Parallel Computer Research in the Wisconsin Wind Tunnel Project

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

{markhill, larus, david}@cs.wisc.edu
http://www.cs.wisc.edu/~wwt

Abstract

The paper summarizes the Wisconsin Wind Tunnel Project's research into parallel computer design and methods. Our principal design contributions—*Cooperative Shared Memory* and the *Tempest Parallel Programming Substrate*—seek to balance the programming benefits of a shared address space with facilities for low-level performance optimizations.

The project has refined and compared a variety of ideas with a unique mixture of techniques that include micro-architecture-level simulation, software prototyping, and rapid hardware prototyping. An important by-product of this research has been innovative tools, such as the *Wisconsin Wind Tunnel* and the *Executable Editing Library.*

1 Introduction

Parallel computers show great promise (and have shown great promise for 30 years!). Parallel hardware offers an attractive solution for problems whose computation needs outstrip even the rapidly improving uniprocessors. Equally important, users are discovering that parallel computer hardware can be more cost-effective than a uniprocessor, because (a) it often uses microprocessors rather than esoteric supercomputer processors, and, somewhat surprisingly, (b) it makes cost-effective use of expensive large memories. For example, we found that a 16-processor Silicon Graphic multiprocessor was more cost-effective for problems requiring 1GB of memory than a uniprocessor, whenever speedups were larger than just 4.3 [43].

So why is parallel computing such a mixed success? At the high-end, massively parallel processors (MPPs) failed because their custom hardware and software incurred high fixed costs that limited these machines to the most demanding applications, and because their novel and unforgiving programming environments required programs to be crafted for a particular machine.

In the middle range, networks of workstations (NOWs) offer lower hardware cost than an MPP by building on commodity hardware and software. Most NOWs are limited to a message-passing programming model and suffer from high communication latency, attributable to both hardware and software. Academic research (e.g., Berkeley) and the commercial success of the NOW-like IBM SP-1 and SP-2 make it likely that NOWs will remain a promising parallel alternative.

Nevertheless, the success story of parallel computing is symmetric multiprocessors (SMPs). They are cost-effective, because the marginal performance gain—both to reduce job latency of parallel programs and increase job throughput with multiprogramming—of an additional processor tends to be significantly higher than the marginal hardware cost increase [43]. Also, SMP's shared address space programming model is an evolutionary step for uniprocessor software, particularly modern software written using threads [12]. SMPs are increasingly common as commodity processors, such as Intel's P6, and operating systems, such as Microsoft's Windows NT, routinely support small-scale SMP systems. SMPs' Achilles' heel is their shared bus, which limits their size and scalability.

This paper is a summary of research performed by the Wisconsin Wind Tunnel project. Most ideas described herein have been previously published. Appendix A reprinted, with permission, from COMPCON '95 (San Francisco, California, March 1995, pp. 327–332). Abstracts and information on our papers can be found at URL:

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

This work is generously supported by NSF grant MIP-9225097 and Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant #F33615-94-1-1525 and ARPA order no. B550. Additional support has been provided by NSF PYI/NYI Awards CCR-9157366, MIPS-8957278, and CCR-9357779, NSF Grant CCR-9101035, 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. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

Existing systems accommodate only 20–32 processors. Moreover, faster processors will limit future SMPs to no more than about 16 processors, which is too small for many important applications.

When users outgrow an SMP they are faced with two alternatives: (a) re-write their applications or (b) use distributed shared memory. Re-writing an application to a variety of programming models and vendor-specific programming environments imposes a software cost that will strangle this approach to parallel computing. Although considerable efforts have succeeded in standardizing message-passing with MPI and data-parallel programming with HPF, the problem of two programming models remains a fundamental barrier to scalable parallel computing. We believe the shared-memory programming model will dominate the next wave of parallel computing, as SMPs become ubiquitous and shared address spaces become the programming norm.

The second option to expand beyond an SMP is d*istributed shared memory* (DSM), which extends the shared memory programming model beyond a bus. Traditional DSM implementations are either (mostly) hardware shared memory (HSM) or virtual shared memory (VSM). HSM machines, such as Kendall Square KSR-1, Stanford DASH, MIT Alewife, and Sequent STiNG use hardware to retrieve and cache blocks, implement a global physical address space, and run a single (multithreaded) kernel. By contrast, VSMs, such as Yale/Princeton Ivy, Rice Munin, and Rice Treadmarks, use virtual memory mechanisms to cache pages, do not need a physical global address space, and run a kernel per node. In many respects, HSMs and VSMs complement each other. HSMs approach SMP performance, but require extensive custom hardware (e.g., cache controllers, directories, or attraction memories) and custom operating systems—key drawbacks of MPPs. VSMs, on the other hand, use commodity hardware and software, sometimes with modest hardware accelerators. In general, VSMs' performance is lower because their coherence is at a virtual memory page granularity (e.g., 4K bytes). To improve performance, some VSM systems, such as MIT CRL and Rice Treadmarks, use alternative programming models that reduce the coherence traffic necessary for fine-grain sharing [4, 20].

The goal of our research is combine the best of HSMs and VSMs (and new results from our research). Like HSMs, we support the SMP's fine-grain coherence, but seek to reduce hardware complexity (and cost) and, at the same time, permit programmers with a richer collection of mechanisms for improving performance. Like VSMs, we exploit existing hardware and rely on user-level support to improve performance over simple shared memory. Finally, like uniprocessors, our work rests on the belief that software must be portable to machines at a wide range of cost-performance points.

The next sections discuss our design proposals, methods, and technology transfer. Appendix A describes Tempest and Tempest implementations in greater detail.

2 Design Overview

2.1 Cooperative Shared Memory

Our initial design, *Cooperative Shared Memory*, was an evolutionary extension to conventional HSM software and hardware. Cooperative Shared Memory asks programmers to identify expected data sharing behavior through the Check-In/Check-Out (CICO) performance annotations so that the system could efficiently handle subsequent references, with less complex hardware than a traditional HSM (e.g., Dir_1SW [18]).

The key CICO annotations are:

- *check_out_X*: expect exclusive data access,
- *check_out_S*: expect shared data access, and
- *check_in*: expect end of data access.

CICO annotations serve several purposes [26]. First, they can direct the underlying coherence protocol to perform performance enhancing operations (e.g., cache flush on *check_in*). Second, when inserted by the programmer, they can force a reasoning about shared-memory communication that may suggest code improvements (e.g., moving communication out of an inner loop). Third, when inserted automatically (e.g., with a trace-based tool like Cashier [10]), they can show a programmer where communication actually occurs.

CICO annotations can be used aggressively to optimized the expected case, because they never affect program semantics. Even randomly-inserted CICO annotations will not change a program's possible executions. This separation of performance and correctness makes automatic use of CICO annotations much simpler than software cache coherence management, which must be correct to preserve semantics.

CICO alters program behavior and permits simpler coherence protocols, such as Dir_1SW [18]. Dir_1SW uses several state bits and a single pointer/counter field per block and, more importantly, does not have to implement complex composite state transitions, as required by the more conventional Dir_NNB protocol. The pointer/counter either identifies a single writer or counts readers. Simple hardware entirely handles cases conforming to the CICO model by updating the pointer/counter and forwarding data to a requesting processor. No cases require multiple messages (beyond a single request-response pair) or transient states. Programs not conforming to the CICO model or using CICO incorrectly run correctly, but trap to system

software that performs more complex operations (in a manner similar to MIT Alewife [1]). Hardware for example, need not handle the complex case of a writer encountering many extant shared copies, because proper use of *check_in's* make this case rare.

Extensive simulations showed that Cooperative Shared Memory reduced hardware complexity relative to an HSM, allowed moderate interconnection network latency to be tolerated, and was a good design point [42,34].

We next asked if we could:

- reduce hardware complexity further,
- tolerate even longer interconnection network latencies, and
- provide solutions at many price-performance points.

A particularly frustrating aspect of performance optimization in a shared memory model, such as CSM, is that sometimes a message is exactly the right communication mechanism [25,6]. CSM directives can approximate a message send, but the approximation is not semantically perfect and costs performance. So, we added the question:

- can messages be integrated with coherent shared memory in a portable way?

Our affirmative answer to these questions is the *Tempest* parallel programming substrate.

2.2 Tempest

Tempest is a portable interface that provides caching, naming, and communication mechanisms that allow user-level software to provide applications with SMP-like shared memory, message-passing, or hybrid models. Tempest portably integrates shared memory and messages, while allowing:

- simpler hardware, including running on an unmodified NOW,
- the selected use of message-passing or custom coherence protocols to tolerate greater communication latencies, and
- many implementations, including simulated hypothetical machines (e.g., Typhoon), an MPP (Blizzard on a CM-5), a NOW (Blizzard on Wisconsin COW), and a NOW with selected hardware acceleration (Wisconsin COW with T0, designed with the help of Sun Microsystems).

Tempest provides two classes of messages. First, active messages—like Berkeley's [41]—transfer control information (e.g., requests for data), and small amounts of data (e.g., a 32-byte "cache" block) [33]. Second, bulk data transfer primitives—like CM-5 channels—provide higher bandwidth for large messages, which can afford the higher start-up cost.

As defined so far, Tempest provides messages, which supports message-passing applications and message passing within shared-memory applications, but does not integrate messaging and shared memory. Tempest's final two mechanisms provide this integration.

First, Tempest allows user-level software to control how pages in a special segment are mapped, using virtual memory mechanisms similar to those used by VSM systems. Consistent use of these mechanisms provides page-based coherent shared memory.

The final, novel Tempest mechanism is f*ine-grain access control*, which allows user-level software to tag blocks (e.g., 32 bytes) as read-write, read-only, or invalid. Combined with the other mechanisms, fine-grain access control supports shared-memory implementations that maintain coherence on the same granularity—i.e., cache blocks—as SMPs.

With Tempest, a standard library, a compiler run-time system, or a demanding application program can implement a shared-memory coherence protocol. Most programmers will develop their applications using transparent shared memory (obtained by linking their program with a standard protocol). After a program runs correctly, a programmer can improve its performance by selecting alternative library protocols to manage key data structures that are bottlenecks [14,7,9,35]. If no protocol performs well, the programmer can write a custom protocol for the program. Similarly, compilers can use custom protocols to implement higher-level programming language constructs or optimize compiled programs [27].

A commonly misunderstood aspect of Tempest is who—in practice—will write the policies. Tempest leaves policy to user-level software. People have interpreted this to mean every programmer must write their own coherence policy. Just as very few programmer today write assembly language or make kernel modifications, we expect that most programmers will rely on coherence policies provided by compilers or system libraries.

2.3 Tempest Implementations

Tempest would be a paper tiger if not implemented. Tempest would be a mildly interesting design if it could not be implemented on several platforms. We have spend considerable effort to ensure that neither situation happens. The current Tempest systems are:

- Blizzard/CM-5 implements Tempest on a Thinking Machines CM-5 using access control implemented with either (a) executable editing or (b) purposely set bad ECC [40]. It requires significant kernel modifications to add user-level virtual memory to the native CMOST operating system.

- Blizzard/COW implements Tempest on a network of 40 dual-processor SPARCstations-20s connected with Myricom Mryinet. Access control is implemented with (a) executable editing, (b) purposely set bad ECC, or (c) an Mbus board implemented in cooperation with Sun Microsystems. System code is in the form of loadable device derives that require no changes to the Solaris 2.4 kernel.
- Several simulated implementations that explore new design options, including Typhoon [38], Typhoon-1, and Typhoon-2 [39].

Appendix A describes Tempest and the implementations in more detail.

3 Methods

Research in parallel computer design requires a mix of evaluation methods. Abstract modeling helps in the initial phases, but the real work is at the level of detail that can be evaluated only with simulation and prototyping. Detailed evaluation methods are necessary because abstractions in engineering, in general, and computer systems, in particular, are approximations. Extended manipulation of abstractions, without reference to the details, often leads to unworkable solutions. Simulating and building are necessary.

3.1 Our Approach

A widely held, but false, dichotomy is whether it is better to simulate (build wind tunnels) or prototype (build airplanes). In reality, architects must rely on multiple methods and understand and exploit the continuum between the flexibility of simulation and single-design-point fidelity of prototypes. Even the most realistic and concrete academic machine remains but a wind tunnel model of a commercial product.

The Wisconsin Wind Tunnel project employed a mixture of complementary methods:

- Micro-architectural level simulation (e.g., of Typhoon, Typhoon-1, and Typhoon-2) using the Wisconsin Wind Tunnel [37,36,15,2,8] and other simulators.
- New tools for performance measurement and modelling [30, 29, 28].
- User- and system-software prototyping and development on existing commercial platforms (e.g., Blizzard/CM-5 and Blizzard/COW).
- Surgical hardware prototyping, as exemplified by the MBus card we designed with Sun Microsystems to accelerate COW. Surgical refers to the approach of building no more than what needs to be built. This board is not a prototype of optimal hardware, but it demonstrated that no unforeseen hardware or software problems stand in the way of Tempest acceleration.

Two of the important methodology artifacts of our project are the Wisconsin Wind Tunnel (WWT) and the Executable Editing Library (EEL).

3.2 Wisconsin Wind Tunnel

We developed and implemented an innovative, execution-driven simulation system called the Wisconsin Wind Tunnel (WWT) [37]. WWT runs a parallel shared-memory program on a parallel computer (Thinking Machines CM-5) and uses execution-driven, distributed, discrete-event simulation to accurately calculate program execution time. WWT directly executes all shared-memory program instructions and memory references that hit in the hypothetical machine's cache. WWT's speed and the CM-5's memory capacity permit evaluations to use more realistic workloads than are feasible with other simulation technique.

3.3 Executable Editing Library

Executable editing changes executable (compiled) code by removing existing instructions and adding *foreign code* that observes or modifies a program’s execution. It is an effective technique for measuring and modifying program behavior since executables hold an entire program (including libraries) and editing them does not require source code or modification to system tools such as compilers and linkers.

Executable editing is widely used for three purposes: emulation, observation, and optimization. An edited executable can emulate features that hardware does not provide. For example, the Wisconsin Wind Tunnel architecture simulator [37] drives a distributed, discrete-event simulation of a parallel computer from the logical cycle times of processors directly executing a parallel program. The underlying hardware (a SPARC processor in a Thinking Machines CM-5) does not provide a cycle counter or an efficient mechanism for interleaving computation and simulation. The Wind Tunnel system edits programs so that they update a cycle timer and return control at timer expirations. Similarly, one version of the Blizzard distributed shared-memory system [40] edits programs to insert fine-grain access tests before shared loads and stores. These tests permit data sharing at cache-block granularity, which reduces the false sharing incurred by page-granularity distributed shared-memory systems.

EEL (Executable Editing Library) is a library for building tools to analyze and modify an executable (compiled) program [28]. Currently, however, tools of this sort are difficult and time-consuming to write and are usually closely tied to a particular machine and operating system. EEL

supports a machine- and system-independent editing model that enables tool builders to modify an executable without being aware of the details of the underlying architecture or operating system or being concerned with the consequences of deleting instructions or adding foreign code.

4 Technology Transfer

Our project employes many methods of technology transfer. Like other academics, we write conference papers, visit and give talks at companies, and have students graduate to industrial jobs.

Like a few projects, we hold semi-annual affiliates meetings. These are very effective at engaging industry in a dialog. They force industrial people to spend a block of time thinking about our project, which leads them to make many useful suggestions. Furthermore, students get excited when they see that people care about their work. Recent meetings have included representatives from Cray, IBM, Intel, Portland Group, Sun, and Thinking Machines.

Finally, as a new experiment, one of us (Hill) is spending an academic year on sabbatical in an industrial product group (at Sun Microsystems). His very positive experiences with technology exchange appears is his position paper (elsewhere in these proceedings).

5 Summary

The paper summarizes research into parallel computer design and methods performed by the Wisconsin Wind Tunnel Project. Our principal design contributions—*Cooperative Shared Memory* and the *Tempest Parallel Programming Substrate*—seek to portably surpass the benefits of SMP-like shared-memory. We refine and compare design proposals with a mixture of techniques, including micro-architecture-level simulation, software prototyping, and "surgical" hardware prototyping. Furthermore, the evaluation requirements of our design provided impetuous to the methodological advances embodied in the *Wisconsin Wind Tunnel* and the *Executable Editing Library.*

6 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.

References

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

[2] Douglas C. Burger and David A. Wood. Accuracy vs. Performance in Parallel Simulation of Interconnection Networks. In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

[3] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating System Principles (SOSP)*, pages 152–164, October 1991.

[4] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.

[5] 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.

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

[7] 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.

[8] Sashikanth Chandrasekaran and Mark D. Hill. Optimistic Simulation of Parallel Architectures Using Program Executables. In *Proceedings of Tenth Workshop on Parallel and Distributed Simulation (PADS '96)*, May 1996.

[9] Trishul Chilimbi, Thomas Ball, Stephen Eick, and James Larus. StormWatch: A Tool for Visualizing Memory System Protocols. In *Proceedings of Supercomputing '95*, December 1995.

[10] Trishul M. Chilimbi and James R. Larus. Cachier: A Tool for Automatically Inserting CICO Annotations. In *Proceedings of the 1994 International Conference on Parallel Processing (Vol. II Software)*, pages II–89–98, August 1994.

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

[12] Helen Custer. *Inside Windows NT*. Microsoft Press, 1993.

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

[14] 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 *Proceedings of Supercomputing '94*, pages 380–389, November 1994.

[15] Babak Falsafi and David A. Wood. Cost/Performance of a Parallel Computer Simulator. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS '94)*, July 1994.

[16] 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.

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

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

in it Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)/.

[19] Mark D. Hill, James R. Larus, and David A. Wood. Tempest: A Substrate for Portable Parallel Programs. In *COMPCON '95*, pages 327–332, San Francisco, California, March 1995. IEEE Computer Society.

[20] Kirk L. Johnson, M. Frans Kaashoek, and Deborah A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *Proceedings of the 15th ACM Symposium on Operating System Principles (SOSP)*, pages 213–228, December 1995.

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

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

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

[24] 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 International Workshop)*, pages 326–341. Springer-Verlag, August 1993.

[25] 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.

[26] James R. Larus, Satish Chandra, and David A. Wood. CICO: A Shared-Memory Programming Performance Model. In Jeanne Ferrante and Tony Hey, editors, *Portability and Performance for Parallel Processors*, chapter 5, pages 99–120. John Wiley & Sons, 1994.

[27] 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.

[28] 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.

[29] Alvin R. Lebeck and David A. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27(10):15–26, October 1994.

[30] Alvin R. Lebeck and David A. Wood. Active Memory: A New Abstraction for Memory-System Simulation. In *Proceedings of the 1995 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 220–230, May 1995.

[31] 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.

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

[33] 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*, page ?, May 1996.

[34] Shubhendu S. Mukherjee and Mark D. Hill. An Evaluation of Directory Protocols for Medium-Scale Shared-Memory Multiprocessors. In *Proceedings of the 1994 International Conference on Supercomputing*, pages 64–74, Manchester, England, July 1994.

[35] 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)*, pages 68–79, July 1995.

[36] 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*, September 1993.

[37] 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.

[38] 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.

[39] 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*, May 1996.

[40] 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 297–307, October 1994.

[41] 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.

[42] 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 it CMG Transactions,/ Spring 1994.

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

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

Appendix A. Tempest: A Portable Substrate for Parallel Programs[1]

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

1. Reprinted from Compcon '95 with the permission of the IEEE. Copyright © 1995 IEEE [19].

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.

1.1 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—LOADs of invalid blocks or STOREs to invalid or read-only blocks—transfer control to an application-supplied handler. Section 1.2 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 application with both a shared address space and efficient communication. Section 1.3 discusses how Tempest supports different programming styles.

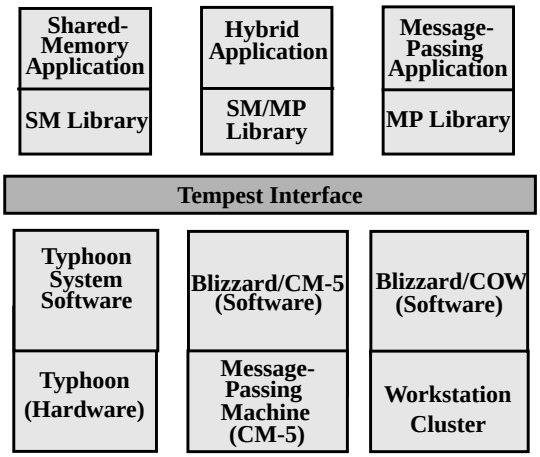

FIGURE 1. The Tempest interface. This figure summarizes the paper: Section 1.2 describes the Tempest interface, our substrate for parallel programming on a wide range of platforms. Section 1.3 discusses Tempest's support for different programming styles (above Tempest). Section 1.4 describes alternative Tempest imple-

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 1.4 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).

Section 1.5 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 implemen-

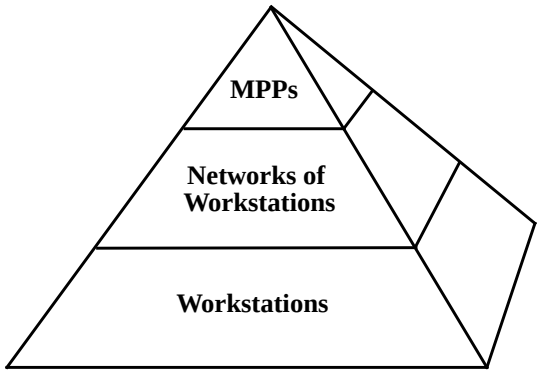

FIGURE 2. The parallel machine pyramid.

tations 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.

1.2 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) [38]. 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 [41]. 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 [32,3,21]. 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 [31].

Tempest provides mechanisms to implement programming paradigms, but leaves policy to user-level code [13]. 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 [44].

	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.

1.3 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 [16]) or NUMA (no caching) shared memory (e.g., Split-C [11]) 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 [6,25]) 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 [14]. 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 iterations. Section 1.5 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

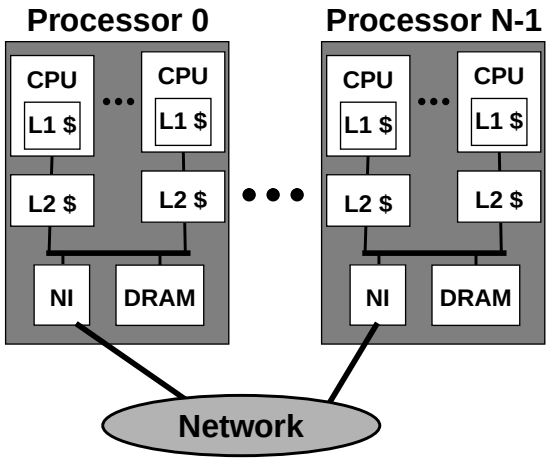

FIGURE 3. Base parallel machine hardware.

structures and built-in functions. The C** data parallel programming language [24] 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) [27] 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.

1.4 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 [38]. 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.

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

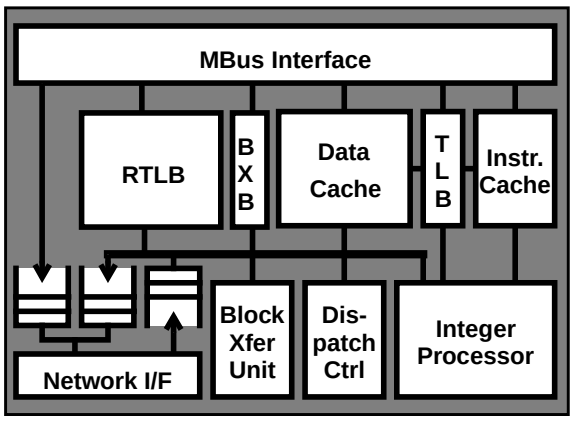

FIGURE 4. Typhoon's Network Interface.

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 [40]. The CM-5 provides no support for shared memory but does fit the machine model depicted in Figure 3.[1] 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 [36].

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 *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.

1. RTLB misses delay the processor while the NI loads the entry from memory. Special mappings treat private memory as Read-Write [38].

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

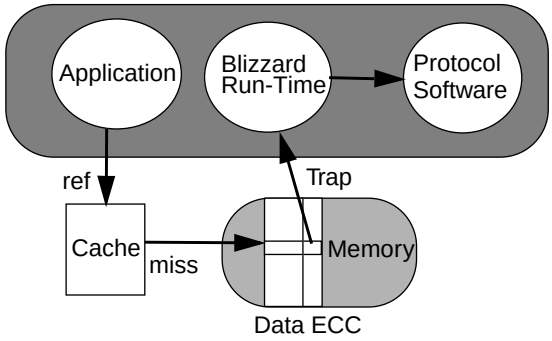

FIGURE 5. Blizzard-E: Tempest on a CM-5

To increase portability, we developed the all-software *Blizzard-S*. Blizzard-S modifies executable programs (`a.out` files) with a tool based on EEL [28] 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.

1.5 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. [38] used simulations on the Wisconsin Wind Tunnel [37] to compare Typhoon against a CC-NUMA machine modeled after the Stanford DASH [31]. 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. [40] 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. [14] 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 1.3. On the CM-5, the shared-memory EM3D ran as fast as a native message-passing version.

1.6 Related Work

Several interfaces share Tempest's goal of providing portability among parallel machines. PVM [16] is a widely-used, coarse-grain message-passing system. Berkeley's Active Messages [41] provides a portable interface for fine-grain messages, but, unlike Tempest, no support for transparent caching. DSM systems, such as Rice's Munin [3] and Treadmarks [21], 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 [5], Rice Munin [3], and Stanford FLASH [23]. 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 [17] and Kendall Square KSR-1 [22].

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 [13]. 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 [37].

1.7 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.