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
`http://www.cs.wisc.edu/~wwt`
April 4, 2001

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 on-line.

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 Glen Ammons, Tom Ball, E. Ender Bilir, Doug Burger, Satish Chandra, Sashikanth Chandrasekaran, Trishul Chilimbi, Anne Condon, Ross M. Dickson, Glen Ecklund, Babak Falsafi, Charles Fischer, Alain Kägi, Ying Hu, Steve Huss-Lederman, Rahmat Hyder, Alvin Lebeck, James Lewis, Mike Litzkow, Shubhendu Mukherjee, Subbarao Palacharla, Manoj Plakal, Steven Reinhardt, Brad Richards, Anne Rogers, Timothy Schimke, Eric Schnarr, Yannis Schoinas, Daniel Sorin, Steve Swartz, Brian Toonen, Frank Tränkle, Guhan Viswanathan, Zhichen Xu.

2 Project Funding Sources

This work has been 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 Grants CCR-9101035, MIP-9225097, and MIPS-9625558, NSF PYI/NYI Awards CCR-9157366, MIPS-8957278, and CCR-9357779, 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, IBM, Sun Microsystems, Thinking Machines Corporation, and Xerox Corporation. Much of this research was conducted on computing resources purchased through NSF Institutional Infrastructure Grants No. CDA-9024618 and CDA-9623632 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.

3 Overview

The *Wisconsin Wind Tunnel Project* focuses on trade-offs for designing cost-effective parallel machines supporting shared memory. In this work we have published more than five dozen papers which are listed in this annotated bibliography (`ftp://ftp.cs.wisc.edu/wwt/annobib.{ps,pdf}`[1]).

The project focus has centered on three phases of parallel machine design. The first phase examined *Cooperative Shared Memory* (`ftp://ftp.cs.wisc.edu/wwt/tocs93_csm.{ps,pdf}`) that simplified shared memory hardware by allowing software to manage data movement.

The second phase proposed the more-general *Tempest* (`ftp://ftp.cs.wisc.edu/wwt/nsf96_summary.{ps,pdf}`) interface that enabled programmers, compilers, and program libraries to implement and use message passing, transparent shared memory, and hybrid combinations of the two.

We are now moving into a third phase which returns to a greater hardware emphasis and seeks to use system-wide prediction and speculation to improve performance even as communication latencies (measured in instruction opportunities) increase (e.g., `ftp://ftp.cs.wisc.edu/wwt/isca98_cosmos.{ps,pdf}` and `ftp://ftp.cs.wisc.edu/wwt/isca99_multifacet.{ps,pdf}`).

Furthermore, our parallel machine design foci have and continue to serve as catalysts for a constellation of related research. This research include examinations of SMP cluster design, network interface design, coherence protocol verification, execution-driven simulation, data parallel compilation, path profiling, and cache conscious data allocation and reorganization.

Finally, we have developed and distributed many tools, including software for parallel execution-driven simulation (`Wisconsin Wind Tunnel`) executable editing (`Executable Editing Library`), and cache-conscious data allocation and reorganization (`ccmalloc` and `ccmorph`).

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.

4 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.

4.1 World Wide Web

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

4.2 Anonymous FTP

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

1. The notation `ftp:annobib.{ps,pdf}` means either `ftp:annobib.ps` or `ftp:annobib.pdf`

5 The Papers

Below we divide our papers into (1) *Overviews* (2) *Tempest, Typhoon, and Blizzard,* (3) *Custom Protocols,* (4) *Compiling for Tempest,* (5) *Hardware Design,* (6) *Tools,* (7) *Cooperative Shared Memory,* (8) *Wisconsin Wind Tunnel,* (9) *Path Profiling, (10) Network Interfaces, (11) Lamport Clocks, and (12) Miscellaneous.*

5.1 Overviews

The current version of this annotated bibliography is in `annobib.{ps,pdf}`.

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. (Frequently updated. Web location is: `http://www.cs.wisc.edu/~wwt`).

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.

This paper summarizes our project as of June 1995 (`ftp://ftp.cs.wisc.edu/wwt/nsf96_summary.{ps,pdf}`). It uses our Compcon 1995 paper as an appendix on Tempest.

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.

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.*

5.2 Tempest, Typhoon, and Blizzard

This paper proposes an interface for user-level shared memory called *Tempest* and describes a hardware implementation called *Typhoon* (`ftp://ftp.cs.wisc.edu/wwt/isca94_typhoon.{ps,pdf}`).

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 (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 (`ftp://ftp.cs.wisc.edu/wwt/asplos6_fine_grain.{ps,pdf}`).

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.

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 is an overview of Tempest (`ftp://ftp.cs.wisc.edu/wwt/compcon95_tempest.{ps,pdf}`).

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.

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.

This report gives the Tempest Interface Specification (`ftp://ftp.cs.wisc.edu/wwt/tr95_tempest_spec.{ps,pdf}`).

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

This document describes Tempest, an architectural user/system interface for distributed-memory parallel systems. By providing user-level access to both messaging and memory-management functions, Tempest not only supports message passing and shared memory, the two dominant parallel programming models, but allows users to construct hybrid models as well. The most unusual feature of Tempest is fine-grain memory access control. An access tag is associated with every block of memory at a granularity typical of a hardware cache block (e.g., 32-128 bytes). Loads and stores that conflict with the access tag invoke a user-specified handler that can perform arbitrary operations to resolve the conflict. The ability to dynamically manage data

access at a fine grain allows data migration and replication without relying on compile-time analysis or a restricted programming model to guarantee correctness. This migration and replication can be performed without renaming using Tempest's virtual memory management operations. To effect data transfer between nodes, Tempest provides two types of message-passing. Fine-grain messaging provides the short, low-latency messages required to implement cache coherence protocols and support fine-grain parallelism. Bulk data transfer operations are optimized to provide high bandwidth for large messages. Timers and thread management complete the list of Tempest features.
More recent versions of this document may exist. (`http://www.cs.wisc.edu/~wwt`) points to the most recent version.

This paper describes StormWatch, a tool for analyzing the interaction between a program and memory system (`http://scxy.tc.cornell.edu/sc95/proceedings/505_TCHI/SC95.HTM`).

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

Recent research has offered programmers increased options for programming parallel computers by exposing system policies (e.g., memory coherence protocols) or by providing several programming paradigms (e.g. message passing and shared memory) on the same platform. Increased flexibility can lead to higher performance, but it is also a double-edged sword that demands a programmer understand his or her application and system at a more fundamental level. Our system, Tempest, allows a programmer to select or implement communication and memory coherence policies that fit an application's communication patterns. With it, we have achieved substantial performance gains without making major changes in programs. However, the process of selecting, designing, and implementing coherence protocols is difficult and time consuming, without tools to supply detailed information about an application's behavior and interaction with the memory system.
StormWatch is a new visualization tool that aids a programmer through four mechanisms: tightly-coupled bidirectionally linked views, interactive filters, animation, and performance slicing. Multiple views present several aspects of program behavior simultaneously and show the same phenomenon from different perspectives. Real-time linking between views enables a programmer to explore levels of abstraction by changing a view and observing the effect on other views. Interactive filters, along with bidirectional linking, can isolate the effects of statements, loops, procedures, or files. StormWatch can also animate a program's dynamic behavior to show the evolution of program execution and communication. Finally, performance slicing captures causality among events. The examples in the paper illustrate how StormWatch helped us substantially improve the performance of two applications.

This paper describes hardware support for Tempest running on a network of workstations (`ftp://ftp.cs.wisc.edu/wwt/isca96_dcpld.{ps,pdf}`).

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.

This paper investigates hardware support for fine-grain distributed shared memory (DSM) in networks of workstations. To reduce design time and implementation cost relative to dedicated DSM systems, we decouple the functional hardware components of DSM support, allowing greater use of off-the-shelf devices.
We present two decoupled systems, Typhoon-0 and Typhoon-1. Typhoon-0 uses an off-the-shelf protocol processor and network interface; a custom access control device is the only DSM-specific hardware. To demonstrate the feasibility and simplicity of this access control device, we designed and built an FPGA-based version in under one year. Typhoon-1 also uses an off-the-shelf protocol processor, but integrates the network interface and access control devices for higher performance.
We compare the performance of the two decoupled systems with two integrated systems via simulation. For six benchmarks on 32 nodes, Typhoon-0 ranges from 30% to 309% slower than the best integrated system, while Typhoon-1 ranges from 13% to 132% slower. Four of the six benchmarks achieve speedups of 12 to 18 on Typhoon-0 and 15 to 26 on Typhoon-1, compared with 19 to 35 on the best integrated system. Two

benchmarks are hampered by high communication overheads, but selectively replacing shared-memory operations with message passing provides speedups of at least 16 on both decoupled systems. These speedups indicate that decoupled designs can potentially provide a cost-effective alternative to complex high-end DSM systems.

This paper further examines how much hardware support is beneficial for supporting the Tempest interface. (`ftp://ftp.cs.wisc.edu/wwt/toc98_decoupled.{ps,pdf}`).

Steven K. Reinhardt, Robert W. Pfile, and David A. Wood. Hardware Support for Flexible Distributed Shared Memory. *IEEE Transactions on Computer Systems*, Vol. 47, No. 10, October 1998,pp. 1056-1072.

Workstation-based parallel systems are attractive due to their low cost and competitive uniprocessor performance. However, supporting a cache-coherent global address space on these systems involves significant overheads. We examine two approaches to coping with these overheads. First, DSM-specific hardware can be added to the off-the-shelf component base to reduce overheads. Second, application-specific coherence protocols can avoid some overheads by exploiting programmer (or compiler) knowledge of an application's communication patterns. To explore the interaction between these approaches, we simulated four designs that add DSM acceleration hardware to a collection of off-the-shelf workstation nodes. Three of the designs support user-level software coherence protocols, enabling application-specific protocol optimizations. To verify the feasibility of our hardware approach, we constructed a prototype of the simplest design. Measured speedups from the prototype match simulation results closely.
We find that even with aggressive DSM hardware support, custom protocols can provide significant speedups. In addition, the custom protocols are generally effective at reducing the impact of other overheads, including those due to less aggressive hardware support and larger network latencies. However, for three of our benchmarks, the additional hardware acceleration provided by our most aggressive design avoids the need to develop more efficient custom protocols.

This paper describes and analyzes an extended version of Blizzard, called Scirocco, that runs on a cluster of SMPs (`ftp://ftp.cs.wisc.edu/wwt/pact98.{ps,pdf}`).

Ioannis Schoinas, Babak Falsafi, Mark D. Hill, James R. Larus, and David A. Wood. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, October 1998.

Software fine-grain distributed shared memory (FGDSM) provides a simplified shared-memory programming interface with minimal or no hardware support. Originally software FGDSMs targeted uniprocessor-node parallel machines. This paper presents Sirocco, a family of software FGDSMs implemented on a network of low-cost SMPs. Sirocco takes full advantage of SMP nodes by implementing inter-node sharing directly in hardware and overlapping computation with protocol execution. To maintain correct shared-memory semantics, however, SMP nodes require mechanisms to guarantee atomic coherence operations. Multiple SMP processors may also result in contention for shared resources and reduce performance. SMP nodes also impact the cost trade-off. While SMPs typically charge higher price-premiums, for a given system size SMP nodes substantially reduce networking hardware requirement as compared to uniprocessor nodes.
In this paper, we ask the question "are SMPs cost-effective building blocks for software FGDSM?" We present experimental measurements on Sirocco implementations ranging from an all-software system to a system with minimal hardware support. Together with simple cost models we show that low-cost SMP nodes: (i) result in competitive performance with uniprocessor nodes, (ii) substantially reduce hardware requirement and are more cost-effective than uniprocessor nodes, (iii) significantly benefit from hardware support for coherence operations, and (iv) are especially beneficial for FGDSMs with high-overhead coherence operations.

This paper presents a retrospective of the origins and evolution of the Tempest and Typhoon designs (`ftp://ftp.cs.wisc.edu/wwt/isca94_retrospective.{ps,pdf}).`

Steven K. Reinhardt, James R. Larus, and David A. Wood. Reflections on ’Tempest and Typhoon: User-level Shared Memory’. In Gurindar Sohi, editor, *25 years of the International Symposia of Computer Architecture: Selected Papers*, pages 98–101. 1998.

Tempest and Typhoon have emerged as among the most influential contributions of the Wisconsin Wind Tunnel project, a collaborative effort with Prof. Mark D. Hill, several staff members, and a large group of graduate students. This retrospective focuses on the origins of the Tempest and Typhoon ideas and their subsequent evolution.
This paper was selected by a committee of former ISCA program chairs as one of the forty most influential papers in the 25 year history of ISCA.

5.3 Custom Protocols

The papers in this section focus on using custom protocols to improve parallel program performance.

This paper examines customizing protocols to applications using the Tempest interface running on Blizzard (`ftp://ftp.cs.wisc.edu/wwt/sc94_protocols.{ps,pdf}`).

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.

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 speed-ups were obtained on *Blizzard*, a fine-grained DSM system running on a 32-node Thinking Machines CM-5.

This paper compares three irregular codes running on CHAOS and Tempest (`ftp://ftp.cs.wisc.edu/wwt/ppopp95_irregular.{ps,pdf}`).

S. S. Mukherjee, S. D. Sharma, M. D. Hill, J. R. Larus, A. Rogers, and J. 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.

Irregular computation problems underlie many important scientific applications. Although these problems are computationally expensive, and so would seem appropriate for parallel machines, their irregular and unpredictable run-time behavior makes this type of parallel program difficult to write and adversely affects run-time performance.
This paper explores three issues—partitioning, mutual exclusion, and data transfer—crucial to the efficient execution of irregular problems on distributed-memory machines. Unlike previous work, we studied the same programs running in three alternative systems on the same hardware base (a Thinking Machines CM-5): the CHAOS irregular application library, Transparent Shared Memory (TSM), and eXtensible Shared Memory (XSM). CHAOS and XSM performed equivalently for all three applications. Both systems were somewhat (13%) to significantly faster (991%) than TSM.

This paper describes a language for writing coherence protocols. (`ftp://ftp.cs.wisc.edu/wwt/pldi96_teapot.{ps,pdf}`).

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.

Recent shared-memory parallel computer systems offer the exciting possibility of customizing memory coherence protocols to fit an application's semantics and sharing patterns. Custom protocols have been used to achieve message-passing performance-while retaining the convenient programming model of a global address space-and to implement high-level language constructs. Unfortunately, coherence protocols written in a conventional language such as C are difficult to write, debug, understand, or modify. This paper describes Teapot, a small, domain-specific language for writing coherence protocols. Teapot uses continuations to help reduce the complexity of writing protocols. Simple static analysis in the Teapot compiler eliminates much of the overhead of continuations and results in protocols that run nearly as quickly as hand-written C code. A Teapot specification can be compiled both to an executable coherence protocol and to input for a model checking system, which permits the specification to be verified. We report our experiences coding and verifying several protocols using Teapot, along with measurements of the overhead incurred by writing a protocol in a higher-level language.

This paper describes our experience with using Teapot. (`ftp://ftp.cs.wisc.edu/wwt/dsl97_experiences.{ps,pdf}`).

S. Chandra, M. Dahlin, B. Richards, R. Y. Wang, T. E. Anderson, and J. R. Larus. Experience with a language for writing coherence protocols. In *USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

In this paper, we describe our experience with Teapot, a domain-specific language for addressing the cache coherence problem. The cache coherence problem arises when parallel and distributed computing systems make local replicas of shared data for reasons of scalability and performance. In both distributed shared memory systems and distributed file systems, a coherence protocol maintains agreement among the replicated copies when the underlying data are modified by programs running on the system. Unfortunately, cache coherence protocols are notoriously difficult to implement, debug, and maintain. Furthermore, the details of the protocols depend on the requirements of the system under consideration and are highly varied. This paper presents case studies detailing the successes and shortcomings of using Teapot for writing coherence protocols in two distinct systems. The first system, loosely coherent memory (LCM), implements a particular flavor of distributed shared memory suitable for data-parallel programming. The second system, the xFS distributed file system, implements a high-performance, serverless file system. Our overall experience with using Teapot has been positive. In particular, Teapot's language features resulted in considerable simplifications in the protocol code for both systems. Furthermore, Teapot's close coupling between implementation and formal verification allowed us to achieve much higher confidence in our protocol implementations than had previously been possible, reducing the time needed to build the protocols. By using Teapot to solve real problems in complex systems, we also discovered several shortcomings of the Teapot design. Most noticeably, we found Teapot lacking in support for multithreaded environments, for expressing actions that transcend several cache blocks, and for blocking system calls. We conclude that domain-specific languages can be valuable in the specific problem domain of cache coherence. Drawing on our experience, we also provide guidelines for domain-specific languages in the broader context of systems software.

This paper describes a new technique for cache coherence protocols (`ftp://ftp.cs.wisc.edu/wwt/isca95_dsi.{ps,pdf}`).

Alvin R. Lebeck and David A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48-49, June 1995.

This paper introduces dynamic self-invalidation (DSI), a new technique for reducing cache coherence overhead in shared-memory multiprocessors. DSI eliminates invalidation messages by having a processor automatically invalidate its local copy of a cache block before a conflicting access by another processor. Eliminating invalidation overhead is particularly important under sequential consistency, where the latency of invalidating outstanding copies can increase a program's critical path.
DSI is applicable to software, hardware, and hybrid coherence schemes. In this paper we evaluate DSI in the context of hardware directory-based write-invalidate coherence protocols. Our results show that DSI reduces

execution time of a sequentially consistent full-map coherence protocol by as much as 41%. This is comparable to an implementation of weak consistency that uses a coalescing write-buffer to allow up to 16 outstanding requests for exclusive blocks. When used in conjunction with weak consistency, DSI can exploit tear-off blocks-which eliminate both invalidation and acknowledgment messages- for a total reduction in messages of up to 26%.

5.4 Compiling for Tempest

Discusses and compares the strengths of compiler- and hardware-implemented shared memory (`ftp://ftp.cs.wisc.edu/wwt/hw_sw_sm.{ps,pdf}`).

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 compares four shared-memory and message-passing programs running on detailed architectural simulators of comparable machines. (`ftp://ftp.cs.wisc.edu/wwt/asplos6_sm_mp.{ps,pdf}`).

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 (ASP-LOS VI)*, Pages 61-75, October 1994.

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 (`ftp://ftp.cs.wisc.edu/wwt/asplos6_lcm.{ps,pdf}`).

J. R. Larus, B. Richards, and G. 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 (ASP LOS VI)*, pages 208–218, October 1994.

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 explores the performance of a suite of HPF programs on a Blizzard implementation on a cluster of workstations (`ftp://ftp.cs.wisc.edu/wwt/lcpc96_hpf.{ps,pdf}`).

S. Chandra and J. R. Larus. HPF on Fine-Grain Distributed Shared Memory: Early Experience. In U. Banerjee, A. Nicolau, D. Gelernter, and D. Padua, editors, *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*. August 1996.

This paper examines the performance of a suite of HPF applications on a network of workstations using two different compilation approaches: generating explicit message-passing code, and generating code for a shared address space provided by a fine-grain distributed shared memory system (DSM). Preliminary experiments indicate that the DSM approach performs with usually a small slow down compared to the message passing approach on regular programs, yet enables efficient execution of non-regular programs.

This book chapter describes the C** data parallel programming language and its implementation using Tempest (`ftp://ftp.cs.wisc.edu/wwt/PPUC.{ps,pdf}`).

J. R. Larus, B. Richards, and G. Viswanathan. Parallel programming in C: A large-grain data-parallel programming language. In G. V. Wilson and P. Lu, editors, Parallel Programming Using C++, chapter 8, pages 297–342. MITP, 1996.**

C** is a large-grain data-parallel programming language. It preserves the principal advantages of SIMD data parallelism—comprehensible and near-determinate parallel execution—while relaxing SIMD's constricted execution model. We have used C** a vehicle for experimenting with parallel language features and with implementation techniques that exploit program-level control of a parallel computer's memory system. This paper both describes the language and summarizes progress in language design and implementation since the previous C** paper.

This paper describes how a compiler-directed predictive protocol can improve shared-memory communication for iterative data-parallel programs (`ftp://ftp.cs.wisc.edu/wwt/sc96_compiler_sm.{ps,pdf}`).

Guhan Viswanathan and James R. Larus. Compiler-directed Shared-Memory Communication for Iterative Parallel Applications. In *Proceedings of Supercomputing '96*, November 1996.

Many scientific applications are iterative and specify repetitive communication patterns. This paper shows how a parallel-language compiler and a predictive cache-coherence protocol in a distributed shared memory system together can implement shared-memory communication efficiently for applications with unpredictable but repetitive communication patterns. The compiler uses static analysis to identify program points where potentially repetitive communication occurs. At runtime, the protocol builds a communication schedule in one iteration and uses the schedule to pre-send data in subsequent iterations. This paper contains measurements of three iterative applications (including adaptive programs with unstructured data accesses) that show that a predictive protocol increases the number of shared-data requests satisfied locally, thus reducing the remote data access latency and total execution time.

This paper describes how to optimize communication in HPF programs on fine-grain distributed shared memory (`ftp://ftp.cs.wisc.edu/wwt/ppopp97_hpf.{ps,pdf}`).

S. Chandra and J. R. Larus. Optimizing Communication in HPF programs for Fine-Grain Distributed Shared Memory. In Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP), June 1997.

Unlike compiler-generated message-passing code, the coherence mechanisms in shared-memory systems work equally well for regular and irregular programs. In many programs, however, compile-time information about data accesses would permit data to be transferred more efficiently—if the underlying shared-memory system offered suitable primitives. This paper demonstrates that cooperation between a compiler and a memory coherence protocol can improve the performance of High Performance Fortran (HPF) programs running on a fine-grain distributed shared memory system up to a factor of 2, while retaining the versatility and portability of shared memory. As a consequence, shared memory's performance becomes competitive with message passing for regular applications, while not affecting (or in some cases, even improving) its large advantage for irregular codes. This paper describes the design of our implementation and reports experimental results.

5.5 Hardware Design

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 (`ftp://ftp.cs.wisc.edu/wwt/ics94_directory.{ps,pdf}`).

S. S. Mukherjee and M. 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.

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_iB, 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_1B* 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_iB*, *Coarse Vector*, and *Gray-software* depends on whether one wants to optimize for few sharers (*Dir_iB*), 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 the cost and performance of adding synchronization hardware to an existing network of workstations. (`ftp://ftp.cs.wisc.edu/wwt/ics96_synch.{ps,pdf}`).

Rahmat S. Hyder and David A. Wood. Synchronization Hardware for Networks of Workstations: Performance vs. Cost. In *Proceedings of the 1996 International Conference on Supercomputing*, May 1996.

Networks of workstations (NOWs) are gaining popularity as lower-cost alternatives to massively-parallel processors (MPPs) because of their ability to leverage high-performance commodity workstations and data networks. However, fast data networks may not suffice if applications require frequent global synchronization, e.g., barriers, reductions, and broadcasts. Many MPPs provide hardware support specifically to accelerate these operations. Separate synchronization networks have also been proposed for NOWs, but such add-on hardware only makes sense if the performance improvement is commensurate with its cost. In this study, we examine the cost/performance trade-off of add-on synchronization hardware for an emulated 32-node NOW, running an aggregate workload of twelve shared-memory, message-passing, and data-parallel workloads. For low-latency messaging (e.g., ~10 ms), add-on hardware is cost-effective only if its per-node cost is less than 8% of the base workstation cost. For higher-latency messages (e.g., ~100 ms), add-on hardware is cost-effective if it costs less than 23% of the base cost. At these higher latencies and typical prices, a 32-node NOW with an add-on synchronization network is cost effective for 10 of the 12 benchmarks, compared to a uniprocessor with the same memory capacity.

This paper proposes *Reactive NUMA (R-NUMA)*, a hardware DSM design that unifies S-COMA and CC-NUMA (`ftp://ftp.cs.wisc.edu/wwt/isca97.{ps,pdf}`).

Babak Falsafi and David A. Wood. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 229–240, June 1997.

This paper proposes and evaluates a new approach to directory-based cache coherence protocols called Reactive NUMA (R-NUMA). An R-NUMA system combines a conventional CC-NUMA coherence protocol with a more-recent Simple-COMA (S-COMA) protocol. What makes R-NUMA novel is the way it dynamically reacts to program and system behavior to switch between CC-NUMA and S-COMA and exploit the best aspects of both protocols. This reactive behavior allows each node in an R-NUMA system to independently choose the best protocol for a particular page, thus providing much greater performance stability than either CC-NUMA or S-COMA alone. Our evaluation is both qualitative and quantitative. We first show the theoretical result that R-NUMA's worst-case performance is bounded within a small constant factor (i.e., two to three times) of the best of CC-NUMA and S-COMA. We then use detailed execution-driven simulation to show that, in practice, R-NUMA usually performs better than either a pure CC-NUMA or pure S-COMA protocol, and no more than 57% worse than the best of CC-NUMA and S-COMA, for our benchmarks and base system assumptions.

This paper advocates the multiprocessor support sequential consistency or related simple models (`ftp://ftp.cs.wisc.edu/wwt/computer98_sccase.{ps,pdf}`).

M. D. Hill. Multiprocessors should support simple memory consistency models. IEEE Computer, 31, 1998.

Many future computers will be shared-memory multiprocessors. These hardware systems must define for software the allowable behavior of memory. A reasonable model is sequential consistency (SC), which makes a shared memory multiprocessor behave like a multiprogrammed uniprocessor. Since SC appears to limit some of the optimizations useful for aggressive hardware implementations, researchers and practitioners have defined several relaxed consistency models. Some of these models just relax the ordering from writes to reads (processor consistency, IBM 370, Intel Pentium Pro, and Sun TSO), while others aggressively relax the order among all normal reads and writes (weak ordering, release consistency, DEC Alpha, IBM PowerPC, and Sun RMO).
This paper argues that multiprocessors should implement SC or, in some cases, a model that just relaxes the ordering from writes to reads. I argue against using aggressively relaxed models because, with the advent of speculative execution, these models do not give a sufficient performance boost to justify exposing their complexity to the authors of low-level software.
Keywords: multiprocessors, parallel computing, shared memory, memory consistency models.
Revised version of *Technical Report 1353*, Computer Sciences Department, University of Wisconsin–Madison, October 1997.

This paper describes a novel hybrid coherence technique that combines aspects of directory protocols with traditional snooping.

E. Ender Bilir, Ross M. Dickson, Ying Hu, Manoj Plakal, Daniel J. Sorin, Mark D. Hill, and David A. Wood. Multicast Snooping: A New Coherence Method Using a Multicast Address Network. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, Atlanta, Georgia, May 1999.

This paper proposes a new coherence method called “multicast snooping” *that* dynamically adapts between broadcast snooping and a directory protocol. Multicast snooping is unique because processors predict which caches should snoop each coherence transaction by specifying a multicast “mask.” Transactions are delivered with an ordered multicast network, such as an Isotach network, which eliminates the need for acknowledgment messages. Processors handle transactions as they would with a snooping protocol, while a simplified directory operates in parallel to check masks and gracefully handle incorrect ones (e.g., previous owner missing). Preliminary performance numbers with mostly SPLASH-2 benchmarks running on 32 processors show that we can limit multicasts to an average of 2-6 destinations (<< 32) and we can deliver 2-5 multicasts per network cycle (>> broadcast snooping’s 1 per cycle). While these results do not include timing, they do provide encouragement that multicast snooping can obtain data directly (like broadcast snooping) but apply to larger systems (like directories).

5.6 Tools

This paper describes the EEL Executable Editing Library (`ftp://ftp.cs.wisc.edu/wwt/pldi95_eel.{ps,pdf}`).

J. R. Larus and E. 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.

EEL (Executable Editing Library) is a library for building tools to analyze and modify an executable (compiled) program. The systems and languages communities have built many tools for error detection, fault isolation, architecture translation, performance measurement, simulation, and optimization using this approach of modifying executables. 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.

This paper describes a new abstraction for efficient memory system simulation (`ftp://ftp.cs.wisc.edu/wwt/sigmetrics95_am.{ps,pdf}`).

A. R. Lebeck and D. 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.

This paper describes the *active memory* abstraction for memory-system simulation. In this abstraction—designed specifically for on-the-fly simulation, memory references logically invoke a user-specified function depending upon the reference’s type and accessed memory block state. Active memory allows simulator writers to specify the appropriate action on each reference, including “no action” for the common case of cache hits. Because the abstraction hides implementation details, implementations can be carefully tuned for particular platforms, permitting much more efficient on-the-fly simulation than the traditional trace-driven abstraction.
Our SPARC implementation, *Fast-Cache*, executes simple data cache simulations two or three times faster than a highly-tuned trace-driven simulator and only 2 to 7 times slower than the original program. Fast-Cache implements active memory by performing a fast table look up of the memory block state, taking as few as 3 cycles on a SuperSPARC for the no-action case. Modeling the effects of Fast-Cache’s additional lookup instructions qualitatively shows that Fast-Cache is likely to be the most efficient simulator for miss ratios between 3% and 40%.

This paper describes how instruction scheduling within EEL can reduce the cost of program instrumentation and reschedule legacy code. (`ftp://ftp.cs.wisc.edu/wwt/wcsss96_eel.{ps,pdf}`).

Eric Schnarr and James R. Larus. Instruction Scheduling and Executable Editing. In *Workshop on Compiler Support for System Software (WCSSS ‘96)*, Tucson, Arizona, February 1996

Modern microprocessors offer more instruction-level parallelism than most programs and compilers can currently exploit. The resulting disparity between a machine's peak and actual performance, while frustrating for computer architects and chip manufacturers, opens the exciting possibility of low-cost or even no-cost instrumentation for measurement, simulation, or emulation. Instrumentation code that executes in previously unused processor cycles is effectively hidden. These microprocessors also pose another problem, which arises from the machine-specific instruction scheduling necessary for high performance. Different implementations of an architecture, such as the many x86 processors, may benefit from different schedules, which either requires multiple executables or a way to reschedule existing programs for new machines.
We investigated both opportunities by adding an instruction scheduler to the EEL executable editing library. On first-generation, 2 and 3-way superscalar SPARC processors, this simple, local scheduler hid an average of 17% (8-22%) of the overhead cost of profiling instrumentation in the SPECINT benchmarks and an average of 28% (5-53%) of the profiling cost in the SPECFP benchmarks. On a second-generation, 4-way superscalar UltraSPARC, the scheduler hid an average of 16% (8-21%) of the profiling cost in the SPECINT benchmarks and 65% (7-136%) in the SPECFP benchmarks. We also used the scheduler to reschedule uninstrumented code previously compiled for the SuperSPARC. Scheduling that takes into account the UltraSPARC's out-of-order execution improved the SPECFP benchmarks by an average of 9% (1-33%).

This paper describes how instruction scheduling within EEL can reduce the cost of program instrumentation. (`ftp://ftp.cs.wisc.edu/wwt/micro29_eel.{ps,pdf}`).

E. Schnarr and J. R. Larus. *Instruction scheduling and executable editing. In 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 29), December 1996.*

Modern microprocessors offer more instruction-level parallelism than most programs and compilers can currently exploit. The resulting disparity between a machine's peak and actual performance, while frustrating for computer architects and chip manufacturers, opens the exciting possibility of low-cost instrumentation for measurement, simulation, or emulation. Instrumentation code that executes in previously unused processor cycles is effectively hidden. On two superscalar SPARC processors, a simple, local scheduler hid an average of 13% of the overhead cost of profiling instrumentation in the SPECINT benchmarks and an average of 33% of the profiling cost in the SPECFP benchmarks.

This paper describes a new approach to profile shared-memory performance. (`ftp://ftp.cs.wisc.edu/wwt/ppopp97_memprof.{ps,pdf}`).

Z. Xu, J. R. Larus, and B. P. Miller. Shared-memory performance profiling. In *Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP),* June 1997.

This paper describes a new approach to finding performance bottlenecks in shared-memory parallel programs and its embodiment in the Paradyn Parallel Performance Tools running with the Blizzard fine-grain distributed shared memory system. This approach exploits the underlying system's cache coherence protocol to detect data sharing patterns that indicate potential performance bottlenecks and presents performance measurements in a data-centric manner. As a demonstration, Paradyn helped us improve the performance of a new shared-memory application program by a factor of four.

This paper describes optimizations that dramatically accelerate the cycle-accurate simulation of an out-of-order processor (ftp://ftp.cs.wisc.edu/wwt/asplos98_fastsim`.{ps,pdf}`).

Eric Schnarr and James R. Larus. Fast Out-Of-Order Processor Simulation Using Memoization. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS VIII),* October 1998.

Our new out-of-order processor simulator, FastSim, uses two innovations to speed up simulation 8-15 times (vs. Wisconsin SimpleScalar) with no loss in simulation accuracy. First, FastSim uses speculative direct-execution to accelerate the functional emulation of speculatively executed program code. Second, it uses a variation on memoization -- a well-known technique in programming language implementation -- to cache microarchitecture states and the resulting simulator actions, and then "fast forwards" the simulation the next time a cached state is reached. Fast-forwarding accelerates simulation by an order of magnitude, while producing exactly the same, cycle-accurat result as conventional simulation.

This paper describes how to automatically accelerate the cycle-accurate simulation of an out-of-order processor (ftp://ftp.cs.wisc.edu/wwt/plidi01_facile.pdf).

Eric Schnarr, Mark D. Hill, and James R. Larus. Facile: A Language and Compiler for High-Performance Processor Simulators. In Programming Language Design and Implementation, 2001.

Architectural simulators are essential tools for computer architecture and systems research and development. Simulators, however, are becoming frustratingly slow, because they must now model increasingly complex micro-architectures running realistic workloads. Previously, we developed a technique called fast-forwarding, which applied partial evaluation and memoization to improve the performance of detailed architectural simulations by as much as an order of magnitude .
While writing a detailed processor simulator is difficult, implementing fast-forwarding is even more complex. This paper describes Facile, a domain-specific language for writing detailed, accurate micro-architecture simulators. Architectural descriptions written in Facile can be compiled, using partial evaluation techniques, into fast-forwarding simulators that achieve significant performance improvements with far less programmer effort. Facile and its compiler make this performance-enhancing technique accessible to computer architects.

5.7 Cooperative Shared Memory

Introduces cooperative shared memory (a precusor to Tempest) (`ftp://ftp.cs.wisc.edu/wwt/tocs93_csm.{ps,pdf}`):

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

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 (`ftp://ftp.cs.wisc.edu/wwt/isca93_mechanisms.{ps,pdf}`):

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.

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 (`ftp://ftp.cs.wisc.edu/wwt/p4_cico.{ps,pdf}`):

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 (`ftp://ftp.cs.wisc.edu/wwt/cce_electrostatics.{ps,pdf}`).

F. Traenkle, M.D. Hill, and S. Kim. Solving Microstructure Electrostatics on a Proposed Parallel Computer. *Computers and Chemical Engineering*, 19:743–757, 1995.

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 (`ftp://ftp.cs.wisc.edu/wwt/traenkle_ms.{ps,pdf}`).

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 (`ftp://ftp.cs.wisc.edu/wwt/icpp94_cachier.{ps,pdf}`).

T. M. Chilimbi and J. 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.

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.

5.8 Wisconsin Wind Tunnel

Describes the Wisconsin Wind Tunnel (`ftp://ftp.cs.wisc.edu/wwt/sigmetrics93_wwt.{ps,pdf}`):

S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. 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 (`ftp://ftp.cs.wisc.edu/wwt/usenix93_kernel.{ps,pdf}`):

S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. 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.

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 (`ftp://ftp.cs.wisc.edu/wwt/wwt_tutorial.{ps,pdf}`).

S. S. Mukherjee, A. Kagi, and D. Burger. A programming tutorial for the wisconsin wind tunnel. Unpublished manuscript, revised January 1995.

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 `ftp://ftp.cs.wisc.edu/wwt/pads94_costperf.{ps,pdf}`).

B. Falsafi and D. A. Wood. Cost/performance of a parallel computer simulator. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (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.

This paper examines a range of network simulation models that trade off the accurate calculation of contention against reductions in simulation time. (`ftp://ftp.cs.wisc.edu/wwt/ipps95_netsim.{ps,pdf}`).

D. C. Burger and D. A. Wood. Accuracy vs. performance in parallel simulation of interconnection networks. In *Proceedings of the 9th International Parallel Processing Symposium*, April 1995.

Parallel simulation is emerging as the dominant technique for studying parallel computers. However, the interconnection networks of these machines can be modeled at many different levels of abstraction, allowing researchers to trade off accuracy and performance. In this paper, we use the Wisconsin Wind Tunnel, a parallel simulator for cache-coherent shared-memory machines, to study the trade-offs of accuracy versus performance for six different network simulation models. We evaluate these models for a variety of parallel applications, cache-coherence protocols, and topologies. We show that only the two most expensive models-which model contention at individual links-are robust in the presence of high network loads or non-uniform traffic patterns.

This paper examines the use of optimistic algorithms to perform parallel simulations of parallel machines using program executables. (`ftp://ftp.cs.wisc.edu/wwt/pads96_optimistic.{ps,pdf}`).

S. Chandrasekaran and M. D. Hill. Optimistic simulation of parallel architectures using program executables. In *Proceedings of Tenth Workshop on Parallel and Distributed Simulation (PADS '96),* May 1996.

A key tool of computer architects is computer simulation at the level of detail that can execute program executables. The time and memory requirements of such simulations can be enormous, especially when the machine under design—the *target*—is a parallel machine. Thus, it is attractive to use parallel simulation, as successfully demonstrated by the *Wisconsin Wind Tunnel* (WWT). WWT uses a conservative simulation algorithm and eschews network simulation to make lookahead adequate. Nevertheless, we find most of WWT's slowdown to be due to the synchronization overhead in the conservative simulation algorithm.
This paper examines the use of optimistic algorithms to perform parallel simulations of parallel machines. We first show that we can make optimistic algorithms work correctly even with WWT's direct execution of program executables. We checkpoint processor registers (integer, floating-point, and condition codes) and use executable editing to log the value of memory words just before they are overwritten by stores. Second, we consider the performance of two optimistic algorithms. The first executes programs optimistically, but performs protocol events (e.g., sending messages) conservatively. The second executes everything optimistically and is similar to Time Warp with lazy message cancellation. Unfortunately, both approaches make parallel simulation performance worse for the default WWT assumptions. We conclude by speculating on the performance of optimistic simulation when simulating (1) target network details, and (2) on hosts with high message latencies and no synchronization hardware.

This paper presents in detail the analytical models used to derive the results in (`pads94_costperf.{ps,pdf}`), (`ftp://ftp.cs.wisc.edu/wwt/tomacs_perf.{ps,pdf}`).

B. Falsafi and D. A. Wood. Modeling cost/performance of a parallel computer simulator. *ACM Transactions on Modeling and Computer Simulation*, 7(1), January 1997.

This paper describes the Wisconsin Wind Tunnel II (WWT II) simulator (`ftp://ftp.cs.wisc.edu/wwt/paid97_wwt2.{ps,pdf}`).

Shubhendu S. Mukherjee, Steven K. Reinhardt, Babak Falsafi, Mike Litzkow, Steve Huss-Lederman, Mark D. Hill, James R. Larus, and David A. Wood. Wisconsin Wind Tunnel II: A Fast and Portable Parallel Architecture Simulator. In *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

The design of future parallel computers requires rapid simulation of target designs running realistic workloads. These simulations have been accelerated using two techniques: direct execution and the use of a parallel host. Historically, these techniques have been considered to have poor portability. This paper identifies and describes the implementation of four key operations necessary to make such simulation portable across a variety of parallel computers. These four operations are: calculation of target execution time, simulation of features of interest, communication of target messages, and synchronization of host processors. Portable implementations of these four operations have allowed us to easily run the Wisconsin Wind Tunnel II (WWT II)—a parallel, discrete-event, direct-execution simulator—across a wide range of platforms, such as desktop workstations, a SUN Enterprise server, a cluster of workstations, and a cluster of symmetric multiprocessing nodes. We plan to release WWT II in August, 1997. We also plan to port WWT II to the IBM SP2.
We find that for two benchmarks, WWT II demonstrates both good performance and good scalability. Uniprocessor WWT II simulates one target cycle of a 32-node target machine in 114 and 166 host cycles respectively for the two benchmarks on a SUN UltraSPARC. Parallel WWT II achieves speedups between 4.1-5.4 on 8 host processors in our three parallel machine configurations.

5.9 Path Profiling

This paper describes a new and efficient technique for recording the execution frequency of paths through a routine (`ftp://ftp.cs.wisc.edu/wwt/micro96.{ps,pdf}`).

T. Ball and J. R. Larus. Efficient path profiling. In *29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 29)*, page 46–57, December 1996.

A *path profile* determines how many times each acyclic path in a routine executes. This type of profiling subsumes the more common basic block and edge profiling, which only approximate path frequencies. Path profiles have many potential uses in program performance tuning, profile-directed compilation, and software test coverage.
This paper describes a new algorithm for path profiling. This simple, fast algorithm selects and places profile instrumentation to minimize run-time overhead. Instrumented programs run with overhead comparable to the best previous profiling techniques. On the SPEC95 benchmarks, path profiling overhead averaged 31%, as compared to 16% for efficient edge profiling. Path profiling also identifies longer paths than a previous technique, which predicted paths from edge profiles (average of 88, versus 34 instructions). Moreover, profiling shows that the SPEC95 *train* input datasets covered most of the paths executed in the *ref* datasets.

This paper extends the previous paper on path profiling (`micro96.{ps,pdf}`) to take advantage of hardware counters on UltraSPARC (and other processors) (`ftp://ftp.cs.wisc.edu/wwt/pldi97_paths.{ps,pdf}`).

G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*, June 1997.

A program profile attributes run-time costs to portions of a program's execution. Most profiling system suffer from two major deficiencies: first, they only apportion simple metrics, such as execution frequency or elapsed time to static, syntactic units, such as procedures or statements; second, they aggressively reduce the volume of information collected and reported, although aggregation can hide striking differences in program behavior. This paper addresses both concerns by exploiting the hardware counters available in most modern processors and by incorporating two concepts from data flow analysis—flow and context sensitivity—to report more context for measurements. This paper extends our previous work on efficient path profiling to generalized flow sensitive profiling, which associates hardware performance metrics with a path through a procedure. In addition, it describes a data structure, the calling context tree, that efficiently captures calling contexts for procedure-level measurements.
Our measurements found the SPEC95 benchmarks execute a small number (3-28) of hot paths that account for 9–98% of their L1 data cache misses. Moreover, these hot paths are concentrated in a few routines, which have complex dynamic behavior.

This paper shows how to use path profiles to improve the precision of data-flow analysis in a compiler (`ftp://ftp.cs.wisc.edu/wwt/pldi98_dataflow.{ps,pdf}`).

G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *Proceedings of the SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, June 1998.

Data-flow analysis computes its solutions over the paths in a control-flow graph. These paths—whether feasible or infeasible, heavily or rarely executed—contribute equally to a solution. However, programs execute only a small fraction of their potential paths and, moreover, programs' execution time and cost is concentrated in a far smaller subset of *hot paths*.

This paper describes a new approach to analyzing and optimizing programs, which improves the precision of data flow analysis along hot paths. Our technique identifies and duplicates hot paths, creating a *hot path graph* in which these paths are isolated. After flow analysis, the graph is reduced to eliminate unnecessary duplicates of unprofitable paths. In experiments on SPEC95 benchmarks, path qualification identified 2–112 times more non-local constants (weighted dynamically) than the Wegman-Zadek conditional constant algorithm, which translated into 1–7% more dynamic instructions with constant results.

5.10 Network Interfaces

This paper explores a class of Coherent Network Interfaces that uses snooping cache coherence mechanisms to improve communication performance between a processor and a network interface. (`ftp://ftp.cs.wisc.edu/wwt/isca96_cni.{ps,pdf}`).

S. S. Mukherjee, B. Falsafi, M. D. Hill, and D. A. Wood. Coherent network interfaces for fine-grain communication. In Proceedings of the 23rd Annual International Symposium on Computer Architecture, pages 247–258, May 1996.

Historically, processor accesses to memory-mapped device registers have been marked uncachable to insure their visibility to the device. The ubiquity of snooping cache coherence, however, makes it possible for processors and devices to interact with cachable, coherent memory operations. Using coherence can improve performance by facilitating burst transfers of whole cache blocks and reducing control overheads (e.g., for polling).
This paper begins an exploration of network interfaces (NIs) that use coherence—coherent network interfaces (CNIs)—to improve communication performance. We restrict this study to NI/ CNIs that reside on coherent memory or I/O buses, to NI/CNIs that are much simpler than processors, and to the performance of fine-grain messaging from user process to user process.
Our first contribution is to develop and optimize two mechanisms that CNIs use to communicate with processors. A cachable device register—derived from cachable control registers—[39,40] is a coherent, cachable block of memory used to transfer status, control, or data between a device and a processor. Cachable

queues generalize cachable device registers from one cachable, coherent memory block to a contiguous region of cachable, coherent blocks managed as a circular queue.
Our second contribution is a taxonomy and comparison of four CNIs with a more conventional NI. Microbenchmark results show that CNIs can improve the round-trip latency and achievable bandwidth of a small 64-byte message by 37% and 125% respectively on the memory bus and 74% and 123% respectively on a coherent I/O bus. Experiments with five macrobenchmarks show that CNIs can improve the performance by 17-53% on the memory bus and 30-88% on the I/O bus.

This paper does an in-depth simulation study of the impact of data transfer and buffering alternatives on network interface design (`ftp://ftp.cs.wisc.edu/wwt/hpca98_impact.{ps,pdf}`).

S. S. Mukherjee and M. D. Hill. The impact of data transfer and buffering alternatives on network interface design. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, February 1998.

The explosive growth in the performance of microprocessors and networks has created a new opportunity to reduce the latency of fine-grain communication. Microprocessor clock speeds are now approaching the gigahertz range. Network switch latencies have dropped to tens of nanoseconds. Unfortunately, this explosive growth also exposes processor accesses to the network inter face (NI) as a critical bottleneck for fine-grain communication. Researchers have proposed several techniques, such as using block loads and stores, User-Level DMA, and Coherent Network Interfaces, to alleviate this NI access bottleneck.
This paper is the first to systematically identify, examine, and evaluate the key parameters that underlie these design alternatives. We classify these parameters into two categories: data transfer and buffering parameters. The data transfer parameters capture how messages are transferred between internal memory structures (e.g. processor caches, main memory) of a computer and a memory bus NI. The buffering parameters capture how and where an NI buffers incoming network messages.
We evaluate seven memory bus NIs that we believe capture the essential components of the design space exposed by these data transfer and buffering parameters. These seven NIs abstract the data transfer and buffering parameters of the NIs in TMC CM-5, Fujitsu AP3000, Princeton User-Level DMA, Digital Memory Channel, MIT StarT-JR, and two Coherent Network Interfaces (CNI512Q and CNI32Qm).
Our results indicate that a high-performance NI design should effectively use the block transfer mechanism of the memory bus, minimize processor involvement for data transfer, directly transfer messages between an NI and the processor (at least in the common case), provide plentiful buffering (possibly in main memory), and minimize processor involvement to buffer incoming network messages. The relative importance of these parameters depends both on the specific NI design and the characteristics of the application.
As a corollary of this study, we find that, contrary to conventional wisdom, mapping an NI to the processor registers is usually not the ideal choice. This is because processor register memory is a precious resource, which does not provide adequate buffering for many applications.

This paper evaluates alternative designs for address translation structures in network interfaces to support zero-copy transfers (minimal messaging).
(`ftp://ftp.cs.wisc.edu/wwt/hpca98_nitrans.{ps,pdf}`).

I. Schoinas and M. D. Hill. Address translation mechanisms in network interfaces. In *Proceedings of the Fourth IEEE Symposium on High-Performance Computer Architecture*, February 1998.

Good network hardware performance is often squandered by overheads for accessing the network interface (NI) within a host. NIs that support user-level messaging avoid frequent operating system (OS) action yet unnecessary copying can still result in low performance. We explore improving application messaging performance by eliminating all unnecessary copies (minimal messaging). For minimal messaging, NIs must support address translation and must do so more richly than has been done in the past. NI address translation should flexibly support higher-level abstractions, map all user space, exploit translation locality, and degrade gracefully when locality is poor. We classify NI address translation implementations based on where the lookup and the miss handling are performed (CPU or NI). We present alternative designs and we consider how they interact with the OS. We provide simulation results that evaluate the alternative design points and we demonstrate feasibility with a real implementation using Myrinet. We find: (a) NIs need not have hardware lookup structures, as software schemes are fast enough; (b) it is difficult for an NI to handle its own translation misses unless commercial operating systems are substantially modified to view an NI as CPU peer; (c) in the conventional situation where the operating system views the NI as a device, minimal messaging should be used only when the translation is present, while a single-copy protocol is used when it is not and (d) alternatively, one can currently get acceptable performance when the CPU handle misses if the kernel provides very fast trap interfaces but microprocessor and operating system trends may make this alternative less viable in the long run.

This paper argues that a processor's interactions with a network interface should be treated more like a memory access, rather than like a disk interface access. (`ftp://ftp.cs.wisc.edu/wwt/computer98_nicase.{ps,pdf}`).

S. S. Mukherjee and M. D. Hill. Making network interfaces less peripheral. IEEE Computer, "(?):?," " 1998." A talk-only version of this paper appears in *Hot Interconnects V*, 1997.

Much of a computer's value depends on how well it interacts with networks. To enhance this value, designers must improve the performance of networks delivered to users. Fortunately, the performance of networks is improving rapidly. Unfortunately, this dramatic improvement in network performance is seldom delivered to users. A key bottleneck is the host network interface (NI), which connects a network to a host computer. This bottleneck gets more severe as network and host performance continue to improve.
The problem with current NIs is that they were designed with an interface similar to that of a disk interface. Most current NIs require applications to use an operating system call, are placed on the I/O bus, do not allow processors to cache their registers, and force processors to interact with them with in-order and non-speculative accesses. The last two problems are partially due to the presence of "side-effects" in current NI designs.
While this kind of an interface may have been adequate in the past, we argue that future NIs should appear to their hosts more like memory than like a disk. Memory is virtualized without requiring operation system intervention (in the common case), is on the memory bus, can be cached, can be accessed out of order and speculatively, and is free of any side-effects. We discuss how to do the same for NIs, so that the dramatic improvements in network performance can be delivered to users.

This paper describes an efficient approach for low-overhead network interface that allows simultaneous access in an SMP (`ftp://ftp.cs.wisc.edu/wwt/hpca99_pdq.{ps,pdf}).`

Babak Falsafi and David A. Wood. Parallel Dispatch Queue: A Queue-Based Programming Abstraction To Parallelize Fine-Grain Communication Protocols. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture (HPCA-5)*, pages 182–192, 1999.

This paper proposes a novel queue-based programming abstraction, Parallel Dispatch Queues (PDQ), that enables efficient parallel execution of fine-grain software communication protocols. Parallel systems often use fine-grain software handlers to integrate a network message into computation. Executing such handlers in parallel requires acceess to a set of data structures, PDQ allows messages to include a synchronizion key protecting handler accesses to a group of protocol resources. By simply synchronizing message in a queue prior to dispatch, PDQ not only eliminates the overhead of acquiring/releasing synchronization primitives but also prevents busy-waiting within handlers.

5.11 Lamport Clocks

This paper illustrates the use of Lamport Clocks as a verification technique to demonstrate that a bus-based cache-coherence protocol ensures sequential consistency (`ftp://ftp.cs.wisc.edu/wwt/tr98_lamport.{ps,pdf}`).

D. J. Sorin, M. Plakal, M. D. Hill, and A. E. Condon. Lamport Clocks: Reasoning About Shared Memory Correctness. Technical Report 1367, Computer Sciences Department, University of Wisconsin–Madison, 1998.

Modern shared memory implementations use many complex, interacting optimizations, forcing industrial product groups to spend much more effort in verification than in design. Current formal verification techniques are somewhat non-intuitive to system designers and verifiers, and these formal methods do not scale well to practical systems.

This paper seeks to give verifiers and designers a reasoning technique that is precise (unlike informal reasoning) and intuitive (unlike some formal models). To prove that a system obeys the desired consistency model, we would like a tool that allows us to create a total order of events. We modestly extend Lamport's logical clock work from distributed systems and apply it to shared memory systems. We use these so-called Lamport clocks to timestamp events and thereby create a total order. This total order can then be examined to see if it satisfies the desired consistency model. Lamport clocks are purely a reasoning tool, and they are never instantiated in hardware.

We demonstrate the value of Lamport clocks by showing that sequential consistency (SC) is obeyed by a variety of snooping bus-based coherence protocols, ranging from a simple cache-less system to a split-transaction out-of-order bus. We present timestamping schemes for all of the above systems and, in the case of the split-transaction bus, we use the timestamps to formally prove that the system satisfies SC.

This paper illustrates the use of Lamport Clocks as a verification technique to demonstrate that a directory cache-coherence protocol ensures sequential consistency (`ftp://ftp.cs.wisc.edu/wwt/spaa98_lamport.{ps,pdf}`).

M. Plakal, D. J. Sorin, A. E. Condon, and M. D. Hill. Lamport Clocks: Verifying a Directory Cache-Coherence Protocol. In *Proceedings of the 10th Annual ACM Symposium on Parallel Architectures and Algorithms*, June 1998.

Modern shared-memory multiprocessors use complex memory system implementations that include a variety of non-trivial and interacting optimizations. More time is spent in verifying the correctness of such implementations than in designing the system. In particular, large-scale Distributed Shared Memory (DSM) systems usually rely on a directory cache-coherence protocol to provide the illusion of a sequentially consistent shared address space. Verifying that such a distributed protocol satisfies sequential consistency is a difficult task. Current formal protocol verification techniques complement simulation, but are somewhat non-intuitive to system designers and verifiers, and they do not scale well to practical systems.

In this paper, we examine a new reasoning technique that is precise and (we find) intuitive. Our technique is based on Lamport's logical clocks, which were originally used in distributed systems. We make modest extensions to Lamport's logical clocking scheme to assign timestamps to relevant protocol events to construct a total ordering of such events. Such total orderings can be used to verify that the requirements of a particular memory consistency model have been satisfied.

We apply Lamport clocks to prove that a non-trivial directory protocol implements sequential consistency. To do this, we describe an SGI Origin 2000-like protocol in detail, provide a timestamping scheme that totally orders all protocol events, and then prove sequential consistency (i.e., a load always returns the value of the "last" store to the same address in timestamp order).

This paper extends the use of Lamport clocks as a verification technique to systems that use relaxed memory models, specifically the Sun TSO and Compaq Alpha memory models (`ftp://ftp.cs.wisc.edu/wwt/hpca99_lamport.{ps,pdf}`).

Anne E. Condon, Mark D. Hill, Manoj Plakal, and Daniel J. Sorin. Using Lamport Clocks to Reason About Relaxed Memory Models. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, January 1999.

Cache coherence protocols of current shared-memory multiprocessors are difficult to verify. Our previous work proposed an extension of Lamport's logical clocks for showing that multiprocessors can implement sequential consistency (SC) with an SGI Origin 2000-like directory protocol and a Sun Gigaplane-like split-transaction bus protocol. Many commercial multiprocessors, however, implement more relaxed models, such as SPARC Total Store Order (TSO), a variant of processor consistency, and Compaq (DEC) Alpha, a variant of weak consistency.

This paper applies Lamport clocks to both a TSO and an Alpha implementation. Both implementations are based on the same Sun Gigaplane-like split-transaction bus protocol we previously used, but the TSO implementation places a first-in-first-out write buffer between a processor and its cache, while the Alpha implementation uses a coalescing write buffer. Both write buffers satisfy read requests for pending writes (i.e., do bypassing) without requiring the write to be immediately written to cache. Analysis shows how to apply Lamport clocks to verify TSO and Alpha specifications at the architectural level

This paper provides a framework to specify I/O architectures at the system level, and provides a sample implementation whose conformance to the specification is proved using our extension of Lamport Clocks. The Technical Report includes the full proof while the conference version has a sketch of the proof (`ftp://ftp.cs.wisc.edu/wwt/{spaa99_io,tr1398_io}.{ps,pdf}`).

Mark D. Hill, Anne E. Condon, Manoj Plakal, and Daniel J. Sorin. A System-Level Specification Framework for I/O Architectures. In *Proceedings of the Eleventh ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1999.

Mark D. Hill, Anne E. Condon, Manoj Plakal, and Daniel J. Sorin. A System-Level Specification Framework for I/O Architectures. Technical Report 1398, Computer Sciences Department, University of Wisconsin–Madison, March 1999.

A computer system is useless unless it can interact with the outside world through input/output (I/O) devices. I/O systems are complex, including aspects such as memory-mapped operations, interrupts, and bus bridges. Often, I/O behavior is described for isolated devices without a formal description of how the complete I/O system behaves. The lack of an end-to-end system description makes the tasks of system programmers and hardware implementors more difficult to do correctly.

This paper proposes a framework for formally describing I/O architectures called Wisconsin I/O (WIO). WIO extends work on memory consistency models (that formally specify the behavior of normal memory) to handle considerations such as memory-mapped operations, device operations, interrupts, and operations with side effects. Specifically, WIO asks each processor or device that can issue k operation types to specify ordering requirements in a $k \times k$ table. A system obeys WIO if there always exists a total order of all operations that respects processor and device ordering requirements and has the value of each “read” equal to the value of the most recent “write” to that address.

This paper then presents examples of WIO specifications for systems with various memory consistency models including sequential consistency (SC), SPARC TSO, an approximation of Intel IA-32, and Compaq Alpha. Finally, we present a directory-based implementation of an SC system, and we sketch a proof which shows that the implementation conforms to its WIO specification.

This paper develops a table-based methodology for specifying cache coherence protocols, spcifies two protocols, and shows a technique for verification (`ftp://ftp.cs.wisc.edu/wwt/tr1412_lamport.{ps,pdf}`).

Daniel J. Sorin, Manoj Plakal, Mark D. Hill, Anne E. Condon, Milo M. Martin, and David A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. Technical Report 1412, Computer Sciences Department, University of Wisconsin–Madison, March 2000.

In this paper, we develop a specification methodology that documents and specifies a cache coherence protocol in eight tables: the states, events, actions, and transitions of the cache and memory controllers. We then use this methodology to specify a detailed, low-level three-state broadcast snooping protocol with an unordered data network and an ordered address network that allows arbitrary skew. We also present a detailed, low-level specification of a new protocol called Multicast Snooping, and, in doing so, we better illustrate the utility of the table-based specification methodology. Lastly, we demonstrate a technique for verification of the

Multicast Snooping protocol, through the sketch of a manual proof that the specification satisfies a sequentially consistent memory model.

5.12 Miscellaneous

This paper explores paging on distributed-shared-memory machines (`ftp://ftp.cs.wisc.edu/wwt/sc94_paging.{ps,pdf}`).

D. C. Burger, R. S. Hyder, B. P. Miller, and D. A. Wood. Paging tradeoffs in distributed-shared-memory multiprocessors. In *Proceedings of Supercomputing '94,* pages 590–599, November 1994.

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.

This paper shows when parallel computing is cost-effective (`ftp://ftp.cs.wisc.edu/wwt/computer95_cost.{ps,pdf}`).

D. A. Wood and M. D. Hill. Cost-effective parallel computing. *IEEE Computer,* 28(2):69–72, February 1995.

Many academic papers imply that parallel computing is only worthwhile when applications achieve nearly linear speedup (i.e., execute nearly *p* times faster on *p* processors). This note shows that parallel computing is cost-effective whenever speedup exceeds *costup*—the parallel system cost divided by uniprocessor cost. Furthermore, when applications have large memory requirements (e.g., 512 megabytes), the costup—and hence speedup necessary to be cost-effective—can be much less than linear.

This paper evaluates protocol scheduling policies for a software DSM running on an SMP cluster. The paper quantifies when it is beneficial to dedicate a processor on an SMP node to running coherence protocol (`ftp://ftp.cs.wisc.edu/wwt/hpca97.{ps,pdf}`).

B. Falsafi and D. A. Wood. Scheduling communication on an SMP node parallel machine. In *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, pages 128–138, February 1997.

Distributed-memory parallel computers and networks of workstations (NOWs) both rely on efficient communication over increasingly high-speed networks. Software communication protocols are often the performance bottleneck. Several current and proposed parallel systems address this problem by dedicating one general-purpose processor in a symmetric multiprocessor (SMP) node specifically for protocol processing. This scheduling convention reduces communication latency and increases effective bandwidth, but also reduces the peak performance since the dedicated processor no longer performs computation.
In this paper, we study a parallel machine with SMP nodes and compare two protocol processing policies: Fixed, which uses a dedicated protocol processor; and Floating, where all processors perform both computation and protocol processing. The results from synthetic microbenchmarks and five macrobenchmarks show that: i) a dedicated protocol processor benefits light-weight protocols much more than heavy-weight protocols; ii) Fixed improves performance over Floating when communication becomes the bottleneck, which is more likely when the application is very communication-intensive, overheads are very high, or there are multiple (i.e., more than two) processors per node; iii) a system with optimal cost-effectiveness is likely to include a dedicated protocol processor, at least for light-weight protocols.

This paper evaluates the performance tradeoffs associated with different consistency models and coherence granularities. (`ftp://ftp.cs.wisc.edu/wwt/ppopp97_grain.{ps,pdf}`)

Y. Zhou, L. Iftode, J. P. Singh, K. Li, B. R. Toonen, I. Schoinas, M. D. Hill, and D. A. Wood. Relaxed consistency and coherence granularity in DSM systems: a performance evaluation. In *Sixth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, June 1997.

During the past few years, two main approaches have been taken to improve the performance of software shared memory implementations: relaxing consistency models and providing fine-grained access control. Their performance tradeoffs, however, are not well understood. This paper studies these tradeoffs on a platform that provides access control in hardware but runs coherence protocols in software. We compare the performance of three protocols across four coherence granularities, using 12 applications on a 16-node cluster of workstations. Our results show that no single combination of protocol and granularity performs best for all the applications. The combination of a sequentially consistent (SC) protocol and fine granularity works well with 7 of the 12 applications. The combination of a multiple-writer, home-based lazy release consistency (HLRC) protocol and page granularity works well with 8 out of the 12 applications. For applications that suffer performance losses in moving to coarser granularity under sequential consistency, the performance can usually be regained quite effectively using relaxed protocols, particularly HLRC. We also find that the HLRC protocol performs substantially better than a single-writer lazy release consistent (SW-LRC) protocol at coarse granularity for many irregular applications. For our applications and platform, when we use the original versions of the applications ported directly from hardware-coherent shared memory, we find that the SC protocol with 256-byte granularity performs best on average. However, when the best versions of the applications are compared, the balance shifts in favor of HLRC at page granularity.

This paper presents an analytic model for evaluating the performance of ILP multiprocessors. (`ftp://ftp.cs.wisc.edu/wwt/isca98_model.{ps,pdf}`).

D. Sorin, V. Pai, S. Adve, M. Vernon, and D. Wood. Analytic Evaluation of Shared-Memory Parallel Systems with ILP Processors. In *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

This paper develops and validates an analytical model for evaluating specific types of architectural alternatives for shared-memory multiprocessors with processors that aggressively exploit instruction-level parallelism (ILP). Compared to simulation, the analytical model is many orders of magnitude faster to solve, yielding highly accurate system performance estimates in seconds.

The model input parameters characterize the ability of an application to exploit instruction-level parallelism as well as the interaction between the application and the memory system architecture. A new simulation methodology is described that allows these parameters to be generated 250-400 times faster than with a detailed execution-driven simulator.

Finally, this paper shows that this analytical model can be used to gain insights into application performance and to evaluate architectural design trade-offs. Thus, a designer can use this analytical model to quickly determine the important regions of the design space for an application or architecture, allowing a more focused and effective use of simulation time and resources for more detailed studies

This paper presents analytical techniques for modeling servers that have high service time variability. (ftp://ftp.cs.wisc.edu/wwt/sigmetrics00_model.{ps,pdf}).

Derek L. Eager, Daniel J. Sorin, and Mary K. Vernon. AMVA Techniques for High Service Time Variability. In *Proceedings of the 2000 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 217–228, June 2000.

Motivated by experience gained during the validation of a recent ApproximateMean Value Analysis (AMVA) model of modern shared memory architectures, this paper re-examines the ‘‘standard’’ AMVA approximation for non-exponential FCFS queues. We find that this approximation is often inaccurate for FCFS queues with high service time variability. For such queues, we propose and evaluate: (1) AMVA estimates of the mean residual service time at an arrival instant that are much more accurate than the standard AMVA estimate, (2) a new AMVA technique that provides a much more accurate estimate of mean center residence time than the standard AMVA estimate, and (3) a new AMVA technique for computing the mean residence time at a ‘‘downstream’’ queue which has a more bursty arrival process than is assumed in the standard AMVA equations. Together, these new techniques increase the range of applications to which AMVA may be fruitfully applied, so that for example, the memory system architecture of shared memory systems with complex modern processors can be analyzed with these computationally efficient methods.

This paper proposes using general prediction logic to accelerate coherence protocols. (`ftp://ftp.cs.wisc.edu/wwt/isca98_cosmos.{ps,pdf}`).

S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In Proceedings of the 25th Annual International Symposium on Computer Architecture, June 1998.

Most large shared-memory multiprocessors use directory protocols to keep per-processor caches coherent. Some memory references in such systems, however, suffer long latencies for misses to remotely-cached blocks. To ameliorate this latency, researchers have augmented standard coherence protocols with optimizations for specific sharing patterns, such as read-modify-write, producer-consumer, and migratory sharing. This paper seeks to replace these directed solutions with general prediction logic that monitors coherence activity and triggers appropriate coherence actions.
This paper takes the first step toward using general prediction to accelerate coherence protocols by developing and evaluating the *Cosmos* coherence message predictor. Cosmos predicts the source and type of the next coherence message for a cache block using logic that is an extension of Yeh and Patt's two-level *PAp* branch predictor. For five scientific applications running on 16 processors, Cosmos has prediction accuracies of 62% to 93%. Cosmos' high prediction accuracy is a result of predictable coherence message signatures that arise from stable sharing patterns of cache blocks.

This paper discusses using a generational copying garbage collector to implement cache-conscious data placement. (`ftp://ftp.cs.wisc.edu/wwt/ismm98_cache_gc.{ps,pdf}`).

Trishul M. Chilimbi and James R. Larus. Using Generational Garbage Collection to Implement Cache-Conscious Data Placement. In *Proceedings of the International Symposium on Memory Management,* pages 37–48, October 1998.

The cost of accessing main memory is increasing. Machine designers have tried to mitigate the consequences of the processor and memory technology trends underlying this increasing gap with a variety of techniques to reduce or tolerate memory latency. These techniques, unfortunately, are only occasionally successful for pointer-manipulating programs. Recent research has demonstrated the value of a complementary approach, in which pointer-based data structures are reorganized to improve cache locality.

This paper studies a technique for using a generational garbage collector to reorganize data structures to produce a cache-conscious data layout, in which objects with high temporal affinity are placed next to each other, so that they are likely to reside in the same cache block. The paper explains how to collect, with low overhead, real-time profiling information about data access patterns in object-oriented languages, and describes a new copying algorithm that utilizes this information to produce a cache-conscious object layout.

Preliminary results show that this technique reduces cache miss rates by 21–42%, and improves program performance by 14–37% over Cheney's algorithm. We also compare our layouts against those produced by the Wilson-Lam-Moher algorithm, which attempts to improve program locality at the page level. Our cache-conscious object layouts reduces cache miss rates by 20–41% and improves program performance by 18–31% over their algorithm, indicating that improving locality at the page level is not necessarily beneficial at the cache level.

This paper discusses the problem of implementing cache-conscious structure layouts and describes two semi-automatic tools for achieving this. (`ftp://ftp.cs.wisc.edu/wwt/pldi99_cache_layout.{ps,pdf}`).

Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-Conscious Structure Layout. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI),* May 1999.

Hardware trends have produced an increasing disparity between processor speeds and memory access times. While a variety of techniques for tolerating or reducing memory latency have been proposed, these are rarely successful for pointer-manipulating programs.

This paper explores a complementary approach that attacks the source (poor reference locality) of the problem rather than its manifestation (memory latency). It demonstrates that careful data organization and layout provides an essential mechanism to improve the cache locality of pointer-manipulating programs and consequently, their performance. It explores two placement techniques—*clustering* and *coloring*—that improve cache performance by increasing a pointer structure's spatial and temporal locality, and by reducing cache-conflicts.

To reduce the cost of applying these techniques, this paper discusses two strategies—*cache-conscious reorganization* and *cache-conscious allocation*—and describes two semi-automatic tools—`ccmorph` and `ccmalloc`—that use these strategies to produce cache-conscious pointer structure layouts. `ccmorph` is a transparent tree reorganizer that utilizes topology information to cluster and color the structure. `ccmalloc` is a cache-conscious heap allocator that attempts to co-locate contemporaneously accessed data elements in the same physical cache block. Our evaluations, with microbenchmarks, several small benchmarks, and a couple of large real-world applications, demonstrate that the cache-conscious structure layouts produced by `ccmorph` and `ccmalloc` offer large performance benefits—in most cases, significantly outperforming state-of-the-art prefetching.

This paper discusses the problem of cache-conscious structure definition and explores two techniques—structure field reordering and structure splitting—that address this. (`ftp://ftp.cs.wisc.edu/wwt/pldi99_cache_def.{ps,pdf}`).

Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-Conscious Structure Definition. In *Proceedings of the SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, May 1999.

A program's cache performance can be improved by changing the organization and layout of its data—even complex, pointer-based data structures. Previous techniques improved the cache performance of these structures by arranging distinct instances to increase reference locality. These techniques produced significant performance improvements, but worked best for small structures that could be packed into a cache block.

This paper extends that work by concentrating on the internal organization of fields in a data structure. It describes two techniques—*structure splitting* and *field reordering*—that improve the cache behavior of structures larger than a cache block. For structures comparable in size to a cache block, structure splitting can increase the number of hot fields that can be placed in a cache block. In five Java programs, structure splitting reduced cache miss rates 10–27% and improved performance 6–18% beyond the benefits of previously described cache-conscious reorganization techniques.

For large structures, which span many cache blocks, reordering fields, to place those with high temporal affinity in the same cache block can also improve cache utilization. This paper describes `bbcache`, a tool that recommends C structure field reorderings. Preliminary measurements indicate that reordering fields in 5 active structures improves the performance of Microsoft SQL Server 7.0 2–3%.

This paper summarizes cache-conscious work from Chilimbi's thesis and related papers. (`ftp://ftp.cs.wisc.edu/wwt/computer00_conscious.{ps,pdf}`).

Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Making Pointer-Based Data Structures Cache Conscious. In *IEEE Computer*, TBD 2000.

Processor and memory technology trends portend a continual increase in the relative cost of accessing main memory. Machine designers have tried to mitigate the effect of this trend through a hierarchy of caches and a variety of other hardware and software techniques. These techniques, unfortunately, have only been partially successful for pointer-manipulating programs.

This paper explores a complementary approach of enlisting programmers and tool writers in the task of improving the cache locality of accesses to pointer-based data structures. Throughout, we exploit the location transparency of pointer-based data structures that allow changes to the memory (and cache) layout of nodes, records, fields, etc. We discuss how programmers can manually improve cache performance with techniques, such as clustering, compression, and coloring. We then explore how to lessen a programmer's burden with the help of semi-automatic and automatic tools for changing structure layout to improve cache performance. Techniques include reordering fields in a structure definition, carefully placing nodes in memory at allocation time, and dynamically reorganizing extant data structures.