# The Wisconsin Wind Tunnel Project: An Annotated Bibliography*

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

## Abstract

This document lists contributors to the Wisconsin Wind Tunnel Project, gives a brief description of the project, and presents references and abstracts to its principal papers, including how to obtain them online.

## 1 Contributors

The Wisconsin Wind Tunnel project is co-directed by Professors Mark D. Hill, James R. Larus, and David A. Wood. Significant contributions to have been made by **Doug Burger, Satish Chandra, Trishul Chilimbi, Glen Ecklund, Babak Falsafi, Alain Kägi, 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 Tränkle,** and **Guhan Viswanathan.**

*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. Thinking Machines Corporation—especially Dave Douglas, Adam Greenberg, Danny Hillis, Roger Lee, and Steve Swartz—contributed advice and assistance for building the Wisconsin Wind Tunnel. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

## 2 Overview

Consensus is emerging about the lowest and highest levels of massively-parallel computers: these machines will be built from workstation-like nodes and programmed in high-level parallel languages—like HPF—that support a shared address space in which processes uniformly reference data. Unfortunately, no consensus has emerged on the communication model–shared memory or message passing—to support parallel languages.

The Wisconsin Wind Tunnel project is currently developing a consensus about the middle-level interface—below languages and compilers and above system software and hardware. Our first proposed interface was *Cooperative Shared Memory*, which is an evolutionary extension to conventional shared-memory software and hardware [see `tocs93_csm.ps.Z`]. Cooperative Shared Memory asks programmers to identify the expected data sharing behavior with *Check-In/Check-Out (CICO)* performance annotations so that the system can handle references efficiently without complex hardware (e.g., $Dir_1SW$).

Recently, we have developed a new interface—called *Tempest*—between user-level protocol handlers and system-supplied mechanisms [see `isca94_typhoon.ps.Z`]. Tempest provides the mechanisms that allow programmers, compilers, and program libraries to implement and use message passing, transparent shared memory, and hybrid combinations of the two. Tempest mechanisms are low-overhead messages, bulk data transfer, virtual memory management, and fine-grain access control. The most novel mechanism—fine-grain access control—allows user software to tag blocks (e.g., 32 bytes) as read-write, read-only, or invalid, so the local memory can be used to transparently cache remote data.

We are developing implementations of Tempest on a Thinking Machines CM-5, a cluster of dual-processor Sun workstations, and a hypothetical hardware platform. Each will use a different method to implement fine-grain access control.

To refine our design ideas, we have developed and implemented an execution-driven simulation system called the Wisconsin Wind Tunnel (WWT) [`sigmetrics93_wwt.ps.Z`]. The released version of 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 techniques. Unreleased versions of WWT model shared-memory, message-passing, and Tempest target programs and allow each CM-5 node to host multiple target nodes.

The Wisconsin Wind Tunnel Project is so named because we use our tools to cull the design space of parallel supercomputers in a manner similar to how aeronautical engineers use conventional wind tunnels to design airplanes. Needless to say, we neither design airplanes nor blow air.

# 3 On-Line Access

On-line information on the Wisconsin Wind Tunnel Project can be obtained through world wide web/mosaic, anonymous ftp, and gopher. If these fail or you can't print the compressed postscript, e-mail your postal mail address to `wwt@cs.wisc.edu` and we will send hardcopies.

## 3.1 World Wide Web/Mosaic

Our World Wide Web URL is `http://www.cs.wisc.edu/p/wwt/Mosaic/wwt.html`. Our papers can be accessed by buttoning *Technical Papers.*

## 3.2 Anonymous FTP

Anonymous ftp to `ftp.cs.wisc.edu` and `cd wwt`. We recommend that you `get README`.

## 3.3 Gopher

Our gopher server is experimental.

```
gopher gopher.cs.wisc.edu
Select 2 (University of Wisconsin-Madison Computer Sciences D
Select 8 (Wisconsin Wind Tunnel Project)
```

# 4 The Papers

Below we divide our papers into (1) *Cooperative Shared Memory,* (2) *Wisconsin Wind Tunnel,* (3) *Tempest, Typhoon, Blizzard, etc.,* and (4) *Miscellaneous.* The most-recent version of this document is in `annobib.ps.Z`. Isn't recursion wonderful?

David A. Wood Mark D. Hill, James R. Larus. The Wisconsin Wind Tunnel Project: An Annotated Bibliography. unpublished manuscript (revised frequently). (Mosiac location `http://www.cs.wisc.edu/p/wwt/Mosaic/wwt.html`).

This document lists contributors to the Wisconsin Wind Tunnel Project, gives a brief description of the project, and presents references and abstracts to its principal papers, including how to obtain them on-line.

## 4.1 Cooperative Shared Memory

Introduces cooperative shared memory (`tocs93_csm.ps.Z`):

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, Oct. 1992.

We believe the paucity of massively-parallel, shared-memory machines follows from the lack of a shared-memory programming performance model that can inform programmers of the cost of operations (so they can avoid expensive ones) and can tell hardware designers which cases are common (so they can build simple hardware to optimize them). *Cooperative shared memory*, our approach to shared-memory design, addresses this problem.

Our initial implementation of cooperative shared memory uses a simple programming model, called *Check-In/Check-Out (CICO)*, in conjunction with even simpler hardware, called $Dir_1SW$. In CICO, programs bracket uses of shared data with a `check_out` directive marking the expected first use and a `check_in` directive terminating the expected use of the data. A *cooperative prefetch* directive helps hide communication latency. $Dir_1SW$ is a minimal directory

protocol that adds little complexity to message-passing hardware, but efficiently supports programs written within the CICO model.

Examines the complexity and performance of alternative directory protocols for cooperative shared memory (`isca93_mechanisms.ps.Z`):

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.

This paper explores the complexity of implementing directory protocols by examining their *mechanisms*—primitive operations on directories, caches, and network interfaces. We compare the following protocols: $Dir_1B$, $Dir_4B$, $Dir_4NB$, $Dir_nNB$, $Dir_1SW$ and an improved version of $Dir_1SW$ ($Dir_1SW^+$). The comparison shows that the mechanisms and mechanism sequencing of $Dir_1SW$ and $Dir_1SW^+$ are simpler than those for other protocols.

We also compare protocol performance by running eight benchmarks on 32 processor systems. Simulations show that $Dir_1SW^+$'s performance is comparable to more complex directory protocols. The significant disparity in hardware complexity and the small difference in performance argue that $Dir_1SW^+$ may be a more effective use of resources. The small performance difference is attributable to two factors: the low degree of sharing in the benchmarks and Check-In/Check-Out (CICO) directives.

**Keywords:** Shared-memory multiprocessors, memory systems, cache coherence, directory protocols, and hardware mechanisms.

Describes and examines the benefits of the check-in, check-out (CICO) programming performance model (`p4_cico.ps.Z`):

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.

A programming performance model provides a programmer with feedback on the cost of program operations and is a necessary basis to write efficient programs. Many shared-memory performance models do not accurately capture the cost of interprocessor communication caused by non-local memory references, particularly in computers with caches. This paper describes a simple and practical programming performance model—called *check-in, check-out (CICO)*—for cache-coherent, shared-memory parallel computers. CICO consists of two components. The first is a collection of annotations that a programmer adds to a program to elucidate the communication arising from shared-memory references. The second is a model that calculates the communication cost of these annotations. An annotation's cost models the cost of the memory references that it summarizes and serves as a metric to compare alternative implementations. Several examples demonstrate that CICO accurately predicts cache misses and identifies changes that improve program performance.

This paper discusses solving microstructure electrostatics with cooperative shared memory (`cce_electrostatics.ps.Z`).

F. Traenkle, M.D. Hill, and S. Kim. Solving Microstructure Electrostatics on a Proposed Parallel Computer. *To appear in: Computers and Chemical Engineering*, ?(?):?, ? Univ. of Wisconsin Computer Sciences Technical Report 1223 (April 1994).

The programming models presented by parallel computers are diverse and changing. We study a new parallel programming model—cooperative shared memory (CSM)—with a collaborative effort between chemical engineers and computer scientists. Since CSM machines do not (yet) exist we evaluate our applications and machine designs with the Wisconsin Wind Tunnel (WWT), which runs CSM programs and calculates the performance of hypothetical parallel computers.

The application considered is the class of three–dimensional elliptic partial differential equations (Laplace, Stokes, Navier) with solutions represented by boundary integral equations. The parallel algorithm follows naturally from our use of the Completed Double Layer Boundary Integral Equation Method (CDLBIEM).

A major result is the demonstration that coding CDLBIEM is much simpler under CSM than with the message passing model, and yet performance (computational times and speed ups) is comparable, a fact that may be of great interest to designers of future machines. With WWT, we can also examine performance as a function of machine parameters such as cache

size and network bandwidth and latency. The possibility of tweaking simultaneously the algorithm and architecture to outline pathways of evolution for future parallel machines is an important concept explored in this work.

Master's thesis that explains on the contents of the above paper (`traenkle_ms.ps.Z`).

F. Traenkle. Parallel Programming Models and Boundary Integral Equation Methods for Microstructure Electrostatics. Master's thesis, University of Wisconsin–Madison, 1993.

The programming models presented by parallel computers are diverse and changing. We study the implementation of our application in different parallel programming models with a collaborative effort between chemical engineers and computer scientists.

The application considered is the class of three-dimensional elliptic partial differential equations (Laplace, Stokes, Navier) with solutions represented by boundary integral equations. These partial differential equations appear in basic microscopic descriptions of heterogeneous structured continua. As an example, we present results for the macroscopic dielectric constants and thermal conductivities of two-phase materials. The parallel algorithm follows naturally from our use of the Completed Double Layer Boundary Integral Equation Method (CDLBIEM).

The application is implemented in the message-passing programming model using the standard send-receive message-passing primitives in the CMMD library and the static shared-memory model in the form of Split-C, both running on the Thinking Machines CM-5 parallel computer. Furthermore, we study its implementation in a new parallel programming model - cooperative shared memory (CSM). Since CSM machines do not (yet) exist we evaluate our application and machine designs with the Wisconsin Wind Tunnel (WWT), which runs CSM programs and calculates the performance of hypothetical parallel computers.

A major result is the demonstration that coding CDLBIEM is much simpler under CSM than with the message-passing model or Split-C, and yet performance (computational times and scaleup) is comparable, a fact that may be of great interest to designers of future machines.

This paper describes Cachier, a tool for automatically inserting CICO annotations in programs (`icpp94_cachier.ps.Z`).

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)*, page ?, August 1994. To appear.

Shared memory in a parallel computer provides programmers with the valuable abstraction of a shared address space–through which any part of a computation can access any datum. Although uniform access simplifies programming, it also hides communication, which can lead to inefficient programs. The check-in, check-out (CICO) performance model for cache-coherent, shared-memory parallel computers helps a programmer identify the communication underlying memory references and account for its cost. CICO consists of annotations that a programmer can use to elucidate communication and a model that attributes costs to these annotations. The annotations can also serve as directives to a memory system to improve program performance. Inserting CICO annotations requires reasoning about the dynamic cache behavior of a program, which is not always easy. This paper describes Cachier, a tool that automatically inserts CICO annotations into shared-memory programs. A novel feature of this tool is its use of both dynamic information, obtained from a program execution trace, as well as static information, obtained from program analysis. We measured several benchmarks annotated by Cachier by running them on a simulation of the Dir1SW cache coherence protocol, which supports these directives. The results show that programs annotated by Cachier perform significantly better than both programs without CICO annotations and programs that were annotated by hand.

Keywords: Shared-memory, parallel programming performance models, parallel programming tools, cache-coherence, directory protocols.

## 4.2 Wisconsin Wind Tunnel

Describes the Wisconsin Wind Tunnel (`sigmetrics93_wwt.ps.Z`):

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.

We have developed a new technique for evaluating cache coherent, shared-memory computers. The Wisconsin Wind Tunnel (WWT) runs a parallel shared-memory program on a parallel computer (CM-5) and uses execution-driven, distributed, discrete-event simulation to accurately calculate program execution time. WWT is a virtual prototype that exploits similarities between the system under design (the target) and an existing evaluation platform (the host). The host directly executes all target program instructions and memory references that hit in the target cache. WWT's shared memory uses the CM-5 memory's error-correcting code (ECC) as valid bits for a fine-grained extension of shared virtual memory. Only memory references that miss in the target cache trap to WWT, which simulates a cache-coherence protocol. WWT correctly interleaves target machine events and calculates target program execution time. WWT runs on parallel computers with greater speed and memory capacity than uniprocessors. WWT's simulation time decreases as target system size increases for fixed-size problems and holds roughly constant as the target system and problem scale.

Describes operating system support for the Wisconsin Wind Tunnel (`usenix93_kernel.ps.Z`):

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.

This paper describes a kernel interface that provides an untrusted user-level process (an *executive*) with protected access to memory management functions, including the ability to create, manipulate, and execute within subservient contexts (address spaces). *Page motion callbacks* not only give the executive limited control over physical memory management, but also shift certain responsibilities out of the kernel, greatly reducing kernel state and complexity.

The *executive interface* was motivated by the requirements of the Wisconsin Wind Tunnel (WWT), a system for evaluating cache-coherent shared-memory parallel architectures. WWT uses the executive interface to implement a fine-grain user-level extension of Li's shared virtual memory on a Thinking Machines CM-5, a message-passing multicomputer. However, the interface is sufficiently general that an executive could act as a multiprogrammed operating system, exporting an alternative interface to the threads running in its subservient contexts.

The executive interface is currently implemented as an extension to CMOST, the standard operating system for the CM-5. In CMOST, policy decisions are made on a central, distinct control processor (CP) and broadcast to the processing nodes (PNs). The PNs execute a minimal kernel sufficient only to implement the CP's policy. While this structure efficiently supports some parallel application models, the lack of autonomy on the PNs restricts its generality. Adding the executive interface provides limited autonomy to the PNs, creating a structure that supports multiple models of application parallelism. This structure, with autonomy on top of centralization, is in stark contrast to most microkernel-based parallel operating systems in which the nodes are fundamentally autonomous.

How to use the Wisconsin Wind Tunnel (`wwt_tutorial.ps.Z`).

Alain Kagi and Shubhendu S. Mukherjee. A Programming Tutorial for the Wisconsin Wind Tunnel. unpublished manuscript, revised August 1993.

This tutorial gives a brief introduction to programming, compiling, and executing parallel shared-memory applications on the Wisconsin Wind Tunnel (WWT), a *virtual prototyping system*. The WWT currently runs only on a Thinking Machines CM-5, so we assume that the reader has access to one and knows how to log in and run programs and is familiar with basic Unix(TM) functionality.

The tutorial illustrates how to parallelize a simple sequential application; how to use the Cooperative Shared Memory (CSM) model and different cache coherence protocols; and how to execute, debug and profile parallel applications on the WWT. The tutorial should give you enough information to get started writing your own programs for the WWT.

Shows that parallel simulation can have better cost/performance than sequential simulation (`pads94_costperf.ps.Z`).

Babak Falsafi and David A. Wood. Cost/Performance of a Parallel Computer Simulator. In *Proceedings of PADS '94*, July 1994.

This paper examines the cost/performance of simulating a hypothetical *target* parallel computer using a commercial *host* parallel computer. We address the question of whether parallel simulation is simply faster than sequential

simulation, or if it is also more cost-effective. To answer this, we develop a performance model of the Wisconsin Wind Tunnel (WWT), a system that simulates cache-coherent shared-memory machines on a message-passing Thinking Machines CM-5. The performance model uses Kruskal and Weiss's fork-join model to account for the effect of event processing time variability on WWT's conservative fixed-window simulation algorithm. A generalization of Thiebaut and Stone's footprint model accurately predicts the effect of cache interference on the CM-5. The model is calibrated using parameters extracted from a fully-parallel simulation ($p - N$), and validated by measuring the speedup as the number of processors ($p$) ranges from one to the number of target nodes ($N$). Together with simple cost models, the performance model indicates that for target system sizes of 32 nodes and larger, parallel simulation is more cost-effective than sequential simulation. The key intuition behind this result is that large simulations require large memories, which dominate the cost of a uniprocessor; parallel computers allow multiple processors to simultaneously access this large memory.

## 4.3 Tempest, Typhoon, Blizzard, etc.

Discusses and compares the strengths of compiler- and hardware-implemented shared memory (`hw_sw_sm.ps.Z`).

> 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.
>
> Many parallel languages presume a shared address space in which any portion of a computation can access any datum. Some parallel computers directly support this abstraction with hardware shared memory. Other computers provide distinct (per-processor) address spaces and communication mechanisms on which software can construct a shared address space. Since programmers have difficulty explicitly managing address spaces, there is considerable interest in compiler support for shared address spaces on the widely available message-passing computers.
>
> At first glance, it might appear that hardware-implemented shared memory is unquestionably a better base on which to implement a language. This paper argues, however, that compiler-implemented shared memory, despite its shortcomings, has the potential to exploit more effectively the resources in a parallel computer. Hardware designers need to find mechanisms to combine the advantages of both approaches in a single system.

This paper proposes an interface for user-level shared memory called *Tempest* and describes a hardware implementation called *Typhoon* (`isca94_typhoon.ps.Z`).

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

This paper discusses various techniques for fine-grain access control and three implementation of them in Blizzard (`asplos6_fine_grain.ps.Z`).

> 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)*, October 1994. To appear.

This paper discusses implementations of fine-grain memory access control, which selectively restricts reads and writes to cache-block-sized memory regions. Fine-grain access control forms the basis of efficient cache-coherent shared memory. This paper focuses on low-cost implementations that require little or no additional hardware. These techniques permit efficient implementation of shared memory on a wide range of parallel systems, thereby providing shared-memory codes with a portability previously limited to message passing.

This paper categorizes techniques based on where access control is enforced and where access conflicts are handled. We incorporated three techniques that require no additional hardware into Blizzard, a system that supports distributed shared memory on the CM-5. The first adds a software lookup before each shared-memory reference by modifying the program's executable. The second uses the memory's error correcting code (ECC) as cache-block valid bits. The third is a hybrid. The software technique ranged from slightly faster to two times slower than the ECC approach. Blizzard's performance is roughly comparable to a hardware shared-memory machine. These results argue that clusters of workstations or personal computers with networks comparable to the CM-5's will be able to support the same shared-memory interfaces as supercomputers.

This paper compares four shared-memory and message-passing programs running on detailed architectural simulators of comparable machines. (`asplos6_sm_mp.ps.Z`).

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)*, October 1994. To appear.

Message passing and shared memory are two techniques parallel programs use for coordination and communication. This paper studies the strengths and weaknesses of these two mechanisms by comparing equivalent, well-written message-passing and shared-memory programs running on similar hardware. To ensure that our measurements are comparable, we produced two carefully tuned versions of each program and measured them on closely-related simulators of a message-passing and a shared-memory machine, both of which are based on same underlying hardware assumptions.

We examined the behavior and performance of each program carefully. Although the cost of computation in each pair of programs was similar, synchronization and communication differed greatly. We found that message-passing's advantage over shared-memory is not clear-cut. Three of the four shared-memory programs ran at roughly the same speed as their message-passing equivalent, even though their communication patterns were different.

This paper shows how a custom memory system, built on Blizzard, can help support C**, a high-level parallel language (`asplos6_lcm.ps.Z`).

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)*, October 1994. To appear.

Higher-level parallel programming languages can be difficult to implement efficiently on parallel machines. This paper shows how a flexible, compiler-controlled memory system can help achieve good performance for language constructs that previously appeared too costly to be practical.

Our compiler-controlled memory system is called Loosely Coherent Memory (LCM). It is an example of a larger class of Reconcilable Shared Memory (RSM) systems, which generalize the replication and merge policies of cache-coherent shared-memory. RSM protocols differ in the action taken by a processor in response to a *request* for a location and the way in which a processor *reconciles* multiple outstanding copies of a location. LCM memory becomes temporarily inconsistent to implement the semantics of C** parallel functions efficiently. RSM provides a compiler with control over memory-system policies, which it can use to implement a language's semantics, improve performance, or detect errors. We illustrate the first two points with LCM and our compiler for the data-parallel language C**.

This paper examines customizing protocols to applications using the Tempest interface running on Blizzard (`sc94_protocols.ps.Z`).

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*, November 1994. To appear.

Recent distributed shared memory (DSM) systems and proposed shared-memory machines have implemented some or all of their cache coherence protocols in software. One way to exploit the flexibility of this software is to tailor a coherence protocol to match an application's communication patterns and memory semantics. This paper presents evidence that this approach can lead to large performance improvements. It shows that application-specific protocols substantially improved the performance of three application programs—*appbt*, *em3d*, and *barnes*—over carefully tuned transparent shared memory implementations. The speedups were obtained on *Blizzard*, a fine-grained DSM system running on a 32-node Thinking Machines CM-5.

## 4.4 Miscellaneous

This paper proposes three new multicast directory protocols for cache-coherent shared-memory machines, and shows that even with very little state they can be competitive with a full-map directory protocol (`ics94_directory.ps.Z`).

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

This paper considers alternative directory protocols for providing cache coherence in shared-memory multiprocessors with 32 to 128 processors, where the state requirements of $Dir_N$ may be considered too large. We consider $Dir_i$B, $i = 1, 2, 4$, $Dir_N$, *Tristate* (also called *superset*), *Coarse Vector*, and three new protocols. The new protocols—*Gray-hardware*, *Gray-software*, *Home*—are optimizations of *Tristate* that use gray coding to favor near-neighbor sharing.

Our results are the first to compare all these protocols with complete applications (and the first evaluation of *Tristate* with a non-synthetic workload). Results for three applications—`ocean` (one-dimensional sharing), `appbt` (three-dimensional sharing), and `barnes` (dynamic sharing)—for 128 processors on the Wisconsin Wind Tunnel show that (a) $Dir_1$B sends 15 to 43 times as many invalidation messages as $Dir_N$, (b) *Gray-software* sends 1.0 to 4.7 times as many messages as $Dir_N$, making it better than *Tristate*, *Gray-hardware*, and *Home*, and (c) the choice between $Dir_i$B, *Coarse Vector*, and *Gray-software* depends on whether one wants to optimize for few sharers ($Dir_i$B), many sharers (*Coarse Vector*), or hedge one's bets between both alternatives (*Gray-software*).

**Keywords:** Shared-memory multiprocessors, cache coherence, directory protocols, and gray code.

This paper explores paging on distributed-shared-memory machines (`sc94_paging.ps.Z`).

Douglas C. Burger, Rahmat S. Hyder, Barton P. Miller, and David A. Wood. Paging Tradeoffs in Distributed-Shared-Memory Multiprocessors. In *Proceedings of Supercomputing 94*, November 1994. To appear.

Massively parallel processors have begun using commodity operating systems that support demand-paged virtual memory. To evaluate the utility of virtual memory, we measured the behavior of seven shared-memory parallel application programs on a simulated distributed-shared-memory machine. Our results (i) confirm the importance of gang CPU scheduling, (ii) show that a page-faulting processor should spin rather than invoke a parallel context switch, (iii) show that our parallel programs frequently touch most of their data, and (iv) indicate that memory, not just CPUs, must be "gang scheduled". Overall, our experiments demonstrate that demand paging has limited value on current parallel machines because of the applications' synchronization and memory reference patterns and the machines' high page-fault and parallel-context-switch overheads.