initialize all blocks on the page to `TPPI_Blk_Invalid`. In the latter case, when the access is retried after the thread is resumed it will generate a block access fault. This approach may be favored because it keeps protocol-specific code out of the page fault handler.

> **Rationale:** The page fault handler will typically use `TPPI_alloc_and_map` to add a page at the desired address. It may need to send a request to a remote node to determine the appropriate page mode; in this case, the `TPPI_alloc_and_map` and the resumption of the faulting thread will be performed by the response message handler.

A.2.1.5 Retrieving per-page information

```
typedef struct {
        implementation-specific
} TPPI_PageInfo;

int TPPI_get_page_info(void *va, TPPI_PageInfo *info_ptr);
```

Provides information about the virtual page containing the arbitrarily-aligned address `va` in the user-managed virtual segment. The return value is 0 if the page is not mapped, 1 if it is mapped, and -1 if there is an error (because `va` is not in the user-managed virtual segment or `info_ptr` is not suitably aligned). If `info_ptr` is non-null and the return value is 1, the `mode`, `user_ptr`, and `home` values provided when the page was mapped are written to the fields of the same name in the structure pointed to by `info_ptr`. The exact definition of `TPPI_PageInfo` is implementation-dependent, but it must contain at least these three fields: `TPPI_PageMode mode`, `void *user_ptr`, and `TPPI_NodeId home`.

```
void *TPPI_get_page_user_ptr(void *va);
```

Returns the `user_ptr` pointer supplied to `TPPI_alloc_and_map` when the page containing the arbitrarily-aligned address `va` was mapped. The return value is undefined if the page is not in the user-managed virtual segment or is not mapped.

```
TPPI_PageMode TPPI_get_page_mode(void *va);
```

Returns the `mode` value supplied to `TPPI_alloc_and_map` when the page containing the arbitrarily-aligned address `va` was mapped. The return value is undefined if the page is not in the user-managed virtual segment or is not mapped.

```
TPPI_NodeId TPPI_get_page_home(void *va);
```

Returns the `home` value supplied to `TPPI_alloc_and_map` when the page containing the arbitrarily-aligned address `va` was mapped. The return value is undefined if the page is not in the user-managed virtual segment or is not mapped.

A.2.2 Block access control

The semantics of block access tags are discussed in Section A.1. Tag values are initialized during page allocation (see Section A.2.1.3). Note that tags can also be modified as a side-effect of sending or receiving blocks using the `Ba` item type (see Section A.2.3).

A.2.2.1 Access tag values

```
typedef enum {
    TPPI_Blk_Busy, TPPI_Blk_Invalid, TPPI_Blk_ReadOnly,
TPPI_Blk_Writable
} TPPI_BlkAccTag;
```

#define TPPI_NUM_BLK_ACC_TAGS 4

```
#define TPPI_MAX_BLK_ACC_TAG(TPPI_NUM_BLK_ACC_TAGS - 1)
```

The `TPPI_BlkAccTag` type enumerates the possible access tag values for a block. The enumeration constants will be named as specified and valued from 0 to 3 inclusive, but their ordering is implementation-dependent. The constants `TPPI_NUM_BLK_ACC_TAGS` and `TPPI_MAX_BLK_ACC_TAG` indicate the number of supported access tags and the largest access tag value, respectively.

A.2.2.2 Tag block size

#define TPPI_TAG_BLK_SHIFT *implementation-specific*

#define TPPI_TAG_BLK_SIZE (1 << TPPI_TAG_BLK_SHIFT)

The implementation's minimum tag block size in bytes (see Section A.1) is exported in the constant `TPPI_TAG_BLK_SIZE`. `TPPI_TAG_BLK_SHIFT` is the width in bits of an offset within a block (i.e., $\log_2$(`TPPI_TAG_BLK_SIZE`)).

A.2.2.3 Specifying memory blocks

A memory block is specified with two parameters: an address and a length in bytes. These appear as a pair of arguments: `void *blk_va, int blk_len`. Whenever an address/length pair is used to specify a block as the target of a Tempest operation, the length must be a power of two and must be greater than or equal to the implementation's minimum block size. The address does not have to be aligned; it may point anywhere within the block.

> **Rationale:** For systems with hardware support, it is trivial to ignore unused address bits, so forcing the user to align addresses introduces unnecessary overhead. For software-based systems, the potential exists for address alignment to be inlined at the call site, with common subexpression elimination allowing a single alignment operation to serve for multiple Tempest function calls. Assuming these optimizations, there is little performance advantage in forcing the user to perform alignment.

If the specified length is greater than the minimum block size, the block is called a *superblock*. Operations on superblocks are subject to the following constraints:

- The operation should be viewed as a non-atomic series of operations on the constituent minimal blocks. The atomicity guarantees of Section A.2.2.6 only apply to the minimal-block operations.
- If a tag change operation (of type `TPPI_BlkTagChange`) is applied to a superblock, it must be a valid tag change for each of the constituent minimal blocks. Note that this does not mean that all of the minimal blocks must have the same original tag (though this is likely to be the case). For example, `TPPI_Blk_Invalidate` could be applied to a superblock in which some minimal blocks are tagged `TPPI_Blk_ReadOnly` and others `TPPI_Blk_Writable`.

Otherwise, it should be transparent to the user whether an operation is applied to a single minimal block or a superblock.

A.2.2.4 Reading access tags

```
TPPI_BlkAccTag TPPI_get_blk_acc(void *va);
```

Returns the block access tag associated with the block containing `va`. The result is undefined if the address is not in the user-managed virtual segment or is not mapped.

A.2.2.5 Changing access tags

```
typedef enum {
        TPPI_Blk_Validate_RW, TPPI_Blk_Upgrade_RW, TPPI_Blk_Validate_RO,
        TPPI_Blk_Downgrade_RO, TPPI_Blk_Invalidate, TPPI_Blk_Mark_Busy,
        TPPI_Blk_No_Tag_Change, TPPI_Blk_Invalid_To_Busy, TPPI_Blk_Busy_To_Invalid
} TPPI_BlkTagChange;
```

Block access tag modifications are made using the `TPPI_BlkTagChange` constants. These not only specify the desired tag value but also imply the current value of the tag, as specified in Table A.2. If the user applies a tag change operation to a block whose tag is not one of those implied by the operation (i.e., the operation does not appear in the row corresponding to that tag in Table A.2), the resulting state of the block is indeterminate. Note that `TPPI_Blk_Invalidate`, `TPPI_Blk_Mark_Busy`, and `TPPI_Blk_Validate_RW` (and of course `TPPI_Blk_No_Tag_Change`) can be applied to any block, regardless of its initial state, though in some cases the same tag change may be performed more efficiently using a different operation.

Rationale: Reducing access to a block typically requires that the block be flushed from any hardware caches, while increasing or not changing block access does not. These cache flushes

Table A.2: Block tag change enumeration values (type `TPPI_BlkTagChange`).

Current Tag	New Tag[a]			
	Invalid	Busy	ReadOnly	Writable
Invalid	`No_Tag_Change`, `Invalidate`	`Invalid_To_Busy`, `Mark_Busy`	`Validate_RO`	`Validate_RW`
Busy	`Busy_To_Invalid`, `Invalidate`	`No_Tag_Change`, `Mark_Busy`	`Validate_RO`	`Validate_RW`
ReadOnly	`Invalidate`	`Mark_Busy`	`No_Tag_Change`	`Upgrade_RW`, `Validate_RW`
Writable	`Invalidate`	`Mark_Busy`	`Downgrade_RO`	`No_Tag_Change`, `Validate_RW`

a. All values are prefixed by `TPPI_Blk_`, e.g., `TPPI_Blk_No_Tag_Change`. Where two values are listed in a single entry, the first is preferred.

can be very expensive and need to be avoided when possible. A simple "set tag" function is not sufficient to identify when flushes are necessary, and requiring the implementation to look up the current tag before it is changed (to determine is a flush is required) may also be expensive. The current tag state is usually implied by the user protocol state, so the user code typically has enough information to supply the tag change operation with no extra overhead.

void TPPI_change_blk_acc(void *blk_va, int blk_len, TPPI_BlkTagChange chg);

Changes the access tag of the block specified by (`blk_va`, `blk_len`).

void TPPI_change_blk_acc_and_copy(void *blk_va, int blk_len, TPPI_BlkTagChange chg, void *from);

Copies data from memory starting at `from` to the block specified by (`blk_va`, `blk_len`) and changes the tag of the block according to `chg`.

A.2.2.6 Atomicity of data access and tag changes

Threads may be executed concurrently on implementations with multiprocessor nodes, so while one thread is in the middle of a tag change, other threads may issue loads and stores. Tempest operations that combine data transfer and access tag changes (including TPPI_change_blk_acc_and_copy , `send_*Ba*`, and `recv_Ba`) provide the following useful semantics:

- If data is read from a block and the block's access is downgraded from `Writable`, the block data that is read is guaranteed to reflect all writes that complete before the tag change.

- If data is written to a block and the block's access is upgraded, any load or store that does not fault but would have faulted given the previous access tag is guaranteed to be performed after the block's contents are updated with the new data.

A.2.2.7 Block access fault handlers

```
typedef void (*TPPI_BlkAccFaultHandlerPtr)(void *va, void *user_ptr, TPPI_NodeId home);

void TPPI_register_blk_acc_fault_handler(TPPI_PageMode mode, TPPI_BlkAccTag tag, int acc,
                                         TPPI_BlkAccFaultHandlerPtr fn);
```

Registers function `fn` as the block access fault handler for accesses of type `acc` (which should be one of the defined constants `TPPI_ReadAccess` or `TPPI_WriteAccess`) to blocks tagged with `tag` on pages of mode `mode`. The handler will be invoked as

```
void (*fn)(void *va, void *user_ptr, TPPI_NodeId home)
```

where `va` is an address within the block on which the faulting access was performed and `user_ptr` and `home` are the values supplied to `TPPI_alloc_and_map` when the page containing `va` was mapped. The actual address that was accessed by the faulting thread and `va` will be in the same minimal block, but are not necessarily related otherwise.

> **Rationale:** Implementations using hardware external to a commodity processor will only observe the cache miss that results from a faulting access, not the faulting access itself. In this case, the relationship between the observed address and the accessed address will be determined by the processor implementation.

Only five of the eight `tag`/`acc` combinations are meaningful (see Table A.1); specifying handlers for the other three (`TPPI_Blk_ReadOnly`/`TPPI_ReadAccess`, `TPPI_Blk_Writable`/`TPPI_ReadAccess`, and `TPPI_Blk_Writable`/`TPPI_WriteAccess`) may have undesirable implementation-dependent effects and should be avoided. (Ideally, the implementation will detect attempts to specify handlers for the other cases and warn the user.)

A.2.3 Fine-grain messaging

Fine-grain messaging provides low-overhead message sending and reception, optimized for short message lengths.

> **Rationale:** Both cache coherence protocols and fine-grain parallel applications employ short asynchronous messages whose contents are immediately consumed on receipt (e.g., cache miss or remote read requests and responses). Much of the message data originates in the sender's registers and is consumed in the receiver's registers. The memory-to-memory transfers provided by most message-passing models (and by Tempest's bulk data transfer operations) are inappropriate for these applications since both the management of memory buffers and the need to copy data into and out of these buffers add significant overhead.

Tempest's fine-grain messaging facility is based on Active Messages [vECGS92]. In the Active Message model, the first word of every message is the starting program counter of the handler to be executed at the receiver. Messages are queued and the handlers are executed serially by the protocol thread.

A.2.3.1 Node identifiers

```
typedef implementation-specific TPPI_NodeId;
unsigned TPPI_num_nodes;
TPPI_NodeId TPPI_self_address;
```

`TPPI_NodeId` is an unsigned integer type of implementation-defined size used to hold node identifiers. Node identifiers are in the range 0 to *n*-1 for *n*-node systems. Two integer variables, `TPPI_num_nodes` and `TPPI_self_address`, provide the number of available nodes and the local node's identifier, respectively.

A.2.3.2 Sending

typedef void (*TPPI_MessageHandlerPtr)(TPPI_NodeId src, int size);

void TPPI_send_*typelist*(TPPI_NodeId dest, TPPI_MessageHandlerPtr pc, *arglist*);

This set of functions sends a message to the specified node, where the message will be handled by executing code starting at the specified program counter. The body of the message is constructed using the specified (possibly empty) item list. In the current C binding, the item list specification is split: the types of the items in are encoded in a string that is part of the function name, while the parameters describing the items are part of the argument list. A given item may require more than one parameter.

> **Rationale:** Abstractly, `TPPI_send` is a polymorphic function that takes an arbitrary number of arguments selected from a set of types (word, memory block, and memory region). Unfortunately, C does not support this polymorphism. The C++ binding (when complete) will have a single `TPPI_send` function that is overloaded to support all possible message formats.

The following item types are available (with the type string given in parentheses):

- *Word (W).* A single machine word is sent. The corresponding parameter is the word value, of type `int`.
- *Block with access change (Ba).* The contents of a memory block are sent, and the memory block's access tag is modified. The corresponding parameters are the block specifier (`void *blk_va, int blk_len`) (see Section A.2.2.3) and the tag change (type `TPPI_BlkTagChange`) (see Section A.2.2.5).
- *Region (R).* The contents of a region of memory are sent. The region must start on a word boundary and contain an integral number of words. The corresponding parameters are the region start address (type `void *`) and the region length in bytes (type `int`). The region length must be a multiple of the word size.
- *Forward (F).* Data from the current received message is sent. This option is only valid

when the send is called in the context of a message handler. The corresponding parameter is the number of bytes to forward (type `int`), which must be a multiple of the word size.

For example, the following call sends a word of data (`word`) along with a memory block of size `blk_len` at address `blk_va`, atomically changing the block's tag from ReadOnly to Invalid:

```
TPPI_send_WBa(dest, handler_pc, word, blk_va, blk_len,
                    TPPI_Blk_Invalidate);
```

As a syntactically special case, the '_' in the function name is elided when a message with no body is sent, e.g., `TPPI_send(dest, pc)`.

The message body is constructed by concatenating data items, in the specified order, into an untyped stream of words.

A.2.3.3 Receiving

On the receiver, the system logically queues the message until the protocol thread is idle. The sender-specified function is invoked with two parameters: the source node (type `TPPI_NodeId`) and the size of the message body in bytes (type `int`). The message body is provided as a logical queue of words. Data is read from this queue and consumed using the following calls, which correspond to the types available for sending:

int TPPI_recv_W();

The next word is returned.

void TPPI_recv_Ba(void *blk_va, int blk_len, TPPI_BlkTagChange chg);

The next `blk_len` bytes are read from the queue and written to the memory block specified by (`blk_va`, `blk_len`) (see Section A.2.2.3), whose access tag is changed (see Section A.2.2.5).

void TPPI_recv_R(void *va, int len);

The next `len` bytes of data are read from the queue and written to the specified region of memory. The region must start on a word boundary and contain an integral number of words.

In addition, message data can be consumed using a "forwarded block" item in a send operation. Note that even though the send and receive operations use the same types, the message body is transferred as a typeless word stream, so the types used to send and receive a particular message do not need to match. However, the receive handler must consume the entire message body. If a receive handler leaves data in the message queue when

it terminates, some implementations may interpret this data as part of a separate message. The resulting behavior is undefined.

A.2.3.4 Message size limit

#define TPPI_MAX_AM_BYTES *implementation-specific*

A Tempest implementation will typically have an upper bound on the size of message that can be supported in terms of the number of bytes in the message body. This upper bound is exported in the constant TPPI_MAX_AM_BYTES and is guaranteed to be at least `(TPPI_TAG_BLK_SIZE + 16)`.

Rationale: This minimum size allows for a block and 16 bytes of control information (e.g., four 32-bit words, or two 32-bit words and a 64-bit address).

Implementation note: Tempest provides an Active Message interface without enforcing an Active Message implementation. In an actual Active Message implementation, the entire message is put into a single packet. On a system that cannot support a `(TPPI_TAG_BLK_SIZE + 16)`-byte payload in a single packet, packetization and reassembly must be supported, but messages that are "small enough" may still be handled in a true Active Message fashion.

A.2.4 Bulk data transfer

Bulk data transfer provides high-bandwidth, connection-oriented, memory-to-memory data movement between nodes.

Rationale: A memory-to-memory transfer model is desirable because it simplifies system flow control and buffering issues by inherently providing buffer space on both the sender and receiver, and it can be efficiently supported with typical DMA hardware. A connection-oriented model allows connection set-up overhead to be amortized over multiple transfers when a repetitive communica
tion pattern exists.

Implementation note: All memory-to-memory transfers could be implemented on top of a suitable Active Messages layer.

A.2.4.1 Channel allocation

```
typedef void (*TPPI_ChannelHandlerPtr)(NodeId, int channel);
int TPPI_set_channel_src(TPPI_NodeId dest, TPPI_ChannelHandlerPtr fn);
int TPPI_set_channel_dst(TPPI_NodeId src, int channel, void *buffer, int bytes,
                         TPPI_ChannelHandlerPtr fn);
```

Channel allocation requires allocation of an endpoint on both the source (sending) and destination (receiving) nodes. The source node must first call TPPI_set_channel_src`()` to obtain a channel ID number.[1] The arguments are the destination node and a pointer to a function which

1. The channel ID may be relative to a source/destination pair, i.e., distinct channel IDs are only required when there a multiple active channels between a given source and destination.

will be invoked at the completion of each send. The sender must communicate the returned channel ID to the destination node (typically via an active message). The destination node then calls TPPI_set_channel_dst() to initialize the receiving end, passing in the source node ID, the channel ID from the source, the address and length of a receive buffer, and a pointer to a function that will be invoked at the completion of each receive.

Both the send and receive callbacks are invoked with the ID of the corresponding node and the channel ID. In either case, a null function pointer may be provided in which case no callback will be performed. On the source node, invocation of the callback means only that the send buffer can be reused; it does not imply that the data has been received at the destination.

```
int TPPI_set_channel_dst_notify(TPPI_NodeId src, int channel,
                        void *buffer, int bytes,
                        TPPI_ChannelHandlerPtr fn);

int TPPI_is_channel_estd(TPPI_NodeId dst, int channel);
```

TPPI_set_channel_dst_notify() performs the same function as TPPI_set_channel_dst() and sends an active message back to the source to notify it that the endpoint has been established. The source node may use TPPI_is_channel_estd() to poll for the establishment of the destination endpoint. It will return non-zero only after the arrival of the notification message.

A.2.4.2 Sending data

void TPPI_channel_send(TPPI_NodeId dest, int channel, void *buffer, int bytes);

The source node calls TPPI_channel_send() to initiate a data transfer of bytes bytes starting at the pointer buffer. The transfer may be asynchronous; the send callback, if any, will be invoked when the buffer memory may be reused. The number of bytes specified in the send must exactly match the number specified by the destination node in its call to TPPI_set_channel_dst().

int TPPI_is_channel_ready(TPPI_NodeId src, int channel);

The destination node may call TPPI_is_channel_ready() to poll for the arrival of data (in lieu of specifying a receive callback function). This function will return non-zero when the destination has received the number of bytes specified in its call to TPPI_set_channel_dst(). It will continue to return non-zero until the endpoint is reset via TPPI_reset_channel().

void TPPI_reset_channel(TPPI_NodeId src, int channel);

The destination node must call TPPI_reset_channel() to reset the receive endpoint of the channel after each data transmission before the source can perform another send. The destination and source nodes must synchronize to guarantee that the destination has called TPPI_reset_channel() before the source calls TPPI_channel_send().

```
void TPPI_reset_channel_notify(TPPI_NodeId src, int channel_id);

int TPPI_is_channel_reset(TPPI_NodeId dst, int channel);
```

TPPI_reset_channel_notify() performs the same function as TPPI_reset_channel() and sends an active message back to the source to notify it that the endpoint has been reset. The source node may use TPPI_is_channel_reset() to poll

for this event. It will return non-zero only after the arrival of the notification message.

A.2.4.3 Channel deallocation

void TPPI_destroy_channel_src(TPPI_NodeId dest, int channel);

void TPPI_destroy_channel_dst(TPPI_NodeId dest, int channel);

As with allocation, both the source and destination nodes must explicilty deallocate their endpoints. Results are unpredictable if either endpoint is deallocated before all of the data that sent on the channel has been received at the destination.

A.2.4.4 User pointers

```
void TPPI_set_src_channel_user_ptr(TPPI_NodeId dst, int channel,
                void *ptr);
void *TPPI_get_src_channel_user_ptr(TPPI_NodeId dst, int chan-
                nel);
void TPPI_set_dst_channel_user_ptr(TPPI_NodeId src, int channel,
                void *ptr);
void *TPPI_get_dst_channel_user_ptr(TPPI_NodeId src, int chan-
                nel);
```

Each endpoint (source and destination) contains storage for an arbitrary pointer so that the user may associate application-specific structures with the channel. These functions provide access to that storage.

A.2.4.5 Transfer size limit

#define TPPI_MAX_CHANNEL_BYTES *implementation-specific*

The constant TPPI_MAX_CHANNEL_BYTES indicates the maximum number of bytes that can be transferred through a channel between calls to `TPPI_reset_channel()`.

A.2.5 Timers

Efficient timers are useful for implementing protocol time-outs and providing flexible forward-progress guarantees.

typedef void (*TPPI_TimerHandlerPtr)(void *user_ptr);

void TPPI_schedule_timer(int ticks, TPPI_TimerHandlerPtr fn, void *user_ptr);

Schedules a timer event for `ticks` units of time in the future. The units for `ticks` are implementation-dependent. After the timer event occurs, the handler function `fn` will be invoked by the protocol thread with the single argument `user_ptr`.

A.2.6 Thread management

void TPPI_resume_va(void *blk_va, int blk_len);

Resumes the set of threads suspended due block access faults on a particular block. The thread

must be blocked due to a page fault or block access fault. For page faults, the faulting instruction is reissued. For block access faults, the faulting instruction may be reissued or the access may be completed without reissuing (e.g., if the faulting access was a buffered store). The block access tag may or may not be checked again; that is, if the faulting access still conflicts with the tag value, whether the access completes or another block access fault occurs is implementation-dependent. Thus a user protocol must eventually change the access tag to make the access legal in order to achieve forward progress.

void TPPI_sleep(volatile int *sem_ptr);

Increments the semaphore pointed to by `sem_ptr`. If the resulting semaphore value is greater than zero, the calling thread is suspended until the semaphore value is less than or equal to zero.

void TPPI_wakeup(volatile int *sem_ptr);

Decrements the semaphore pointed to by `sem_ptr`. If the resulting semaphore value is equal to zero, any threads waiting on the semaphore (via `TPPI_sleep()`) will be resumed.

void TPPI_atomic_incr(volatile int *sem_ptr);

Increments the semaphore pointed to by `sem_ptr`. Because Tempest implementations may schedule threads concurrently or preemptively, it is unsafe for users to perform read-modify-write operations directly on semaphores.

Appendix B

Access Control Via Bus Snooping

The three Typhoon designs described in Chapter 3 and the Typhoon-0 prototype described in Chapter 4 implement fine-grain access control using bus-based snooping hardware. This appendix discusses the general requirements of this approach.

This access-control hardware leverages the support for bus-based cache coherence found in nearly all modern microprocessors. I will assume an invalidation-based protocol that employs copyback, allocate-on-write caches, where a write miss results in a read-invalidate (read-for-ownership) bus transaction. Update-based protocols and write-through caches do not necessarily preclude implementing access control via bus snooping, but these features are uncommon in current systems so they will not be addressed here.

The three sections of this appendix cover enforcing tag semantics, handling transactions that cause block access faults, and maintaining hardware cache consistency modifying tag values.

B.1 Enforcement of tag semantics

Access control is enforced by a bus monitor, a hardware module with associated tag storage that sits on the bus and observes the requests made by the processor(s). For each processor request, the monitor looks up the tag for the corresponding memory block. According to the block tag and the type of transaction, the bus monitor may also affect the transaction in one of two ways:

- If the bus transaction indicates a reference that conficts with the tag, the monitor must prevent the transaction from completing successfully and invoke a block access fault handler. The following section (Section B.2) discusses techniques for suspending the transaction.
- On a read miss to a ReadOnly block, the monitor forces the processor to load the block in a read-only state, so that subsequent reads hit but a subsequent write requires an invalidate request. The monitor relies on detecting this invalidate request to prevent writes to the block. These features are typically supported using dedicated open-collector bus signals that provide a wired-OR of the corresponding outputs of all snooping agents.

Table B.1 summarizes the actions that the bus monitor takes in response to observed processor transactions. In the cases labeled “fault”, the monitor suspends the transaction and invoke a block access fault handler. An invalidate request implies that a cache has a read-only copy of a block, so an invalidate should not occur when the block’s access tag is Invalid.

Table B.1: Bus monitor snooping behavior.

Bus request	Access tag		
	Writable	ReadOnly	Invalid
read	no action	force read-only	fault
read-invalidate	no action	fault	fault
invalidate	no action	fault	(not allowed)

B.2 Block access faults

The particular method used to handle faulting transactions depends on the features of the native bus protocol. There are three major alternatives: defer the transaction directly, cause the processor to retry the transaction, or abort the transaction and reissue the access via software.

If the processor supports cache-to-cache transfers on a bus with split transactions or deferred responses, the monitor may handle faulting transactions directly. The bus monitor simply takes responsibility for responding, as if performing a cache-to-cache transfer, and delays the response until the remote data has arrived. Because there is no software involved in resuming the computation, the latency from the arrival of the miss response to the successful completion of the faulting access can be greatly reduced. This approach assumes that another processor is available to execute the access fault handler.[1] Also, the processor that incurs the access fault may continue to compute if it has a non-blocking cache, possibly generating more access faults whose latency can be overlapped with the first. The bus monitor must have a set of buffers to track the pending transactions for which it is responsible. Ideally, the number of buffers matches the number of processors times the number of outstanding requests each processor supports; if fewer buffers are available, the monitor can abort transactions that fault when all buffers are occupied.

A second alternative—applicable to protocols such as Sun's MBus that do not support deferred transactions—is to retry the transaction rather than abort it. In this case, the processor's bus interface will rearbitrate for the bus and reexecute the transaction. As with the direct approach, the latency of resuming the access is low because no software is involved, but another processor must execute the access fault handler. To avoid consuming bus bandwidth with pointless retries—interfering with the execution of the fault handler, message handlers, and other computation threads—the system should be able to mask the

1. This requirement can be avoided if the processor can be interrupted while a memory access is outstanding, as in the Alewife system [AKK+93]. Unfortunately, current commercial microprocessors do not provide this feature.

faulting processor from arbitration, preventing it from reacquiring the bus until the access can be satisfied. In some machines, this can be done by modifying an arbiter register. The Stanford DASH system [LLG+92] uses this technique, although it requires hardware modification to the commercial system on which the nodes are based. The simulated Typhoon systems in Chapter 3 also use this approach.

Finally, the transaction can be aborted, typically by responding with an error status. The faulting processor will invoke an exception handler, which must recognize that the cause of the bus error is an access fault and wait for notification to restart the faulting instruction. (If the access fault handler will be executed on the same processor, the exception handler must save the current state and begin execution of the access fault handler.) Aborting transactions is a heavy-handed approach to access faults. The performance cost of taking exceptions on deeply pipelined superscalar processors is high. The latency of restoring processor state and restarting the access lies on the critical path for misses. The processor must provide a restartable (but not necessarily precise) exception on an aborted bus transaction. The Typhoon-0 prototype described in Chapter 4 uses this approach.

B.3 Tag value modification

In addition to enforcing access control based on existing tag values, the bus monitor must allow modification of tag values. Because the bus monitor cannot observe or affect accesses which hit in the processor caches, it must maintain these invariants:

- If a block's tag is Invalid, it is not present in any processor caches.
- If a block's tag is ReadOnly, it may be present in one or more processor caches in a read-only state.
- If a block's tag is Writable, it may be present in one or more processor caches in any state. The native bus protocol will guarantee that at most one cached copy is writable at any given time.

Upgrading access—that is, changing from Invalid to ReadOnly or Writable, or from ReadOnly to Writable—relaxes the restrictions on caching blocks, so no action is required

other than modifying the value in tag storage. However, when access is downgraded—from Writable to ReadOnly, or from ReadOnly or Writable to Invalid—copies that may reside in processor caches must be invalidated.[1] If the original access tag is Writable, the only up-to-date copy may be in a cache; this modified copy must be written back to memory. These operations must occur atomically with the update of the value in tag storage. In general, to downgrade from Writable, the bus monitor must acquire the bus and atomically perform a read-invalidate, write the block data to memory, and modify the value in tag storage. (The PowerPC 60X bus [AAWC94] supports a block flush transaction that forces any cache with a modified copy to do a writeback, which could be used in place of the read-invalidate/write sequence.) To downgrade from ReadOnly, a simple invalidate is sufficient, with no memory write.

1. On a transition from Writable to ReadOnly, it is theoretically sufficient to remove write access from any cached copies. In practice, ownership must also be taken away from the cache, which in most protocols requires an invalidation.