To appear: *COMPCON Spring 95*, March 1995.

Tempest: A Substrate for Portable Parallel Programs

Mark D. Hill, James R. Larus, and David A. Wood
Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton St.
Madison, WI 53706 USA
wwt@cs.wisc.edu

Abstract

This paper describes Tempest, a collection of mechanisms for communication and synchronization in parallel programs. With these mechanisms, authors of compilers, libraries, and application programs can exploit—across a wide range of hardware platforms—the best of shared memory, message passing, and hybrid combinations of the two. Because Tempest provides mechanisms, not policies, programmers can tailor communication to a program's sharing pattern and semantics, rather than restructuring the program to run with the limited communication options offered by existing parallel machines. And since the mechanisms are easily supported on different machines, Tempest provides a portable interface across platforms. This paper describes the Tempest mechanisms, briefly explains how they are used, outlines several implementations on both custom and stock hardware, and presents preliminary performance results that demonstrate the benefits of this approach.

1 Introduction

Uniprocessor computers flourish, in part, because they share a programming model suitable for programs written in many styles and high-level languages. The common model allows programmers to select a language appropriate for their applications and to transfer most programs between computers without worrying about the underlying machine architecture. Computers did not always provide such a congenial environment. Several decades ago, every program was crafted for a particular machine in its own, machine-specific assembly language.

Parallel computers still languish at this stage. They do not share a common programming model or support many vendor-independent languages. To address this problem, the Wisconsin Wind Tunnel research project developed the *Tempest* interface, which provides a common parallel computer programming model. Figure 1 summarizes this paper by showing how Tempest provides a substrate that allows compilers and programmers to exploit different programming styles across a wide range of parallel systems.

Tempest provides the mechanisms necessary for efficient communication and synchronization: active messages, bulk data transfer, virtual memory management, and fine-grain access control. The first two are commonly-used mechanisms for short, low-overhead messages and efficient data transfer, respectively. The latter two mechanisms allow a program to control its memory, so it can implement a shared address space. Fine-grain access control is a novel mechanism that associates a tag with a small block of memory (e.g., 32–128 bytes). The system checks this tag at each `LOAD` or `STORE`. Invalid operations—`LOAD`s of invalid blocks or `STORE`s to invalid or read-only blocks—transfer control to an application-supplied handler. Section 2 describes Tempest in more detail.

Because Tempest provides mechanisms, not policies, it supports many programming styles. Current parallel machines are designed for a single programming style—message passing or shared memory—which forces programmers to fit a program to a machine rather than allowing them to choose the tools appropriate for the task at hand. Programs written for a particular parallel machine are rarely portable, which has limited the appeal and use of these machines. By separating mechanism from policy, Tempest allows a programmer to tune a program without restructuring it. In particular, Tempest supports custom shared-memory coherence protocols that provide an appli-

---

This paper is a summary of research performed by the Wisconsin Wind Tunnel project. Many ideas described above were previously introduced in other papers: Tempest, Typhoon, and Stache [21], Blizzard [22], custom protocols [6], and Loosely Coherent Memory [15]. Abstracts [9] and information on our papers can be found at URL:

`http://www.cs.wisc.edu/~wwt`

This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550, NSF PYI/NYI Awards CCR-9157366, MIPS-8957278, and CCR-9357779, NSF Grants CCR-9101035 and MIP-9225097, DOE Grant DE-FG02-93ER25176, University of Wisconsin Graduate School Grant, Wisconsin Alumni Research Foundation Fellowship and donations from A.T.&T. Bell Laboratories, Digital Equipment Corporation, Sun Microsystems, Thinking Machines Corporation, and Xerox Corporation. Our Thinking Machines CM-5 was purchased through NSF Institutional Infrastructure Grant No. CDA-9024618 with matching funding from the University of Wisconsin Graduate School.

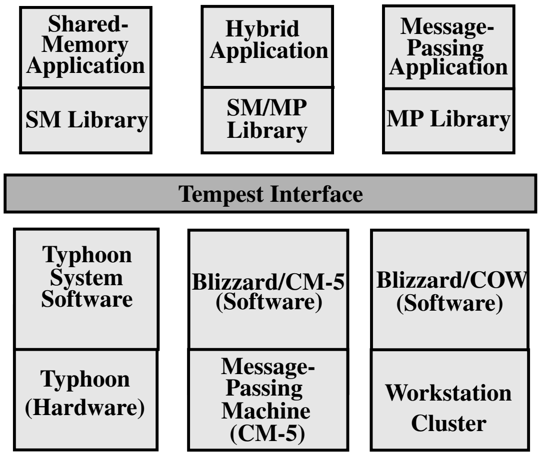

FIGURE 1. The Tempest interface. This figure summarizes the paper: Section 2 describes the Tempest interface, our substrate for parallel programming on a wide range of platforms. Section 3 discusses Tempest's support for different programming styles (above Tempest). Section 4 describes alternative Tempest implementations

cation with both a shared address space and efficient communication. Section 3 discusses how Tempest supports different programming styles.

Tempest's success depends on effective implementations throughout the parallel machine pyramid (Figure 2). Uniprocessor and multiprocessor workstations and servers form the base of this pyramid. Most programs are, and will continue to be, developed on these inexpensive and ubiquitous machines. Larger jobs with low communication requirements may require a step up to networks of desktop workstations (NOWs). Networks of dedicated workstations, possibly with additional special hardware, can trade higher cost for increased performance. Finally, at the pyramid's apex, supercomputers and massively parallel processors (MPPs) offer the highest performance for those able to pay for it.

Section 4 describes several Tempest implementations. *Typhoon* is a proposed high-end design. It uses a network interface chip containing the inter-processor network interface, a processor to run access-fault handlers, and a reverse translation lookaside buffer to implement fine-grain access control. The *Blizzard* system implements Tempest on existing machines without additional hardware. It currently runs on a non-shared-memory Thinking Machines CM-5 and uses one of two techniques to implement fine-grain access control. *Blizzard-E* uses virtual memory page protection and the memory system's ECC (error correcting code) to detect access faults. *Blizzard-S* rewrites an executable program to add tests before shared-memory `LOAD` and `STORE` instructions. We are currently porting Blizzard to the Wisconsin COW (a **C**luster **O**f **W**orkstations).

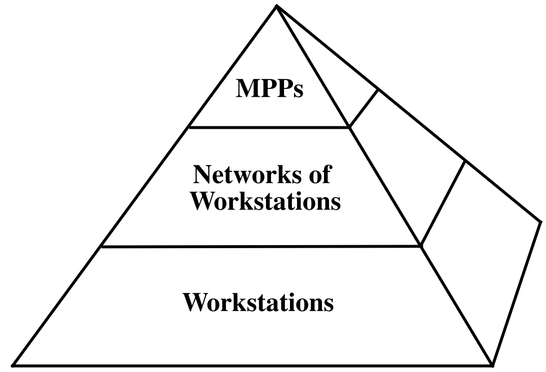

FIGURE 2. The parallel machine pyramid.

Section 5 presents preliminary performance numbers, which show that, with adequate hardware support, shared memory implemented on Tempest is competitive with hardware shared memory. In addition, Blizzard implementations on stock hardware offer acceptable shared-memory performance on current machines. However, the real benefits and large performance improvements come from the custom coherence protocols made possible by Tempest.

2 Tempest Mechanisms

To form a portable parallel programming substrate, Tempest must provide mechanisms that suffice to implement most parallel programming abstractions and that permit efficient implementations across a broad range of parallel machines.

As a common denominator, Tempest assumes a distributed memory hardware base constructed from *P* processing nodes (see Figure 3) [21]. To simplify the exposition, this paper assumes a single program multiple data (SPMD) programming model with one processor per node and one computation thread per processor. Each thread runs in a private address space augmented by an optional shared segment. Shared-memory and hybrid applications can use Tempest mechanisms (or Tempest shared-memory libraries) to manage the shared address space.

The four types of Tempest mechanism are:

Active messages are short, low-latency messages [23]. They are useful for sending control, synchronization, or short data messages. Upon receipt of an active message, the system invokes the handler specified by the message and passes two arguments: the sender's processor number and the message length. The handler reads the message body from the incoming message queue.

Bulk data transfer efficiently moves large quantities of data between nodes, much like conventional DMA. In most systems, a single transfer is less costly than a sequence of shorter messages, so Tempest supports both mechanisms.

Virtual memory management allows an application to control its virtual address space. With this mechanism, Tempest programs can support page-granularity shared memory similar to distributed shared memory (DSM) systems [18,1,10]. These systems use virtual memory page protection to identify non-local data (by mapping it out of a processor's address space). Unfortunately, large pages (typically, 4–8K) causes expensive false sharing when an application places writable data for two processors on the same page.

Fine-grain access control alleviates this problem by greatly reducing the granularity of access control. It associates a tag with each small, aligned memory block (e.g., 32–128 bytes) and atomically checks a referenced block's tag at every `LOAD` or `STORE` instructions. The tags are *Invalid*, *Read-Only*, and *Read-Write*. `LOAD`s of *Invalid* blocks and `STORE`s to *Invalid* or *Read-Only* blocks invoke user-level handlers. This mechanism enables Tempest to support coherence at the same granularity as hardware shared-memory systems [17].

Tempest provides mechanisms to implement programming paradigms, but leaves policy to user-level code [5]. Table 1 summarizes the Tempest mechanisms that support different programming paradigms. This code may reside in unprivileged libraries, be generated by a compiler, or be written specifically for an application. By separating policy from mechanism, Tempest avoids the pitfalls inherent in system-level policies that are too general and expensive or too specific and incomplete [24].

	Active Messages	**Bulk Data Transfer**	**Virtual Memory Mgmt.**	**Fine-Grain Access Control**
Message Passing	X	X		
Data Parallelism	X	X		
NUMA Shared Memory	X			
Coherent Shared Memory	X		X	X
Hybrid	X	X	X	X

TABLE 1. Use of Tempest mechanisms.

3 Using Tempest

Perhaps the best way to understand Tempest is to see how it is used. With its mechanisms, coarse-grain message passing (e.g., PVM [7]) or NUMA (no caching) shared memory (e.g., Split-C [4]) are easily implemented.

More interesting are cache-coherent shared memory and hybrid models that exploit program locality by caching data at processors that reference them. *Stache* is an application-level library that uses Tempest mechanisms to implement sequentially consistent, transparent shared memory. A unique feature is that Stache uses a programmable fraction of a node's physically local memory to cache data from remote processors (the "stache"). This large, fully-associative cache reduces memory latency and message traffic by keeping data that does not fit in the hardware cache near the processor that accessed it.

Stache is similar to DSM systems in some respects. Each page in the user-managed shared segment has a "home" node. When a non-home processor first references a page, it is not mapped and, consequently, the reference causes a page fault that invokes a Tempest user-level handler. That handler allocates a local page frame, maps the page, and obtains the referenced location from its home.

Stache differs from DSM systems because it uses fine-grain access control to mitigate false sharing. When a new page is allocated, all its blocks are tagged *Invalid*. The protocol then obtains the referenced block from its home node. Only this block's tag is changed. A subsequent reference to another block in the page causes a fine-grain access control fault, which invokes a handler to obtain the block. Fine-grain access control permits processors to read and write different blocks on the same page without false sharing.

Stache, and other sequentially-consistent shared-memory protocols, send more messages than necessary for some communication patterns. For example, Stache and other write-invalidate protocols require four messages to update a value in a producer and consumer relationship: consumer request, producer response, producer invalidate, and consumer acknowledgment. This excess communication is a consequence of "one-size fits all" coherence policies, which implement widely-applicable semantics that can be unnecessarily general in many situations.

Tempest mechanisms enable a compiler or programmer to retain the advantages of shared memory (a shared address space and caching [3,14]) but communicate more efficiently by customizing a coherence protocol to an application's sharing patterns and semantics. To demonstrate these ideas, we developed custom update protocols for three applications: NAS Appbt, Berkeley EM3D, and SPLASH Barnes [6]. The three protocols differ substantially in how they detect sharing. Appbt's protocol exploits the application's static and predictable sharing pattern to send updates directly. Barnes' dynamic and changeable sharing requires updates to be forwarded through a home node that maintains a sharing list. Finally, EM3D's sharing pattern is static, but unknown until run time. EM3D uses an augmented version of Stache to record the sharing in the first iteration and a direct update protocol for subsequent

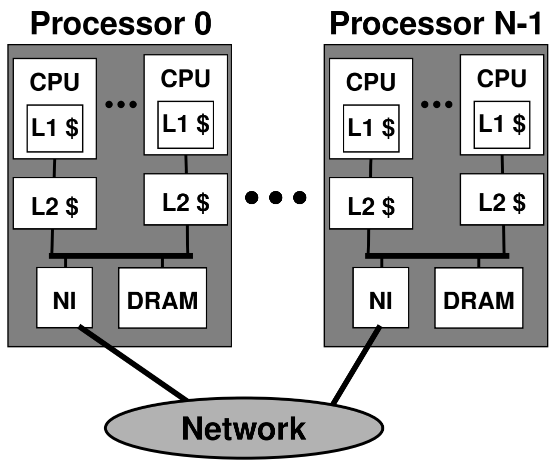

FIGURE 3. Base parallel machine hardware.

iterations. Section 5 presents results that demonstrate the large gains possible from custom coherence protocols.

Custom protocols can also help support high-level parallel programming languages, which offer semantically attractive constructs that can be difficult to implement efficiently on parallel machines. An example is the copy-in, copy-out semantics that Fortran 90 provides for some data structures and built-in functions. The C** data parallel programming language [13] offers this semantics for general routines and data structures. We used Tempest to assist a compiler in efficiently supporting this language semantics. Loosely Coherent Memory (LCM) [15] implements fine-grain copy-on-write operations, which allows C** programs to run correctly, even when compiler cannot analyze their sharing pattern because of pointers or function calls.

4 Implementing Tempest

To develop and demonstrate the Tempest interface, we implemented it on several platforms with different levels of hardware support. *Typhoon* is a hardware implementation that uses a highly-integrated custom chip. *Blizzard* is a software-only system that runs on an unmodified CM-5.

Our implementations assume a base architecture of P nodes connected by a point-to-point network (see Figure 3). Each node is similar to a workstation, with one or more commodity processors with caches, a MOESI cache-coherent memory bus, memory (DRAM), and memory controller (not shown). A parallel machine built from these nodes connects them with a point-to-point network that is accessed through a network interface (NI).

Typhoon implements Tempest through the network interface chip depicted in Figure 4 [21]. Typhoon's Network Interface (NI) includes a reverse translation lookaside buffer (RTLB) to implement fine-grain access control, a processor to run user-level handlers, DMA logic to support block transfers, and the network interface itself.

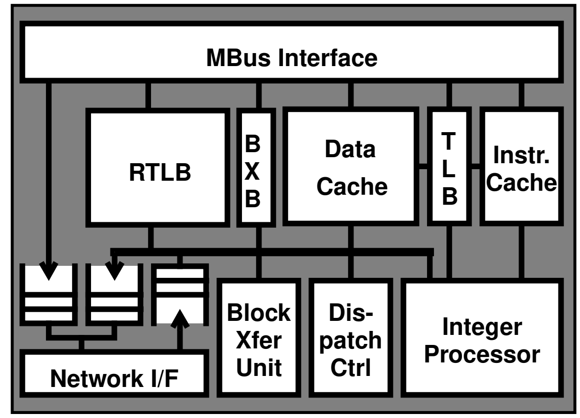

FIGURE 4. Typhoon's Network Interface.

Typhoon logically validates access control tags on all `LOAD`s and `STORE`s—without modifications to a node's processor, cache, or memory controller. Consider the situation when a processor loads a block that it has not accessed before. The reference misses in the processor's hardware cache(s) and appear on the memory bus. As the memory processes the request, the NI snoops the physical address and uses its RTLB to find the block's tag.[1] If the tag is *Read-Write*, the NI remains inactive and the block is loaded into hardware cache(s), where it can be subsequently accessed at full speed. If the tag is *Read-Only*, the NI asserts the "shared" line, so subsequent `LOAD`s succeed but `STORE`s access the memory bus again for another tag check. On `STORE`s to *Read-Only* blocks or `LOAD`s and `STORE`s of *Invalid* blocks, the NI delays the requesting processor and runs a user-level handler on its processor. In all cases, the NI follows the bus's snooping protocol and appears to be another processor. In some sense, the NI is the agent for other nodes in the system that helps achieve global coherence with only locally-coherent hardware.

Blizzard implements Tempest on a CM-5 [22]. The CM-5 provides no support for shared memory but does fit the machine model depicted in Figure 3.[2] The CM-5's network interface is mapped into a user program's address space and provides fast messages. The Tempest virtual memory management mechanisms are provided by an extended CM-5 node kernel [19].

Blizzard implements fine-grain access control through two alternative methods. First, *Blizzard-E* uses a CM-5 diagnostic mode to intentionally set double-bit ECC errors in *Invalid* blocks. As depicted in Figure 5, a `LOAD` or `STORE` that misses in the CM-5's hardware cache goes to memory for a cache-line fill. The fill succeeds for valid tags, but the ECC error for an *Invalid* tag causes a trap, which Blizzard-E vectors to a user-level handler. The

1. RTLB misses delay the processor while the NI loads the entry from memory. Special mappings treat private memory as Read-Write [21].

2. Blizzard does not use the CM-5 vector units.

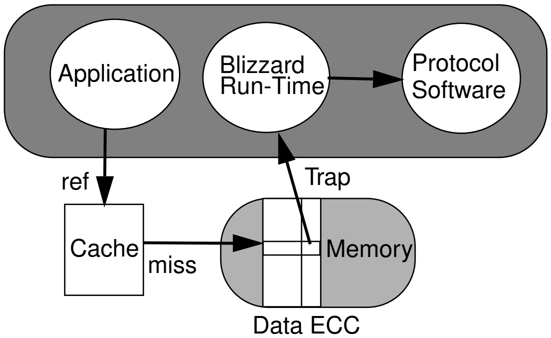

FIGURE 5. Blizzard-E: Tempest on a CM-5

Read-Only state is synthesized with page-level protection. No ECC coverage is lost with this approach, because Blizzard-E verifies that ECC errors arise from *Invalid* blocks, *Invalid* blocks do not contain valid data, and Blizzard sets double-bit errors on multiple doublewords in a memory block. Blizzard-E, however, will not work on processors that do not allow restartable exceptions on ECC errors.

To increase portability, we developed the all-software *Blizzard-S*. Blizzard-S modifies executable programs (`a.out` files) with a tool based on EEL [16] to add an explicit tag check before all `LOAD`s and `STORE`s that could access the shared segment. The current version uses several optimizations to reduce the frequency of tests and implement them in five instructions, in the best case. Protocol software and application executables (before EEL) are identical for Blizzard-S and Blizzard-E.

We are currently porting Blizzard to a network of dedicated workstations. The Wisconsin COW (**C**luster **O**f **W**orkstations) is built from 40 Sun SPARCstation-20 workstations, each with two Ross HyperSparc processors. The nodes will be interconnected with a Myricom Myrinet. Blizzard/COW will implement fine-grain access control three ways: with ECC (like Blizzard-E), by executable editing (like Blizzard-S), and with custom hardware that snoops the memory bus. Blizzard/COW presents some new challenges, including longer network latencies, a commodity operating system (Solaris 2.4), and dual processors.

5 Preliminary Performance

We have reported preliminary performance results for these ideas in several papers. The numbers, unfortunately, are not directly comparable, because that they come from different systems (simulation or implementation), different Tempest implementations, different benchmarks, and different protocols. Reinhardt et al. [21] used simulations on the Wisconsin Wind Tunnel [20] to compare Typhoon against a CC-NUMA machine modeled after the Stanford DASH [17]. The results showed that Typhoon performs very closely to the all-hardware implementation when both systems ran their base coherence protocols. Typhoon performed slightly worse when a program's working set fit in the CC-NUMA's 256KB hardware cache and slightly better when it did not. However, Typhoon performed up to 35% better for EM3D when running a custom update protocol that would be difficult to implement in hardware.

Schoinas et al. [22] present early measurements for Blizzard running on a 32-node CM-5. The results show that Blizzard-S is a viable implementation that runs than two times slower than Blizzard-E, in the worst case. More recent versions of Blizzard-S closed this gap to 1.5X and run some programs faster than Blizzard-E—when high miss rates makes Blizzard-S's lower miss overhead more important that its higher lookup overhead at each access.

Finally, Falsafi et al. [6] demonstrate the enormous potential of custom coherence protocols. They improved the 32-processor Blizzard-E performance of NAS Appbt, Berkeley EM3D, and SPLASH Barnes by factors of 5.7, 16.0 and 1.4—over optimized shared memory versions—by changing the coherence protocols, as described in Section 3. On the CM-5, the shared-memory EM3D ran as fast as a native message-passing version.

6 Related Work

Several interfaces share Tempest's goal of providing portability among parallel machines. PVM [7] is a widely-used, coarse-grain message-passing system. Berkeley's Active Messages [23] provides a portable interface for fine-grain messages, but, unlike Tempest, no support for transparent caching. DSM systems, such as Rice's Munin [1] and Treadmarks [10], support shared memory, but since their coherence is limited to page granularity, they require more complex semantic models to mitigate the adverse effects of false sharing. Tempest's fine-grain access control avoids page-level false sharing.

Several other systems also support custom protocols, including MIT Alewife [2], Rice Munin [1], and Stanford FLASH [12]. We are not aware, however, of another system that gives a user complete, protected control over protocols. Some Tempest protocols have predecessors. In particular, Stache is similar to a DSM protocol extended to cache-sized blocks and to a software implementation of the hardware COMA protocols of the Data Diffusion Machine [8] and Kendall Square KSR-1 [11].

Several machines share features with Tempest implementations. The MIT J-Machine shares Tempest's goal of providing mechanisms, not policy, but uses a custom processor [5]. Stanford FLASH is similar in many respects to Typhoon. FLASH, however, uses a custom memory controller, rather than a snooping device, runs handlers on all hardware caches misses, and runs protocols in privileged mode without address translation. Blizzard's kernel interface and ECC use come from its ancestor, the Wisconsin Wind Tunnel [20].

7 Summary

The Tempest mechanisms provide a substrate for portable and efficient parallel programs. A programmer or compiler writer can use these mechanisms to implement an efficient parallel program through the time-proven process of successive refinement. Most programmers will start with a shared memory program that uses a pre-written transparent shared-memory library such as Stache. As the program develops, a programmer will find bottlenecks, which can be eliminated without restructuring the program by choosing another shared-memory protocol, such as the update protocols discussed in this paper. Of course, programmers seeking the highest level of performance can both write their own protocols and use message passing where appropriate. Tempest supports all of these approaches across a wide range of parallel systems.

8 Acknowledgments

We would like to thank the many people who made important contributions to the Wisconsin Wind Tunnel project: Douglas Burger, Satish Chandra, Sashikanth Chandrasekaran, Trishul Chilimbi, Glen Ecklund, Babak Falsafi, Alain Kagi, Sangtae Kim, Rahmat Hyder, Alvin Lebeck, James Lewis, Shubhendu Mukherjee, Subbarao Palacharla, Steven Reinhardt, Brad Richards, Anne Rogers, Timothy Schimke, Eric Schnarr, Yannis Schoinas, Steve Swartz, Frank Trankle, and Guhan Viswanathan. We would also like to thank David Patterson and Guri Sohi for their helpful comments on this paper.

References

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

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

[3] Satish Chandra, James R. Larus, and Anne Rogers. Where is Time Spent in Message-Passing and Shared-Memory Programs? In *Proc. of the Sixth Inter. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pp. 61–75, October 1994.

[4] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. of Supercomputing '93*, pp. 262–273, Nov. 1993.

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

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

[7] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. PVM 3 users's Guide and Reference Manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1994.

[8] Erik Hagersten. Toward Scalable Cache Only Memory Architectures. The Royal Institute of Technology Swedish Institute of Computer Science, Oct. 1992. Stockholm, Sweden. Ph.D. Thesis, Swedish Institute of Computer Science Dissertation Series 08.

[9] Mark D. Hill, James R. Larus, and David A. Wood. The Wisconsin Wind Tunnel Project: An Annotated Bibliography. *Computer Architecture News*, 22(5):19–26, December 1994. (Mosaic location is: http://www.cs.wisc.edu/p/wwt/Mosaic/wwt.html).

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

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

[12] Jeffrey Kuskin et al. The Stanford FLASH Multiprocessor. In *Proc. of the 21st Annual Inter. Symposium on Computer Architecture*, pp. 302–313, April 1994.

[13] James R. Larus. C**: a Large-Grain, Object-Oriented, Data-Parallel Programming Language. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages And Compilers for Parallel Computing (5th Inter. Workshop)*, pp. 326–341. Springer-Verlag, August 1993.

[14] 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.

[15] James R. Larus, Brad Richards, and Guhan Viswanathan. LCM: Memory System Support for Parallel Language Implementation. In *Proc. of the Sixth Inter. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pp. 208–218, October 1994.

[16] James R. Larus and Eric Schnarr. EEL: Machine-Independent Executable Editing, 1994. Submitted for publication.

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

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

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

[20] Steven 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 *Proc. of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pp. 48–60, May 1993.

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

[22] Ioannis Schoinas, Babak Falsafi, Alvin R. Lebeck, Steven K. Reinhardt, James R. Larus, and David A. Wood. Fine-grain Access Control for Distributed Shared Memory. In *Proc. of the Sixth Inter. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VI)*, pp. 297–307, Oct. 1994.

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

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