Zoran Radović

Software Techniques for Distributed Shared Memory
Dissertation at Uppsala University to be publicly examined in Auditorium Minus, Museum Gustavianum, Friday, November 18, 2005 at 13:15 for the Degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract

In large multiprocessors, the access to shared memory is often nonuniform, and may vary as much as ten times for some distributed shared-memory architectures (DSMs). This dissertation identifies another important nonuniform property of DSM systems: nonuniform communication architecture, NUCA. High-end hardware-coherent machines built from large nodes, or from chip multiprocessors, are typical NUCA systems, since they have a lower penalty for reading recently written data from a neighbor's cache than from a remote cache. This dissertation identifies node affinity as an important property for scalable general-purpose locks. Several software-based hierarchical lock implementations exploiting NUCA are presented and evaluated, NUCA-aware locks are shown to be almost twice as efficient for contended critical sections compared to traditional lock implementations.

The shared-memory "illusion" provided by some large DSM systems may be implemented using either hardware software or a combination thereof. A software-based implementation can enable cheap cluster hardware to be used, but typically suffers from poor and unpredictable performance characteristics.

This dissertation advocates a new software-hardware trade-off design point based on a new combination of techniques. The two low-level techniques, fine-grain deterministic coherence and synchronous protocol execution, as well as profile-guided protocol flexibility, are evaluated in isolation as well as in a combined setting using all-software implementations. Finally, a minimum of hardware trap support is suggested to further improve the performance of coherence protocols across cluster nodes. It is shown that all these techniques combined could result in a fairly stable performance on par with hardware-based coherence.

Keywords: synchronization, distributed shared memory, write permission cache, nonuniform communication architecture, node affinity, locality, hardware-software trade-off, profiling, flexibility, trap-based memory architecture

Zoran Radović, Department of Information Technology, Uppsala University, Uppsala University, Department of Information Technology, Box 337, SE-751 05 Uppsala, Sweden

© Zoran Radović 2005

ISSN 1104-2516
ISBN 91-554-6385-1
urn:isbn:uu.diva-6088 (http://urn.kb.se/resolve?urn=urn:nbn:uu:diva-6088)
Mojoj dragoj porodici! Svih uzrasta i još neotkrivenih predispozicija... ;-)
Publications by the Author

This dissertation summarizes and discusses the results and contributions presented in several papers listed below.


I. Oskar Grenholm, Zoran Radović, and Erik Hagersten. Latency-hiding and Optimizations of the DSZOOM Instrumentation System. Technical Re-
port 2003-029, Department of Information Technology, Uppsala University, May 2003.

Paper A is a summary of my Master’s thesis [90] that presents the core DSZOOM idea and a first single-node evaluation called DSZOOM-EMU. I am the principal author of papers B and C, which improve on paper A.

I am also the principal author of papers D, E, and F that introduce numerous all-software NUCA locks.

Henrik Löf is the principal author of papers G and H that present a runtime system concept that enables unmodified Pthreads binaries to run transparently on clustered hardware. I am responsible for the binary instrumentation support of that study.

Paper I introduces write permission caching, and is a summary of Oskar Grenholm’s Master’s thesis [42], which was co-advised by Professor Erik Hagersten and me.

Papers J, K, and L are results of a close cooperation with Håkan Zeffer. Håkan and I jointly optimized the DSZOOM system.

Paper M introduces trap-based memory architectures. Håkan Zeffer and I jointly designed and implemented software-based coherence protocols.
Contents

1 Introduction & Motivation .............................................. 1
  1.1 Programming Models ............................................. 2
    1.1.1 Critical Sections and Concurrency ...................... 2
    1.1.2 Hardware Primitives for Mutual Exclusion .............. 3
    1.1.3 POSIX Threads ............................................ 4
    1.1.4 ANL Macros .............................................. 4
    1.1.5 Language Support for Concurrency ...................... 4
    1.1.6 Future Parallel Programming Languages ................ 6
  1.2 Implementing Shared Memory Architectures .................... 7
    1.2.1 The Cache Coherence Problem ............................. 7
    1.2.2 Computer Clusters ....................................... 9
    1.2.3 Software-Based Distributed Shared Memory .............. 10
    1.2.4 Hardware/Software Trade-Offs ....................... 11
  1.3 Nonuniform Communication Architectures ....................... 11
  1.4 This Dissertation ........................................... 14
    1.4.1 Efficient Synchronization for NUCAs ................... 14
    1.4.2 The DSZOOM System .................................... 16
    1.4.3 Summary of the Contributions ........................... 17
  2 Experimentation Environment ...................................... 19
    2.1 Hardware: Sun's WildFire Prototype ....................... 19
    2.2 Programs: SPLASH-2 Benchmark Suite ..................... 20
    2.3 Simulation: Simics & VASA ................................ 21
  3 NUCA Locks .................................................................. 23
    3.1 Introduction .................................................. 23
    3.2 Background and Related Work ................................. 24
      3.2.1 Atomic Primitives ...................................... 24
      3.2.2 Simple Locking Algorithms ............................. 26
      3.2.3 Software Queuing Locks ................................ 27
      3.2.4 Alternative Approaches ................................. 28
    3.3 Key Ideas behind RH Lock ..................................... 29
    3.4 The RH Lock Algorithm ....................................... 30
    3.5 Hierarchical Backoff Locks .................................. 33
      3.5.1 The HBO Lock ........................................... 33
      3.5.2 The HBO_GT Lock ...................................... 35
5.3.2 Simulated Protocols ........................................... 93
5.3.3 Benchmarks ......................................................... 96
5.3.4 Simulation Results ............................................... 97
5.4 Related Work ......................................................... 106

Sammanfattning (Summary in Swedish) 109

Acknowledgments 111

References 113
1. Introduction & Motivation

During the 1990s, shared-memory UNIX servers made parallel computing popular. They are commonly used today in the commercial and technical server markets. One of the prime reasons for the final acceptance of multiprocessors in the wider marketplace was their shared-memory programming abstraction, which made parallel systems easier to program and manage [14, 21, 25, 70, 78, 112]. Many programming paradigms and language primitives have been designed assuming the existence of shared-memory, including POSIX threads [49], the Argonne National Laboratory (ANL) parallel macro package [13], OpenMP [29], parallel implementations of Java [39], or Unified Parallel C (UPC) [17].

Early UNIX servers were typically populated with processors that could only process one thread at a time. It is common that the processors of such systems spend a majority of their time waiting for memory. In contrast, the chip multiprocessor (CMP) and/or the simultaneous multithreading (SMT) processors are able to process multiple threads simultaneously [88, 121]. The modern processors, such as IBM’s Power4 and Power5 [119, 57], AMD’s dual-core Opteron, Intel’s Xeon and Pentium 4, or Sun’s UltraSPARC 4 [118], are all capable of running several threads in parallel. Also, Sun’s upcoming Niagara architecture is said to run as many as 32 parallel threads on each chip [65]. When a thread must wait for memory, other threads in the same processor chip can still execute their own code. This radically improves chip utilization and increases application throughput. However, it is currently not clear how successful this multi-core architectural transition will be since there are several topics in both software and computer architecture community that need to be addressed. This dissertation touches both areas and is therefore divided into two parts.

The first part of this dissertation considers a cache-coherent multiprocessor system where all processors of a system share the same physical memory address space. That system is called a shared-memory multiprocessor. While the various optimizations of spin-lock implementations have in some instances led to enhanced performance, most solutions do not consider or exploit the node affinity of nonuniform shared-memory multiprocessors as an important property for scalable general-purpose locks. In this first part, we address the synchronization problem for such systems. The second part addresses an ef-
1.1 Programming Models

The majority of today's parallel programs are written in a sequential programming language such as C or Fortran coupled with the message passing library (MPI), which is inspired by a classical theory on communicating sequential processes (CSP) [47]. Hoare's paper on CSP suggests that input and output are basic primitives of programming and that parallel composition of communicating sequential processes is a fundamental program structuring method.

There is usually only one instance of the MPI program that is independently executed by each processor in the computing system. Programmers specify communication explicitly through MPI library routines for sending and receiving data. These routines are called two-sided message-passing routines since the processors on both sides of the communication need to be involved in any exchange. One processor calls the send routine and the other processor calls the receive routine, though not necessarily at the same time. Other library calls are provided for collective communication. One of the major advantages of MPI is its simplicity; programmers can get great flexibility with relatively few abstractions.

In shared-memory architectures, communication is implicit since the memory is typically accessible by all the processors. Consequently, programming paradigms for shared-memory multiprocessors focus on constructs for expressing concurrency and synchronization. Process-based models assume that all data associated with a process is private, by default, unless otherwise specified (using UNIX system calls such as shmat (2) and shmdt (2)). In contrast, threads assume that all memory is global. Usually, a thread is defined as a single sequential flow of control within a program, for example a multithreaded Web browser.

1.1.1 Critical Sections and Concurrency

Synchronization mechanisms are usually needed if we consider a system consisting of $N$ processes ($p_1, p_2, \ldots, p_N$). Each process could have a segment of code, called a critical section, in which the process may be changing shared variables or updating other resources of the computing system. When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is mutually exclusive.
When a programmer writes a program in which several concurrent threads are competing for resources, he or she must take precautions to ensure fairness. One of the possible definitions of a fair system is when each thread gets enough access to limited resources to make reasonable progress. The story of the dining philosophers, presented in 1965 by Edsger Dijkstra, is often used to illustrate various problems that can occur when many synchronized threads are competing for limited resources. There are at least three problems that can occur in such systems: starvation, deadlock, and livelock. Starvation occurs when one or more threads in the program are blocked from gaining access to a resource and, as a result, cannot make progress. Deadlock occurs when two or more threads are waiting on a condition that cannot be satisfied, such as when two (or more) threads are each waiting for the other(s) to do something. A system is in livelock when no processor is making forward progress in its computation even though transactions are being executed in the system.

Another unwanted situation that programmers of parallel applications strive to avoid is called a race condition. A race condition is a situation in which two or more threads or processes are reading or writing some shared data, and the final result depends on the timing of how the threads are scheduled. Race conditions can lead to unpredictable results and subtle program bugs.

There are numerous proposals to solve the critical section problem ranging from simple two-process algorithms to more general N-process solutions. The first N-process algorithm was published by Dijkstra in 1965 [30]. Dijkstra’s algorithm, which is based on an earlier two-process algorithm by Dekker, is livelock-free but not starvation-free. A related algorithm published by Knuth in 1966 was the first starvation-free solution [62].

1.1.2 Hardware Primitives for Mutual Exclusion

Today, most contemporary computer designers provide support for simple atomic operations in hardware. Software libraries usually provide higher-level synchronization operations that are constructed from simple atomic operations. The atomic operations may be implemented either as single instructions or through speculative read-write instruction pairs like load-locked and store-conditional.

All experiments and simulations in this dissertation are based on Sparc architectures. SPARC-V9 provides three hardware primitives for mutual exclusion:

- Compare and Swap
- Load and Store Unsigned Byte
- Swap

Each of these instructions is atomic, i.e., no other store can be performed between the load and store elements of the instruction.
1.1.3 POSIX Threads

Historically, hardware vendors have implemented their own proprietary versions of threads, making portability a concern for software developers. For UNIX systems, a standardized C language threads programming interface has been specified by the IEEE POSIX 1003.1c standard. Implementations of POSIX threads are usually referred to as Pthreads.

Pthreads are defined as a set of C language programming types and procedure calls, implemented with a pthread.h header/include file and a thread library; though this library may be part of another library, such as libc. The primary motivation for using Pthreads in high performance and technical computing is to realize potential program performance gains. When compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.

1.1.4 ANL Macros

In the late 1980s, Boyle et al. presented an abstract view of multiprocessors (a programming model) that allows us to write programs that are portable among a broad class of machines [13]. Their targets are three categories of machines:

- machines that have a globally shared memory
- machines that have no shared memory and in which the processors communicate by sending messages; and
- machines that are composed of loosely coupled “clusters” of processors.

This programming model is usually referred to as ANL macros since it was developed at the Argonne National Laboratory. There are numerous versions of these macros used mainly by the research community. For example, the SPLASH-2 benchmark suite [124] which is frequently used by almost all computer architecture research groups is written with so called PARMACS macros [7, 6] which are closely related to the original ANL proposal. All macro packages provide macros to create a process, allocate a segment of shared memory, and synchronize processes using locks, barriers and event flags. Similar functions can be achieved directly by using UNIX system calls. Table 1.1 shows the names of the ANL macros and UNIX calls that accomplish each of these operations.

1.1.5 Language Support for Concurrency

Modern parallel programming languages usually provide automatic support for various synchronization events as part of the language itself. For example in the Java language, a critical section can be a block or a method and is
<table>
<thead>
<tr>
<th>Operation</th>
<th>ANL Macros</th>
<th>UNIX System Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create new process</td>
<td>CREATE</td>
<td>fork/exec</td>
</tr>
<tr>
<td>Allocate shared memory region</td>
<td>G_MALLOC</td>
<td>shmget/shmat</td>
</tr>
<tr>
<td>Create synchronization object</td>
<td>LOCKINIT, BARINIT, PAUSEINIT</td>
<td>semget</td>
</tr>
<tr>
<td>Execute synchronization operation</td>
<td>LOCK/ UNLOCK BARRIER, SETPAUSE/ WAITPAUSE</td>
<td>semop</td>
</tr>
</tbody>
</table>

Table 1.1: Operations for creating parallel applications [101].

identified with the `synchronized` keyword as illustrated by the following example,

```java
public synchronized int method_name() {
    ...
}
```

The Java platform associates a lock with every object and the lock is acquired upon entering a critical section. Whenever a thread enters a synchronized method, the thread that called the method locks the object whose method has been called. Other threads cannot call a synchronized method on the same object until the object is unlocked. The acquisition and release of a lock is done automatically and atomically by the Java run-time system. This ensures that race conditions cannot occur in the underlying implementation of the threads, thus ensuring data integrity.

Java also provides support for explicit locks which are more flexible than the synchronized keyword because the lock can protect a few statements in a method. To create an explicit lock the programmer must instantiate an implementation of the `Lock` interface, usually `ReentrantLock`. Methods `lock` and `unlock` are then used to acquire and release the lock.

Another approach to provide concurrency based upon the existence of multiple threads in the shared memory programming paradigm is called OpenMP. That is an application program interface, jointly defined by a group of major computer hardware and software vendors. The goal is to provide a portable
and scalable model for developers of shared memory parallel applications. OpenMP is an explicit programming model, offering the programmer full control over parallelization. It uses the fork-join model of parallel execution. OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code. For example, the MASTER directive specifies a region that is to be executed only by the master thread of the team. All other threads on the team skip this section of code. For example:

```c
...  
#pragma omp master
   {
      x = x + 1;
   }
...
```

To ensure atomicity of critical sections, OpenMP provides the CRITICAL directive that specifies a region of code that must be executed by only one thread at a time:

```c
...  
#pragma omp parallel shared(x, y)
   {
...  
#pragma omp critical
      {
         x = x + 1;
         y = y * 42;
      }
...  
} /* end of parallel section */
...
```

### 1.1.6 Future Parallel Programming Languages

New parallel computer architectures sometimes require new parallel programming languages. There are numerous attempts and ongoing projects that strive to provide a modern and more productive parallel language. Some of the proposals are presented by participants of the DARPA's high productivity computing systems (HPCS) program, which has the goal of increasing supercomputer productivity by the year 2010. Their motivation is presented at http://www.highproductivity.org/:

"The DARPA High Productivity Computing Systems is focused on providing a new generation of economically viable high productivity computing systems"
for the national security and industrial user community in the 2007-2010 time-frame. The goal is to provide systems that double in productivity (or value) every 18 months."

Companies such as Cray, Inc., Sun Microsystems or IBM are all proposing at least one new language that will more easily and efficiently exploit modern computer architectures by providing a higher level of expression than current parallel languages. The main target here is a high-performance computing. All these new languages are heavily inspired by many traditional languages, such as High-Performance Fortran (HPF), the Java Programming Language, NextGen, Scala, Eiffel, Self, Standard ML, Haskell, Scheme, etc.

Cray's Chapel proposal is being developed in conjunction with California Institute of Technology (Jet Propulsion Laboratory). The main language features are support for data-locality optimizations and data distribution for data-driven placement of subcomputations. Chapel's parallel concepts are based on ideas from HPF, ZPL, and the Cray MTA's extensions to Fortran and C.

Sun's Fortress Programming Language is a general-purpose, statically typed, component-based programming language designed for robustness and high productivity. Also here, high-performance software is the main concern while the main inspiration is Java and HPF. One of the design principles is that parallelism is encouraged everywhere (for example, it is intentionally harder to write a sequential loop than a parallel loop).

1.2 Implementing Shared Memory Architectures

One major complication associated with an architectural support for shared-memory parallel languages in multiprocessing computer systems relates to maintaining the coherency of program data shared across multiple computing nodes/processors.

1.2.1 The Cache Coherence Problem

In general, the cache-coherent system must implement an "ordering policy" that defines an order of operations (in particular memory operations) initiated by different processors. Traditional shared-memory implementations manage the shared memory address space by dividing the memory into blocks or cache lines; for example, 64 bytes in size. During the execution of a system's workload, cache lines often move between the nodes. This "movement" needs to be performed such that operations on the cache line occur in a manner that is consistent with the ordering/memory model. The exact definition of a consistent view of memory is defined by a memory consistency model [1]. The simplest and most intuitive memory consistency model is sequential consistency [58],
introduced by Leslie Lamport, who declared that a multiprocessing system had sequential consistency if:

"... the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."

This definition was one of the first statements on what constituted “correct” behavior for a multiprocessing system. Without a “coordination mechanism,” one of the processors may perform an update that is not properly reflected in another node. Maintaining a unified, coherent view of shared-memory locations is thus essential from the standpoint of program correctness. The protocols that maintain coherence for multiprocessor systems are called cache coherence protocols.

Small multiprocessor systems with few processors often implement the coherence using hardware-based coherence protocols. These protocols control the contents of the processor’s caches by copying, replicating and invalidating their contents at a cache line granularity. The protocol also makes sure that the memory content is updated when needed. The smaller shared-memory systems often provide a uniform view of the shared memory, i.e., the unloaded access time to the memory is uniform. This is often referred to as uniform memory architecture (UMA) or symmetric multiprocessor (SMP). Larger multiprocessor systems supporting shared memory are typically build from several nodes, each node with some number of processors as well as some memory. The architecture of one such node can be reminiscent of a UMA system. A distributed coherence protocol, implemented in hardware, gives the illusion that the sum of memory of all the nodes is the shared memory of the system. The protocol is also responsible for keeping the coherence between all the system’s caches. The access time to this shared memory with respect to the processors of one node is not uniform. Accessing memory physically located in the own node is of course much faster that accessing memory in a remote node. Such systems are called cache-coherent nonuniform memory architectures (CC-NUMAs) or hardware-based distributed shared memory architectures (DSMs).

There are numerous events that can take place inside a DSM system, and some of them are explained here:

- **read request**: one of the processor desires to cache a copy of a block that it can read

1The term symmetric multiprocessor (SMP) is also sometimes used to label the functional property of sharing without any timing considerations, i.e., all processors may access the entire memory and any processor can handle any I/O device. As such, the term SMP can be used to describe both UMAs and CC-NUMAs.
• write request: one of the processors desires a copy of the block it can both read and write
• cache-to-cache miss: a miss that requires another processor's cache to supply the data (often caused by an access to recently modified shared data)

1.2.2 Computer Clusters

There are many different types of computer clusters. Most of them do not provide support for shared memory. Clusters have been used since Tandem introduced them in 1975 for fault tolerance. Digital Equipment Corporation (DEC) also offered VAX clusters in 1983. Generally speaking, a computer cluster is a group of individual systems (PCs, workstations, blades, or multiprocessors/servers) that typically behave and perform as a single system under one management framework. A cluster environment consists of a number of computing nodes, a cluster interconnect component, and cluster software. For example, Sun’s latest “supercluster” is defined below [117].

• Compute node: Each server in the cluster environment, sometimes called a compute node, has one or more processors, data resources, and its own memory, operating system, as well as application software. A node can be a single server or a domain in a high-end server. SMP servers provide large memory, processing and data movement capacity as a versatile building block of the cluster.

• Cluster interconnect: Cluster nodes are connected to each other by the cluster interconnect. The cluster interconnect carries the message and data traffic between the nodes. The interconnect fabric and the communication protocol forms the network.

• Cluster software: Distributed resources and capabilities of individual cluster nodes are managed collectively and made available to the users and applications as a single resource by the cluster software. The cluster software determines the purpose and function of the cluster.

The supercluster is available in 2-, 4-, and 8-node configurations. With up to eight Sun Fire 15K servers, clusters based on Sun Fire Link [113] offer almost 2 TFLOPS peak performance—a total of about 800 UltraSPARC III Cu processors. However, these systems do not currently support the global view of a common shared memory; users are instead forced to use a message passing programming interface during the development time of their parallel applications.

So-called Beowulf clusters have gained popularity [115]. Beowulf systems are built from several PCs, workstations, or blades, connected by a “standard”

---

2In many Sun servers, like the Sun Fire 6800, Sun Fire 12K, and Sun Fire 15K servers, processors can be partitioned into a number of completely independent domains, each of which runs a separate copy of the Solaris operating system and participates as a member in the cluster.
interconnect. Each cluster node runs its own operating system, and there is no globally shared data. The Beowulf model suits certain classes of applications well, especially many independent jobs running in parallel. Some loosely coupled parallel algorithms can also run well on Beowulfs. Thus, Beowulf systems are here to stay for many years to follow. However, while this is a great price/performance choice for many applications, it is in many ways a step backwards in terms of technology. Beowulf clusters do not efficiently support the popular abstraction of a common memory, shared in a coherent way among the cluster nodes, and cannot execute certain common parallel programming paradigms and languages. Basically, Beowulf clusters forces the applications that run across several processors to explicitly communicate shared data. This can be a cumbersome task, especially for dynamically scheduled and load-balanced systems. It also forces programmers to describe the problem using a message-passing paradigm even though the algorithm may be best described using well established shared-memory constructs. One solution to run shared-memory applications on such clusters, called shared virtual memory (SVM), was introduced by Li and Hudak [74]. A first SVM prototype was implemented on an Apollo ring in the mid 1980s. In this dissertation, we refer to this class of systems as software-based DSM systems, SW-DSMs.

1.2.3 Software-Based Distributed Shared Memory

Traditional software-based shared memory relies on the page protection system to detect when a coherence activity is needed [10, 33, 50, 51, 54, 59, 74, 72, 73, 75, 114, 116]. A page can be put in the exclusive state in a node, giving the processors of that node write permission to the page, while the processors of the other nodes would have no access rights to the page and would trap the first time they access it. The trap will wake up the SW-DSM software-protocol engine of that node and send a message to a dedicated home node’s protocol engine requesting a copy of the page. The home node would forward the request to the protocol engine of the node with the exclusive copy of the page, which would downgrade its access to read-only access right of the page and send a copy of the page to the requesting node’s protocol engine. There are three major drawbacks of this scheme: (1) the interrupt-based asynchronous messages between the protocol engines add substantial latency; (2) the large coherence units (i.e., a page), instead of commonly used smaller cache line size, creates an ample of false sharing; and (3) each protocol action requires a large quantity of data to be transferred. Another major drawback is that the synchronization is slow because it is typically implemented through explicit messages.
1.2.4 Hardware/Software Trade-Offs

Chapter 9 of Culler and Singh [28] provides a detailed discussion of hardware/software trade-offs for shared-memory implementations. In this dissertation, we address primarily high design and implementation cost.

There is a wide range of options for hardware/software trade-offs for implementing coherent shared memory, ranging from all-hardware coherence to implementations relying on no specific hardware support at all. It is fairly common, however, that systems rely on both hardware as well as run-time software to improve the performance. Most hardware-based DSMs rely on hardware performance counters to guide the page migration software [70, 44]. The Sun WildFire system has hardware support for detecting pages in need of replication [44]. Replicated copies of each page can be instantiated by software, but the coherence between the multiple copies is kept by hardware on a cache-line sized basis. The hardware/software boundary is also sometimes crossed in order to handle some of the "corner cases" of the coherence protocol. One such example is the MIT Alewife machine that adds efficient support for trapping to a software handler on the rare event of massive sharing [2]. These systems rely on hardware coherence and use software to simplify part of the design.

The trade-off between hardware/software is even more apparent in systems with programmable coherence engines or dedicated coherence processors such as Stanford's FLASH [67], Sun's S3mp [86] and Wisconsin's Typhoon-0 [98]. Here, the entire coherence protocol is controlled by specialized software, which enables flexible protocol adoptions as well as protocol bug correction. SMTp is a more recent proposal [23] in which the coherence protocol is run by one SMT thread that handles coherence actions on processor cache misses. While these systems rely on software handlers for coherence, they require either dedicated coherence processors capable of snooping the memory bus [67, 86, 98] or extended memory controllers together with SMT processor pipeline modifications [23].

1.3 Nonuniform Communication Architectures

Shared-memory architectures with a nonuniform memory access time to the shared memory (CC-NUMA) are common today [36, 44, 70, 71, 78]. All these computers implement a shared address space where the time to access a particular piece of memory depends on its location. Due to the popularity of NUMA systems, optimizations directed to such architectures have attracted much attention in the past. For example, optimizations involving the migration and replication of data in NUMAs have demonstrated a great performance improvement in many applications [44, 70, 85].
Most NUMA architectures also have the characteristic of a nonuniform communication architecture (NUCA), in which the access time from a processor to other processors' caches varies greatly depending on their placement. In node-based NUMA systems, in particular, processors have much shorter access times to other caches in their group than to the rest of the caches. Recently, technology trends have made it attractive to run more than one thread per chip, using either the chip multiprocessor and/or the simultaneous multithreading approach. Large servers, built from several such chips, can therefore be expected to form NUCAs, since collated threads will most likely share an on-chip cache at some level [8]. In addition, since many of today's applications exhibit a large fraction of cache-to-cache misses [9], optimizations which consider the NUCA nature of a system may also lead to significant performance enhancements.

Many large-scale shared-memory architectures have nonuniform access time to the shared memory. In order to make a key difference, the nonuniformity should be substantial—let's say at least a factor of two between best and worst unloaded latency. Most NUMA architectures also have a substantial difference in latency for cache-to-cache transfer—a nonuniform communication architecture. We consider NUCA to be any architecture in which the unloaded latency for a processor accessing data recently modified by another processor differs at least by a factor of two, depending on where that processor is located (see Figure 1.1 for an example).

![Figure 1.1: Example of a two-node NUCA architecture.](image)

DASH was the first NUCA machine [71]. Each DASH node consists of four processors connected by a snooping bus. A cache-to-cache transfer from a cache in a remote node is 4.5 times slower than a transfer from a cache in the same node. We call this the NUCA ratio. Sequent's NUMA-Q has a similar topology, but its NUCA ratio is closer to 10 [78]. Both DASH and NUMA-Q have a remote access cache (RAC), located in each node, that simplifies the implementation of the node-local cache-to-cache transfer. Table 1.2 shows several NUCA examples with their NUCA ratios.
Sun's WildFire system can have up to four nodes with up to 28 processors each, totaling 112 processors [44]. Parts of each node's memory can be turned into an RAC using a technique called coherent memory replication (CMR) [44]. Accesses to data allocated in WildFire's CMR cache have a NUCA ratio of about six, while accesses to other data only have a minor latency difference between node-local and remote cache-to-cache transfers.

Compaq's DS-320 (which was also code-named WildFire) can connect up to four nodes, each with four processors sharing a common DTAG and directory controller [36]. Its NUCA ratio is roughly 3.5.

<table>
<thead>
<tr>
<th>Year</th>
<th>NUCA Example</th>
<th>NUCA Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1992</td>
<td>Stanford DASH [71]</td>
<td>≈ 4.5</td>
</tr>
<tr>
<td>1996</td>
<td>Sequent NUMA-Q [78]</td>
<td>≈ 10</td>
</tr>
<tr>
<td>1999</td>
<td>Sun WildFire [44]</td>
<td>≈ 6</td>
</tr>
<tr>
<td>Today &amp; Future</td>
<td>CMP [88] and/or SMT [121]</td>
<td>≈ 6–10</td>
</tr>
</tbody>
</table>

Table 1.2: NUCA ratios for different architectures.

Future microprocessors can be expected to run many more threads on a chip through a combination of CMP and SMT technology. This can already be seen in the Pentium 4’s Hyperthreading and the IBM Power4’s dual CMP processors on a chip. The Piranha CMP proposal has eight CMP threads running on each chip [8]. Larger systems, built from many such CMPs, are expected to have a NUCA ratio of between six and ten depending on the technology chosen.

It is possible that several levels of non-uniformity will be present in future large-scale servers. A simple example of this would be one of today's NUMA architectures populated with CMP processors instead of traditional single-threaded processors. This would create a hierarchical NUMA and NUCA property of the system.

Not all architectures are NUMAs or NUCA. The recent SunFire 15k architecture can have up to 18 nodes, each with four processors, memory, and directory controllers [22]. The nodes are connected by a fast backplane. It has a flavor of both NUMA and NUCA. However, both of its NUMA and NUCA ratios are below two. The SGI Origin 2000 is a NUMA architecture with a NUMA ratio of around three for reasonably sized systems [70]. However, it does not efficiently support cache-to-cache transfers between adjacent processors and also has a NUCA ratio below two.
1.4 This Dissertation

This dissertation is divided into two parts. We present novel software techniques for distributed shared memory architectures. In particular, these research questions are our main focus:

- Hardware/Software interdependence for synchronization (section 1.4.1)
- Hardware/Software trade-offs in DSM implementations (section 1.4.2)

1.4.1 Efficient Synchronization for NUCAs

This dissertation presents several NUCA-aware locks that fulfill general performance goals for locks, as given by Culler et al. [28]:

1. **Low latency.** If a lock is free and no other processors are trying to acquire it at the same time, a processor should be able to acquire it with low latency.
2. **Low traffic.** If many or all processors try to acquire a lock at the same time, they should be able to acquire the lock one after the other with as little generation of traffic or bus transactions as possible.
3. **Scalability.** Neither latency nor traffic should scale quickly with the number of processors used.
4. **Low storage cost.** The information needed for a lock should be small and should not scale quickly with the number of processors.
5. **Fairness.** Ideally, processors should acquire locks in the order their requests are issued. At the least, starvation or substantial unfairness should be avoided. Since starvation is usually unlikely, the importance of fairness must be traded off with its impact on performance.

In fact, we paid attention only to the first four goals and partly downplayed the last one (with the exception of goals for starvation avoidance). We also paid attention to the data locality created by our lock algorithms; in other words, our additional goal was to maximize the node locality of NUCA architectures. Today, low latency in low contention situations and scalability in high contention situations is becoming more important as processor counts grow.

On a NUCA, it is attractive from a performance point of view to hand the lock over to a waiting neighbor, rather than a thread from a remote node. Favoring the neighbor will improve the lock handover time, as well as the access to the critical data that most likely reside in its cache. However, in such a scheme attention must also be given to starvation avoidance.

The goal of this work is to create a new set of locks that efficiently exploit communication locality in a NUCA while minimizing the potential risk of starvation. In order to be generally usable, such a lock should scale to a large number of nodes, handle contended locks well, have reasonable memory space requirements, and introduce minimal overhead for the uncontested locks.
The RH lock is our first NUCA-aware lock proposal that exploits 2-node NUCA's [95, 94]. The goal of the RH lock is (1) to create a lock that minimizes the global traffic generated at lock handover and (2) to maximize the node locality of NUCA architectures. In the RH lock proposal (sections 3.3 and 3.4), every node contains a copy of the lock. The total lock storage for a 2-node case is thus $2 \times \text{sizeof(lock)}$. The global traffic at lock handover is minimized by simply making sure that only one thread per node (the "node winner") performs remote-spinning if the lock is currently not owned by a thread in the node. The node locality of NUCA's is increased by handing over the lock to another thread running in the same node. This not only cuts down the lock-handover time, but creates locality in the critical section work, since its data structures already reside in the node. The RH lock is implemented on the SPARC-V9 architecture with three atomic operations. This lock could easily be adapted to noncoherent systems as well, assuming the existence of remote put, get, and atomic operations in the networking hardware.

There are few drawbacks with this original RH lock proposal. First, the lock implementation is vulnerable to starvation. Second, the implementation is not particularly portable since the allocation and a physical placement of the memory in different nodes is sometimes difficult or even impossible task for many machines. Finally, this proof-of-concept implementation supports only two NUCA nodes. A set of simple hierarchical backoff (HBO) locks that are based on traditional software spin-lock implementations with exponential backoffs are presented in section 3.5.1. All HBO proposals use only one atomic operation (compare-and-swap) and all of them are dependent on a per-thread node_id information [96]. In summary, the goals for HBO locks are the following:

- Efficiently exploit communication locality (create node affinity) in a NUCA
- Handle contended locks well
- Introduce minimal overhead for uncontested locks (the most common case)
- Scale to many NUCA nodes
- Reasonable memory space requirements
- Minimizing the potential risk of starvation
- Simple and portable implementation

RH and HBO locks are compared with other software-based locks using simple synchronization microbenchmarks. The total traffic in the system is significantly reduced for NUCA-aware locks compared with other software-based locks. An application study demonstrates superior performance for applications compiled with RH and HBO locks and with high lock contention and competitive performance for other programs.
1.4.2 The DSZOOM System

With the advent of low-latency high-bandwidth networking hardware, clusters of computers strive to offer the same processing power as high-end servers for a fraction of the cost. In such environments, shared memory has been traditionally limited to software page-based distributed shared-memory systems, which control access to shared memory using the memory’s page protection to implement coherence protocols. Unfortunately, false sharing and inter-task and/or poll-based asynchronous protocol messaging force such systems to resort to weak consistency shared-memory models, which complicate the shared-memory programming model. The DSZOOM system [90, 91, 92, 93], presented in this dissertation, is a fine-grained distributed software-based shared-memory implementation that differs from most other software DSM implementations. The DSZOOM system is an all-software solution that creates the “illusion” of a single shared memory across the entire cluster using a software run-time layer, attached between the application and the hardware. DSZOOM differs from most other software-based DSM implementations in five major areas:

1. **Fine-grain coherence**: Instead of relying on page-based coherence units, the code instrumentation [43, 69] technique is used to expand load and store instructions that may touch a shared data into a sequence of instructions that performs in-line coherence checks. This work is originally inspired by DEC’s Shasta [105, 103, 100] and Wisconsin’s Blizzard-S [109, 108] proposals from the mid 1990s, which are two cost-effective fine-grain software DSM implementations.

2. **Protocol in requesting processor**: If the coherence check determines that global coherence action is required, the entire protocol is run in the requesting processor, which otherwise would have been idle waiting for the coherence action to be resolved.

3. **No asynchronous interrupts**: Since the entire coherence protocol is run in the requesting processor/node, there is no need to send asynchronous interrupts to “coherence agents” in other nodes, i.e., this also removes the overhead of asynchronous interrupts from the remote latency timing path.

4. **Deterministic coherence protocol**: The global coherence protocol is designed to avoid all traditional corner cases of cache coherence protocols. Only one thread at a time can have the exclusive right to produce global coherence activity to each piece of data. This greatly simplifies the task of designing an efficient and correct coherence protocol.

5. **Thread-safe protocol implementation**: While most other implementations have a single protocol thread per node, our scheme allows several threads in the same node to perform protocol actions at the same time.
The DSZOOM technology should be applicable to a wide range of cluster implementations. It assumes a cluster built from "standard" SMP nodes. This allows each node to contain several processors, whose caches and memory content are kept coherent within the node by a hardware-coherence protocol. The proposal is assuming a high-bandwidth, low-latency cluster interconnect, supporting put, get, and atomic operations to a remote node's memory, but with no hardware support for globally coherent shared memory. Example of cluster interconnects are the common Beowulf SCI interconnects, Sun's "supercluster" Sun Fire Link [113], and the InfiniBand standard [52].

Section 4.3 extends DSZOOM's flexibility with simple synchronous coherence protocols that support more than 50 different optimization combinations (coherence flags). The coherence protocol can be parameterized by setting a set of appropriate coherence flags when the application is started, much like the optimization flags of compilers. The system combines code instrumentation and page protection mechanisms. Fine-grain access control checks applied at shared loads and stores avoid false sharing without any application rewriting or memory model weakening. The page protection mechanism minimizes unnecessary global memory replication and, in particular, works as an efficient bandwidth-reduction technique for update-based protocols. A very simplistic low-overhead profiling mode of the system is capable of finding a set of appropriate coherence flags for the studied applications. This is an automatic single-run process based on the profile run feedback.

Chapter 5 explores a new design point in the trade-off between hardware and software to implement shared memory. We call our proposal the trap-based memory architecture (TMA). In TMA, the coherence-trap hardware is responsible for access control checks and to generate a trap whenever a coherence violation is detected. Coherence is then maintained in software by coherence-trap interrupt handlers, which allow great protocol flexibility.

We evaluate one simplistic implementation of TMA called TMA Lite. Detailed full-system simulation of 4-node systems, based on chip multiprocessors, shows that our TMA Lite system can perform on par with a highly optimized hardware DSM system running on the same node and interconnect hardware.

1.4.3 Summary of the Contributions
In summary, this work includes novel research activities in these areas:
- Identifying a new property of DSMs: nonuniform communication architecture (NUCA).
- Presenting several lock algorithms that exploit NUCAs.
- Improving the traditional microbenchmarks for synchronization evaluations.
• A novel software-based shared-memory proposal, DSZOOM, and an implementation proposal thereof.
• A performance characterization of the proposed implementation.
• A latency-hiding technique for memory-store operations: write permission cache (WPC).
• A coherence flag approach that provides per-application flexibility in software-based DSM systems.
• A cost-effective processor extension that provides higher transparency of software DSM systems: trap-based memory architecture (TMA).
2. Experimentation Environment

This chapter gives an overview of the experimentation environment that is used throughout this work. Basically, the identical experimentation hardware is used since the start of this research: a 2-node Sun WildFire machine (described in section 2.1). All application studies use well-known workloads from the SPLASH-2 benchmarks suite [124] that are shortly presented in section 2.2. In addition, several simple benchmarks are introduced and used to evaluate many properties of software-based locking algorithms studied. One of our latest papers proposes a hardware support for software-based shared memory why we naturally need to rely on simulation techniques. Section 2.3 gives a short introduction to our full-system simulation approach.

2.1 Hardware: Sun’s WildFire Prototype

The experimentation hardware used in this dissertation is an experimental prototype multiprocessor called WildFire, built by Sun Microsystems. The WildFire is a cache-coherent nonuniform memory architecture (CC-NUMA) with unusually large nodes (up to 28 processors per node). The individual nodes in the WildFire system can be any of Sun’s E series multiprocessors (E5x00, E5x00, E4x00, or E3x00). WildFire can connect two to four multiprocessors by replacing one dual-processor (or I/O) board with a WildFire Interface (WFI) board, yielding up to 112 processors (4×28). The WFI board supports one coherent address space across all four multiprocessor nodes (it plugs into the bus and sees all memory requests). Each WFI has three ports that connect to up to three additional WildFire nodes, each with a raw bandwidth of 800 MB/sec in each direction (Figure 2.1). A hardware configuration used in this dissertation is shown below:

- **E5000 nodes.** The individual nodes in our WildFire configuration are Sun Enterprise E5000 symmetric multiprocessors [112]. The server has 16 UltraSPARC II (250 MHz) processors and 4 GB uniformly shared memory with an access time of 330 ns (imbench latency [81]) and a total bandwidth of 2.7 GB/sec. Each processor has a 16 kB on-chip instruction cache, a 16 kB on-chip data cache, and a 4 MB second-level off-chip data cache.

- **WildFire configuration.** The configuration used in this dissertation, is a 2-node machine, built from two E5000 nodes with 16 processors each.
access time to local memory is the same as above, 330 ns, while accessing data located in the other E6000 node takes about 1700 ns (Imbench latency). The entire system runs a slightly modified version of the Solaris 2.6 operating system. The WildFire system is a highly configurable machine and is well suited for research environments. To read more on WildFire, see Hagersten and Koster [44], Noordergraaf and van der Pas [85], or Hennessy and Patterson [45]. An important optimization of the WildFire system is the support for page migration and replication. The machine uses page-level mechanisms supported by the operating system to migrate/replicate a page in memory, although coherence is still maintained at the cache-block level. This technique is called **coherent memory replication** (CMR) [44].

![Diagram of WildFire system](image)

*Figure 2.1: Sun's WildFire prototype connects up to four Sun E-series multiprocessors by inserting one WF1 board in each node. I/O boards are not shown in this figure.*

### 2.2 Programs: SPLASH-2 Benchmark Suite

The benchmarks that are used in this dissertation are well-known workloads from the SPLASH-2 benchmark suite [124], originally developed for hardware multiprocessors. The SPLASH-2 suite is a mixture of eight complete applications and four computational kernels. The programs are representative for scientific, engineering, and computing graphic fields. Only unmodified benchmarks from the original Stanford University distribution are used.
in this dissertation. A short description of each application from the original SPLASH-2 paper by Woo et al. [124] is given here:

- **Barnes** — This application simulates the interaction of a system of bodies (galaxies or particles, for example) in three dimensions over a number of time steps, using the Barnes-Hut hierarchical N-body method.
- **Cholesky** — The blocked sparse Cholesky factorization kernel factors a matrix into the product of a lower triangular matrix and its transpose.
- **FFT** — The FFT kernel is a complex 1-D version of the radix-$\sqrt{n}$ six-step FFT algorithm. This kernel is optimized to minimize interprocessor communication.
- **FMM** — Like Barnes, the FMM application also simulates a system of bodies over a number of timesteps. However, it simulates interactions in two dimensions using a different hierarchical N-body method called the adaptive Fast Multipole Method.
- **LU-c & LU-nc** — A classical method for blocked LU decomposition. It factors a dense matrix into the product of a lower triangular and upper triangular matrix. LU-c is a more optimized version with contiguous allocation of data.
- **Ocean-c & Ocean-nc** — The Ocean application studies large-scale ocean movements based on eddy and boundary currents. Ocean-c is a more optimized version of Ocean-nc.
- **Radiosity** — This program computes the equilibrium of light in a scene using the interactive hierarchical diffuse radiosity method.
- **Radix** — The integer radix sort kernel.
- **Raytrace** — This application renders a three-dimensional scene using ray tracing.
- **Volrend** — This application renders a three-dimensional volume using a ray casting technique.
- **Water-Nsquared & Water-Spatial** — Water simulations with and without spatial data structure. This application evaluates forces and potentials that occur over time in a system of water molecules. Water-Spatial solves the same problem as Water-Nsquared, but uses a more efficient algorithm.

2.3 Simulation: Simics & VASA

We use the Simics full-system simulator [79] extended with an out-of-order processor model, memory hierarchy and a system interconnect model provided by the simulation package called Vasa [122]. Vasa is a highly configurable package developed for the Virtutech Simics simulator that can model complex SMT/CMP processors in great detail. We simulate a SPARC-V9 system running an unmodified Solaris 9 operating system. Our processor model
is based on the Simics Micro Architectural Interface which guarantees correctness leaving timing modeling to processor and memory system models.
3. NUCA Locks

This chapter identifies node affinity as an important property for scalable general-purpose locks. Nonuniform communication architectures (NUCAs), for example CC-NUMAs built from a few large nodes or from chip multiprocessors, have a lower penalty for reading data from a neighbor’s cache than from a remote cache. Lock implementations that encourage handing over locks to neighbors will improve the lock handover time, as well as the access to the critical data guarded by the lock, but will also be vulnerable to starvation.

We propose a set of simple software-based hierarchical backoff locks (RH and HBO) that create node affinity in NUCAs. A solution for lowering the risk of starvation is also suggested. Our new locks are compared with other software-based lock implementations using simple benchmarks, and are shown to be very competitive for uncontested locks while being more than twice as fast for contended locks. An application study also demonstrates superior performance for applications with high lock contention and competitive performance for other programs.

3.1 Introduction

The scalability of a shared-memory application is often limited by contention for some critical section, such as modification of shared data guarded by mutual exclusion locks. The simplest, and most widely used, test-and-use lock implementation suffers from poor performance at high contention: the more contented the critical section gets, the lower is the rate at which new threads can enter it. This very inappropriate behavior is mostly due to the vast amount of traffic generated at each lock handover event.

Some alternative lock implementations’ traffic is less dependent on the number of contenders, which is why their lock hand-over rates do not decrease as significantly at high contention. The simplest way to limit the contention traffic is to apply some backoff strategy that causes the threads to access the common lock variable less frequently the longer they have waited. The more advanced queue-based locks instead maintain a first come, first served order between the contending threads. Each contender will only spin on the dedicated flag set at its predecessor’s release of the lock, and contenders ordered after it will not be affected [26, 80, 82]. However, the complicated software
queuing locks are less efficient for uncontested locks, which have led to the
creation of even more complicated adaptive hybrid proposals in the quest for
a general-purpose solution [76].

Shared-memory architectures with a nonuniform memory access time to the
shared memory (CC-NUMAs) are gaining popularity. Most systems that form
NUMA architectures also have the characteristic of a nonuniform communi-
cation architecture (NUCA), in which the access time from a processor to other
processors' caches varies greatly depending on their placement. In node-based
NUMA systems; in particular, processors have much shorter access times to
other caches in their group than to the rest of the caches. Recently, technol-
yogy trends have made it attractive to run more than one thread per chip, using
either the chip multiprocessor (CMP) and/or the simultaneous multithreading
(SMT) approach. Large servers, built from several such chips, can therefore
be expected to form NUCAs, since collated threads will most likely share an
on-chip cache at some level [8].

Due to the popularity of NUMA systems, optimizations directed to such ar-
chitectures have attracted much attention in the past. For example, optimiza-
tions involving the migration and replication of data in NUMAs have demon-
strated a great performance improvement in many applications [44, 70, 85]. In
addition, since many of today's applications exhibit a large fraction of cache-
to-cache misses [9], optimizations which consider the NUCA nature of a sys-
tem may also lead to significant performance enhancements.

3.2 Background and Related Work

Ideally, synchronization primitives should provide good performance under
both high and low contention without requiring substantial programmer effort.
Mutual exclusion (lock-unlock) operations can be implemented in a variety of
different ways, including: atomic memory primitives; nonatomic memory
primitives (load-linked/store-conditional), and explicit hardware lock-unlock
primitives (Cray's Xmp lock registers, DASH's lock-unlock operations on di-
rectory entries, or Goodman's queue-on-lock-bit). In this dissertation, we will
concentrate on implementing locks entirely in software using the atomic mem-
ory primitives, that are available in the majority of modern processors.

3.2.1 Atomic Primitives

In this dissertation, we make reference to three atomic operations:

1. \texttt{taw(address)} atomically writes a nonzero value to the \texttt{address} memory
   location and returns its original contents;
2. \texttt{swap(address, value)} atomically writes a value to the address memory location and returns its original contents; and

3. \texttt{cas(address, expected_value, new_value)} atomically checks the contents of a memory location \texttt{address} to see if it matches an \texttt{expected_value} and, if so, returns its original contents and replaces it with a \texttt{new_value}.

SPARC-V9 provides \texttt{tas}, \texttt{swap}, and \texttt{cas} and is our target architecture. Sparc's \texttt{cas} operation is called \textit{Compare and Swap}, and the \texttt{tas} operation is called \textit{Load and Store Unsigned Byte}. Sections A.9 and A.29 of the SPARC architecture manual (version 9) describe these operation in more detail [123].

\textbf{Compare and Swap (CASA, CASXA)}

Compare-and-swap is an atomic operation which compares a value in a processor register to a value in memory, and, if and only if they are equal, swaps the value in memory with the value in a second register. Both 32-bit (CASA) and 64-bit (CASXA) operations are provided. The compare-and-swap operation is atomic in the sense that once begun, no other processor can access the memory location specified until the compare has completed and the swap (if any) has also completed and is potentially visible to all other processors in the system.

<table>
<thead>
<tr>
<th>Suggested Assembly Language Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{casa [reg]l immasi, regr2, regrd}</td>
</tr>
<tr>
<td>\texttt{casa [reg]l%asi, regr2, regrd}</td>
</tr>
<tr>
<td>\texttt{casxa [reg]l immasi, regr2, regrd}</td>
</tr>
<tr>
<td>\texttt{casxa [reg]l%asi, regr2, regrd}</td>
</tr>
</tbody>
</table>

"The CASA instruction compares the low-order 32 bits of register \texttt{r[r2]} with a word in memory pointed to by the word address in \texttt{r[r1]}. If the values are equal, the low-order 32 bits of register \texttt{r[rd]} are swapped with the contents of the memory word pointed to by the address in \texttt{r[r1]}, and the high-order 32 bits of register \texttt{r[rd]} are set to zero. If the values are not equal, the memory location remains unchanged, but the zero-extended contents of the memory word pointed to by \texttt{r[r1]} replace the low-order 32 bits of \texttt{r[rd]} and the high-order 32 bits of register \texttt{r[rd]} are set to zero.

The CASXA instruction compares the value in register \texttt{r[rs2]} with the doubleword in memory pointed to by the doubleword address in \texttt{r[rs1]}, If the values are equal, the value in \texttt{r[rd]} is swapped with the doubleword pointed to by the doubleword address in \texttt{r[rs1]}. If the values are not equal, the contents of the doubleword pointed to by \texttt{r[rs1]} replaces the value in \texttt{r[rd]}, but the memory location remains unchanged."
Load and Store Unsigned Byte (LDSTUB)
The load-store unsigned byte instruction loads a byte value from memory to a
register and writes the value 0xFF into the addressed byte atomically.

Suggested Assembly Language Syntax

\texttt{ldstub [address], reg\_rd}

"The load-store unsigned byte instruction copies a byte from memory into
\texttt{r[rd]}, and then rewrites the addressed byte in memory to all ones. The fetched
byte is right-justified in the destination register \texttt{r[rd]} and zero-filled on the left."

3.2.2 Simple Locking Algorithms
Two very commonly used busy-wait algorithms are TATAS and TATAS\_EXP.
The contention produced by the traditional test\&set-based spin locks can be reduced by polling (busy-wait code) with ordinary load operations to
avoid generating expensive stores to potentially shared locations (TATAS
algorithm). Rudolph and Segall first proposed an extension to ordinary
test\&set (this was the sole synchronization primitive available on numerous
early systems, such as the IBM 360 series) that performs a read of the lock
before attempting the actual atomic test operation [99]. A typical TATAS
algorithm is shown below.

\texttt{typedef unsigned long bool;}
\texttt{typedef volatile bool tatas\_lock;}

1: void tatas\_acquire(tatas\_lock *L)
2: {
3:   if (tas(L)) {
4:     do {
5:       if (*L)
6:         continue;
7:     } while (tas(L));
8:   }
9: }

10: void tatas\_release(tatas\_lock *L)
11: {
12:   *L = 0;
13: }

Furthermore, the burst of refill traffic whenever a lock is released can be
reduced by using the Ethernet-style exponential backoff algorithm in which,
after a failure to obtain the lock, a requester waits for successively longer periods of time before trying to issue another lock operation [5, 82]. The delay between tas attempts should not be too long; otherwise, processors might remain idle even when the lock becomes free. This is the idea behind the TATAS_EXP lock, and one typical implementation is shown below.

```c
typedef volatile unsigned long tatas_lock;

01 void tatas_exp_acquire(tatas_lock *L)
02  { if (tas(L)) tatas_exp_acquire_slowpath(L); }

03 void tatas_exp_acquire_slowpath(tatas_lock *L)
04  {
05    int b = BACKOFF_BASE, i;
06    do {
07      for (i = b; i; i--) ; // delay
08      b = min(b * BACKOFF_FACTOR, CAP);
09      if (*L)
10         continue;
11    } while (tas(L));
12  }

13 void tatas_exp_release(tatas_lock *L)
14  { *L = 0; }
```

In many implementations, acquire and release functions are in-lined, while the acquire_slowpath routine is linked to the binary code. Backoff parameters must be tuned by trial and error for each individual architecture. The storage cost for TATAS locks is low and does not increase with the number of processors.

3.2.3 Software Queuing Locks

Even with exponential backoff, TATAS locks still induce significant contention. Performance results using backoff with a real tas instruction on older machines can be found in the literature [41, 82]. Queue-based locks may eliminate these problems by letting each process spin on a different local memory location. The first proposal for a distributed, queue-based locking scheme was made by Goodman, Vernon, and Woest [38] in the late 1980s for the cache controllers of the Wisconsin Multicube. Several researchers have proposed locking primitives that incorporate both local spinning and queue-based locking in software [4, 41, 82].

A unique feature of software (or hardware) queuing locks found in many implementations is an explicit starvation avoidance and maximal fairness; in other words, first come, first served order of lock-acquire requests is guaranteed. In addition, queue-based locks provide reasonable latency in the absence of contention, provide good scalability for high-contended locks on many ar-
chitectures, and software implementations can easily be completely in-lined into the application code.

The acquire function of the software-based queue locks perform three basic phases: (1) a flag variable in a shared address space is initialized to the value BUSY; (2) the content at the lock location in memory is swapped with the address value pointing to the flag; (3) the thread spins until the prev_flag memory location, a pointer which was returned by the swap, contains the value FREE. The release function of the queue-based locks writes a FREE value to the flag location. Numerous variations of software queuing lock implementations are known [5, 26, 41, 80, 82, 110].

The locking primitive called MCS is one of the first software queue-based lock implementations, originally inspired by the QOSB [38] hardware primitive. The MCS lock was developed by Mellor-Crummey and Scott [82]. During the acquire request, the MCS lock inserts requesters for a held lock into a software queue using atomic operations such as swap and cas. Mellor-Crummey and Scott also describe another version of the MCS lock which only requires the swap operation. Fairness in that case is no longer guaranteed, and the implementation is slightly more complex.

Magnusson, Landin, and Hagersten proposed two software queue-based locking primitives about three years after MCS, namely LH and M [80]. Craig independently developed a lock identical to LH [26]. In this dissertation we will refer to this lock as the CLH lock. The CLH lock requires one fewer remote accesses to transfer a lock than does MCS, and will usually outperform MCS when high lock contention exists [80, 111]. The CLH lock achieves this behavior at the expense of increased latency to acquire an uncontested lock. The M lock achieves the more efficient lock transfer without increased uncontested lock access latency, at the expense of significant additional complexity in the lock algorithm.

3.2.4 Alternative Approaches

The fact that some synchronization algorithms perform well under low-contention periods and others under high-contention periods is the basic idea behind the reactive synchronization presented by Lim and Agarwal [76]. Reactive algorithms will dynamically switch among several software lock implementations. Typically, spin locks (TATAS_EXP) are used during the low-contention phase, and queue-based locks (MCS) are used during the high-contention phase [56]. Reactive algorithms demonstrate modest performance gains.

Goodman, Vernon, and Woest [38] have proposed very aggressive hardware support for locks. They introduced the queue-on-lock-bit primitive (QOLB, originally called QOSB). In this scheme, a distributed, linked list of nodes
waiting on a lock is maintained entirely in hardware (pointers in the processor caches are used to maintain the list of the waiting processors), and the releaser grants the lock to the first waiting node without affecting others. Effective colocation (allocation of the protected data in the same cache line as the lock) is possible; thus, this hardware scheme may reduce the lock hand-over time as well as the interference of lock traffic with data access and coherence traffic. The original QOLB proposal demonstrates good performance [56], but it requires complex protocol support, new instructions, and recompilation of applications. Another purely hardware-based mechanism called Implicit QOLB uses speculation and delays to transparently convert software locks to provide a hardware queued-lock behavior without requiring any software support or new instructions [97]. The load-linked/store-conditional instructions are used to demonstrate a possible implementation.

Stanford DASH uses directories to indicate which processors are spinning on the lock [71]. When the lock is released, one of the waiting nodes is chosen at random and is granted the lock. The grant request invalidates only that node’s caches and allows one processor in that node to acquire the lock with a local operation. This scheme lowers both the traffic and the latency involved in releasing a processor waiting on a lock. A time-out mechanism on the lock grant allows the grant to be sent to another node if the spinning process has been swapped out or migrated.

3.3 Key Ideas behind RH Lock

During the design phase of the RH lock, we paid attention to several general performance goals for locks, as given by Culler et al. [28]. We also paid attention to the data locality created by our lock algorithm; in other words, our additional goal was to maximize the node locality of NUCA architectures.

Queue-based locks implement a first come, first served fairness, which is less desirable on a NUCA machine because of the potentially huge percentage of the lock’s node handoffs. In other words, there is a risk that a contented lock might “jump” back and forth between the nodes, creating an enormous amount of traffic. The goal of the RH lock is to create a lock that minimizes the global traffic generated at lock handover and maximizes the node locality of NUCA architectures. To make this possible in our first proposal of a NUCA-aware lock, it is necessary to have the information about what thread is performing the acquire-release operation and in which node that thread is running.

The RH lock cuts down on lock traffic by making sure that only one thread per node (the “node winner”) tries to retrieve a lock that is currently not owned by a thread in the node. In addition, to increase the node locality of NUCAs, the RH lock is handing over the lock to another thread running in the same
node. We will later refer to this operation as marking the lock value with L_FREE tag. This not only cuts down on the lock-handover time, but also creates locality in the critical section work, since its data structures already reside in the node.

Even if the first come, first served policy may be a too strong requirement for a lock, it must guarantee some fairness and make sure that other nodes eventually get the lock even if there are always local requests for the lock.

3.4 The RH Lock Algorithm

The RH lock algorithm supports only two nodes. Every node contains a copy of the lock. The total lock storage for a 2-node case is thus $2 \times \text{sizeof(lock)}$. Initially, we could decide to logically place a lock in node 0 (mark the lock value as FREE in that node, meaning that both threads from the local node or from another node can acquire the lock).

The copy of the lock that is allocated and placed in node 1 is then marked with a REMOTE value, meaning that the “global” lock is in another node (node 0). At most one node may have its local copy of the lock in state FREE. Thus, the threads from node 1 would “see” a REMOTE tag if they try to acquire the local copy of the lock. The first thread that gets back the REMOTE value is the node winner and is allowed to continue to spin remotely with a larger backoff until the global lock is obtained. Other threads in the same node will spin locally on the local copy until the lock is fetched and released by the node winner.

Our NUCA-aware lock algorithm is shown in Figures 3.1 and 3.2. my_tid is the thread identification number (0, 1, 2, ..., maximum number of threads − 1), and my_node_id is the node number in which thread is placed (0 or 1). Both my_tid and my_node_id must be thread-private values, and preferably efficiently accessible from rh_acquire and rh_release functions. In our implementation, we reserve one of the Sparc’s thread-private global registers ($g2$) for that particular task which is the most efficient way to obtain my_tid and my_node_id. During the rh_acquire function, every thread will swap its own thread identification number into the node-local copy of the lock. If the lock is already in the node and is in the state L_FREE or FREE (lines 6–7 in the rh_acquire function), the acquire operation finishes and the thread can proceed with its critical section. If the lock value is in the REMOTE state (lines 8–11), the function rh_acquire_remote_lock is called. Otherwise, the lock is in the node and some other neighbor thread performs the critical task. In that case, the current thread calls the rh_acquire_slowpath function and spins locally until it succeeds with its own acquire operation. Of course, there is one rare special case when another node is lucky enough to obtain the
Figure 3.1: RH lock-acquire code.
1: void rh_release(rh_lock *L)
2: {
3:     if (be_fair)
4:         *L = FREE;
5:     else {
6:         if (cas(L, my_tid, FREE) != my_tid)
7:             *L = L_FREE;
8:     }
9: }

Figure 3.2: RH lock-release code.

lock before the current thread (line 19 in the rh_acquire_slowpath function). Once again, in that case, the function rh_acquire_remote_lock is called.

To achieve controlled unfairness we use a thread-private be_fair variable that initially is TRUE. The random function (line 6 in the rh_acquire_slowpath function) uses a nonlinear feedback random-number generator. It returns pseudo random numbers in the range from 0 to $2^{31} - 1$. If FAIR_FACTOR is equal to one, the RH lock will behave “as fairly as it can.” During the rh_release operation, the thread first checks if the be_fair variable is TRUE or not (line 3). If thread-private be_fair is TRUE, the lock will be released by writing the FREE value into the lock’s place. Otherwise, the lock can be released only to local/neighbor threads if interest was shown by them (line 7). Or, the lock can be released to the “world” by an atomic cas operation if no other from the same node showed any interest to acquire the same lock (line 6).

We use the following settings and definitions:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FAIR_FACTOR</td>
<td>1, 2, 100</td>
</tr>
<tr>
<td>BACKOFF_BASE</td>
<td>625</td>
</tr>
<tr>
<td>BACKOFF_FACTOR</td>
<td>2</td>
</tr>
<tr>
<td>BACKOFF_CAP</td>
<td>2,500</td>
</tr>
<tr>
<td>REMOTE_BACKOFF_BASE</td>
<td>2,500</td>
</tr>
<tr>
<td>REMOTE_BACKOFF_CAP</td>
<td>10,000</td>
</tr>
<tr>
<td>FREE</td>
<td>maximum number of threads</td>
</tr>
<tr>
<td>REMOTE</td>
<td>FREE + 1</td>
</tr>
<tr>
<td>L_FREE</td>
<td>FREE + 2</td>
</tr>
</tbody>
</table>
3.5 Hierarchical Backoff Locks

While the RH lock can create node affinity in NUCAs, it only works for two computing nodes. In addition, the RH implementation adds unwanted storage cost for locks; the information for locks scales with the number of nodes. This section introduces a new set of hierarchical backoff locks (HBO) that are scalable to many nodes and that have minimal memory overhead for lock storage.

3.5.1 The HBO Lock

The goal of the HBO lock is similar to that of the RH lock, which is that the algorithm should exploit communication locality and reduce global traffic for contended locks. In addition, in this algorithm, we pay attention to adding as little overhead as possible for uncontested locks. Ideally, at low contention or in the absence of contention the algorithm should not add any overhead, it should simply perform an atomic operation directly on the lock variable.

The core idea behind the HBO lock is simple: when a lock is acquired, the node_id of the thread/process is cas-ed into the lock location. In other words, if the lock-value is in the FREE state, it is atomically changed into the node_id, otherwise it remains the same. If a busy lock is held by someone in the same node, the cas will return the thread's own node_id, and the thread will start spinning with a small backoff constant (the same, for example, as the typical TATAS_EXP configuration). If the cas returns a different node_id, the thread will use a larger backoff constant. In this manner, a thread that is executing in the same node in which a lock has already been obtained will be more likely to subsequently acquire the lock when it is freed in relation to other contending threads executing in other nodes. Decreased migration of the lock (and the shared critical-section data structures) from node to node is obtained, and the overall performance is enhanced. This scheme can be expanded in a hierarchical way, using more than two sets of constants, for a hierarchical NUCA. Figure 3.3 shows code for the HBO lock. Emphasized lines are related to the HBO_GT lock (see next section) and should be ignored for the HBO proposal. Note that the "critical path" of the HBO lock (lines 6–9) does not add any significant overhead compared with the simple spin locks (assuming that the node_id information is easily accessible, e.g., it is stored in a thread-private register). That is important for the performance of the lock in the absence of contention.

As in the TATAS_EXP implementation, the acquire and release functions of HBO locks can be in-lined, while the acquire_slowpath routines are linked to the binary code. Backoff parameters must also be tuned by trial and error for each individual architecture.
typedef volatile unsigned long hbo_lock;

01 void hbo_acquire (hbo_lock *l)
02 {
03       unsigned long tmp;
04
05         while (l == is_spinning[my_node_id]) ; // spin
06         tmp = cas(l, FREE, my_node_id);
07        if (tmp == FREE)
08             return; // lock was free, and is now locked
09         hbo_acquire_slowpath(l, tmp);
10    }

11 void backoff(int *b, int cap)
12 {
13     int i;
14     for (; *b; i++; ) ; // delay
15         *b = min(*b * BACKOFF_FACTOR, cap);
16    }

17 void hbo_acquire_slowpath(hbo_lock *l,
18                           unsigned long tmp)
19 {
20     int b;
21     start:
22
23     if (tmp == my_node_id) { // local lock
24         b = BACKOFF_BASE;
25         while (1) {
26             backoff(&b, BACKOFF_CAP);
27             tmp = cas(l, FREE, my_node_id);
28             if (tmp == FREE)
29                 return;
30             if (tmp != my_node_id) {
31                 backoff(&b, BACKOFF_CAP);
32                 goto restart;
33             }
34         }  
35     }
36     else { // remote lock
37         b = REMOTE_BACKOFF_BASE;
38
39         is_spinning[my_node_id] = L;
40        while (1) {
41             backoff(&b, REMOTE_BACKOFF_CAP);
42             tmp = cas(l, FREE, my_node_id);
43             if (tmp == FREE) {  
44                 is_spinning[my_node_id] = dummy;
45                 return;
46             }  
47             if (tmp == my_node_id) {
48                 is_spinning[my_node_id] = dummy;
49                 goto restart;
50             }
51         }
52     }
53    }

Figure 5.3: Acquire and release code for HBO and HBO_GT locks. Emphasized lines are related to the HBO_GT implementation. Note that lines 54–63 are shown on the next page.
```c
restart:
while (L == is_spinning[my_node_id]) // spin
  tmp = cas(L, FREE, my_node_id);
  if (tmp == FREE)
    return;
goto start;
}

void hbo_release(hbo_lock *L)
{ *L = FREE; }
```

*Figure 3.3: (Cont.) Acquire and release code for HBO and HBO_GT locks. Emphasized lines are related to the HBO_GT implementation.*

### 3.5.2 The HBO_GT Lock

When multiple processors in the same node are executing in the slow spin loop (HBO lock, lines 37–52, Figure 3.3), the global coherence traffic through network is created by cas operations of each of the spinning processors. The purpose of the HBO_GT (global traffic throttling) lock is to limit the number of processors that are spinning in the same node and attempting to gain a lock currently owned by another node, thereby reduce global traffic on the network.

Before acquiring a lock, the thread reads the per-node memory location (not necessarily allocated in the local memory) `is_spinning`, compares its content with the lock address, and keeps spinning for as long they are equal (line 5 and 56, Figure 3.3). Then, the thread performs the atomic cas operation (line 6 and 57). If the cas returns a node_id different from the thread's own id, the thread will store the lock address L in the node's `is_spinning` and start to spin for the lock with a fairly large backoff constant. This operation may thus prevent others in the same node from performing lock-acquisition transactions to the lock address (the cas operations) that might otherwise create global coherence traffic on the network. As soon as the thread has acquired the lock, it writes the "dummy value" to the node's `is_spinning`, which allows any waiting neighbor to start spinning. By using this algorithm, there is usually only one thread per node (or a small number of threads) that is performing remote spinning.

### 3.5.3 The HBO_GT_SD Lock

Many of the queue-based locks guarantee starvation freedom. Even though starvation is usually unlikely [28], with multiple threads competing for a lock, it is possible that some threads may be granted the lock repeatedly while others may not and may become starved. Since both HBO and HBO_GT (and all other simple spin-locking algorithms) are vulnerable to starvation, we need a solution that at least lowers the risk of potential starvation, which is es-
43 if (tmp == FREE) {
44    // release the threads from our node
45    is_spinning[my_node_id] = DUMMY;
46    // release the threads from stopped nodes, if any
47    if (stopped_nodes > 0)
48        is_spinning[for each stopped_node] = DUMMY;
49    return;
50 }
51 if (tmp == my_node_id) {
52    is_spinning[my_node_id] = DUMMY;
53    if (stopped_nodes > 0)
54        is_spinning[for each stopped_node] = DUMMY;
55    goto restart;
56 }
57 if (tmp != my_node_id) {
58    // lock is still in some remote node
59    get_angry++;
60    if (get_angry == GET_ANGRY_LIMIT) {
61        stopped_node_id[stopped_nodes++] = tmp;
62        is_spinning[tmp] = 1;
63    }
64 }

Figure 3.4: Part of the HBO_GT_SD lock’s acquire function. Lines 43–50 from the HBO_GT algorithm are replaced with the code in this figure.

Especially important for HBO locks because of their “nonuniform” nature in the locking algorithm. For example, a counter can be maintained of the number of times a lock-request has been denied and, after a certain threshold, a thread’s priority may be increased (a thread can start spinning without any backoff until the lock is obtained). In addition to this simple “thread-centric” solution, HBO_GT_SD provides a “node-centric” mechanism, described in further detail below, which lowers the risk of node starvation. Explicit thread-centric starvation avoidance can be implemented with alternative approaches (reactive synchronization [76], see section 3.2), at the expense of additional complexity in the lock algorithm.

The HBO_GT_SD lock is based on the HBO_GT proposal. The idea behind the node-centric algorithm, which lowers the risk of node starvation, is the following: after a “winning” thread (or a small number of threads), has tried several times to acquire a remote lock owned by another node, it gets “angry.” An angry node will take two measures to get the lock more quickly: (1) it will spin more frequently, and (2) it will set the is_spinning for the other nodes to the lock address and thus prevent more threads in those nodes from trying to acquire the lock.

The details of this algorithm are shown in Figure 3.4 (lines 43–50 from Figure 3.3 are replaced with the code from Figure 3.4). Note that the initialization
of variables `get_angry` and `stopped_nodes` is excluded from the code example.

3.6 Performance Evaluation

Most of the experiments in this section are performed on a Sun Enterprise E6000 server. The hardware DSM numbers have been measured on a 2-node Sun WildFire built from two E6000 nodes.\(^4\)

We have implemented the traditional TATAS lock and the RH lock using the `tas`, `swap`, and `cas` operations available in the SPARC-V9 instruction set. All HBO locks are implemented with only a `cas` operation. The code for TATAS\(_{EXP}\), CLH, and MCS lock is written by Scott and Scherer [111], and the entire implementation framework is compiled with GNU’s `gcc-3.2`, optimization level 3. Note that some of our early studies presented in section 3.6.1 and section 3.6.2 are obtained with GNU’s `gcc-3.0.4`, optimization level 1. The TATAS\(_{EXP}\) lock was previously tuned for a Sun Enterprise E6000 machine by Scott and Scherer [111]. We use identical values in our experiments. By using the GCC’s `static inline` construct, we explicitly in-line TATAS, CLH, and MCS locks. All other locks have a “slowpath” routine called from the corresponding in-line part of the acquire function. Release functions for all locks are in-lined. Machines used for tests were otherwise unloaded.

3.6.1 Uncontested Performance

One important design goal for locks is a low latency acquisition of a free lock [28]. In other words, if a lock is free and no other processors are trying to acquire it at the same time, the processor should be able to acquire it as quickly as possible. This is especially important for applications with little or no contention for the locks, which fortunately is a quite common case.

In this section we obtain an estimate of lock overhead in the absence of contention for three common scenarios. We evaluate the cost of the acquire-release operation (1) if the same processor as the previous owner is the owner of the lock (lock is in its cache); (2) if the lock is in the same node but the previous owner is not the current processor (lock is in the neighbor’s cache); and (3) if the lock was owned by a remote node (lock is in the cache of a processor that is in another node). The pseudocode for this NUCA-aware microbenchmark is shown in Figure 3.5.

\(^4\)Currently, our system has 30 processors, 16 plus 14, and therefore we perform our experiments mainly on a 14 plus 14 configuration.
1: acquire and release all locks;
2: BARRIER

// case 1: previous owner: same processor
3: if (my_tid == 0) {
4:   acquire and release all locks;
5:   start timer;
6:   acquire and release all locks;
7:   stop timer;
8: }
9: BARRIER

// case 2: previous owner: same node
10: if (my_tid == 1) {
11:   start timer;
12:   acquire and release all locks;
13:   stop timer;
14: }
15: BARRIER

// case 3: previous owner: remote node
16: if (my_tid == 2) {
17:   start timer;
18:   acquire and release all locks;
19:   stop timer;
20: }

Figure 3.5: NUCA-aware microbenchmark.

For this experiment we need three threads. Threads with thread identification numbers (tid) zero and one are executing inside cabinet 1 and thread two is running inside cabinet 2. The total number of allocated locks is 2400. The first two lines of the microbenchmark are used to warm the TLBs and are executed by all threads. Results are presented in Table 3.1.

Unsurprisingly, TATAS and TATAS_EXP are the fastest acquire-release operations for this simple test without contention. We observe that our low latency design goal for the RH lock is within the reasonable magnitudes for the same processor and the same node case. For the remote node case RH lock performs poorly compared to other locks. The reason for this is twofold: (1) the rh_acquire function executed by the third thread will always acquire the local copy of the lock before the “global” lock that is in another cabinet (the rh_acquire_remote_lock function is always called), and (2) the rh_acquire function generates some additional remote coherence traffic at line 5. However, we observe that our low latency design goal for the HBO locks is fulfilled, and performance is almost identical with the simplest locks: TATAS and TATAS_EXP.
<table>
<thead>
<tr>
<th>Lock Type</th>
<th>Same Processor</th>
<th>Same Node</th>
<th>Remote Node</th>
</tr>
</thead>
<tbody>
<tr>
<td>TATAS</td>
<td>150 ns</td>
<td>660 ns</td>
<td>2050 ns</td>
</tr>
<tr>
<td>TATAS_EXP</td>
<td>143 ns</td>
<td>613 ns</td>
<td>2070 ns</td>
</tr>
<tr>
<td>MCS</td>
<td>210 ns</td>
<td>732 ns</td>
<td>2120 ns</td>
</tr>
<tr>
<td>CLH</td>
<td>234 ns</td>
<td>805 ns</td>
<td>2530 ns</td>
</tr>
<tr>
<td>RH</td>
<td>198 ns</td>
<td>672 ns</td>
<td>4480 ns</td>
</tr>
<tr>
<td>HBO</td>
<td>152 ns</td>
<td>652 ns</td>
<td>2010 ns</td>
</tr>
<tr>
<td>HBO_GTI</td>
<td>152 ns</td>
<td>643 ns</td>
<td>2010 ns</td>
</tr>
<tr>
<td>HBO_GTI_SD</td>
<td>140 ns</td>
<td>638 ns</td>
<td>2010 ns</td>
</tr>
</tbody>
</table>

Table 3.1: Uncontested performance for a single acquire-release operation.

3.6.2 Traditional Microbenchmark

The traditional microbenchmark that is used by many researchers consists of a tight loop containing a single acquire-release operation. The iteration time for a single-node Sun Enterprise E6000 is shown in Figure 3.6.

![Figure 3.6: Traditional microbenchmark iteration time for a single-node Sun Enterprise E6000 (empty critical section).](image)

The number of iterations performed by every thread in this microbench-
mark is 10,000. The FAIR_FACTOR for the RH lock is equal to one. From
the Figure 3.6 it appears that TATAS_EXP and RH performs much better than
the other locks. Both these locks have an exponential backoff, why waiting
threads will “sample” the lock variable less often. This increases the prob-
ability that the thread releasing a lock will manage to acquire it again right
away. The iteration time for these two locks indeed looks similar to that of
their uncontested performance if the previous owner is the same processor
(see Table 3.1). TATAS has no backoff for the waiting threads, which is why a
handoff to the same processor is less likely to occur, especially at high proc-
sessor counts. Queue-based locks on the other hand will rarely allow the releasing
processor to acquire the lock again directly.

To overcome this problem we altered the microbenchmark to initialize a
global variable last_owner inside the critical section, and force the thread
to observe a new owner before it is allowed to compete for the lock again. A
single remaining thread will be excluded from this requirement in order to run
until completion. Slightly modified traditional microbenchmark is shown in
Figure 3.7.

```c
shared int iterations, total_threads;
shared volatile int last_owner = -1;
shared volatile int total_finished = 0;

1: for (i = 0; i < iterations; i++) {
2:    ACQUIRE(L);
3:    if (my_tid != last_owner) last_owner = my_tid;
4:    // some more statistics goes here
5:    RELEASE(L);
6:    while ((last_owner == my_tid) &&
7:           (total_finished < total_threads - 1))
8:       ; // spin
9: }
10: atomic_increase(total_finished);
```

Figure 3.7: Slightly modified traditional microbenchmark.

Figure 3.8 shows the result from the modified microbenchmark (number of
iterations is still equal to 10,000 and RH’s FAIR_FACTOR is one). The iteration
time here is an offset by the time it takes to perform the extra work in the
critical section. The queue-based CLH and MCS performs the best. This is be-
bcause only their third phase of the lock-acquire function (see section 3.2.3) is
performed at lock-handover time. The releasing thread will perform an single
store upgrade cache miss to its flag (its cache already contains a valid copy
in shared state) and the next thread will need a single load cache miss for its
prev_flag, i.e., the same memory location as the releasing thread’s flag.
For all the other locks the releasing thread will need to perform a store miss to
Figure 3.8: Slightly modified traditional microbenchmark iteration time for a single-node Sun Enterprise 6000.

$L$ (its cache does not contain a valid copy) and the next thread will perform a load cache miss and a store upgrade miss to $L$. TATAS is the only lock showing a distinct degradation caused by increased traffic as more processors are added.

The iteration time for the modified benchmark on a 2-node Sun WildFire is shown in Figure 3.9 ("Iteration Time" diagram). In this study, we use round-robin scheduling for thread binding to different cabinets. The RH lock outperforms all other tested locks for all runs with more than two threads, and is comparable to other locks for one (no contention) and two threads.

Figure 3.9 ("Node Handoffs" diagram) shows the ratio of node handoffs for each lock type, reflecting how likely it is for a lock to migrate between nodes each time it is acquired. We ignore the TATAS values for more than 16 processors. The graph clearly shows the key advantage of the RH lock. The RH lock consistently shows low node handoff numbers for all the three settings of FAIR_FACTOR.

The simple spin locks also show a node handoff ratio below 50 percent, which could be expected since local processors can acquire a released lock much faster than remote processors. The queue locks are expected to show a node handoff ratio equal to $(N/2)/(N-1)$, since $N/2$ of the processors reside in the other node and we do not allow the same processor to acquire the lock.
Figure 3.9: Iteration time and node handoffs study for a slightly modified traditional microbenchmark for a 2-node WildFire system. GNU's gcc-3.0.4 compiler, optimization level 1.
twice in a row. However, the queue-based locks often show unnatural behavior with large variation in the node handoff ratio. Our only explanation for this is pure lock. At 12 and 22 processors, the CLH shows a low handover ratio. This also explains the varied performance in Figure 3.9, for example the good CLH performance at 12 and 22 processors. At 8–10 processors, the node handoff ratio is fairly normal for both queue-based locks. The “Iteration Time” diagram of Figure 3.9 shows that a critical section guarded by the RH lock takes less than halft the time to execute compared with the same critical section guarded by any other lock.

It appears that the simplistic regular microbenchmark that we, as well as most other lock studies, use sometimes makes processors in the same node more likely to queue up after each other making the lock ratio substantially lower than expected. We also suspect that RH's job of creating locality is greatly simplified by this highly regular benchmark.

Figure 3.10 shows the performance of HBO locks. The FAIR_FACTOR for the RH lock is equal to one. Note that this study uses GNU's gcc-3.2 compiler. As in Figure 3.9, we can observe that a critical section guarded by NUCA-aware locks (especially by RH, HBO_GT, and HBO_GT_SD) takes about half the time to execute compared with the same critical section guarded by any other software-based lock.

3.6.3 New Microbenchmark

No real applications have a fixed number of processors pounding on a lock. Instead, they have a fixed number of processors spending most of their time on noncritical work, including accesses to uncontested locks. They rarely enter the “hot” critical section. The degree of contention is affected by the ratio of noncritical work to critical work. The unnatural node-handoff behavior of the traditional lock microbenchmark led us to a new lock benchmark, which we feel better reflects the expected behavior of a real application. In the new microbenchmark, the number of processors is kept constant. Each thread performs some amount of noncritical work, which consist of one static delay and one random delay of similar sizes, between attempts to acquire the lock. Initially, the length of the noncritical work is chosen such that there is little contention for the critical section and all lock algorithms perform the same. More contention is modeled by increasing the number of elements of a shared vector that are modified before the lock is released. The pseudocode of the new benchmark is shown in Figure 3.11.

Figure 3.12 (“Iteration Time” diagram) shows that the two queue-based locks perform almost identically for the new benchmark. Figure 3.12 (“Node Handoffs” diagram) shows also their node handover to be close to the expected values: \((N/2)/(N - 1)\). The simple spin locks still perform in an unpre-
Figure 3.10: Slightly modified traditional microbenchmark on a 2-node Sun WildFire system. GNU’s gcc-3.2 compiler, optimization level 3.

dictable way. This is clearly correlated to their unpredictable node handover. (TATAS values are measured for a critical_work of 0–1300 because its
shared int cs_work[MAX_CRITICAL_WORK];
shared int iterations;

for (i = 0; i < iterations; i++) {
    ACQUIRE(L);
    int j;
    for (j = 0; j < critical_work; j++)
        cs_work[j]++;
    RELEASE(L);
    int non_cs_work[MAX_PRIVATE_WORK];
    int j, random_delay;
    for (j = 0; j < private_work; j++)
        non_cs_work[j]++;
    random_delay = random() % private_work;
    for (j = 0; j < random_delay; j++)
        non_cs_work[j]++;
}

Figure 3.11: New microbenchmark.

performance is poor for higher levels of contention.) The NUCA-aware locks perform better the more contention there is, which can be explained by its decreasing amount of node hand-over. This is exactly the behavior we want from a lock: the more contention there is, the better it should perform.

In Table 3.2 we present the amount of traffic generated by our new microbenchmark. The numbers are normalized to the TATAS_EXP which generates 15.1 million local and 8.9 million global transactions. Once again, software queuing locks perform almost identically. We can also observe that NUCA-aware locks generate less than half the amount of global transactions compared with any other software-based locks on this NUCA machine. The global traffic is reduced by a factor of 15 compared to the traditional TATAS locks. The table also shows that the NUCA locks do produce a slightly higher amount of local traffic compared with the queue-based locks.

3.6.4 Application Performance

In this section, we evaluate the effectiveness of our new locking mechanisms using explicitly parallel programs from the SPLASH-2 suite [124]. Table 3.3 shows SPLASH-2 applications with the corresponding problem sizes and lock statistics (Total Locks is the number of allocated locks, and Lock Calls is the total number of acquire-release lock operations during the execution).

Problem size is a very important issue in this context. Generally, the larger
Figure 3.12: New microbenchmark on a 2-node Sun WildFire system, 28-processor runs.

the problem size, the lower the frequency of synchronization relative to computation. On the one hand, using large problem sizes will make synchroniza-
<table>
<thead>
<tr>
<th>Lock Type</th>
<th>Local Transactions</th>
<th>Global Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>TATAS</td>
<td>4.41</td>
<td>4.70</td>
</tr>
<tr>
<td>TATAS_EXP</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>MCS</td>
<td>0.53</td>
<td>0.65</td>
</tr>
<tr>
<td>CLH</td>
<td>0.54</td>
<td>0.63</td>
</tr>
<tr>
<td>RH</td>
<td>0.54</td>
<td>0.28</td>
</tr>
<tr>
<td>HBO</td>
<td>0.60</td>
<td>0.30</td>
</tr>
<tr>
<td>HBO_GT</td>
<td>0.60</td>
<td>0.30</td>
</tr>
<tr>
<td>HBO_GT_SD</td>
<td>0.61</td>
<td>0.29</td>
</tr>
</tbody>
</table>

Table 3.2: Normalized local and global traffic generated for the new microbenchmark (critical_work = 1500, 28 processors).

<table>
<thead>
<tr>
<th>Program</th>
<th>Problem Size</th>
<th>Total Locks</th>
<th>Lock Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>2M particles</td>
<td>130</td>
<td>69,193</td>
</tr>
<tr>
<td>Cholesky</td>
<td>128,000</td>
<td>67</td>
<td>74,284</td>
</tr>
<tr>
<td>FFT</td>
<td>1M points</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>FMM</td>
<td>32k particles</td>
<td>2,052</td>
<td>80,528</td>
</tr>
<tr>
<td>LU-c</td>
<td>1024 x 1024 matrices, 16 x 16 blocks</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>LU-nn</td>
<td>1024 x 1024 matrices, 16 x 16 blocks</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Ocean-c</td>
<td>512 x 514</td>
<td>6</td>
<td>6,304</td>
</tr>
<tr>
<td>Ocean-nn</td>
<td>256 x 256</td>
<td>6</td>
<td>6,656</td>
</tr>
<tr>
<td>Radiosity</td>
<td>room, -e 5000.0 -e 0.0, -e 0.10</td>
<td>3,975</td>
<td>295,627</td>
</tr>
<tr>
<td>Radix</td>
<td>4M integers, radix 1024</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>Raytrace</td>
<td>car</td>
<td>26</td>
<td>366,450</td>
</tr>
<tr>
<td>Volrend</td>
<td>head</td>
<td>67</td>
<td>38,456</td>
</tr>
<tr>
<td>Water-Nsq</td>
<td>2,197 molecules</td>
<td>2,206</td>
<td>112,415</td>
</tr>
<tr>
<td>Water-Sp</td>
<td>2,197 molecules</td>
<td>222</td>
<td>510</td>
</tr>
</tbody>
</table>

Table 3.3: The SPLASH-2 programs. Only emphasized programs are studied further. Lock statistics are obtained for 32-processor runs.

function operations seem less important. On the other hand, small problem sizes might lead to very low speedup of the application, rendering it uninteresting on a machine of this scale, even though we chose the fairly standard problem sizes found in many other related investigations. We also chose to further examine only applications with more than 10,000 lock calls, as in the cases of Barnes, Cholesky, FMM, Radiosity, Raytrace, Volrend, and Water-Nsq. For each application, we vary the synchronization algorithm used and measure the execution time on a 2-node Sun WildFire machine. Programs are compiled
with GNU's gcc-3.0.4, optimization level -O1. (With this particular compiler, this is the highest level of optimization that does not break the correctness of execution for all applications and lock implementations.)

<table>
<thead>
<tr>
<th>Program</th>
<th>TATAS</th>
<th>TATAS_EXP</th>
<th>MCS</th>
<th>CLH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>1.54 (0.05)</td>
<td>1.43 (0.01)</td>
<td>1.83 (0.15)</td>
<td>1.54 (0.10)</td>
</tr>
<tr>
<td>Cholesky</td>
<td>2.31 (0.07)</td>
<td>2.04 (0.04)</td>
<td>2.09 (0.03)</td>
<td>2.25 (0.11)</td>
</tr>
<tr>
<td>FMM</td>
<td>4.24 (0.33)</td>
<td>4.19 (0.19)</td>
<td>4.33 (0.06)</td>
<td>4.46 (0.07)</td>
</tr>
<tr>
<td>Radiosity</td>
<td>1.66 (0.06)</td>
<td>1.75 (0.07)</td>
<td>1.75 (0.07)</td>
<td>1.75 (0.07)</td>
</tr>
<tr>
<td>Raytrace</td>
<td>2.00 (0.91)</td>
<td>1.71 (0.18)</td>
<td>1.41 (0.28)</td>
<td>1.38 (0.32)</td>
</tr>
<tr>
<td>Volrend</td>
<td>1.70 (0.03)</td>
<td>1.57 (0.10)</td>
<td>1.48 (0.28)</td>
<td>1.75 (0.16)</td>
</tr>
<tr>
<td>Water-Nsq</td>
<td>2.37 (0.03)</td>
<td>2.25 (0.06)</td>
<td>2.20 (0.04)</td>
<td>2.45 (0.03)</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>2.47 (0.21)</td>
<td>2.13 (0.09)</td>
<td>2.22 (0.14)</td>
<td>2.31 (0.13)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>RH</th>
<th>HBO</th>
<th>HBO_GT</th>
<th>HBO_GT_SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>1.54 (0.14)</td>
<td>1.50 (0.04)</td>
<td>1.69 (0.06)</td>
<td>1.44 (0.10)</td>
</tr>
<tr>
<td>Cholesky</td>
<td>2.23 (0.06)</td>
<td>2.06 (0.09)</td>
<td>2.34 (0.03)</td>
<td>2.13 (0.11)</td>
</tr>
<tr>
<td>FMM</td>
<td>4.27 (0.13)</td>
<td>4.37 (0.09)</td>
<td>4.59 (0.27)</td>
<td>4.27 (0.09)</td>
</tr>
<tr>
<td>Radiosity</td>
<td>1.44 (0.07)</td>
<td>1.45 (0.09)</td>
<td>1.68 (0.04)</td>
<td>1.51 (0.03)</td>
</tr>
<tr>
<td>Raytrace</td>
<td>0.62 (0.01)</td>
<td>0.77 (0.01)</td>
<td>0.70 (0.01)</td>
<td>0.72 (0.01)</td>
</tr>
<tr>
<td>Volrend</td>
<td>1.61 (0.09)</td>
<td>1.68 (0.14)</td>
<td>1.33 (0.10)</td>
<td>1.24 (0.03)</td>
</tr>
<tr>
<td>Water-Nsq</td>
<td>2.21 (0.01)</td>
<td>2.14 (0.03)</td>
<td>2.09 (0.02)</td>
<td>2.14 (0.01)</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td>1.99 (0.07)</td>
<td>2.00 (0.07)</td>
<td>2.06 (0.08)</td>
<td>2.02 (0.05)</td>
</tr>
</tbody>
</table>

Table 3.4: Application performance for eight locking algorithms for 28-processor runs, 14 threads per WildFire node. Execution time is given in seconds and the variance is shown in parentheses.

Table 3.4 presents the execution times in seconds for 28-processor runs for all eight studied locking schemes.\(^2\) Variance is given in parentheses in the same table. Normalized speedup for TATAS, TATAS_EXP, MCS, CLH, and HBO_GT_SD is shown in Figure 3.13. For Barnes, the MCS lock is much worse than the ordinary TATAS_EXP, and for Volrend and Water-Nsq that is also the case for the CLH lock. As expected from our new microbenchmark study, on average, queue-based locks perform about the same as the TATAS with exponential backoff.

We chose to further investigate only Raytrace. This application renders a 3D scene using ray tracing and is one of the most unpredictable SPLASH-2 programs [28]. In this application, locks are used to protect task queues; locks

\(^2\)An unmodified version of Radiosity will not execute correctly with software queuing locks on any optimization level. We did not investigate this any further.
Figure 3.13: Normalized speedup for 28-processor runs on a 2-node Sun WildFire.

are also used for some global variables that track statistics for the program. A large amount of work is usually done between synchronization points. Execution time in seconds for all eight synchronization algorithms for single-, 28-, and 30-processor runs is shown in Table 3.5.

NUCA-aware locks demonstrate very low measurement variance for both 28- and 30-processor runs. In the same table, we also demonstrate that MCS and CLH locks are practically unusable for 30-processor runs. They are extremely sensitive for small disturbances produced by the operating system itself. Traditionally, original software queuing locks exhibit poor behavior in the presence of multiprogramming, because a process near the end of the queue, in addition to having to wait for any process that is preempted during its critical section, must also wait for any preempted process ahead of it in the queue. This unwanted behavior of the queue-based locks has been studied further by Scott on the same architecture and on the Sun Enterprise 10000 multiprocessor [110, 111].

Speedup for Raytrace is shown in Figure 3.14. There is a significant decrease in performance for all other locks above 12 processors, while the NUCA-aware locks continue to moderately scale all the way up to 28 processors. We also present the normalized traffic numbers for all synchronization algorithms in Table 3.6.
<table>
<thead>
<tr>
<th>Lock Type</th>
<th>1 Processor</th>
<th>28 Processors</th>
<th>30 Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>TATAS</td>
<td>5.02</td>
<td>2.90 (0.91)</td>
<td>2.70 (0.45)</td>
</tr>
<tr>
<td>TATAS_EXP</td>
<td>5.26</td>
<td>1.71 (0.18)</td>
<td>2.05 (0.26)</td>
</tr>
<tr>
<td>MCS</td>
<td>5.05</td>
<td>1.41 (0.28)</td>
<td>&gt; 200 s</td>
</tr>
<tr>
<td>CLH</td>
<td>5.30</td>
<td>1.38 (0.32)</td>
<td>&gt; 200 s</td>
</tr>
<tr>
<td>RH</td>
<td>5.08</td>
<td>0.62 (0.01)</td>
<td>0.68 (0.00)</td>
</tr>
<tr>
<td>HBO</td>
<td>5.00</td>
<td>0.77 (0.01)</td>
<td>0.78 (0.01)</td>
</tr>
<tr>
<td>HBO_GT</td>
<td>5.02</td>
<td>0.70 (0.01)</td>
<td>0.75 (0.00)</td>
</tr>
<tr>
<td>HBO_GT_SD</td>
<td>5.02</td>
<td>0.72 (0.01)</td>
<td>0.80 (0.02)</td>
</tr>
</tbody>
</table>

Table 3.5: Raytrace performance. Execution time is given in seconds and the variance is presented in parentheses.

![Graph showing speedup for Raytrace.](image)

Figure 3.14: Speedup for Raytrace.

### 3.7 Fairness and Sensitivity

The task of the lock-unlock synchronization primitives is to create a serialization schedule for each critical region such that simultaneous attempt to enter the region will be ordered in some serial way. Any serial order will result in a correct execution, as long as starvation is avoided. Fairness is often considered a desirable property, since it can create an even distribution of work between
<table>
<thead>
<tr>
<th>Program</th>
<th>TAPAS</th>
<th>TAPAS_EXP</th>
<th>MCS</th>
<th>CLH</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>1.01 / 0.67</td>
<td>(2.5) 1.00 / 1.00 (1.8)</td>
<td>1.01 / 0.66</td>
<td>1.14 / 0.78</td>
</tr>
<tr>
<td>Cholesky</td>
<td>0.99 / 1.00</td>
<td>(14.3) 1.00 / 1.00 (4.6)</td>
<td>0.96 / 0.87</td>
<td>0.97 / 0.90</td>
</tr>
<tr>
<td>FMM</td>
<td>1.09 / 1.17</td>
<td>(6.8) 1.00 / 1.00 (3.2)</td>
<td>0.99 / 0.83</td>
<td>0.97 / 0.80</td>
</tr>
<tr>
<td>Radiosity</td>
<td>1.06 / 1.08</td>
<td>(4.1) 1.00 / 1.00 (1.9)</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Raytrace</td>
<td>1.15 / 1.24</td>
<td>(7.8) 1.00 / 1.00 (2.9)</td>
<td>0.91 / 0.84</td>
<td>1.04 / 0.78</td>
</tr>
<tr>
<td>Volrend</td>
<td>1.02 / 1.07</td>
<td>(5.2) 1.00 / 1.00 (1.3)</td>
<td>1.02 / 1.05</td>
<td>1.04 / 1.17</td>
</tr>
<tr>
<td>Water-Nsq</td>
<td>1.01 / 1.03</td>
<td>(2.8) 1.00 / 1.00 (1.2)</td>
<td>1.00 / 1.04</td>
<td>1.07 / 1.10</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>1.05 / 1.04</strong></td>
<td><strong>1.00 / 1.00</strong></td>
<td><strong>0.98 / 0.88</strong></td>
<td><strong>1.04 / 0.92</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Program</th>
<th>RH</th>
<th>HBO</th>
<th>HBO_GT</th>
<th>HBO_GT_SD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>1.02 / 0.60</td>
<td>0.92 / 0.61</td>
<td>0.92 / 0.62</td>
<td>0.97 / 0.62</td>
</tr>
<tr>
<td>Cholesky</td>
<td>0.95 / 0.87</td>
<td>0.96 / 0.90</td>
<td>0.96 / 0.90</td>
<td>0.97 / 0.91</td>
</tr>
<tr>
<td>FMM</td>
<td>1.00 / 0.83</td>
<td>0.96 / 0.84</td>
<td>0.99 / 0.89</td>
<td>1.03 / 0.98</td>
</tr>
<tr>
<td>Radiosity</td>
<td>1.00 / 0.85</td>
<td>1.01 / 0.89</td>
<td>0.92 / 0.82</td>
<td>0.99 / 0.98</td>
</tr>
<tr>
<td>Raytrace</td>
<td>0.89 / 0.89</td>
<td>0.83 / 0.89</td>
<td>0.82 / 0.89</td>
<td>0.81 / 0.84</td>
</tr>
<tr>
<td>Volrend</td>
<td>1.01 / 1.03</td>
<td>1.01 / 0.87</td>
<td>1.02 / 0.87</td>
<td>1.01 / 0.86</td>
</tr>
<tr>
<td>Water-Nsq</td>
<td>1.03 / 1.02</td>
<td>0.98 / 0.97</td>
<td>0.96 / 0.98</td>
<td>0.99 / 0.88</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>0.98 / 0.81</strong></td>
<td><strong>0.95 / 0.81</strong></td>
<td><strong>0.94 / 0.81</strong></td>
<td><strong>0.97 / 0.85</strong></td>
</tr>
</tbody>
</table>

Table 3.6: Normalized traffic (local/global) for all locking algorithms for 28-processor runs. Number of transactions in millions for TAPAS_EXP is shown in parentheses.

Threads. Without fairness, the threads may arrive unevenly at a barrier, even though they have performed the same amount of work. This will force the early arriving threads to wait for the last arriving thread while performing no useful work. However, the importance of fairness must be traded off with its impact on performance [28].

It is not clear that fairness always results in the fastest execution. Assume that all threads are expected to enter the same critical region exactly once between two barriers. All threads arrive roughly at the same time to the critical section. Here, the serialization scheme with the shortest time between two contenders entering the critical section would be preferable over the fairness scheme that strictly scheduled the threads according to their arrival time.

The queue-based locks implement a first come, first served order between simultaneous attempts to enter a critical region and guarantee both fairness and starvation avoidance. The TAPAS locks rely on the coherence implementation for its serialization, and cannot make such guarantees. They are dependent on such guarantees made by the underlying coherence mechanism.

A very simple fairness study is performed with the new microbenchmark
by measuring the finish times of all individual threads in the benchmark. The results for all lock algorithms are shown in Figure 3.15. As expected, we can see that the queue-based locks are the fairest locks in this experiment (produce the most vertical line); the percentage difference in completion time between the first and the last processor is only 2.1 percent. TATAS_EXP seems to be most "unfair" lock for this benchmark with the same percentage difference of 28.9 percent. NUCA-aware locks perform about the same; for example, the percentage difference for HBO_GT_SD is 5.6 percent.

The sensitivity of the HBO_GT_SD algorithm is studied by running the new microbenchmark algorithm on a 2-node Sun WildFire machine. Results for 26-processor runs are shown in Figure 3.16.

Figure 3.16 (a) shows results from experiments where we have varied the REMOTE_BACKOFF_CAP parameter (see Figure 3.3), and Figure 3.16 (b) shows results from experiments with the GET_ANGRY_LIMIT parameter (see Figure 3.4). All results are normalized against "optimal" REMOTE_BACKOFF_CAP and GET_ANGRY_LIMIT values for our experimentation hardware, namely 10,000 and 100. The MCS and the HBO_GT algorithms are used for comparison.

It is interesting to note that small values of a REMOTE_BACKOFF_CAP parameter increase node handoffs, while high values (for example, 160,000)
decrease handoffs significantly, i.e., create more node locality in the locking algorithm. However, this extra locality comes at a cost of an increased execution time. Similarly, small values of a GET_ANGRY_LIMIT parameter increase both iteration time and node handoffs since the lock will start to "move" between the nodes at a very high rate.
Figure 3.16: Sensitivity study of the HBO_GT_SD algorithm with a new microbenchmark algorithm. Iteration time and node handoff values are both normalized, and MCS and HBO_GT are used for comparison.
4. The DSZOOM System

The DSZOOM system is fine-grained and software-based distributed shared memory implementation. The all-software protocol is implemented assuming some basic low-level primitives in the cluster interconnect and an operating system bypass functionality, similar to the InfiniBand standard. All interrupt- and/or poll-based asynchronous protocol processing is completely removed by running the entire coherence protocol in the requesting processor. This not only removes the asynchronous overhead, but also makes use of a processor that otherwise would stall. The technique is applicable to both page-based and fine-grain software-based shared memory.

4.1 Introduction

Clusters of multiprocessors provide a powerful platform for executing parallel applications. To allow for shared-memory applications to run on such clusters, software distributed shared-memory (SW-DSM) systems support the illusion of shared memory across the cluster via a software run-time layer between the application and the hardware. This approach can potentially provide a cost-effective alternative to hardware shared-memory systems for executing certain classes of workloads. SW-DSM technology can also be used to connect several large hardware distributed shared-memory (HW-DSM) systems and thereby extend their upper scalability limit.

Most SW-DSM systems keep coherence between page-sized coherence units [73, 18, 59, 114, 116]. The normal per-page access privilege of the memory-management unit offers a cheap access control mechanism for these SW-DSM systems. The large page-size coherence units in the earlier SW-DSM systems created extra false sharing and caused frequent page transfers of large pages between nodes. In order to avoid most of the false sharing, weaker memory models have been used to allow many update actions to be lumped to a specific point in time, such as the lazy release consistency (LRC) protocol [58].

Fine-grain SW-DSM systems with a more traditional cache-line-sized coherence unit have also been implemented. Here, the access control check is either done by altering the error-correcting codes (ECC) [108] or by in-line code snippets (small fragments of machine code) [108, 105]. The small cache-line
size reduces the false sharing for these systems, but the explicit access-control check adds extra latency for each load or store operation to global data. The most efficient access check reported to date is three extra instructions adding three extra cycles for each load to global data [107].

The coherence protocol of SW-DSM systems is often implemented as communicating software agents running in the different nodes sending requests and replies to each other. Each agent is responsible for accessing its local memory and for keeping a directory structure for “its part” of the shared address space. The agent where the directory structure for a specific coherence unit resides is called its home node. The interrupt cost, associated with receiving a message, for asynchronous protocol processing is typically a large component of the slow remote latency, not the actual wire delay in the network or the software actually implementing the protocol [12, 51].

In this chapter, we suggest a new efficient approach for software-based coherence protocols. While other work has proposed elaborate schemes for cutting down on the overhead associated with interrupting and/or polling caused by the asynchronous communication between the agents [11, 83], our implementation has completely eliminated the protocol-agent interactions. In DSZOOM, the entire coherence protocol is implemented in the protocol handler running in the requesting processor. This also makes use of a processor that otherwise would have been idle. Rather than relying on a “directory agent” located in the home node, as the synchronization point for the coherence of a cache line, we use a remote atomic fetch-and-set operation to allow for protocol handlers running in any node, not just the home node, to temporarily acquire atomic access to the directory structure of the cache line.

We have implemented the DSZOOM system between the nodes of a Sun UltraSPARC system without relying on its hardware-based coherence capabilities. All loads and stores are instead performed to the node’s local “private” memory. We use binary instrumentation to insert fine-grain access control checks before shared-memory loads and stores. Global coherence is resolved by a coherence protocol implemented in C that copies data to the node’s “private” local memory by performing loads and stores from and to remote memory.

A total of twelve unmodified SPLASH-2 applications, developed for fine-grain hardware multiprocessors, are studied. We compare the performance of a DSZOOM system with that of a Sun Enterprise E6000 server as well as the hardware-coherent WildFire system.

This chapter also extends the basic DSZOOM system with several novel optimization options and adds a low-overhead profiling mode that suggests appropriate coherence flags for the 12 applications studied. When compared with a hardware DSM system built from identical node hardware, interconnect and coherence strategy, the base system is trailing by 45 percent on average (slowdown-factor range is 0.98–3.02) for these 12 applications. The profiled
coherence flags brought the numbers down to 17 percent on average (0.69–1.85) and hand-tuned coherence flags down to 11 percent (0.69–1.66). It was a bit surprising, but very encouraging, to note that the profile-based coherence flags propelled the software DSM to outperform the hardware DSM for four of the applications.

4.2 Basic DSZOOM System

This section gives an overview of the basic DSZOOM system that relies on code instrumentation to maintain fine-grain coherence (load and store operations to shared memory are augmented with coherence checks). The provided protocols assume a high bandwidth, low latency cluster interconnect, supporting fast user level mechanisms for put, get, and atomic operations to remote nodes' memories, such as InfiniBand [52] or Sun Fire Link [113]. The system further assumes that the write order between any two endpoints in the network is preserved. These network assumptions make it possible to remove interrupt- and/or poll-based asynchronous protocol processing found in the majority of software DSM implementations [93, 11].

4.2.1 Cluster Network Model

DSZOOM assumes a cluster interconnect with an inexpensive user-level mechanism to access memory located in other nodes, similar to the remote put/get semantics found in the cluster version of the InfiniBand standard. Some of the memory in the other nodes is mapped into a node's I/O space and can be accessed using ordinary load and store operations. The different cluster nodes run different kernel instances and do not share memory with each other in a coherent way; in other words, no invalidation messages are sent between the nodes to maintain coherence when replicated data are altered in one node. This removes the needs for the complicated coherence scheme implemented in hardware and allows the NIC to be connected to the I/O bus, e.g., PCI or SBUS, rather than to the memory bus. In order to prevent a "wild node" from destroying crucial parts of other nodes' memories, the incoming transactions are sent through a network MMU (IOMMU). Each kernel needs to set up appropriate IOMMU mapping to the remotely accessible part of its memory before the other nodes are accessed. Given the correct initialization of the IOMMU, user-level accesses to remote memory are enabled.
4.2.2 Node Model

In our experimental setup, each DSZOOM node consists of a multiprocessor, e.g., the Sun Enterprise E6000 server with up to 28 processors. The node hardware keeps coherence among the caches and the memory within each node. The InfiniBand-like interconnect, as described above, connects the nodes.

4.2.3 Blocking Directory Protocol Overview

Most of the complexity of a coherence protocol is related to the race conditions caused by multiple simultaneous requests for the same cache line. Blocking directory coherence protocols have been suggested to simplify the design and verification of hardware DSM systems [44]. The directory blocks new requests to a cache line until all previous coherence activity to the cache line has ceased. The requesting node sends a completion signal upon completion of the activity, which releases the block for the cache line. This eliminates all the race conditions, since each cache line can only be involved in one ongoing coherence activity at any specific time.

The DSZOOM protocol implements a distributed version of a blocking protocol. A processor that has detected the need for global coherence activity will first acquire a lock associated with the cache line before starting the coherence activity. A remote fetch-and-set operation to the corresponding directory entry in the home node will bring the directory entry to the processor and also atomically acquire the cache line's "lock." If the most significant bit/byte of the directory entry returned is set, the cache line is "busy" by some other coherence activity. The fetch-and-set operation is repeated until the most significant bit is zero. (A random back-off scheme can be used to avoid a live-lock situation, but has not been employed in DSZOOM yet.) Now, the processor has acquired the exclusive right to perform coherence activities on the cache line and has also retrieved the necessary information in the directory entry using a single operation. The processor now has the same information as, and can assume the role of, the "directory agent" in the home node of a more traditional SW-DSM implementation. Once the coherence activity is completed, the lock is released and the directory is updated by a single put transaction. No memory barrier is needed after the put operation since any other processor will wait for the most significant bit/byte of the directory entry to become zero before the directory entry can be used. Thus, the latency of the remote write will not be visible to the processor.

To summarize, we have enabled the requesting processor to momentarily assume the role of a traditional "directory agent," including access to the directory data, at the cost of one remote latency and the transfer of two small network packets. This has the advantage of removing the need for asynchronous interrupts in foreign nodes and also allows us to execute the protocol in the
requesting processor that most likely would be idle waiting for the data. A further advantage is that the protocol execution is divided between all the processors in the node, not just one processor at a time as suggested in some other proposals.

4.2.4 The Invalidation-Based Protocol

The invalidation-based protocol states, modified, shared and invalid (MSI), are explicitly represented by global data structures in the nodes’ memories. Bits of a memory operation’s effective address determine the location of a coherence unit’s directory location, i.e., its “home node.” All coherence units in invalid state store a “magic” data value, as independently suggested by Scales et al. [105] and Chiou et al. [24] (Schoiras et al. [108] use the same technique in the Blizzard-S system). This significantly reduces the number of directory accesses caused by load operations, since the directory only has to be consulted on a read miss.\(^1\)

To reduce the number of accesses to remote directory entries caused by global store operations, each node has one byte of local state (MTAG) per global coherence unit (similar to Shasta’s private state table [104]), indicating if the coherence unit is locally writable. Before each global store operation, the MTAG byte is checked. The directory only has to be consulted if the MTAG indicates that the node currently does not have write permission to the coherence unit. The directory will assume the role of MTAG in the home nodes, and hence, no extra MTAG state is needed for home nodes. To avoid race conditions, the corresponding MTAG entry has to be locked before a write permission check is carried out. Otherwise, a coherence unit can be downgraded between the consultation and the point in time when the store is performed.

Figure 4.1 illustrates the protocol activity caused by a 2-hop write miss (transactions A1-A3) and a 3-hop read miss (transactions B1-B5). The state transitions for coherence units A and B can be found below each node. \(atomic1\), \(get64\) and \(put1\) correspond to a 1-byte remote atomic operation, a remote 64-byte get and a remote 1-byte put.

2-hop Write Miss Example: The requestor in node1, \(reqA\), checks its local MTAG for write permission of coherence unit A. Since node1 does not have write permission, the store protocol is called and the coherence activity is started. \(reqA\) acquires exclusive access to A’s directory located in node0’s memory (A1). When the directory is locked, the data is retrieved from the home node with a remote \(get64\) operation (A2). To end the coherence activity, \(reqA\) releases and updates the directory with a single remote \(put1\) operation.

\(^1\)The directory also has to be consulted in the rare case when the real data value is equal to the magic value [108, 105].
(A3), which is off the critical path.

![Diagram of 2-hop write miss and 3-hop read miss examples for the invalidation-based protocol.]

Figure 4.1: 2-hop write miss and 3-hop read miss examples for the invalidation-based protocol.

**3-hop Read Miss Example:** The requestor in node3, reqB, tries to read coherence unit B. However, the load returns the “magic” value and the load protocol is called. reqB locks the directory entry and determines the identity of the node holding the data (B1). The data happens to reside in node2, in a modified state. A second remote atomic operation (B2) to node2's MTAG structure disables write permission on that node. The data is fetched with a remote get64 operation (B3). Node2’s MTAG and the home’s directory are then released and updated. These two putl operations (B4 and B5) are also off the critical path.

### 4.2.5 Binary Instrumentation

Efficient binary instrumentation (or compiler support) is very important for the overall DSZOOM performance. *Binary instrumentation* is a technique usually described as a low-cost, medium-effort approach of inserting sequences of machine instructions into a program in executable or object format. In our early work [91, 92, 93, 77], we perform instrumentation with Laru's executable editing library (EEL) [69], which was successfully used in several similar projects based on the UltraSPARC architecture, e.g., Blizzard-S [109].
and Sirocco-S [107]. In our recent work [128, 129], however, we perform code instrumentation with our Java-based tool on assembly output of a GCC compiler. The following code example shows the code snippet for one global floating-point fine-grain access control check.

1: ld [addr], %reg //original load
2: fcmps %fcc0, %reg, %reg
3: nop
4: fbe,pt %fcc0, hit
5: //call global coherence load routine ...

hit:

The “magic” value in this case is an integer corresponding to an IEEE floating-point NaN (not a number). Only instructions 1–4 are executed if the loaded data is valid, i.e., the %reg is a non-magic value.2 Thus, this access control check is comparable in cost to the most efficient access check reported to date; three extra instructions adding three extra cycles for each load to global data [107].

4.2.6 Write Permission Cache

DSZOOM's access control checks for stores represent the largest part of the total instrumentation cost [128]. Most of this overhead comes from the fact that the locally cached directory entry (MTAG) must be checked atomically for each global store operation. This section describes write permission cache (WPC) that hides some of the instrumentation cost for stores [128]. While Shasta reduces its instrumentation overhead by statically merging coherence actions at instrumentation time [105] (batching), a WPC dynamically merges store coherence checks at runtime. Instead of releasing the MTAG lock after a store is performed, a thread holds on to the write permission and the MTAG lock, hoping that the next store will be to the same coherence unit. The identity of the coherence unit is stored in a dedicated register, which is consulted before the next store is performed. (DSZOOM reserves UltraSPARC's application registers [123] for fast WPC checks.) If indeed the next store is to the same coherence unit, the store overhead is reduced to a few ALU operations and a conditional branch instruction. When a store to another coherence unit appears, a WPC miss occurs. Only then, a lock release followed by a lock acquire must be performed.

Lines 01 to 17 of Figure 4.2 show how an original store instruction expands into a store snippet. Ry is a temporary register, Rx contains the value to be stored and addr is the effective address of this particular store operation. (CU_id[addr] refers to addr’s coherence unit identifier.) Lines 12

2 Line 3 can be eliminated if the code is executed on a SPARC-V9 architecture.
01: original_store:
02:  ST Rx, addr
03:  original_store_snippet:
04:  LOCK(MTAG_lock[CU_id[addr]])
05:  LD Ry, MTAG_value[CU_id[addr]]
06:  if (Ry != WRITE_PERMISSION)
07:    call st_protocol
08:  LOCK(MTAG_lock[CU_id[addr]])
09:  UNLOCK(MTAG_lock[CU_id[addr]])
10:  wpc_fast_path_snippet:
11:  if (WPC != CU_id[addr])
12:    call wpc_slow_path_snippet
13:  ST Rx, addr
14:  wpc_slow_path_snippet:
15:  UNLOCK(MTAG_lock[WPC])
16:  WPC = CU_id[addr]
17:  LOCK(MTAG_lock[CU_id[addr]])
18:  LD Ry, MTAG_value[CU_id[addr]]
19:  if (Ry != WRITE_PERMISSION)
20:    call st_protocol

Figure 4.2. Original and WPC store snippets.


and 17 acquire and release the MTAG lock. Lines 13 and 14 load and check the MTAG value for permission. If the processor does not have write permission, the store protocol is called at line 15. Finally, at line 16, the original store is performed.

Lines 21 to 37 of Figure 4.2 show a WPC snippet. The snippet consists of a fast- and a slow-path. The slow-path snippet is called when a WPC miss occurs. The lock of the currently cached coherence unit identifier is then released (line 32). Lines 34, 35 and 36, lock, load and check the MTAG for permission. Again, if the processor does not have write permission, the store protocol is called. Note that the lock is kept at the end of the WPC snippet. Holding on to the MTAG lock raises WPC related deadlock issues. We address these in section 4.6.

Figure 4.3 shows that the WPC hit rate for SPLASH-2 benchmarks [124] varies greatly depending on the application, the number of WPC entries and the coherence unit size. (GCC 3.3.4, optimization level 3, 16-processor runs.) For example, two entries demonstrate much better hit rate, which is most significant for fft and ocean applications.

62
Figure 4.3: WPC hit rate for different applications and coherence unit sizes (64-4096 bytes).
4.3 Extending DSZOOM’s Flexibility

It is well known that design choices made for coherence implementations may suit some applications better than others may. While a large coherence unit size may speed up one application, other applications may experience a slowdown due to increased false sharing. Another design choice that can go either way depending on the application is the choice between update or invalidate coherence. History has provided hardware capable of running multiple choices of coherence protocols, such as SPARCcenter 1000/2000’s dual coherence protocols: update and invalidate [19]. However, this choice was made at boot time and all applications, as well as the operating system, had to share a common strategy. Thus, to our knowledge, these systems were always run in invalidate mode.

This chapter proposes a transparent and [semi-] automatic technique for providing suitable per-application optimizations for software-based coherence. Our coherence protocol can be parameterized by setting the appropriate coherence flags when an application is started, much like the optimization flags of compilers. The coherence flags’ settings can specify the required coherence strategy (invalidate/update), coherence unit size, write instrumentation strategy, and a number of filtering strategies. This way, the beneficial coherence setting can be provided to an application without the need to modify the source code or even to recompile.

Even though our experiments only provide a handful of coherence flags to the application, the total number of possible coherence flag combinations is currently 54. Testing all these options for the optimal setting for each application is not feasible. We provide a simple low-overhead profiling tool that finds a close-to-optimal setting from a single profiling run. The profiling is always performed using a smaller “training” data set than the one used for the performance run. Where available, the training set is much smaller and the profiling run is consequently faster than the “production” run (between 1.1 and 52.4 times faster; 4.4 times on average).

One of the key observations of this work is that the WPC technology can be used as an efficient software-based store buffer in an update-based system. To be more specific, multiple stores could be merged before the MTAG lock is released and the data is distributed to other nodes.

In this section, we extend DSZOOM’s flexibility with a new update-based protocol based on store-buffer filtering that outperforms the base protocol for some applications, typically when the number of read misses is high. In a fine-grained software DSM system, the gain is twofold because only store operations must be instrumented. Section 4.3.2 presents additional bandwidth reduction techniques.
4.3.1 The Update-Based Protocol

The update-based protocol is based on write permission. All nodes have read permission to all data whereas only one has read-write permission for each coherence unit. The states read-write (W) and read (R or RW) are explicitly represented in the nodes' memories. Remote directory traffic is tamed with an update version of the MTAG optimization described in section 4.2.4. In addition, the update protocol is race-free, i.e., the corresponding MTAG entry has to be locked before a write permission check is carried out. Moreover, to guarantee data integrity, data must be distributed to other nodes before a MTAG is released.

3-hop Write Miss Example: Figure 4A shows coherence activity caused by an update 3-hop write miss. atomic1 and put1 correspond to a 1 byte remote atomic operation and a remote 1 byte put. The state transitions for coherence unit D can be found below the nodes. The requestor, reqD, first checks its local MTAG for write permission of coherence unit D. Since node2 does not have write permission, the store protocol is called and the coherence activity is started. reqD locks the directory located at the home node (D1). The directory indicates that the write permission is located on node0. Hence, D2 locks node0's MTAG. This removes write permission on that node. D3 and D4 updates the MTAG (IW) and the directory (node2's id) respectively. Node2 is now in state read-write and has correct data (data has to be distributed before a lock is released; the network order is preserved).

4.3.2 Bandwidth Reduction Techniques

Update-based coherence protocols have to deal with the potential bandwidth problem introduced by excessive data pushing. In this section, we present two mechanisms (filtering strategies) that address the bandwidth problem: dirty-data and private-data filtering.

Dirty-Data Filtering

Scaling the coherence unit size has two potential benefits: (1) a large coherence unit size can reduce the number of coherence misses, and (2) the WPC hit rate is improved. Hence, the number of locks taken and the instrumentation overhead are reduced. However, update-based coherence protocols can be very sensitive to coherence unit size scaling. A large coherence unit wastes bandwidth when exposed to write-write false sharing or if processors only write parts of the coherence unit before distributing the data. We address this problem with a dirty-data WPC that tracks modifications of a current cache line. Hence, only modifications (dirty data) are distributed to other nodes. A coherence unit is divided into smaller parts. The lock is obtained per coherence unit whereas the WPC points to one part of the coherence unit, the 'hot
Figure 4.4: 3-hop write miss example for the update-based protocol.

spot.” Data are marked dirty when the hot spot is moved.

Figure 4.5 shows how the dirty-data WPC handles a stream of store references. The coherence unit size is 512 bytes and the “hot spot” size is 64 bytes. When the store to a0 appears, the snippet code locks coherence unit A, a0 is put in the hot spot and marked as dirty. The next store to a2 misses in the hot spot, but is to the same coherence unit. The hot spot is moved to a2, which is marked as dirty. The store to b0 triggers both a hot spot and a coherence-unit miss. a0 and a2 are pushed to the other nodes and coherence unit A is unlocked. Coherence unit B is locked and b0 is both put in the hot spot and marked dirty. The store to b2 is handled just as the one to a2.

**Private-Data Filtering**

benchmarks often show a large amount of stores that are to node-private data. For some applications, this is true even at page granularity. For example, 89 (77) percent of the global stores in lu-c (ocean-c) are to private pages. We exploit this application property to reduce the global update bandwidth consumed with a *private-data* filter. The virtual-memory system is used to keep track of private-to-shared state changes and the page state is used to omit updates to node-private pages.

Our private-data filter is implemented with a page-permission check from a locally cached page directory before each global update. In addition, at
read/write page faults to the shared memory segment, our signal handler updates the page directory with new permission information through a remote-atomic and a remote-put operation and uses `mprotect(2)` to set up local memory mappings. If the node does not have data, a page fetch might be needed. While multiple schemes are possible, our system only allows page-permission upgrades.

4.4 Performance Evaluation

Table 4.1 shows data set sizes for all of the SPLASH-2 applications studied. The reason we cannot run `volrend` is that shared variables are not correctly allocated with the `G_MALLOC` macro, `cholesky` is not run because we were not able to find a large enough working sets.

4.4.1 Compiler and Instrumentation Software

All experiments in this chapter use the GCC 3.3.4 compiler. To simplify instrumentation, we use GCC’s `-fno-delayed-branch` flag that avoids loads and stores in delay slots, and `-fno-app-regs` that reserves UltraSPARC’s thread-private registers [123] for our snippets. These two flags
slow down SPLASH-2 applications with less than 3 percent on average. Note that only the DSZOOM system uses those flags. All benchmarks are compiled with optimization level 3.

We extend DSZOOM’s instrumentation tool with a simplified version of Shasta’s batching technique [105, 128]. The tool implements a read-modify-write batching, which merges load and store coherence checks (to the same effective address) by replacing the load check with the store’s WPC check. We also schedule application instructions into coherence snippets to increase instruction-level parallelism (inspired by EEL [69]).

4.4.2 Hardware Setup

The systems is run in two different modes in order to model a hardware-coherent as well as a software-coherent system built from identical node and interconnect hardware.

The hardware DSM results have been measured on a WildFire system configured as a traditional CC-NUMA with its data migration capability activated and its coherent memory replication (CMR) disabled. There is one common shared address space accessed by all processors in the system. That shared address space is kept coherent by the global hardware-coherence protocol implemented in WildFire’s network interface.

The software DSM results have been measured on a WildFire with its data migration capability as well as its replication (CMR) capability disabled. The
shared address space is allocated twice with one copy physically tied down to the local memory of each node. The shared memory of the application is initialized in a way such that the processors of one node will perform all their loads and stores to the node-local copy of the address space. When a fine-grain coherence access violation is detected, the software-based DSM protocol running in user level is activated. Only the DSM protocol is capable of accessing data located in the remote node’s memory. In order to bypass WildFire’s hardware-coherence protocol and avoid cache pollution, non-cacheable block load and block store SPARC instructions are used by the DSM protocol. The atomic memory operations (ldstub) is used as remote put, get and atomic operations.

The sequential experiments run on two processor types: a 250 MHz UltraSPARC II (USII) and a 900 MHz UltraSPARC III (USIII). The USII processor has a 32 kbyte instruction cache, a 64 kbyte data cache, a 2 kbyte write cache and a 2 kbyte prefetch cache. The second-level cache is 8 Mbyte and off-chip.

### 4.4.3 Instrumentation Overhead

Table 4.2 shows sequential-execution time in seconds for non-instrumented programs (second column). It also reports the factor increase in execution time when both load and store instrumentation is inserted for three invalidation-based configurations: the invalidation-based protocol without WPC (inv), the invalidation-based protocol with a 1-entry WPC (inv-swpc) and the invalidation-based protocol with a 2-entry WPC (inv-dwpc) (see Table 4.3 for abbreviations). All experiments run on both USII and USIII processors with a coherence unit size of 512 bytes. On average, instrumentation overhead for the slower processor (USII) is lowered from 66 percent for the inv protocol to 33 percent when a 2-entry WPC is used (inv-dwpc). For the faster processor, this reduction is even more significant (from 104 percent to 55 percent). The store instrumentation overhead for WPC implementations can be reduced even further if larger coherence unit sizes are used. For example, the store instrumentation overhead for fft is reduced from 178 to 27 percent for the USIII target when a 2-entry WPC is added and the coherence unit is scaled from 64 to 8192 bytes. Note that the instrumentation techniques based on WPC (such as inv-swpc and inv-dwpc) only add ALU instructions to the fast path of the execution when loads and stores are instrumented. This results in an instrumentation overhead that is fairly independent of processor technology, as can be seen in Table 4.2. Traditional instrumentation techniques (such as inv), that also add memory operations for store instrumentation, experience a much higher overhead for UltraSPARC III than for UltraSPARC II processors.
<table>
<thead>
<tr>
<th>Program</th>
<th>Time [s] USH (II)</th>
<th>inv USH (II)</th>
<th>inv-swpc USH (II)</th>
<th>inv-dwpc USH (II)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fit</td>
<td>17.4 (9.7)</td>
<td>2.67 (2.89)</td>
<td>2.18 (2.08)</td>
<td>1.65 (1.52)</td>
</tr>
<tr>
<td>lu-c</td>
<td>53.2 (14.9)</td>
<td>3.56 (5.83)</td>
<td>1.48 (1.79)</td>
<td>1.51 (1.84)</td>
</tr>
<tr>
<td>lu-nc</td>
<td>270.4 (87.4)</td>
<td>2.15 (3.30)</td>
<td>1.53 (1.45)</td>
<td>1.60 (1.49)</td>
</tr>
<tr>
<td>radix</td>
<td>17.7 (22.8)</td>
<td>1.52 (1.75)</td>
<td>1.70 (1.79)</td>
<td>1.66 (1.69)</td>
</tr>
<tr>
<td>barnes</td>
<td>161.0 (52.5)</td>
<td>1.09 (1.15)</td>
<td>1.07 (1.13)</td>
<td>1.10 (1.13)</td>
</tr>
<tr>
<td>firm</td>
<td>155.0 (48.0)</td>
<td>1.15 (1.28)</td>
<td>1.10 (1.17)</td>
<td>1.10 (1.18)</td>
</tr>
<tr>
<td>ocean-c</td>
<td>84.5 (56.5)</td>
<td>1.71 (1.72)</td>
<td>1.49 (1.30)</td>
<td>1.33 (1.18)</td>
</tr>
<tr>
<td>ocean-nc</td>
<td>132.2 (91.4)</td>
<td>1.45 (1.42)</td>
<td>1.38 (1.26)</td>
<td>1.31 (1.19)</td>
</tr>
<tr>
<td>radiosity</td>
<td>198.8 (14.7)</td>
<td>1.08 (1.39)</td>
<td>1.11 (1.18)</td>
<td>1.11 (1.18)</td>
</tr>
<tr>
<td>raytrace</td>
<td>80.5 (23.9)</td>
<td>1.24 (1.36)</td>
<td>1.24 (1.36)</td>
<td>1.24 (1.35)</td>
</tr>
<tr>
<td>water-nsp</td>
<td>162.8 (68.2)</td>
<td>1.18 (1.28)</td>
<td>1.16 (1.27)</td>
<td>1.12 (1.27)</td>
</tr>
<tr>
<td>water-sp</td>
<td>115.7 (48.3)</td>
<td>1.16 (1.28)</td>
<td>1.15 (1.23)</td>
<td>1.17 (1.24)</td>
</tr>
<tr>
<td>Avg.</td>
<td>1.66 (2.04)</td>
<td>1.38 (1.42)</td>
<td>1.33 (1.35)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Sequential instrumentation overhead for 250 MHz UltraSPARC II and 900 MHz UltraSPARC III processor runs.

We believe that the instrumentation overhead can be further tamed with compiler support for instrumentation, as been demonstrated by Niwa et al. [84]. This could potentially close the performance gap to hardware DSM even further. However, this would require a recompilation when a shared-memory application is moved to software DSM.

4.4.4 Instrumentation Overhead Breakdown

In this section, we characterize the overhead of inserted fine-grain access control checks for global loads/stores for all of the studied SPLASH-2 programs. Since the WPC technology is a store optimization technique, the write permission checking code (store snippets) is the focus of this section. To obtain a sequential instrumentation breakdown for different snippets, we ran the applications with just one processor and with only one kind of memory instruction instrumented at a time. This way, the code will never need to perform any coherency work and will therefore never enter the protocol code.

Sequential instrumentation overhead breakdowns for applications compiled with minimum and maximum compiler optimizations are shown in Figures 4.6 and 4.7. The store overhead is the single largest source of the total instrumentation overhead: 61 (34) percent for optimized (non-optimized) code. In addition, the single-WPC checking code (st-swpc) reduces this store overhead to 57 (16) percent. Double-WPC checking code (st-duwpc) further reduces the

70
Figure 4.6: Sequential instrumentation overhead breakdown (non-optimized binaries) for integer loads (intld), floating-point loads (fpfl), the original store snippet (st), a 1-entry WPC store snippet (stk-wpc), a 2-entry WPC store snippet (st-dwpc) and a store snippet with WPC hit rate 1.0 (st-wpc-hr1). Note that this study uses GNU's gcc-3.3.3 compiler.
Figure 4.7: Sequential instrumentation overhead breakdown (optimized binaries) for integer loads (intld), floating-point loads (fpld), the original store snippet (sr), a 1-entry WPC store snippet (st-swpc), a 2-entry WPC store snippet (st-dwpc) and a store snippet with WPC hit rate 1.0 (st-wpc-hr1). Note that this study uses GNU’s gcc-3.3.3 compiler.
original store overhead to 36 (11) percent. As expected, the reduction is most significant for lu-c and lu-nc because they have the highest WPC hit rate, see Figure 4.3, and low shared load/store ratio [124], fft and cholesky perform much better when a 2-entry WPC is used. For radix, the instrumentation overhead slightly increases for the st-swpc and st-dwpc implementations. The low WPC hit rate (see Figure 4.3) is directly reflected in this particular instrumentation breakdown. Finally, the “perfect” WPC checking code (st-wpc-lr1) demonstrates very low instrumentation overheads: 9 percent for optimized and 3 percent for non-optimized code.

### 4.4.5 Parallel Performance

Figure 4.8 shows the performance impact of various coherence protocols and optimizations for 16-processor runs when adding different protocol optimizations as compared to DSZOOM’s base protocol (inv-64). Selecting the most optimal coherence unit size improves fft’s performance with 25 percent. Adding the most appropriate WPC strategy amounts for an additional 40 percent and instrumentation scheduling adds another 5 percent. The base update protocol (upd-swpc-64) results in a slowdown compared with the inv-64 protocol. However, tuning the WPC setting and enabling the private-data filtering improves fft’s update performance with more than 100 percent. Still, the most optimal update protocol is still 38 percent behind the base-inv-64 protocol for fft. Only performance improvements are shown in Figure 4.8, which is why some of the optimizations are not visible for all applications.

There is a large variation among the applications as to which class of optimization is the most important. ocean-c’s update protocol is greatly improved by the private-data filtering, which makes it outperform the best invalidate-based protocol, while lu-c enjoys a great boost to its invalidate protocol from its most optimal WPC setting. While Figure 4.8 can help understanding the importance of the different optimizations (further explained in section 4.4.6), it should be pointed out that the different
Figure 4.8: Invalidation (inv) and update-based (upd) speedup when normalized to DSZOOM's invalidation-based system with a coherence unit size of 64 bytes (inv-64).
Figure 4.9: Hardware DSM vs. four software DSM experiments. (16-processor runs.)
performance improvements reported are somewhat dependent on each other, why the orders in which they are presented do effect their individual contributions.

Figure 4.9 shows parallel performance for 16-processor runs. Here, the software DSM performance can be compared to the execution time of a 2-node Sun WildFire (HW-DSM) and DSZOOM’s base configuration (inv-64). As a comparison, the execution time for inv-dwpc-64 as well as the configuration suggested by the profiling tool (PROFILED), described in section 4.5, are also reported. The rightmost bar (BEST) shows the best performance obtained by testing all coherence flag settings. (See Table 4.4 for (PROFILED) and (BEST) configurations.) To ensure that our results are not affected by application scaling characteristics, we also run all applications with four processors per node (eight in total). These results are almost identical to the ones presented in this chapter (when compared to the hardware DSM) and are omitted because of space.

<table>
<thead>
<tr>
<th>Program</th>
<th>PROFILED</th>
<th>BEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>inv-dwpc-2048</td>
<td>inv-dwpc-2048</td>
</tr>
<tr>
<td>lu-c</td>
<td>inv-swpc-2048</td>
<td>inv-dwpc-2048</td>
</tr>
<tr>
<td>lu-nc</td>
<td>inv-swpc-128</td>
<td>inv-swpc-128</td>
</tr>
<tr>
<td>radix</td>
<td>inv-64</td>
<td>inv-128</td>
</tr>
<tr>
<td>barnes</td>
<td>upd-swpc-64</td>
<td>upd-df-swpc-512</td>
</tr>
<tr>
<td>imm</td>
<td>inv-swpc-64</td>
<td>inv-swpc-64</td>
</tr>
<tr>
<td>ocean-c</td>
<td>upd-pf-dwpc-512</td>
<td>upd-pf-dwpc-1024</td>
</tr>
<tr>
<td>ocean-nc</td>
<td>inv-dwpc-2048</td>
<td>inv-dwpc-1024</td>
</tr>
<tr>
<td>radiosity</td>
<td>upd-swpc-64</td>
<td>upd-swpc-64</td>
</tr>
<tr>
<td>raytrace</td>
<td>upd-swpc-64</td>
<td>inv-swpc-64</td>
</tr>
<tr>
<td>water-nsq</td>
<td>upd-pf-swpc-64</td>
<td>upd-pf-swpc-256</td>
</tr>
<tr>
<td>water-sp</td>
<td>upd-pf-swpc-64</td>
<td>upd-pf-swpc-64</td>
</tr>
</tbody>
</table>

Table 4.4: Classification results from the profile feedback run and the best coherence setting.

On average, the inv-64 protocol is 45 percent slower than the hardware DSM system. This overhead is reduced to 32 percent when the inv-dwpc-64 protocol is used. Optimal coherence unit size, number of WPC entries and instrumentation optimizations further improve the invalidation-based DSZOOM performance with almost 30 percent. The invalidation-based protocol actually outperforms the hardware DSM system when run with fft, ocean-c and ocean-nc. While an invalidation-based coherence protocol together with “the best” coherence unit size offers stable performance, some applications show peak performance when run in an
update-based environment as long as bandwidth usage is kept low. The update-based protocol is able to outperform the hardware DSM system for ocean-c, ocean-nc\(^3\) and radiosity. The number of WPC entries and the private-data filter are the most important optimizations for update mode.

Maybe the most notable performance feature is the similarity between the performance of the best DSZOOM protocol (BEST) and the Sun WildFire system (HW-DSM). The performance of the two systems is within 30 percent of each other for all applications except radix and lu-c. Note also that the continuous and non-continuous versions of lu, ocean and water all achieve a similar performance compared with the hardware DSM. This is typically not the case for traditional software DSMs. On average, DSZOOM is 11 percent slower than the hardware DSM. When radix, the application with the worst locality, is omitted, this slowdown is reduced to only 5 percent on average.

4.4.6 FFT Case Study

To be able to further show the impact of the different optimizations, we provide a case study of a representative application. Figure 4.10 shows DSZOOM performance for (a) the invalidation- and (b) the update-based protocols when coherence unit size is scaled. Execution time is normalized against the hardware DSM system whereas bandwidth (collected with WildFire interconnect counters) is normalized against the inv-64 configuration. Figure 4.10 (a) shows that fft scales with coherence unit size in an invalidation-based environment. The inv-dwpc configuration performs best and outperforms the hardware DSM system when a coherence unit size of 2048 bytes is used. This is because parts of the fft application use two write streams, and hence, shows much better WPC hit rate with a 2-entry WPC than with a 1-entry WPC; see Figure 4.3.

Figure 4.10 (b) shows the update-based protocol with its filters and combinations of them. The performance of update is poor when used with a single WPC entry (upd-swp). Scaling the coherence unit size makes it worse. However, performance is improved when the private-data filter is added (upd-pf-swp). This is because the private-data filter exploits the fact that more than 50 percent of the global stores in fft are to private pages, and hence, manages to reduce the consumed bandwidth with more than 50 percent. Again, when coherence unit size is increased the performance goes down. The dirty-data filter configuration (upd-df-swp) starts out where the upd-swp system started, but improves as coherence unit size scales. This is because the single write stream part improves as coherence

\(^3\)The reason why inv-64 is better than the hardware DSM when run on ocean-nc is that the -fno-delayed-branch actually improves performance on this particular application.
Figure 4.10: DSZOOM performance for fft. The execution time is normalized to that of the HW-DSM whereas the bandwidth is normalized to inv−64.
unit size scales but a 1-entry WPC and a large coherence unit size is a bad match for the part with two write streams. However, since the dirty-data filter removes updates of non-dirty data, coherence unit size scaling can be used. The upd-df-pf-swp configuration shows that the dirty-data and the private-data filter target different kinds of bandwidth. The performance starts out where upd-pf-swp starts but improves when coherence unit size is scaled.

Since fft has high WPC hit rate for a 2-entry WPC, the upd-dwpc configuration shows much better performance than the 1-entry WPC configuration. Again, adding the dirty-data filter both improves the performance and reduces the bandwidth. It is interesting to see that the parallel performance of fft directly follows consumed bandwidth.

4.5 Profiling and Classification of Applications

Previous section demonstrates that a flexible software DSM system can deliver hardware DSM competitive performance. However, choosing optimal coherence flags may be a cumbersome task. This section presents a simple classification algorithm for fast finding of appropriate coherence settings. The classification heuristic is based on feedback from a low-overhead profile run. Our classification algorithm is not general, it is intended to show that also a simple heuristic can be used to achieve appropriate coherence settings, and hence, high performance.

We have tested our profiling mode and classification algorithm with both small and large working set sizes. Our results are almost identical. Thus, for the applications studied, it is possible to use the small working set size during the profile run and reuse the same coherence strategy for "production" runs. While scaling down the workload size on a uniprocessor system can heavily affect the cache performance, the profile mode tracks the entire shared memory and especially coherence traffic. Hence, it is not heavily dependent on machine parameters, such as cache sizes. (Table 4.1 shows the small working set size used for classification and the large working set size used for result runs.)

4.5.1 Low-Overhead Profiling

Our low-overhead profiling is capable to collect 1- and 2-entry WPC hit rate, global update bandwidth and coherence unit size information in a single run with less than 30 percent overhead (avg.). The estimation of coherence unit size uses virtual coherence units. We slice the global memory space into different segments (currently, 2048 bytes each). These segments use different
<table>
<thead>
<tr>
<th>Program</th>
<th>Base Mode</th>
<th>Profile Mode</th>
<th>Profile Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Large Set</td>
<td>Large Set</td>
<td>Training Set</td>
</tr>
<tr>
<td>fft</td>
<td>5.99 s</td>
<td>6.13 s (1.02)</td>
<td>0.18 s (0.03)</td>
</tr>
<tr>
<td>lu-c</td>
<td>28.79 s</td>
<td>22.90 s (0.80)</td>
<td>0.55 s (0.02)</td>
</tr>
<tr>
<td>lu-nc</td>
<td>46.58 s</td>
<td>96.49 s (2.07)</td>
<td>2.90 s (0.06)</td>
</tr>
<tr>
<td>radix</td>
<td>13.09 s</td>
<td>16.86 s (1.29)</td>
<td>3.55 s (0.27)</td>
</tr>
<tr>
<td>Barnes</td>
<td>14.62 s</td>
<td>19.64 s (1.34)</td>
<td>2.77 s (0.39)</td>
</tr>
<tr>
<td>Firm</td>
<td>16.26 s</td>
<td>26.73 s (1.64)</td>
<td>7.66 s (0.47)</td>
</tr>
<tr>
<td>ocean-c</td>
<td>9.37 s</td>
<td>9.04 s (0.96)</td>
<td>1.51 s (0.16)</td>
</tr>
<tr>
<td>ocean-nc</td>
<td>17.71 s</td>
<td>17.44 s (0.98)</td>
<td>2.18 s (0.12)</td>
</tr>
<tr>
<td>radiosity</td>
<td>429 s</td>
<td>9.16 s (1.87)</td>
<td>0.23 s (0.05)</td>
</tr>
<tr>
<td>raytrace</td>
<td>15.27 s</td>
<td>18.69 s (1.22)</td>
<td>14.02 s (0.92)</td>
</tr>
<tr>
<td>water-nsq</td>
<td>1288 s</td>
<td>13.74 s (1.07)</td>
<td>3.41 s (0.27)</td>
</tr>
<tr>
<td>water-sp</td>
<td>9.12 s</td>
<td>10.62 s (1.16)</td>
<td>1.65 s (0.18)</td>
</tr>
<tr>
<td>Avg.</td>
<td></td>
<td>(1.29)</td>
<td>(0.23)</td>
</tr>
</tbody>
</table>

Table 4.5: Performance of the profiling mode for large and training (small) input data sets. Normalized execution time is shown inside parenthesis.

virtual coherence unit sizes. When a coherence miss occurs, permission for the entire virtual coherence unit size is acquired. DSZOOM's runtime system collects the number of misses for all different coherence sizes in a single run. Of course, it is important to divide the memory space in a representative way. We have tried multiple schemes and found that a simple modulo scheme works satisfactorily for our conservative classification algorithm and the applications studied. The virtual-memory system collects non-private store information. When a WPC miss occurs, a local page directory lookup classifies the store as private or non-private. The number of non-private store misses is proportional to the global update bandwidth used in the system when run in update mode. Hence, this information lets one decide if the update-based coherence protocol is a good candidate or not, and if the private-data filter should be used.

Table 4.5 shows performance of the profiling mode for 16-processor runs. Numbers for both training (small) and large data input sets are shown. It is interesting to see that three applications run faster when the profiling mode is turned on. All three applications gain speedup because of the virtual-coherence technique described above. The large coherence unit size used in some of its segments are very beneficial for these applications. The performance of a profile mode is significantly increased for training input sets. For example, the most extreme case (lu-c) runs about 50 times faster than the base mode with a large input set.
4.5.2 Simple Classification Method

This section describes the proposed classification algorithm. Because the DSZOOM system implements multiple memory consistency models (further discussed in section 4.6), and some applications require a stricter memory model than others, the memory consistency model has to be taken into account when choosing coherence flags. If the memory-consistency model requirement for an application is not known, a conservative choice has to be made. We use the memory consistency model and the global update bandwidth to select between invalidate/update. We have used the global update bandwidth data and the low-overhead profile run execution time to estimate bandwidth/sec. Applications that consume less than 100 Mbyte/s are classified as update candidates.

The global update bandwidth with and without the private-data filter is used to make a choice for update filtering techniques. For example, ocean-c consumes more than 400 Mbyte/s without private-data filter. This number is reduced to less than 10 Mbyte/s when the filter is enabled (we enable the private-data filter if it reduces the bandwidth with more than 10 percent). Currently, we do not have a good metric for the dirty-data filter. However, it is reasonable to use this filter when a large coherence unit size is used with the upd-update configuration.

We use the number of coherence misses to different virtual coherence unit sizes to select coherence unit size for an application. Applications with a significant amount of spatial locality (e.g., fft and lu-c) are easily recognized because the number of misses is reduced by 50 percent each time the coherence unit size is doubled. Applications that expose false sharing are also easily recognized since the number of misses increase. However, applications that exploit some locality and at the same time introduce some false sharing are harder to classify. We use a conservative approach for these applications by choosing a small coherence unit size. In addition, an upper limit of 512 bytes for update-based protocols is applied.

Finally, the number of WPC entries has to be selected. Since 1- and 2-entry WPC hit rate is contained in the profile data, this seems like a simple task. However, the WPC hit rate is collected on a system running with a coherence unit size of 64 bytes. This is the reason why the classification of lu-c does not show peak performance (see Table 4.4 and Figure 4.9).

4.6 Memory Consistency, Deadlock, and Scalability

The introduction of the WPC technique raises multiple questions regarding deadlocks and memory consistency models, which are addressed in this section. We also discuss the protocol scalability and the hardware DSM platform.
used for comparison.

4.6.1 Memory Consistency

The invalidation-based protocol of the base architecture (without a WPC implementation) maintains sequential consistency (SC) [68] by requiring all acknowledgments from the sharing nodes to be received before a global store request is granted. Introducing the WPC in an invalidation-based environment will not weaken the memory model. The WPC protocol still requires all the remotely shared copies to be destroyed before granting the write permission. WPC just extends the duration of the permission tenure before the write permission is given up. Of course, if the memory model of each node is weaker than sequential consistency, it will dictate the memory model of the system. The invalidation-based DSZOOM system implements total store order (TSO) [123] since E6000 nodes are used.

For an update-based system without load instrumentation, such as the one we present in section 4.3.1, the sequential consistency property is sacrificed. Our update-based software DSM system with a 1-entry WPC and a 64 bytes coherence unit size implements processor consistency (PC) [37, 35]. Writes from a processor cannot be observed out of issue order by another processor since the node’s hardware keeps write ordering correct per coherence unit. When a processor decides to write to a new coherence unit, the old coherence unit is made available to all other nodes, and hence, the processor store order is preserved. However, the order in which writes from two processors are seen by others may differ. Thus, PC and not TSO is implemented. The memory consistency model gets more relaxed if more than one WPC entry or a coherence unit size larger than 64 bytes is used. Such a system needs multiple updates to push all data to other nodes, and hence, store issue order can get lost. This consistency model is similar to weak-ordering (WO) [31]. Our PC and WO systems do not implement causal correctness [106].

4.6.2 Deadlock Avoidance Mechanisms

To avoid WPC related deadlocks, our runtime system releases a processor’s WPC entries at synchronization points, at failures to acquire MTAG/directory entries and at thread termination. However, since SC (TSO) and PC are supported, flag synchronization not visible to the runtime system may occur. The code in Figure 4.11 can for example lead to deadlock. \( a, b \) and \( flag \) are all global variables initially assigned the value zero. \( a \) and \( b \) are both located on the same coherence unit \( u \). \( flag \) is located on coherence unit \( v \neq u \). Let two processors, \( P0 \) and \( P1 \), execute the code shown in Figure 4.11. \( P0 \) enters the code first (executes line 02) and assigns \( a \) the value one. This implies that \( P0 \)
puts a’s coherence unit u in its WPC. When PO reaches line 03, it starts to spin on the shared variable flag, waiting for P1. P1 enters the code (line 12) and tries to obtain write permission for b. However, since the directory state for b’s coherence unit u is locked and cached by PO’s WPC, we have a deadlock! PO is waiting for P1 to update the flag variable, and P1 is waiting for PO to release the caching of u.

01: /* PO’s code */
02: a = l;
03: while (flag != 1) 12: b = l;
04: /* wait */ 13: flag = l;
05: ...

Figure 4.11: WPC-deadlock code. a, b and flag are all global variables initially assigned the value zero. a and b are located on the same coherence unit whereas flag is located on another.

These WPC related deadlocks are easily avoided with extra runtime system support. We can envision three possible mechanisms for avoiding deadlocks: (1) a processor’s WPC entries can be flushed periodically by the runtime system, (2) the processor waiting for coherence unit u can signal a WPC release to the processor caching write permission for u with a remote interrupt, and (3) PO can detect its lack of forward progress and flush its WPC entries. For more information see [127]. However, we are convinced that the simplest and best solution is to implement WPC deadlock avoidance in the instrumentation tool. A WPC FIFO replacement policy together with simple basic-block analysis can be used to guarantee that all MTAG locks are released before flag synchronizations (not currently implemented). For simplicity, our instrumentation tool is manually guided in the two applications (barnes and fmm) that use flag synchronization.

4.6.3 Protocol Scalability

This work only presents data for a 2-node system since our WildFire machine only contains two E6000 nodes. A different study has used “virtual clustering” [126] to show that our invalidation-based protocol scales with number of nodes [91]. However, we do not believe that our update-based protocol will scale to a large number of nodes. The reason we have not used virtual clustering emulation while testing the update-based protocols is because it is very difficult to model bandwidth in an accurate way. Given a more scalable hardware setup, it would have been very interesting to test how a WPC-based store buffer and bandwidth filters would affect update scalability.

4A countdown register updated by the runtime system can be used.
4.7 Related Work

Many different SW-DSM implementations have been proposed over the years: Blizzard-S [109], Brazos [114], Cashmere-2L [116, 32], CRL [55], GeNIMA [11], Ivy [73, 75], MGS [125], Murin [18], Shasta [105, 103, 100, 102, 32], Sirocco-S [107], SoftFLASH [33], and TreadMarks [59]. Most of them suffer from synchronous interrupt protocol processing. We believe that many of these implementations would benefit from a more efficient protocol implementation; such the one described here.

The DSZOOM's basic approach is derived from several fine-grain SW-DSM systems: Shasta, Blizzard-S, and Sirocco-S. Our "magic"-value technique for fine-grain access control checks is similar to Shasta's "flag"-value and Blizzard's "sentinel"-value optimizations. This technique was independently introduced in Shasta [105] and Blizzard-S [109] for use with all types of loads. There are several other systems that use compiler-generated checks to implement a global address space (for example, Olden [16], Split-C [27], and Midway [10]).

The GeNIMA proposal is closest to our work [11], GeNIMA proposes a protocol and a general network interface mechanism to avoid some of the asynchronous overhead. A processor starting a synchronous communication event, e.g., the requesting processor initiating some coherence actions, checks for incoming messages at the same time. This avoids some of the asynchronous overhead in the home node, but will also add some extra delay while waiting for a synchronous event to happen in the node. The protocol is still implemented as communicating protocol agents.

Several other papers have suggested hardware support for fine-grain remote write operations in the network interface [64, 63]. One of the implementations is the automatic update release consistency (AURC) home-based protocol [50]. This implementation is a page-based SW-DSM which eliminates " diffs"—the compact encoded representation of the differences between the two pages, frequently used in many page-based SW-DSM systems—by using fine-grain remote writes for both the application data and the protocol metadata. The AURC approach usually performs better than all-software home-based LRC implementations.

Traditional implementations of software-based shared memory rely on virtual memory hardware to detect when coherence activity is needed. Early page-based systems [72] suffer from false sharing that arises from fine-grain sharing of data within a page. Two main research directions have evolved to improve the performance of software shared memory implementations: relaxing consistency models [18, 60, 130, 105] and providing fine-grained access control [109, 105].

Page-based systems often rely on weak memory consistency models and
multiple writer protocol to manage the false sharing introduced by their large coherence unit [33, 116, 11]. Carter et al. [18] introduce the release consistency (RC) model in shared virtual memory. Lazy release consistency was introduced by Keleher et al. [69] and home based lazy released consistency (HLRC) by Zhou et al. [130]. The majority of systems implement numerous coherence strategies/protocols. For example, Munin [18] implements both invalidate- and update-based protocols (including delayed-update and write-shared protocols).

Our system can run multiple memory consistency models, including SC (TSO), PC and WO, while page-based systems often rely on RC, LRC or HLRC protocols. Our private-data bandwidth filter is similar to Munin’s time-out mechanism that makes it possible to update only nodes that actually use data. However, contrary to Munin, our private-data filter is completely synchronous, and hence, removes all asynchronous protocol messaging. Moreover, it is designed to be used with a fine-grain system and can be run in processor consistency mode whereas Munin relies on release consistency. Our dirty-data filter can be compared to the multiple writer protocols’ twin and diff strategies. However, it is much more “light weight” and considerably faster to manage. Where twin and diff strategies have to compare the entire page, we simply check 2-4 bits located in a register before an update. Moreover, the dirty-data filter is a bandwidth reduction technique, while twin and diff strategies maintain coherence and memory consistency.

Fine-grained software DSM systems maintain coherence by instrumenting memory operations in the programs [107, 105, 104]. These systems usually provide stable and predictable performance for the majority of parallel benchmarks originally developed for hardware multiprocessors. On the other hand, the instrumentation cost for most of the systems is not negligible. An interesting comparative study of two mature software-based systems from the late 90s shows that the performance gap between fine- and coarse-grain software DSMs can be bridged by adjusting coherence unit size, program restructuring and relaxing memory consistency models [32].

In an early version of Blizzard [34], application-specific software-based protocols, which provided very high performance, were implemented and evaluated. Shasta [105, 104] implements support for multiple coherence granularities within a single application. This mechanism is exposed to the programmer through multiple memory allocation functions. Zhou et al. [131] presents performance tradeoffs for relaxed consistency and coherence granularity on a platform that provides access control in hardware but runs coherence protocols in software. Their study focuses on coherent shared memory systems with a fixed coherence granularity (64, 256, 1,024, and 4,096 bytes). The results show that no single combination of protocol and granularity performs the best for all SPLASH-2 [124] applications studied.
The Stanford FLASH [67] project addresses concerns with hardwired coherence protocols by migrating the entire protocol-engine to software handlers executed on a separate processor. SMTp is a more recent proposal [23] in which the coherence protocol is run by one SMT thread. Multiple systems implement a simple hardware directory protocol backed up with software handlers. The protocol described by Hill et al. [46] uses a single hardware pointer. In addition, the programmer or compiler can annotate programs with Check-In/Check-Out (CICO) directives to minimize the number of software traps. Chaiken and Agarwal [20] describe performance and cost of software extended coherence shared memory as implemented in Alewife. Grahn and Stenström [40] extend Chaiken and Agarwal’s work. While most of the findings in this chapter can be implemented in such systems, they rely on modified memory controllers [23], protocol processors [67] and/or hardware support for \(n\) pointers in hardware [20, 40, 46].
5. A Trap-Based Memory Architecture

The advances in semiconductor technology have set the shared-memory server trend towards processors with multiple cores per die and multiple threads per core. We believe that this technology shift forces a reevaluation of how to interconnect multiple such chips to form larger systems.

This chapter argues that by adding support for "coherence traps" in future chip multiprocessors, large-scale server systems can be formed at a much lower cost. This is due to shorter design time, verification and time to market when compared to its traditional all-hardware counterpart. In the proposed "trap-based memory architecture" (TMA), software trap handlers are responsible for obtaining read/write permission, whereas the coherence trap hardware is responsible for the actual permission check.

In this chapter we evaluate a TMA implementation (called "TMA Lite") with a minimal amount of hardware extensions, all contained within the processor. The proposed mechanisms for coherence trap processing should not affect the critical path and have a negligible cost in terms of area and power for most processor designs.

Our evaluation is based on detailed full system simulation using out-of-order processors with one or two dual-threaded cores per die as processing nodes. The results show that a TMA based distributed shared memory system can on average perform within one percent of a highly optimized hardware based design.

5.1 Introduction

While large-scale hardware-based shared memory systems have been successfully built for many years, the cost in terms of design and verification for each new generation is ever increasing. This is, for example, due to multiple levels of coherence, i.e., intra-snoop and inter-snoop domain coherence. Recently, the continued decrease in transistor size and the increasing delay of wires have lead to the development of chip-multiprocessors (CMPs) [8, 88, 120]. With the next generation CMPs, promising to include as much as 32 hardware threads per chip [66], the mid-range servers of today will actually fit on a single die. This raises a very interesting question: what is the most cost-efficient way to connect multiple such chips together to form larger shared memory
systems without sacrificing performance?

The traditional way of designing a large-scale shared-memory system is by modifying the memory system, leaving the processor core unchanged. We propose the opposite; adding a minimal amount of hardware support, contained within each processor core, combined with an entirely unmodified memory system designed and optimized for single-chip systems. Our proposal is simply an extension of existing trap mechanisms that would enable efficient software handling of coherence. We believe that the cost of these modifications and additions in terms of area, power and engineer-years, are small enough to justify incorporating them in processor designs spanning multiple market segments. The goal being a system design cost and a system time-to-market that is on par with the processor design time. We call our proposal a “trap-based memory architecture,” or simply TMA. This system architecture presents a completely binary-transparent view to the application. All necessary software support is contained in system software. While a software coherence scheme introduce some extra overhead when compared to a hardwired coherence implementation, it also comes with some very attractive properties. The hard limit on the number of nodes in the system is removed, protocol bugs can be fixed with software patches and it is possible to change coherence schemes in a trivial manner.

Since area and complexity are scarce resources in a processor design, this chapter evaluates a minimalistic TMA implementation based on simple coherence trap support and an incoherent commodity interconnect (e.g., InfiniBand [52]). Our proof-of-concept implementation, called TMA Lite, is evaluated using a detailed full-system simulation of systems formed from chip multiprocessors, where each core is a superscalar, multithreaded, dynamically scheduled out-of-order processor. We find TMA performance to be competitive with a highly optimized hardware-based distributed shared memory (DSM) system, with a vastly higher degree of complexity and a significantly longer design time.

Even though trap-based coherence can be used across the entire application domain, we believe that it is especially well suited for system designs targeting the high-performance computing market. This is partly because the recent trend towards cluster-based systems where TMA can provide shared memory and partly because the data access pattern regularity, often displayed by scientific codes, can be exploited by optimized coherence schemes.

The core of the trap-based memory architecture is to detect fine-grained coherence violations in hardware, trigger a coherence trap when one occurs and maintain coherence by software in coherence trap handlers. The detection and handling of coherence violations can be implemented with a minimal amount of extra hardware support by exploiting the trap mechanisms already existing in modern microprocessors.
A system based on coherence traps can be implemented at a fraction of the cost of the corresponding large-scale hardware DSM system, while offering similar performance. Our approach also removes the limit on the number of nodes, since the coherence protocol is implemented in software and all directory state resides in main memory.

5.2 Processor Support

This section describes load and store trap support and discusses some implementation issues. A coherence trap is in many ways similar to a page miss in a software-managed TLB. When a coherence violation is detected, a trap is signaled and the effective address of the faulting load or store is forwarded to the trap handler. The coherence protocol code in the coherence trap handlers are based on the DSZOOM system [93] and are further described in section 5.3.2. The coherence trap handler then fetches valid data and obtains read/write permission. After completing the coherence action, the faulting instruction is retried.

5.2.1 Load Support: “Magic” Value Sentinel

In the TMA Lite system, all coherence units in state invalid store a predefined “magic-value” as independently proposed by Scales et al. [105] and Chiou et al. [24]. Read misses can therefore be detected by comparing the value returned by each load to this predefined value and trigger a read miss coherence trap whenever a match is found. Note that a load coherence trap does not always indicate a coherence miss. This might happen if the loaded value is (and should be) the magic value. These situations have been shown to be rare and are simple to detect and handle within the protocol code [105].

5.2.2 Store Support

The simplest form of store permission support would be to trap on all stores. This strategy has the drawback of sometimes taking traps when the node already has store permission. Ideally, a coherence trap should only occur when the node does not have store permission. In this study, we model these two “extremes” and a hardware implemented store support mechanism based on the previously proposed software-based write permission cache [128]. If an address tag is present in the WPC, it implies that the node already has write permission for the coherence unit it represents. Hence when using a WPC, the check for store permission must only be made when a store misses in the WPC.

89
The hardware WPC offers binary transparency to the system. It can be seen as a small (e.g., 8 or 16 entries) virtually indexed, virtually tagged cache, which is looked up for each store. When the effective address of a store is not present in the WPC, the store is marked as faulting. The faulting instruction will then invoke the write-miss trap handler, which will obtain permission and fill the WPC.

5.2.3 Implementation Issues

While many optimizations can be implemented to speed up the coherence trap handling, this chapter focuses on an implementation with the bare minimum set of hardware changes. The TMA implementation relies on the existing exception-trap vector mechanism as defined by SPARC-V9 [123] and uses reserved entries in the trap vector for the coherence-trap protocol code. Register spill is avoided by using the alternate global registers [123]. That is, no state has to be saved as long as no context switch occurs. The WPC structure can be manipulated through ASI-mapped registers, requiring only a new ASI from an ISA perspective.1

We believe that the load coherence trap can be implemented in many modern processor designs with negligible cost in area and complexity. However, the target processor must be able to take a trap when the read value is returned. That is, a load instruction is not allowed to be committed/removed from the reorder buffer (ROB) until the value is returned. Another alternative could be a checkpoint-based design [61]. Similar techniques have also been described by Qiu et al. [89] and Cain et al. [15]. Because a load trap only needs to be detected before the load is committed, not before the data is forwarded to other dependent instructions, it should be possible to implement without affecting the critical path.

The WPC can be implemented as a small cache accessed in parallel with the TLB2 or as part of the TLB (similar to the RS/6000 TLB design [87]). We model the former and believe that the WPC-based store trap can be implemented in many modern processor designs with negligible cost in area and complexity. Since the WPC is accessed in parallel with the TLB, the WPC access should be off the critical path and not affect the cycle time nor the single thread performance (when turned off). Since our WPC implementation is virtually indexed and virtually tagged, the TMA Lite system can only provide extra scalability on the application level. An implementation using physical addresses, on the other hand, could provide real system level scalability and allow operating systems to boot across multiple nodes.

---

1Address Space Identifiers [123]
2In the case of both a TLB miss and a WPC miss, the TLB miss takes precedence and the WPC miss will be triggered again upon retrying the instruction after the TLB miss has been serviced.
5.3 Performance Evaluation

This section describes our simulation methodology, simulator framework, benchmarks, results and the architectures modeled.

5.3.1 Simulation Methodology

We use the Simics full-system simulator [79] extended with an out-of-order processor model, memory hierarchy and a system interconnect model [122]. We simulate a SPARC-V9 system running an unmodified Solaris 9 operating system. Our processor model is based on the Simics Micro Architectural Interface that guarantees correctness leaving timing modeling to processor and memory system models.

Core and Chip Model: We model a dual-core CMP where each core is a superscalar, dynamically scheduled, 2-way multithreaded out-of-order processor. The processor model is implemented in the following way. Both threads fetch 8 instructions from a dual-ported instruction cache. The scan-stage scan instructions in the fetch buffer until a predicted instruction falls outside the fetch buffer. As long as the program execution follows the fall through path, no fetch bubbles are generated. When a branch is taken, a one-cycle fetch bubble is generated before the scan unit is able to redirect the instruction fetcher. The two threads share branch-target buffer (BTB) and branch predictor, but have separate return address stacks (RAS). Instructions are selected from the dual fetch unit based on the COUNTER fetch policy and inserted into the 8-way superscalar decode-rename stages. After register renaming, instructions are put in the issue queue. Instructions may be issued and executed out of order, but are committed in order. A round robin policy is used between the threads to select which instructions to commit.

Our memory hierarchy simulator models the latency and bandwidth of three levels of lockup-free caches per chip, where the second and third level is shared among cores. We model first-level write-through caches, while the second and the third level caches both implement a write-back strategy. The L3 latencies were chosen to model on-chip tags and off-chip data. Table 5.1 shows simulated chip parameters, which were chosen to resemble a somewhat scaled down Power5 design.

System Configuration: Table 5.2 shows simulated system parameters. We evaluate our system using both a 4-node and an 8-node configuration, where each node consists of a single chip. The 4-node system is built from dual-core chips and the 8-node system from single-core chips. However, all cores contain 2 hardware threads regardless of system configuration. Hence, there are a total of 16 threads per system for both configurations. We model 16 threads in both configurations to avoid application scalability interference in our results.
<table>
<thead>
<tr>
<th>Processor</th>
<th>2-way SMT, single- or dual-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency</td>
<td>3 GHz</td>
</tr>
<tr>
<td>Pipeline Stages</td>
<td>12</td>
</tr>
<tr>
<td>Fetch/Issue/Retire Width</td>
<td>16/6/8</td>
</tr>
<tr>
<td>Instruction Window</td>
<td>256</td>
</tr>
<tr>
<td>Store Buffer</td>
<td>32 entries per thread</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>32KB 2-way, 4 MSHRs, 2 cycle hit</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>64KB 2-way, 4 MSHRs, 2 cycle hit</td>
</tr>
<tr>
<td>L2 Shared Unified Cache</td>
<td>1MB 16-way, 16 MSHRs, 11 cycle hit</td>
</tr>
<tr>
<td>L3 Shared Unified Cache</td>
<td>8MB 16-way, 16 MSHRs, 81 cycle hit</td>
</tr>
<tr>
<td>L1/L2/L3 Block Size</td>
<td>64 bytes</td>
</tr>
<tr>
<td>Memory Latency</td>
<td>200 cycles, load-use</td>
</tr>
<tr>
<td>TMA Lite Store Support</td>
<td>16-entry hardware WPC</td>
</tr>
<tr>
<td>TMA Lite Load Support</td>
<td>Magic-value comparator</td>
</tr>
</tbody>
</table>

Table 5.1: Simulated chip parameters.

<table>
<thead>
<tr>
<th>4-node Configuration</th>
<th>Dual-core chip (4 threads) per node</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-node Configuration</td>
<td>Single-core chip (2 threads) per node</td>
</tr>
<tr>
<td>Interconnect Bandwidth</td>
<td>3 GB/sec per link</td>
</tr>
<tr>
<td>Network Topology</td>
<td>Fully connected</td>
</tr>
<tr>
<td>Remote Memory latency</td>
<td>600 cycles, load-use</td>
</tr>
</tbody>
</table>

Table 5.2: Simulated system parameters.

Our nodes are fully connected to each others with a high-bandwidth, low-latency interconnect. Our network supports two different modes: a hardware-coherent and a non-coherent mode. The hardware-coherent mode supports the coherence messages used by the hardware-only system. While the non-coherent mode support mechanisms for put, get, and atomic operations to remote nodes' memories, similar to InfiniBand [52] or Sun Fire Link [113]. Remote operations are accomplished without interrupting remote processors.

**Interactions with Solaris:** To make the trap handling as realistic as possible, we use reserved trap types in the SPARC-V9 instruction set to implement our coherence traps. We have applied a binary patch that modifies the corresponding trap vector entry in Solaris 9 with our coherence protocol code.

The simulated load sentinel comparator and all three store-permission checks modeled can signal load and store instructions as faulting. A coherence trap is taken when the instruction marked as faulting reaches the commit stage. When a coherence trap is signaled, it is handled just
as a normal trap and the protocol routines are executed just as any other
trap handler instructions in the pipeline of the cycle-accurate simulator,
consuming pipeline resources and polluting the caches.

TMA Lite relies on the operating system (OS) for detecting when a private
page moves to shared state (handled with the virtual memory subsystem). The
OS is also responsible for local memory mappings, network interface mapp-
ings and replication of pages in shared state. The coherence trap handlers are
used to maintain fine-grained coherence between the "private copies" of all
shared pages.

5.3.2 Simulated Protocols

This section describes the hardware-only and the software-based coherence
protocol (executed by coherence trap handlers) modeled in this chapter.
Even though DSZOOM's software-based invalidation protocol is already
introduced in section 4.2.4, in this section we restate some of the details for a
better comparison with the hardware-only protocol.

Hardware Protocol (HW): The hardware-only protocol is a highly optim-
ized non-blocking MOSI directory protocol. The on-chip coherence agent,
responsible for node-to-node coherence, has dedicated directory memory and
uses a fully mapped bit vector to keep track of sharers [70].

Software Protocol (SW): The software-based protocol is a port of the well
tested DSZOOM protocol [128, 93], a distributed synchronous directory co-
herence protocol [44, 93, 128]. Our software protocol assumes a high band-
width, low latency cluster interconnect, supporting fast mechanisms for put,
get and atomic operations to remote nodes' memories, such as InfiniBand [52]
or Sun Fire Link [113]. The protocol further assumes that the write order be-
tween any two endpoints in the network is preserved. These network assump-
tions make it possible for a trapping processor to get read/write permission
without remote interrupt- and/or poll-based asynchronous protocol process-
ing [11, 93].

A processor that has detected the need for global coherence activity (by a
coherence trap) can lock a remote directory entry and independently obtain
read and write permission. The protocol allows several threads in the same
node to perform protocol actions at the same time. The invalidation-based pro-
tocol (MSI) directory is located in the nodes' memories. Effective address bits
of memory operations determine the location of a coherence unit's directory
location, i.e., its "home node."

To reduce the number of accesses to remote directory entries caused by
WPC-fill traps, each node has one byte of local state (MTAG) per global co-
herence unit indicating if the coherence unit is locally writable. The directory
is only consulted if the MTAG indicates that the node currently does not have

93
write permission to the coherence unit. Since a home node can detect with local memory accesses if it has write permission (the directory is located in its local memory), it does not need any extra MTAG state.

2-hop Write Miss Example

Figure 5.1 illustrates the protocol activity caused by a 2-hop write miss to coherence unit A and a 3-hop read miss to coherence unit B. The 2-hop write-miss coherence activity is started when a processor in node1 writes to coherence unit A. The transactions are marked A1-A3. State transitions for coherence unit A and B are shown below each node.

HW: The coherence agent starts the coherence activity when the store operation is at the head of the store buffer and the miss is detected. The coherence agent sends a read-exclusive-request (A1 in Figure 5.1 (a)) to the home node (node0). The home node responds with permission and data (A2). Finally, node1 sends an acknowledgment (A3) to the home node (off the critical path).

SW: The coherence activity is started since the requesting processor in node1 does not have write permission, and hence, takes a coherence trap. The requesting processor acquires exclusive access to A's directory located in node0's memory (indicated by the remote atomic operation A1 in Figure 5.1 (b)). When the directory is locked, it retrieves the data from the home node with a remote get operation (A2). To end the coherence activity, the requesting processor releases and updates the directory with a single remote put operation (A3), which is off the critical path.

3-hop Read Miss Example

Figure 5.1 also illustrates protocol activity caused by a 3-hop read miss to coherence unit B. The coherence activity starts when a processor in node3 reads coherence unit B. The transactions are marked B1-B5.

HW: The coherence agent in node3 detects the coherence miss and sends a read-request to the home node (B1 in Figure 5.1 (a)). The home sends an intervention to node2 (B2) which has coherence unit B in state modified. Node2 responds with data and permission information to node3 (B3) and enters owner state. Node3 ends the coherence activity by sending an acknowledgment (off the critical path) to the home node.

SW: The coherence activity starts with a load coherence trap triggered by the magic-value sentinel. The requesting processor in node3 locks the directory entry and determines the identity of the node holding the data (B1 in Figure 5.1 (b)). The data happens to reside in node2, in a modified state. A second remote atomic operation (B2) to node2's MTAG structure disables write permission on that node. The data is fetched with a remote get operation (B3). Node2's MTAG and the home's directory are then released and updated. These two put operations (B4 and B5) are off the critical path.
Figure 5.1: Protocol examples for a 2-hop write miss and a 3-hop read miss.
Hardware/Software Protocol Discussion

While handling the coherence protocol using trap mechanisms on the requesting processor has advantages, it also has some drawbacks. On most processors, handling a trap is associated with a significant pipeline disruption. For example, in our model, when a coherence trap is signaled, all the trapping hardware thread’s instructions are flushed from the pipeline. The trap handler has to be fetched and the instructions have to fill the pipeline frontend before the actual execution of the coherence routine is started. When the node has read/write permission, the faulting instructions have to be retried. That is, the instruction has to be fetched again and walk through the entire pipeline.

Our software protocol has two major drawbacks compared to its hardware-only counterpart. First, it needs more inter-node hops to solve coherence misses (see Figure 5.1). Second, the software store protocol is not capable of hiding store misses, like store buffers often can in traditional hardware protocols.

Deadlock Avoidance

All WPC-related deadlock issues are solved by the coherence run-time system. A processor’s WPC entries have to be released at synchronization points, context switches, at failures to acquire MTAG/directory entries and at thread termination. To avoid deadlocks caused by user-level flag synchronization, we also periodically flush all WPC entries. This flush can be accomplished by the operating system, for example, on timer interrupts or with a dedicated timer-based trap mechanism. Throughout this study, WPC entries are flushed every 10,000 cycles.

5.3.3 Benchmarks

The benchmarks that are used in this chapter are the well-known workloads from the SPLASH-2 benchmark suite. Data set sizes for the applications studied can be found in Table 5.3. Since we evaluate our system on a cycle-accurate simulator, we had to restrict our evaluation to a subset of the SPLASH-2 benchmarks. The long turnaround time, up to a week of simulation per data point, is also the reason why we use small working set sizes.

It is not clear however, which system that is favored by small working sets. In the hardware-only system, the large third-level cache will capture most of the data set, and hence, almost no extra coherence traffic is generated because of limited sharing space. The TMA Lite system, on the other hand, might get a slightly increased WPC hit rate, but be penalized by the increased rate of synchronization events that forces WPC flushes in order to avoid deadlocks.
<table>
<thead>
<tr>
<th>Program</th>
<th>Problem Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>fft</td>
<td>64k points</td>
</tr>
<tr>
<td>lu-c</td>
<td>512x512 matrices, 16x16 blocks</td>
</tr>
<tr>
<td>lu-wc</td>
<td>512x512 matrices, 16x16 blocks</td>
</tr>
<tr>
<td>radix</td>
<td>256k integers, radix 1024</td>
</tr>
<tr>
<td>water-naq</td>
<td>512 molecules, 3 time steps</td>
</tr>
<tr>
<td>water-s</td>
<td>512 molecules, 3 time steps</td>
</tr>
</tbody>
</table>

Table 5.3: SPLASH-2 benchmarks.

The selected programs were chosen to represent a variety of communication and synchronization requirements. For example, we use fft because of its communication-intensive behavior and radix because its store access pattern is randomized and because it has been shown to perform poorly on all-software DSMs.

We warm the caches similarly to Woo et al. [124]. All applications are compiled with a gcc-3.4.3 compiler (optimization level 3), PARMACS macros for locks and barriers are based on user-level test&test&set spin locks, Pause/Event macros are implemented with the POSIX Pthread library (only radix uses a small amount of pauses).

5.3.4 Simulation Results

We start our performance evaluation by comparing the performance of our coherence trap enabled TMA Lite system to the hardware DSM system. As shown in Figure 5.2, the performance of the TMA Lite system is on average 25 percent behind the hardware DSM. However, we will show in this section how to increase the performance of the TMA Lite system.

We have found the WPC hit rate to be critical for TMA Lite performance. We therefore establish a lower and upper bound (the two “extremes” from section 5.2.2) on the performance that can be obtained with the type of store coherence mechanisms proposed in this chapter.

The lower bound is the performance obtained with an “optimal” WPC implementation, i.e., store traps are only generated if the node does not have write permission. Conversely, we use an implementation trapping on all stores, i.e., no write permission caching is done at all, as an upper bound of the store coherence trap overhead. The lower and upper bound is shown in Figure 5.2 as trap on coherence (lower) and trap on all stores (upper). As can be seen in Figure 5.2, the high penalty of

\footnote{Note that some unnecessary cycles are spent nevertheless filling WPC entries that never will be used.}
Figure 5.2: Software and hardware protocol performance. (4-nodes, coherence unit size: 64 bytes.)
taking a coherence trap makes trapping on all stores a costly strategy. The performance of the trap on coherence bar, on the other hand, is very encouraging when compared to the hardware DSM system.

The hardware DSM system outperforms the lower-bound software for applications with a lot of coherence activity; this is for example the case for fft. On the other hand, when the amount of coherence activity is low, as in water-5, the optimal WPC performs on par with the hardware DSM system.

The performance difference between the TMA Lite system and the optimal WPC comes from the fact that the WPC has to be filled, even though the node might already has permission. It has earlier been shown that the WPC hit rate for radix is very poor [128]. This is the reason why the WPC performs similar to the trap on all stores strategy for this particular application.

We find that we are able to get a significant performance effect from a simple 16-entry WPC compared to the trap-all strategy. The performance obtained by the optimal write permission caching makes more advanced WPC structures look very promising. This is however beyond the scope of this dissertation.

**Trap Protocol Breakdown**

In order to quantify and understand the different components of the software protocol, we have broken down the execution time into five different parts: deadlock avoidance handling (described in section 5.3.2), load protocol, store protocol, WPC fill and the remaining application execution denoted as other protocol cycles. We have divided the store handling into two different parts (store protocol and wpc fill) to highlight the impact of the overhead caused by WPC fills when the node already has permission. As can be seen in Figure 5.3, the overhead caused by WPC fills is non-negligible. For all benchmarks except fft, more time is spent filling the WPC than on the actual store protocol processing. The relative WPC cost is extra large in applications with little coherence activity.

We find that on average 31 percent of the overall execution time is spent on coherence actions, Out of which 7 percent is deadlock avoidance, 31 percent is load miss handling, 18 percent is store miss handling and 44 percent is WPC fill handling. To understand the cost associated with coherence traps further, we have quantified how much time is spent entering and exiting the trap handlers as well as the time spent waiting for remote get operations, lock handling and the overhead of remaining protocol handling.

Figure 5.4 shows a breakdown for each of the various forms of traps, again, we separate WPC fill processing from the store protocol. Each bar is divided into the following five parts: trap enter, remote get, lock acquire, other protocol cycles and trap exit.

**Trap Enter:** Represents the cost of filling the pipeline with the correspond-
Figure 5.3: TMA Lite execution time breakdown. (4-nodes, coherence unit size: 64 bytes, 16-entry WPC, magic-value comparator.)
Figure 5.4: TMA Lite protocol breakdown. (4-nodes, coherence unit size: 64 bytes, 16-entry WPC, magic-value comparator.)
ing trap handler. It accounts for the total number of cycles from when the trap is taken until the first instruction in the trap handler is committed. A fetch unit optimization, similar to the one proposed in the SMTp proposal [23], can almost remove this part of the trap overhead.

**Remote Get:** This part is the total number of cycles spent waiting for remote get operations to finish. That is, when the load or store protocol needs to get valid data. This part is, of course, highly dependent on the remote memory access time. A longer (shorter) remote memory latency will increase (decrease) this component of the trap handling.

**Lock Acquire:** Since the TMA Lite coherence protocol is based on software handlers, atomic updates of the directory state located in main memory are needed. The lock acquire part corresponds to the total number of cycles waiting for remote and local lock acquire operations while inside the protocol code. Note that the lock acquire part of the wpc fill bar corresponds to local lock operations to the MTAG structure.

**Trap Exit:** This part corresponds to the total number of cycles from the actual commit of the retry instruction ending the coherence trap routine until the pipeline is refilled and the instruction that originally caused the coherence trap to be taken is committed. A more advanced trap mechanism that would not require a pipeline flush when a coherence trap is taken may decrease or completely eliminate this pipeline-refill time [53, 132].

**Other Protocol Cycles:** The rest of the cycles of the coherence trap handlers are contained in the other protocol cycles part. These are typically cycles spent manipulating directory bits, setting up RDMA actions or updating the WPC. Note that the WPC is updated also in the store protocol bar.

Figure 5.4 shows that the pipeline fill (trap enter) and the pipeline refill (trap exit) parts do not affect the performance that much. For a more disruptive trap mechanism, this part may be significantly higher. Most of the cycles in the wpc fill bar are consumed updating the WPC (the other protocol cycles part). It is interesting to note that the overhead caused by local lock acquires to the MTAG structure is not severe, despite the costs associated with atomic operations. The store protocol bar, on the other hand, consumes a lot of its time waiting for lock acquires to finish. The reason why this takes so much time is because the directory often are located on remote nodes, and hence, the remote atomic fetch-and-set operation takes much more time than its local counter part. fft, radix and water-nall have a significant amount of write misses where a remote get of valid data is needed.

Most of the cycles in the load protocol breakdown are consumed waiting for exclusive access to the directory (lock acquire) and while getting valid data (remote get) from remote nodes. Because these two kinds of operations are very costly, the trap overhead (trap enter and trap exit) as well as bit manipulation overhead (the other protocol cycles part) are very small.
Almost all cycles spent inside the deadlock avoidance handler are consumed while manipulating the WPC and releasing the locally cached locks.

**Coherence Unit Size Scaling**

Software-based coherence protocols have the advantage of making protocol changes trivial (see chapter 4, section 4.3), enabling a degree of flexibility difficult to implement in hardware within complexity bounds. In this section we present the performance of the TMA Lite system while varying coherence unit size from 64-256 bytes.4 We have broken down the execution time into protocol components, to expose how each component scale with coherence unit size. As Figure 5.5 shows there are significant performance gains that can be achieved by tuning the coherence unit size. Except for radix, all applications benefit from some coherence unit scaling.

The greatest performance improvement is seen for fft and lu-c, whose load and store protocol cycles scale very well with coherence unit size. The WPC-fill cycles are also decreased, especially for lu-c. The performance improvement for the water applications comes from a 54 (60) percent WPC miss rate improvement (when going from 64-128 bytes) for water-n (water-s), and hence, a large reduction in WPC-fill cycles. This is shown by the wpc fill part in Figure 5.5.

Both fft and lu-c outperform the hardware DSM system when a coherence unit size of 128 bytes is used. fft (lu-c) is more than 26 (15) percent faster than the hardware DSM system when a coherence unit size of 256 bytes is used. However, it is well known that not all applications scale with coherence unit size. lu-no, for example, suffers from false sharing when a coherence unit size larger than 128 bytes is used. This is indicated by the store protocol component increase observable in Figure 5.5. The reason why the other cycles part is not constant between TMA Lite configuration while varying the coherence unit size is because extensive trapping affects both threads in the pipeline. More resources are freed/used, the fetch unit has to be redirected, etc.

The right-most bar in Figure 5.5 shows the average TMA Lite system performance when handpicking the best coherence unit for each application. The TMA Lite system performs within one percent of the hardware DSM on average. More advanced protocol optimizations will further improve TMA Lite performance.

**Node Scaling**

In order to test the scalability of the coherence trap proposal, we have modeled both 4- and 8-node hardware DSM and TMA Lite systems. We have chosen to use a total count of 16 hardware threads in all our configurations to avoid

---

4The TMA Lite system allows the coherence unit size to be set at application launch.
Figure 5.5: Normalized TMA Lite performance. (4-nodes, coherence unit size: 64-256 bytes, 16-entry WPC, magic-value comparator.)
application scalability interference in our results. The 4-node configuration consists of four dual-core chips while the 8-node configuration consists of eight single-core chips. The 8-node system will have less node locality than its 4-node counterpart, which will lead to more coherence activity.

By comparing the relative 4- and 8-node performance difference between the DSM and the TMA Lite system, we can get an indication of how the TMA Lite proposal scales compared to the DSM system. As can be seen in Figure 5.6, the TMA Lite system scales similarly to the hardware DSM system, except for water-s. For water-s we have noted an increase in load and store protocol cycles, however not enough to explain all of the performance degradation.

5.4 Related Work

There is a wide range of options for hardware/software trade-offs for implementing coherent shared memory already mentioned in section 1.2.4. Page-based software-only DSM systems [74, 18, 3, 116, 11] rely on the existing TLB hardware to detect permission violations. However, TMA is targeting a much more fine-grained coherence granularity than a virtual memory page. An appealing option in a page-based software DSM is to extend the functionality of the TLB to support fine-grain load and store access control bits. For example, the RS/6000 [87] provides a single access control bit for every 128-byte segment. This would allow the fast trap mechanisms to be already present for handling TLB faults to also handle the coherence faults. This scheme has the advantage of an earlier detection than most other schemes, including our read support. However, such modifications of the TLB are less practical for TLBs with large or mixed page sizes.

The Blizzard-E [109] system demonstrated the use of the existing trap mechanisms to handle read coherence violations. By corrupting the ECC bits of coherence units in a state invalid, an ECC trap is generated for all read misses. Then once the ECC trap is triggered, the trap handler obtains read permission and corrects the ECC code. While having the advantage that no "false" read coherence traps are generated, the scheme have the drawback of requiring a network interface capable of setting this special ECC value in remote nodes5 if a synchronous coherence protocol to be used. While this approach may be suitable for read checks, it is less practical for writes. In the Blizzard-E system as well as in other fine-grained systems such as Shasta [105], Sirocco-S [107] and DSZOOM [93], permission checks were instrumented in the application code thereby breaking the application binary transparency targeted by the trap-based memory architecture.

5A capability currently not supported in e.g. the InfiniBand standard.
The "informing" memory operation proposal [48] is closest to our goal: to keep added hardware complexity low. While this is a very interesting approach for software coherence, the out-of-order processor implementation is non-trivial. The coherence protocol assumes permission for all data in the L1 cache. However, an informing load operation can update the cache and later be squashed from the pipe (due to control speculation or a preceding exception) leaving the cache with data that have not been checked for permission. Horowitz et al. permits the speculative informing load operation to update the cache on a miss and makes sure that this data cannot be read by another informing load operation by extending the data forwarding mechanism [48]. Shared L1 caches are likely to complicate this scheme further.
Sammanfattning (Summary in Swedish)

Smart att få datorer att arbeta tillsammans

Den som någon gång använt en söknation på Internet har med ganska stor sannolikhet använt tjänster från ett datorkluster. Sådana kluster har som största fördel att de är billiga att konstruera. Vissa typer av program kan däremot inte köras effektivt på dagens kluster. Den här avhandlingen föreslår ett sätt att ta fram kluster som klarar just detta.

Ett datorkluster är två eller flera datorer sammankopplade i ett nätverk som med hjälp av klusterprogramvara arbetar tillsammans som en enhet. Ett bra exempel på ett datorkluster är en av världens största sökmotorer, Google, som består av över 10,000 mer eller mindre vanliga PC-datorer. Datorkluster är ingen ny företeelse, de har funnits sedan mitten av 70-talet då de mest användes för att få tillförlitliga datorsystem.


Visa typer av program, som t.ex stora tekniska beräkningar, har dock krävt stora och dyra servrar; det vill säga har inte gått att köra effektivt på billiga kluster. Den här avhandlingen presenterar en ny typ av klusterprogramvara, kallad DSZOOM. Syftet med forskningen är att kunna köra samma program i ett DSZOOM-kluster som i en större och mycket dyrare server. Möjligheten finns då att köpa flera mindre datorsystem och koppla dem samman i ett kluster. Den totala kostnaden för det systemet skulle på så sätt bli mycket lägre. DSZOOM-kluster är alltså ett datorsystem som består av flera datorer (beräkningsnoder)
sammankopplade med ett enkelt men snabbt nätverk, samt själva DSZOOM-klustermjukvaran som med hjälp av speciella minnesprotokoll styr hur delning av programdata mellan olika noder kommer att ske via nätverket.

Arbetet som redovisas i den här avhandlingen har utförts inom ett samarbetsprojekt mellan Uppsala universitet (Institutionen för informationsteknologi), datorföretaget Sun Microsystems och ett institut som stödjer samarbete mellan den akademiska världen och industrin kallat PSCI (Parallel Scientific Computing Institute).


Förutom DSZOOM systemet, presenterar avhandlingens första del också flera algoritmer för att effektivt synkronisera processorer i ett multiprocessorsystem, tex i en server. En ny hierarkisk egenskap hos dagens och framtidens datorsystem identifieras och används i de nyutvecklade synkroniseringsalgoritmerna för att göra dessa så snabba som möjligt. Moderna högpresterande multiprocessorer är oftast uppbyggda av flera beräkningsnoder med flera processorer på varje nod. Processorer som ligger i samma nod brukar kunna kommunicera mycket snabbare med varandra än processorer mellan olika noder. Det kan t ex vara upp till tio gånger snabbare för en processor att kommunicera med en grann än med andra processorer i systemet. De föreslagna synkroniseringsalgoritmerna tar hänsyn till detta och skapar medvetet högre nodlokaltitet än normalt för både programdata skyddat av ett så kallat minnesläs och själva läset. På grund av detta har processorer i samma nod större chans att erhålla läset när en av grannarna blir klar med sitt arbete (så kallad kritisk sektion) och släpper läset till resten av systemet.
Acknowledgments

I remember one of our first research group meetings five years ago. We have just finished a discussion about our future research plans and Professor Erik Hagersten concluded with “det här kommer att bli skoj” — this is going to be fun! This was fun, really fun! At this particular moment, I think that this is a great opportunity to thank you, so I thank you Erik! Erik’s world-class technology insight, encouragement, and invaluable mentoring have followed me all these years.

I would also like to thank all brave members of the Uppsala Architecture Research Team (UART) for their constructive criticism, and primarily, excellent time. The UART team would not exist without you :-). I thank you all: Lars Albertsson, Erik Berg, Martin Karlsson, Henrik Löt, Dan Wallin, and Håkan Zetter.

Oskar Grenholm and Niklas Ekström, students from Uppsala University, contributed to this work during their final six months of study at UART. Many people at Sun Microsystems, Inc. have provided valuable insights and criticism, I am grateful to Karin Hagersten for her extraordinary English language support.

Many thanks to all colleagues at the IT-department who worked hard to provide a nice work and research environment. I would also like to thank Sverker Holmgren and the Division of Scientific Computing at Uppsala University for a privileged use of their Sun WildFire and 15K machines.

Finally, I would like to thank my friends and my family, in particular my parents and my sister. You have always been there for me! And finally finally, a special thanks goes to Dr. Ana Jovanović who somehow managed to listen to the entire RH lock story.

This research is supported in part by Sun Microsystems, Inc., and the Parallel and Scientific Computing Institute (PSCI), Sweden.
References


119


