2004

Directory-based Cache Coherence in SMTp Machines without Memory Overhead using Sparse Directories

Anton Kiriwas
University of Central Florida, akiriwas@gmail.com

Directory-based Cache Coherence in SMTp Machines without Memory Overhead Using Sparse Directories

By

Anton P. Kiriwas

A thesis submitted in partial fulfillment of the requirements
for Honors in the Major
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term

2004

Abstract

As computing power has increased over the past few decades, science and engineering have found more and more uses for this new found computing power. With the advent of multiprocessor machines, we are achieving MIPS and FLOPS ratings previously unthought-of. Distributed shared-memory machines (DSM) are quickly becoming a powerful tool for computing, and the ability to build them from commodity off-the-shelf parts would be a great benefit to computing in general.

In the paper entitled, "SMTp: An Architecture for Next-generation Scalable Multi-threading", Heinrich, et al. presents an architecture for a scalable DSM built from slightly modified machines capable of simultaneous multi-threading (SMT). In this architecture SMT-based machines are connected together via a high-speed network as DSMs with a directory-based cache coherence protocol. What is unique in SMTp is that the cache coherence protocol runs on the second thread in the SMT processors instead of running on an expensive, specialized memory controller. The results of this work show that SMTp can sometimes be even faster than dedicated hardware. In this thesis I intend to present the work on SMTp and extend its capabilities by removing the necessity for memory based directory backing by leveraging the work of Wolf-Dietrich Weber in sparse directories. The removal of the directory backing store will free a large percentage of main memory for work in the system while having only a minor impact on the cache miss rate of applications and overall system throughput.

Acknowledgments

There are many people that helped me every step of the way through this thesis. Without these people everything in here would still be a gobbled mess inside my head instead of the coherent thoughts I am presenting here. First I would like to thank Dr. Mark Heinrich for believing I may be able to benefit the ongoing computer architecture research at UCF. He's helped me understand the current architecture research and prepared me for my up-coming graduate studies. I am not sure I would have the confidence to further my education and pursue my masters and doctorate degrees without the confidence I have obtained while writing this thesis. I would also like to thank Scott Russell for his help in working on the simulator, in writing protocol code, and helping to bring the Sparse project to life.

I would also like to thank my parents for always pushing me to do more than is required, and to be the best I can possibly be in anything I do. There were times when other things in life tried to take priority over classes and research; without my parent's encouragement and guidance this thesis may not have happened.

Finally and most of all I would like to thank Jeannie Dellutro. She was there on the late nights – the many late nights – to make me hot tea or coffee and simply to keep me company. You had better stick around to help me with all my doctoral work. I'll need you then more than ever. I love you.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction and Background

To introduce a paper covering a large number of topics as a computer architecture paper must, we are forced to break the seemingly unrelated topics up and introduce them independently. In the remainder of chapter 1, I will introduce various topics that, when brought together, form the basis of my thesis work. I have chosen to overlook some of the basics of computer architecture for topics that are more advanced and more importantly, pertinent to my paper. Terms unique to a specific system, protocol or design are explained where necessary and properly cited in the references section of this paper. The goal of this paper is to explore the underlying and surrounding work necessary for SMTp with sparse directories and to explain its implementation, benefits and possible future.

1.1 Simultaneous Multithreading Technology

Simultaneous Multithreading, which is often referred to simply as SMT is an obvious extension of the superscalar processor, and a good balance between the computing power of single-threaded processors and a multi-core chip. While other systems, such as superscalar or multithreaded superscalar, or even on-chip multiprocessors may provide speedup from instruction-level parallelism or thread-level parallelism, none can adapt dynamically to both on the fly as well as SMT.

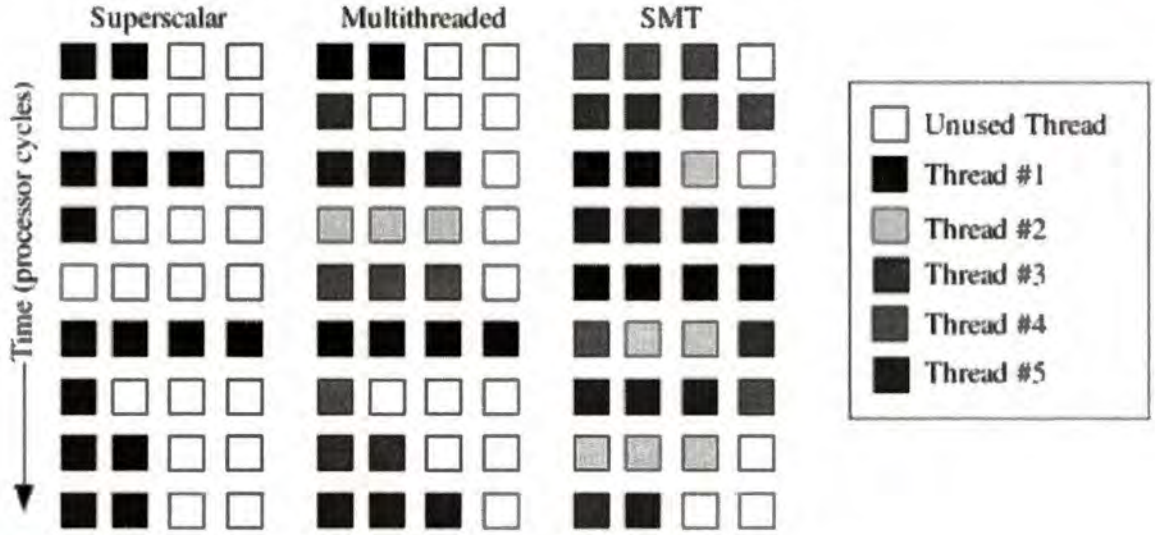

Figure 1: Comparison of functional unit partitioning in competing architectures

In Figure 1 we see that superscalar is able to use multiple instructions from the same process. This scheme helps the CPU's instructions per cycle, but in terms of efficiency of resource usage, the superscalar processor suffers from both horizontal and vertical waste. The multithreaded processor is able to take instructions from different threads each cycle, this allows for long latency operations as well as minimizes vertical waste. As the figure shows, the SMT processor is able to take advantage of both ideas and take instructions from any thread each cycle; this minimizes both horizontal and vertical waste [7].

SMT is implemented with minimal changes to an existing superscalar processor model. The instruction fetch is modified to be able to fetch more instructions per cycle to facilitate the greater execution unit usage efficiency. The other modification is the replication of certain parts of the processor. No other real additions are needed, only replication of architectural units already found in superscalar processors. The items replicated are the execution units, the hardware context registers (such as the program

counter), and some per-thread mechanisms for pipeline flushing, register renaming tables, instruction retirement, trapping, precise instructions and subroutine returns. Also added is a larger cache to deal with the larger strain on the cache system by two simultaneous threads [7].

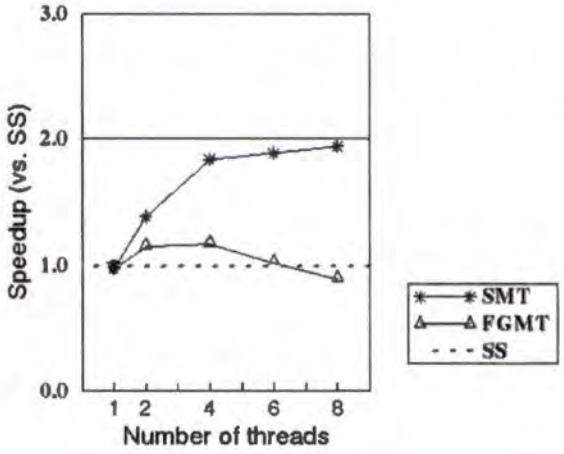

Figure 2: Speedup for multiprogramming Applications

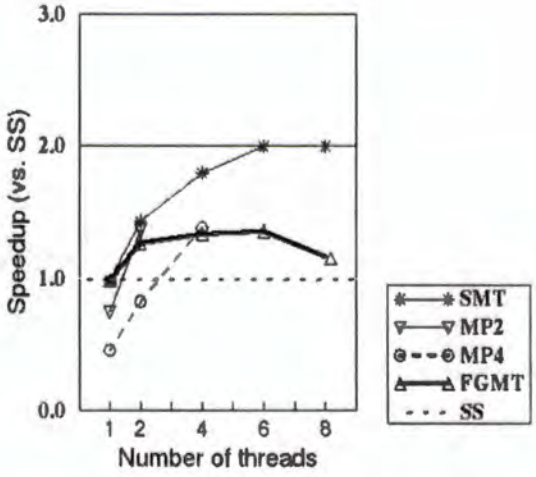

Figure 3: Speedup for parallel applications

The results in Figure 2 show that the speedup for SMT continues to scale up to eight threads, while the performance of fine-grained multithreading fell after four threads. Figure 3 shows that SMT again had better a speedup than fine-grained multithreading,

MP2 and MP4 (multiprocessors with two and four processors and comparable resources to the SMT case). Note that these results are normalized to the performance of superscalar.

Further results from [7] show that SMT has better performance than single-threaded superscalar. In their tests, superscalar's instruction throughput averaged 2.7 per cycle out for multiprogramming workloads and 3.3 per cycle for parallel workloads out of a maximum of eight. SMT achieved a 2.3 times faster execution time on the multiprogramming workload and 1.9 times faster on the parallel workload. SMT was even able to obtain better speedups than multiprocessors because of multiprocessors fixed hardware partitioning between hardware contexts versus SMT's ability to dynamically utilize resources between threads.

1.2 Multiprocessing

As we reach the limits of speed for a uniprocessor machine [8] the need for parallel processing becomes apparent. Even as CPUs become faster and faster, some now reaching almost 4.0 GHz, we are getting less and less of a return. While the CPUs speed increases exponentially (a trend which researchers are not sure can continue – though history tells us will), the speed of main memory has only increased linearly, if even that fast. Single threaded processing could be reaching its limits and parallel processing is paving the way to faster computation.

Table 1: Memory Size Increase vs. Memory Speed Increase

Year	Size	Cycle Time
1980	64 Kb	250 ns
1983	256 Kb	220 ns
1986	1 Mb	190 ns
1989	4 Mb	165 ns
1992	16 Mb	145 ns
1995	64 Mb	120 ns

As parallel processing systems were being developed, two kinds of shared memory systems arose. The first is centralized shared-memory architecture [11]. This design features multiple processors connected to one or more levels of cache that connect to main memory and the I/O system through a shared bus. This design does not scale well and other similar designs have been built replacing the shared bus with a multiple buses or a switch. Even with this, the centralized design does not scale above a few dozen processors [11]. In a shared bus system such as this, the main memory is considered to be equal distance from each processor. This reference has to do with memory access time and not with physical distance. Because of this, these systems are often referred to as UMA systems, which stands for Uniform Memory Access systems.

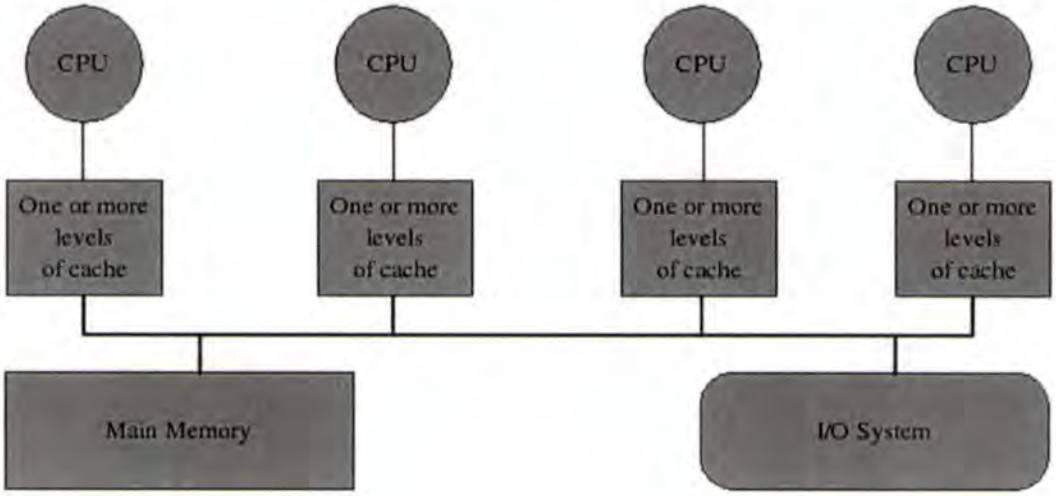

Figure 4: Centralized shared-memory multiprocessor

The second type of multiprocessor has the memory decentralized. In this architecture, each node is now made up of a processor, an I/O unit, and some portion of the overall system's main memory. Connecting these nodes together is a high-speed high-bandwidth network. The benefits of this design over a centralized memory system are that scaling the system is cheaper since it only requires a new node to be added to the system and that the latency for a node-local memory access is much lower. However, this system also has the downside of having higher latencies for node-to-node memory accesses and a more complex model for accessing memory that is discussed further in Section 1.3.1. Because there is no predefined network structure of specifications for a DSM, we have no way of knowing memory access times ahead of time. In fact, in systems such as these the distance from the processor to the memory is said to be non-uniform. This again refers to the memory access time and not physical distance. Because of this, these systems are referred to as NUMA systems, which stands for Non-Uniform Memory Access systems. In the case where a system is said to be cache

coherent (by implementing some cache coherence scheme) we call these systems CC-NUMA for Cache Coherent Non-Uniform Memory Access machines.

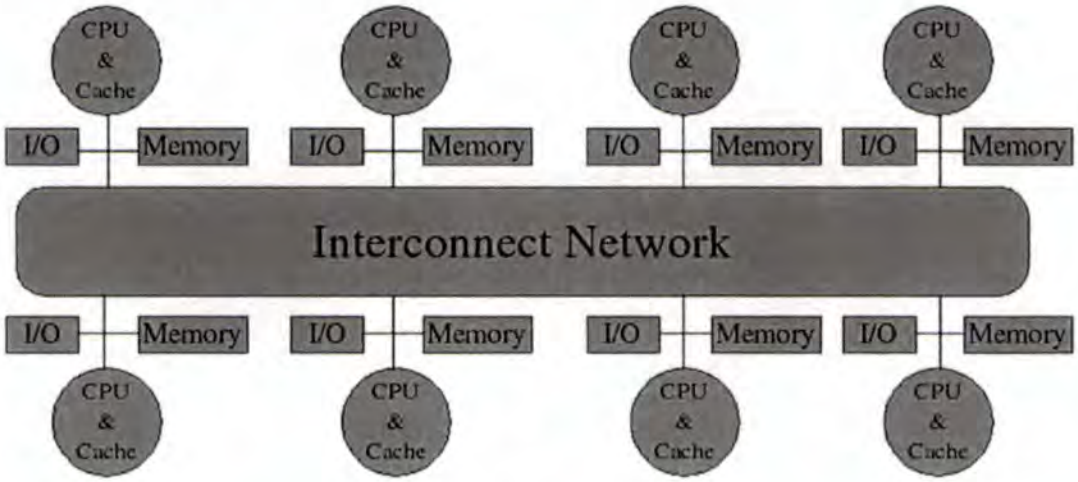

Figure 5: Distributed-memory multiprocessor

At first it appears as though the CC-NUMA machine simply adds more complexity given the interconnections needed and the cache coherence protocols. However, the advantage of NUMA machines is their scalability. UMA machines are only able to scale to between 4 and 6 processors before their shared bus becomes a bandwidth bottleneck and contention-induced latencies soar. NUMA on the other hand can scale up to hundreds or even thousands of processors depending on which interconnect is chosen and which cache coherence protocol (if a CC-NUMA system) implemented.

1.2.1 Message Passing Systems

Although I've used the acronyms NUMA and CC-NUMA interchangeably, there is no requirement that NUMA systems be cache coherent. Some systems distribute the address space between the processing nodes, restricting the access of one node to only its

own address space. Systems referred to as distributed address space machines by definition do not have a shared memory space, and thus force user applications to explicitly pass messages between nodes to solve problems in parallel. These systems typically make use of precompiled libraries to facilitate message passing, such as message-passing interface (MPI) [16]. These systems have the advantage of being highly scalable since each node can still only recognize its own local address space and complicated coherence system is needed. However, this advantage is quickly overshadowed by the shift in programming paradigm required to program them.

The first commercial parallel processing machines that were marketed in the 1980s usually supplied the buyer with a specialized, proprietary message passing library. This meant that the user did not have to spend valuable time programming a library and could instead start immediately coding a parallel solution to the problem at hand. However, it also meant that each machine had its own message passing interface and porting code from one machine to another was basically a complete re-writing of the code. Instances of these proprietary libraries include VERTEX which ran on nCUBE hypercube machines, NX which ran on the Intel iPSC family of multiprocessors EUI on the IBM SP-2, and CMMD on Thinking Machine multiprocessors [1].

The obvious decision to overcome the problem of proprietary interfaces is the same today as it was back then: standards. There was more than one attempt to create a standardized message passing interface including PICL, P4, PARMACS (PARallel MACroS), CHIMP (Common High-level Interface for Message Passing) and Express. In April 1992, the MPI (Message Passing Interface) Forum was setup to establish a

universal standard. In May of 1994, the final draft of the standard was released, only to be revised in June of 1995 [1].

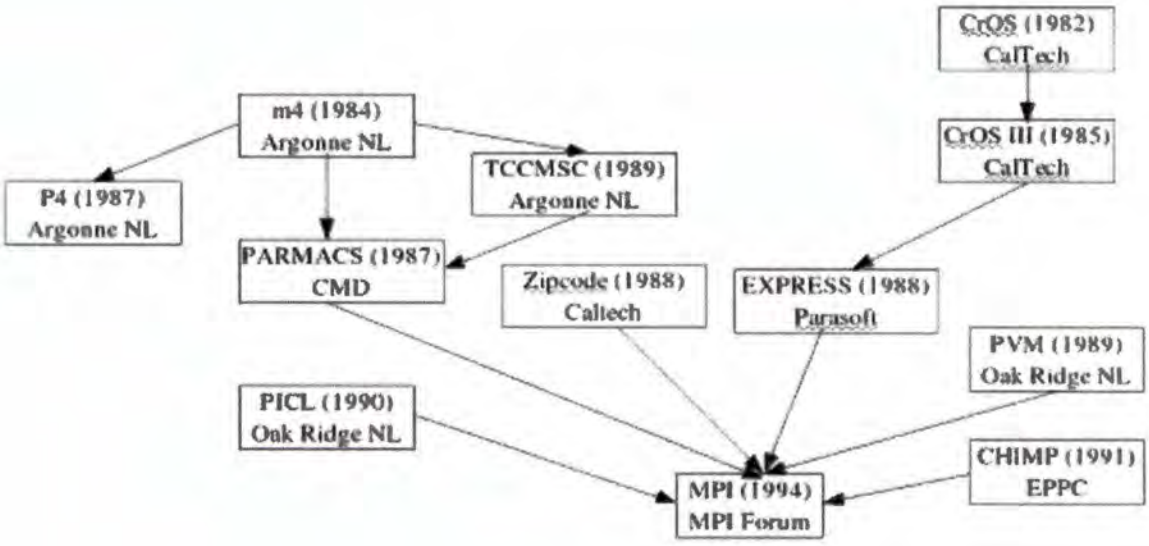

Figure 6: Map of Message Passing Systems

Although message-passing evolved, and by the mid-nineties was standardized and for the most part code could be transferred between systems, the job of solving a problem using parallel processing still relied on the programmer's ability to efficiently make use of the system's resources. This was, unfortunately, unfamiliar to many who had programmed on shared-bus multiprocessors as discussed in section 1.2 or the single address space of the uniprocessor. Before this however, a system evolved that was somewhere between message-passing systems and fully-distributed shared memory machines. In his survey paper, Raina describes computer systems classified as shared virtual memory that often use a larger block of memory than a cache line as the basic unit of sharing. There is usually no hardware assistance in this type of system. Instead, the shared address space is maintained by specialized page fault handlers in the kernel of the

system [19]. These systems are detailed further in [15]. They present two methods for the kernel; a centralized manager and a distributed manager. The centralized manager is a simpler design that unfortunately causes a bottleneck at the manager if there are too many page faults. There are two flavors of the distributed manager. The fixed distributed manager alleviates this bottleneck but spends longer on average locating the owner of a page than the dynamic distributed manager. Their experiments show promising results even when applications share data in a fine-grained fashion (smaller than a page size) as long as most of the data was read-only. This system and systems like it (and descended from it) form the middle ground between message passing and distributed shared memory machines.

1.3 Distributed Shared Memory Machines

As discussed above, there is a gray area where the actual message passing in a message-passing machine becomes transparent to the application programmer by the use of specialized library calls and running kernels. However, true distributed shared memory machines usually implement the shared memory space using specialized hardware. These systems began appearing in the 1970s and continued throughout the 1980s. There were three big name systems in this time, they were the Carnegie Mellon Cm* [21], the IBM RP3 [17], and the BBM Butterfly [4]. As mentioned in section 1.2, these were all considered NUMA systems because there memory access times were non-uniform and dependant on which node in the system contained the data needed.

1.3.1 Cache Coherent Systems

The cache coherence problem is a classic problem that must be dealt with when building a multiprocessor architecture. The source of the problem actually stems from two separate advances in computer architecture. The first is caching, which helps a uniprocessor machine with the "memory-wall" problem [23], and the second is multiprocessing, which allows programs to use multiple processors to increase the overall throughput of the system. Both of these additions help a system to have better performance, but when combined lead to the cache coherence problem.

For a system to be consistent, any memory location read by a processor should return the value that was last written to that memory location, regardless of the caching systems used by either processor. In a shared memory system, this can be a problem because multiple caches may be caching a value and in the case where one writes a value to the cache, that value is not communicated to the other caches. This violates the principle of consistency. To solve this problem, systems have been built that simply do not allow multiple processors to cache the same blocks of memory, others implement a shared-bus system that allows the communication of the needed information between the caches, while other larger systems have implemented directories to keep track of the caching done for each processor and maintain consistency. These methods are explored further in chapter 4.

1.4 Structure of Thesis

The remainder of this thesis is organized to first introduce the previous work that this thesis is based on. This includes the Stanford FLASH Multiprocessor, the simulation environment for the FLASH Multiprocessor, and the architecture, SMTp (simultaneous

multiprocessing with a protocol thread). The thesis will then dive into various forms of multiprocessing including both message passing and cache coherence systems. Various cache coherence systems and protocols are discussed, after which the thesis explains the introduction of the sparse directory [9] idea into SMTp to alleviate the large memory overhead usually associated with directory-based cache-coherent shared memory multiprocessors.

Chapter 2: Previous Work

2.1 Stanford FLASH Multiprocessor

FLASH Stands for Flexible Architecture for SHared memory. It was an architecture developed at Stanford University, designed as a vehicle for experiments in cache coherence and message-passing protocols [13]. As seen in Figure 7, The FLASH machine is made up of interconnected FLASH Nodes, consisting of an R10000 MIPS Microprocessor, 256 MB of DRAM, and a MAGIC chip that controls communication both within and between nodes.

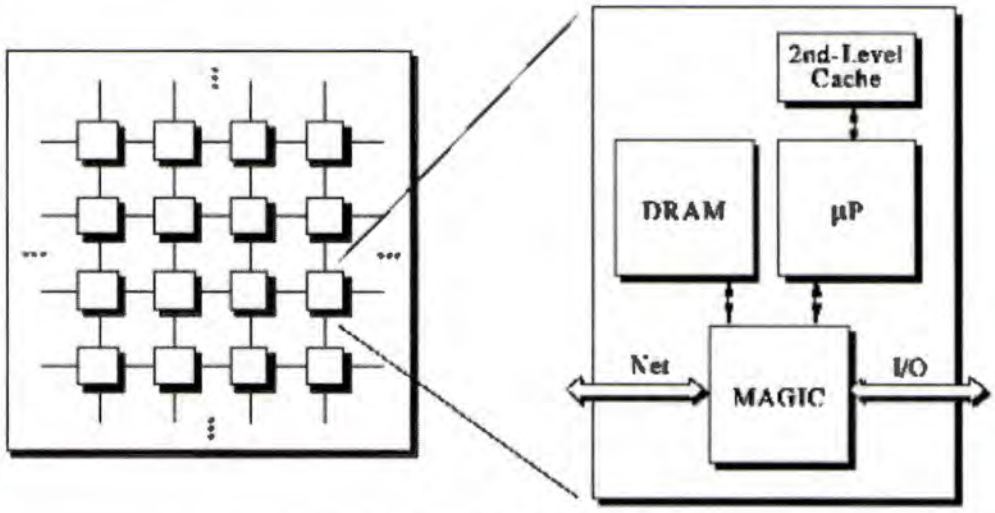

Figure 7: Zoomed in FLASH Node

The original MIPS R10000 (sometimes referred to as the MIPS R10K) implements the MIPS IV instruction set architecture, and is a 64 bit dynamic 4-way superscalar microprocessor. It is able to fetch and decode 4 instructions per cycle, speculatively execute beyond branches with a four-entry branch stack and can issue

instructions out of order, while maintaining sequential memory consistency by graduating instructions in-order. It includes five independent, pipelined, execution units including a non-blocking load/store unit, two 64-bit integer ALUs, and IEEE 754-1985 floating point units. It was originally designed to have a two-way set associative data and instruction cache, each 32 kilobytes, and an external two-way set associative secondary cache [24].

MAGIC stands for Memory And General Interconnect Controller. It is a programmable node controller that is a universal interface to the processor, memory, network and I/O devices [13]. The MAGIC Chip was balances the flexibility of a general-purpose processor where all functionality being done in software, with an inflexible yet fast all-hardware implementation. Figure 8 shows a diagram of the MAGIC architecture.

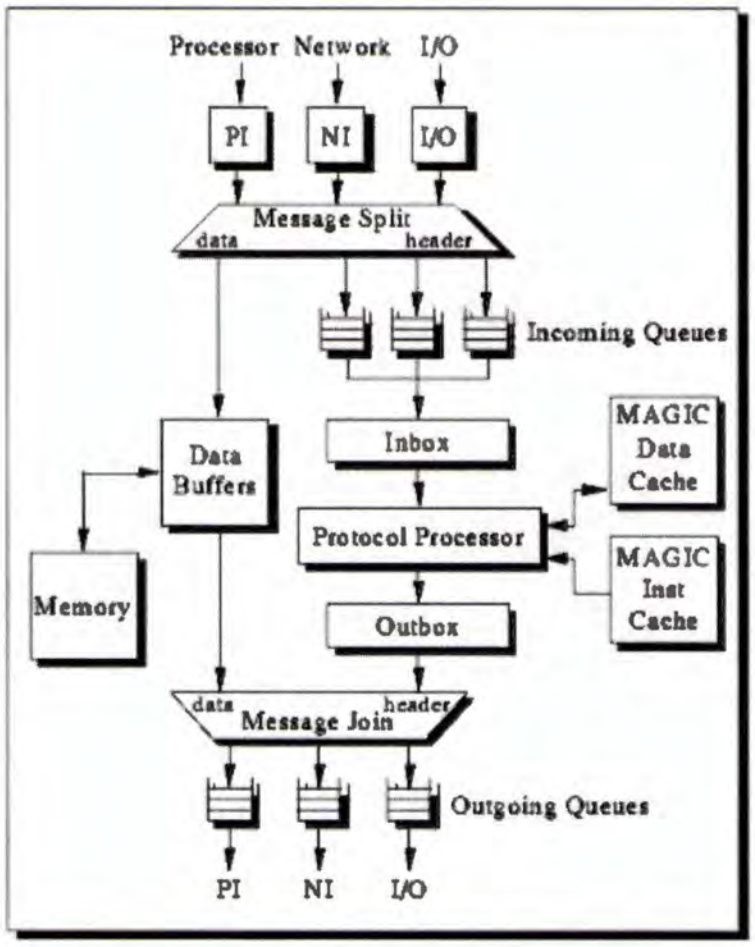

Figure 8: MAGIC Architecture

MAGIC performs two main actions. It is responsible for moving data and maintaining state information. Because these two actions are mainly independent, MAGIC exploits the parallel nature of hardware controllers and splits the message between the data transfer logic and the control pipeline. Message headers are sent to the control pipeline while message data is sent to the data transfer logic [3].

Messages arrive from one of four sources, network, local memory, or I/O. If the message contains data, it is stored in one of the 16 temporary data buffers. The number of this data buffer is used by the control pipeline to deliver the data to the proper destination. Data is kept in its buffer until it is delivered to its destination.

The control pipeline is more interesting and does the majority of the work of MAGIC. The control pipeline is both flexible, in that it can run a variety of protocols, and quick enough to be able to do most of the processing within the latency of the transfer time–that is, not to let processing time become the limiting factor in the system. To achieve the flexibility, a programmable controller is used. This controller has the standard set of RISC instructions as well as a special set of bit-field manipulation instructions for faster implementation of many of the common protocol jobs. MAGIC contains separate data and instruction caches and has special hardware units for dispatching. This means that as soon as a message has been composed by the processor to be sent out, it is placed in the outbox and the processor can immediately start processing the next message from the inbox. Because of this macro-pipelining, up to three messages can be in the system at once [13].

Overall, this allows the FLASH architecture to support a large number of cache coherence protocols in a flexible manner as well as support for message passing protocols, such as NX, MPI, Barrier, and Fetch-and-Op.

2.2 Introduction to SMTp

The idea behind SMTp, which stands for Simultaneous Multithreading with a protocol thread, is to build a distributed shared memory machine, like the FLASH Multiprocessor, without the need for a specialized memory controller such as MAGIC [6]. SMTp can implement the same cache coherence protocols as the FLASH Machine without a specialized node controller, and in some cases can work even faster. This is accomplished by running the protocol on one thread of an SMT processor. This has the advantage of being much faster than the software only implementation was discussed in

section 2.1 because an SMT processor does not have to interrupt the execution of the first thread. It can also, in some cases, be faster than an all hardware specialized memory controller implementation because we have the added benefit of now running at the system's core clock rate approximately 2.0 GHz, instead of the 1/3 processor clock rate as is the case for the MAGIC.

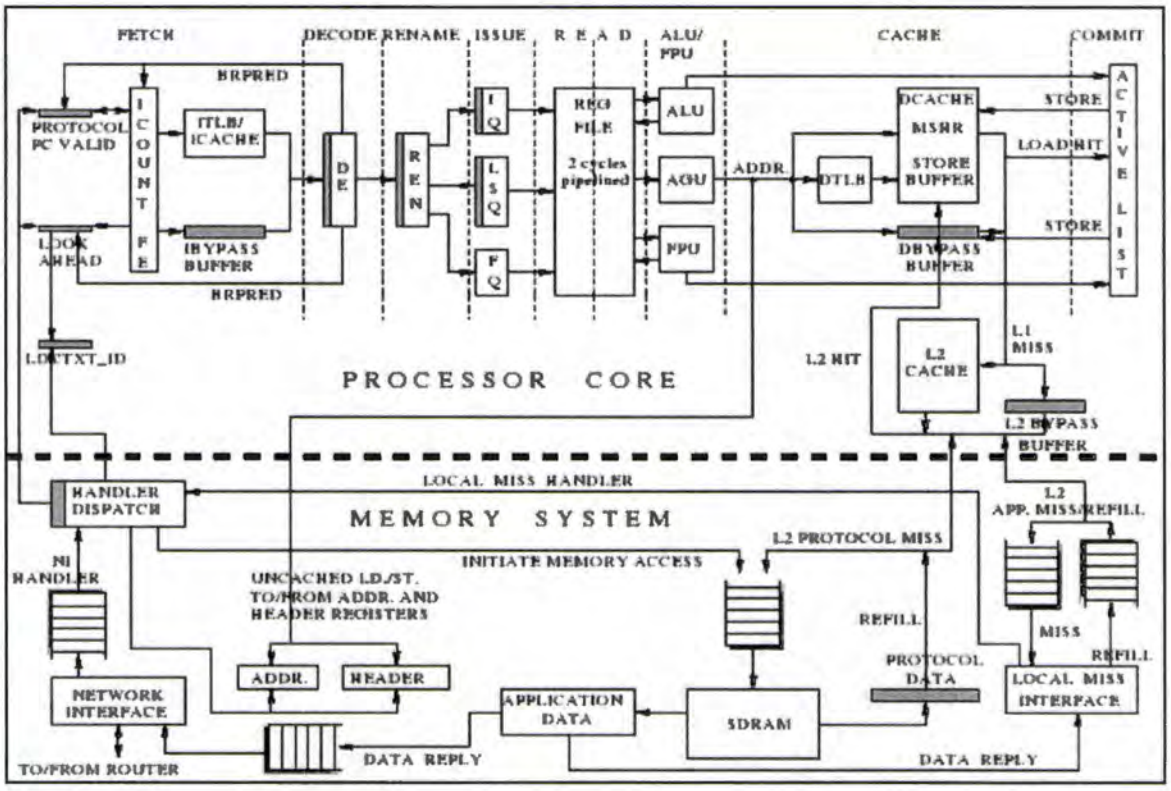

Figure 9: SMTp Node Architecture

The above diagram is adapted from [6] and shows that the SMTp architecture is split between the processor half and the memory controller half. Shaded regions in the figure show additions necessary for SMTp. The protocol thread is bound to the hardware context and not available for the operating system for context switching. The protocol thread operates in parallel with an application level 2 cache miss. The secondary

hardware context used for the protocol shares the level 1 data cache and level 2 cache with the other application threads. On a protocol level 2 cache miss, no handler is triggered (since this recursion would not make sense). Instead, it bypasses the Local Miss Interface and requests the data directly from main memory. The design also assumes a few extra instructions, some of which are already implemented in the Alpha ISA. These instructions are not necessary but do provide a minor performance improvement.

A major problem with SMTp (and thus with SMTp with sparse directories) is deadlock avoidance. In the normal SMT processor, deadlock is not a problem. However, in the SMTp design the application thread progress with a level 2 cache miss depends on the protocol thread handler to run. The protocol thread depends on shared resources between it and the application thread. This leads to possible deadlock with the application thread waiting on the protocol thread and the protocol thread waiting on a resource currently used by the application thread. The solution adopted is to reserve certain resources for the protocol thread to guarantee it will always be able to make progress. For example, if the processor has eight decode queue buffers, one is reserved for the protocol thread and seven may be used by the application thread. Because of this change, some modifications are needed to the decode and rename queue schedule algorithm since it is not purely circular FIFO [6].

Another potential deadlock that must be dealt with is a conflict between the level 1 and level 2 caches. An example of this problem is a protocol thread load/store may conflict (meaning they both map to the same cache line index) with an in-flight application load/store. The solution to this is an extra piece of hardware called a bypass

buffer, which is fully associative. On a protocol thread miss, instead of allocating a cache line, the cache system can now allocate a bypass buffer line. The bypass buffer is made small enough to be able to be accessed in parallel with the normal cache, and to handle the worst pathological case of every outstanding miss in the system conflicting. In tests, the bypass buffer was sized at 16 entries since the system only allowed up to 16 outstanding misses.

Simulation of SMTp was done to test using several SPLASH-2 applications as well as another benchmark adaptation. The table below shows the applications tested and their problem sizes. Explanations for FFT, LU, Radix-Sort, and Ocean can be found in section 3.2 with the discussion of SMTp with sparse directories simulation and testing. FFTW is a 3D fast fourier transform kernel that operates on complex double points. One modification made from the SPLASH-2 suite's set of applications was in Ocean. Ocean's global error lock in the multigrid phase was changed from lock-test-set-unlock to test-lock-test-set-unlock for more efficiency.

Table 2: SMTp Applications and Problem Sizes

Application	Problem Sizes
FFT	1 million points, blocked for DTLB
FFTW	8192 x 16 x 16 points, 32 x 32 block
LU	512 x 512 matrix, 16 x 16 block
Radix-Sort	2 million keys, radix = 32
Ocean	514 x 514 grid, tolerance of 1.0×10^{-5}
Water	1024 molecules, 3 time steps

Table 3: SMTp Simulated Processor Configuration

Parameter	Value
Frequency	2 GHz/4 GHz
Thread contexts	1/2/4+protocol thread
Pipe stages	9
Fetch Policy	ICOUNT (2 threads)
Front-end width	8
Decode queue slots	8
Rename queue slots	8
BTB	256 sets, 4-way
Branch predictor	Tournament (21264)
RAS	32 entries (per thread)
Active list	128 entries (per thread)
Branch stack	32 entries
Integer register	160/192/256
FP register	160/192/256
Integer queue	32 entries
FP queue	32 entries
Unified load/store queue	64 entries
ALU	7 (one for addr. calc.)
Integer mult/div/ latency	6/35 cycles
FPU	3
FP mult latency	Pipelined, 1 cycle
FP div latency	12 (SP)/19 (DP) cycles
Commit width	8
ITLB	128/fully assoc./LRU
DTLB	128/fully assoc./LRU
Page size	4 KB
L1 Icache	32 KB/64B/2-way/LRU
L1 Dcache	32 KB/32B/2-way/LRU
Unified L2 cache	2 MB/128B/8-way/LRU
MSHR	16+1 for returning stores
Store buffer	32
L1 cache hit	1 cycle
L2 cache hit	9 cycles
SMTp Specific	
Res. Decode queue slots	1
Res. Rename queue slots	1
Res. Branch stack slots	1
Res. Integer registers	1
Res. Integer queue slots	1
Res. LSQ slots	1
Res. MSHR	1
Res. Store buffer	1
IBypass buffer	16 lines/fully assoc./LRU
DBypass buffer	16 lines/fully assoc./LRU
L2 Bypass buffer	16 lines/fully assoc./LRU

Table 3 shows the configuration of the simulated processor for SMTp. The 2 Ghz/4 Ghz in the frequency column is in reference to the tests which show that it can be expected that the results from SMTp will continue as the speed of processors continues to increase.

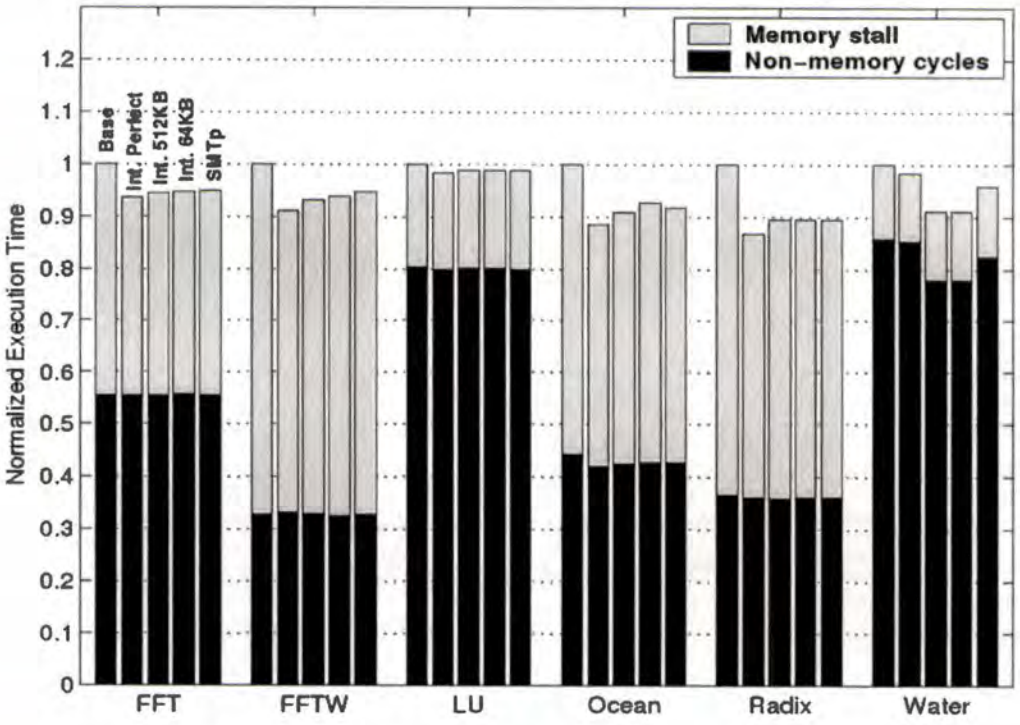

Figure 10: 16 node configuration comparison with SMTp

Figure 10, adapted from [6] shows us a performance comparison between Base, Int64KB, Int512KB, IntPerfect, and SMTp. Base models a conventional DSM machine without an integrated memory controller. Int64KB, Int512KB and IntPerfect are all systems containing integrated memory controllers and programmable dual-issue protocol processors. In this comparison, IntPerfect represents an aggressive hardware approach with the memory controller running at the speed of the processor as opposed to Int512KB and Int64KB whose memory controllers run at half processor speed. This system is probably the best representation of an all-hardware DSM. The results in Figure 10 show

us that in all six benchmarks, SMTp outperformed Base, was almost always (except for the case with Water) faster than Int64KB and often comparable to Int512KB. Overall, it was shown that in any case where there was a gap in the performance between SMTp and Int512KB the cause was data cache pollution in the level 1 cache.

2.3 Future Adaptations with SMTp

SMTp is the merging of many different systems that makes the possibility of building fast and scalable DSM machines from commodity parts very close at hand. The real power of SMTp is the power to have a fully programmable directory controller running at the speed of the processor. This means that as research in cache coherence moves forward, SMTp-based systems move forward as well. As research has shown, different cache coherence schemes work better in different situations [10], this means that another added benefit of SMTp is that scientists and engineers can run different cache coherence protocols depending on their application for the best results. This has only been possible in the past using fully programmable hardware directory controllers which are expensive and do not run at full CPU speed or by using software-only cache coherence which has proven too slow for typical use.

The SMTp paper [6] also presents adaptations and uses for the SMTp architecture in such areas as on-the-fly compression and/or encryption, active-memory address re-mapping, and load value prediction. It also discusses the possibility of implementing these protocol threads on the secondary cores of chip multiprocessors.

Chapter 3: Simulation Environment

3.1 Simulator Design and Implementation

The simulation implementation of SMTp with sparse directory is an extension to the simulator used to simulate SMTp in [6]. It is an execution-driven simulation with simulates one or more MIPS R10000 based processors, caching systems and main memory, as well as the network interconnects between the nodes. This simulator allows us to obtain detailed information on cache hit rates, TLB usage, invalidate messages, processor usage efficiency, etc., which are used in comparing different architectural changes.

3.2 Benchmark Applications

SPLASH stands for Stanford Parallel Applications for Shared-Memory. It was designed as a set of real-world applications to evaluate the performance of both actual shared memory multiprocessors as well as simulated idealized multiprocessors. SPLASH-2 made improvements and changes to the suite and is now the de-facto benchmark application suite for many shared memory multiprocessor researchers.

The SPLASH-2 Suite of applications and computational kernels is an extension from the SPLASH Suite. It is a mixture of complete applications and computational kernels that represent scientific, engineering and graphical computing [22].

3.2.1 Complex 1-D FFT

This fast fourier transform kernel implementation is a version of the six step radix-$\sqrt{n}$ algorithm described in [2]. This version has been optimized to minimize inter-

process communication. The data set consists of n complex data points that are to be transformed, and n complex data points called the roots of unity. These are organized into $\sqrt{n}$ x $\sqrt{n}$ matrices and divided such that every processing unit receives a contiguous set of rows for its local memory. Inter-process communications consists of 3 matrix transpose steps which entail all-to-all communication [22].

3.2.2 Integer Radix Sort

The exact radix sort algorithm used in the kernel is based on the method in [5]. It is an iterative algorithm in which each processor passes over its assigned keys and creates a histogram. These local histograms are accumulated into a global histogram. This global histogram is then used by each processor to permute its keys into a new array for the next iteration of the algorithm. The inter-process communication on integer radix sort is all-to-all during the permutation step [22].

3.2.3 LU Factorization

LU Factorization is also a computational kernel, like the last three sections. The LU kernel factors a dense matrix into the product of a lower triangular matrix and an upper triangular matrix. The original input matrix of size n x n is divided into an N x N array of B x B blocks where n = NB. This serves to exploit temporal locality. Elements within a block are contiguously allocated to improve special locality. To reduce inter-process communication, block ownership is assigned using a 2D scatter decomposition with the blocks being updated by the processors that own them [22]. The running time of LU is n^3 and the parallelism is proportional to n^2.

3.2.4 Ocean

An application rather than a computational kernel, the Ocean application's purpose is to study macro scale ocean movements computed from eddy and boundary currents. This is one of the applications that was improved from the original SPLASH suite. Differences from SPLASH include using a red-black Gauss-Seidel multi-grid equation solver instead of an SOR solver, representing grids as 4D arrays with all sub grids allocated contiguously and locally in the nodes that own them, and it now partitions the grids into square sub grids rather than groups of columns in the previous version. This last improvement serves to improve the communication to computation ratio for the application [22].

3.2.5 Application Problem Sizes

The application problem sizes for our tests will be the same sizes used to test SMTp in [6]. This allows us to give a proper comparison between a baseline SMTp and SMTp with sparse directories.

Table 4: SPLASH-2 Application problem specifications

Application	Problem Specifications
FFT	1 million points, blocked for DTLB
Radix Sort	2 million keys, radix=32
LU	512 x 512 matrix, 16 x 16 block
Ocean	514x514 grid, tolerance of 1.0×10^{-5}

Chapter 4: Cache Coherence Protocols

The cache coherence problem was discussed more fully in section 1.3. This chapter will present a sampling of the cache coherence protocols currently in use. It will also present some that had novel ideas that did not turn out to be beneficial. We will begin with the most basic snoopy protocols implemented on shared bus systems, and move to various directory-based cache coherence protocols. We discuss and analyze the full bit-vector, coarse bit-vector, and dynamic pointer allocation (with discussion of static implementation) protocols.

4.1 Snoopy Protocols

The simplest of all cache coherence protocols is the snoopy protocol. This sort of coherence can only be used in shared-bus systems which limits its scalability. A snoopy system rarely scales to more than 16 nodes. After 16 nodes the bus becomes a bandwidth bottleneck and more intelligent interconnects are needed. The snoopy protocol is still of value however, because many shared memory multiprocessors are made up of multiprocessor processing nodes. To achieve cache coherence within a single node, processors often implement the snoopy protocol. In a snoopy based system, each processor's cache can listen, or snoop, on the shared bus to watch for any transaction that affects that cache. If a cache sees a write to a line it currently has cached it invalidates its local copy, and any time the cache sees a read or write on the bus to a line it has the most recent copy of it responds with the data. Snoopy systems come in two flavors based on their write policy: update-based protocols and invalidation-based protocols.

4.1.1 Write Update Snoopy

Write update is sometimes also referred to as a write broadcast protocol. Write update does exactly what it says it does, instead of invalidating another CPU's copy a CPU broadcasts an update on the bus. Although write update does not strictly require it, it is useful for a CPU's cache to keep track of which other CPU's are caching data from that CPU so that write broadcasts are only sent if necessary. The table below shows a typical exchange for a write update protocol. CPU A reads location X in the memory. This is a cache miss and the value 0 is brought in and cache. Similarly, CPU B reads location X which is also a miss and caches value 0 in its local cache. Next, CPU A writes a value of 1 to location X. Because CPU B is also caching X, CPU A sends out an update on the bus and CPU B is notified and changes its cached value from 0 to 1. Finally, CPU B tries once again to read location X. Since the update changed the value, the look up is a hit and CPU B does not need to go to main memory [11].

Table 5: Write-update snoopy example

Processor Activity	Bus Activity	CPU A Cache	CPU B Cache	Memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Write Broadcast of X	1	1	0
CPU B reads X	Cache hit for X	1	1	1

4.1.2 Snoopy Invalidation Protocols

The write invalidate flavor of the snoopy protocols is by far the more popular form. As the name indicates, in this version, a cache will invalidate the other caches on a write instead of updating them. The example below shows a typical write-invalidate

exchange. Assume two CPUs, A and B, both with caches on a common bus. CPU A reads memory location X, which is a cache miss. CPU A retrieves the cache line containing memory location X (which has a value of 0) and stores it in its cache. Next, CPU B reads from memory location X which is also a cache miss and retrieves the cache line now containing value 0 from the memory and places it in its cache. Now, CPU A writes a value of 1 to memory location X, since it is cache, this value is written to CPU A's cache and instead of updates being broadcast as was the case with write update, an invalidation is sent for memory location X. CPU B's cached copy of memory location X is now invalid and when CPU B reads memory location X it is a miss and it must retrieve the new value of 1 [11].

Table 6: Write-invalidate snoopy example

Processor Activity	Bus Activity	CPU A Cache	CPU B Cache	Memory location X
				0
CPU A reads X	Cache miss for X	0		0
CPU B reads X	Cache miss for X	0	0	0
CPU A writes a 1 to X	Invalidation for X	1		0
CPU B reads X	Cache miss for X	1	1	1

While it may immediately seem as though write update is a superior protocol since it results in one less cache miss than write invalidate in the two cases shown above, write invalidate has become the predominant choice for most designs. Three main concerns when dealing with the differences in performance between write update and write invalidate are writes without reads, write update with multiword cache lines, and read/write timing issues. For the first concern, in the case where there are multiple writes to the same word without any reads in between, the write update protocol requires that

each write be broadcast on the system, whereas in the write invalidate protocol, only one invalidation happens for the first write and each subsequent write is ignored by the other processors until they need to read that word. Our second concern has to deal with the fact that in multiword cache line sizes (which is every cache currently available because of the benefits of spatial locality), each word written in a cache line requires its own write update in the write update scheme, whereas in the write invalidate scheme, the entire line will be invalidated at once. This can actually be a common problem with applications with a lot of spatial locality. A simple example is an application that begins by zeroing out an array. With write update in a 128 byte cache line size system, this would require 128 write updates, while in the write-invalidate we've invalidated the entire line only once. Finally, the update time is faster in the case of the write update protocol, since the value is updated, whereas in the write invalidate protocol, the reader's cache will be invalid and the processor must stall until the valid data can be returned from main memory.

The three points of difference listed above show there are both pros and cons to either protocol, but in systems where bus bandwidth is the limiting factor and often the bottleneck of the entire system, write invalidation protocol has become dominant choice in all but the smallest multiprocessors [11].

4.2 Directory-based Protocols

Directory based schemes all share a commonality between the different variations, and like the snoopy-based protocols, there are update and invalidate flavors of the protocol. In the most basic scheme, a directory keeps track of the state of each cache block that it has control over. These states include shared, in which one or more

processors have that block's data cache and the value is up to date in both memory and caches, uncached meaning that no processors have copies of this data cached, and exclusive which indicates that one processor has exclusive ownership and has written the block so the copy in the memory is currently out of date. This alone would take very little space and would be very convenient. However, the directory must also keep track of which processors have copies of the blocks that are shared. The easiest way to do this is to use what is called a bit-vector. If a block is marked as shared, the bit vector indicates which processors currently have it cached, and if a block is owned exclusively then the bit vector indicates which processor is the owner [11]. The reason for this bit-vector is to improve the efficiency of the directory protocol. In terms of correctness, a directory-based protocol would work even if it did not use a bit vector and instead broadcast out the invalidations to every node. This sort of protocol would have no practical use however. The bit vector allows us to keep precise sharing information so that we send invalidations only to the nodes that need them. This keeps our network traffic to a minimum, allowing systems to scale to thousands of processors.

4.2.1 Coarse Vector

The coarse-vector cache coherence protocol is a natural extension of the bit-vector protocol when the number of nodes in the system is too great for the length of the bit-vector. In this scheme, a single bit in the vector now represents multiple caches instead of a single cache as is the case for bit-vector. The coarseness in a system like this is defined as the number of nodes represented by a single bit in the vector. By this definition, the standard bit vector protocol is a special case of the coarse vector scheme with a coarseness of 1. For example, if a given system has a maximum bit-vector length

of 48 then if the system has less than or equal to 48 nodes then the coarseness is 1. If the system has between 49 and 96 nodes inclusive, then it has a coarseness of 2. A system with between 97 and 192 nodes has a coarseness of 4, etc [9].

4.2.2 Dynamic Pointer Allocation

Dynamic pointer allocation is another version of the directory-based protocol. In the full bit vector and coarse bit vector schemes above, we kept track of which groups of nodes were caching our data and which nodes were not in the most efficient way possible for all cases. The only difference was how large of a group of nodes we were tracking. Dynamic pointer allocation's fundamental purpose is to maintain precise sharing information in large machine sizes without the memory overhead of full bit-vector. A directory entry in dynamic pointer allocation serves as the head of a linked list of nodes sharing. The entries in the linked list are allocated from a statically allocated pool of memory called the pointer/link store.

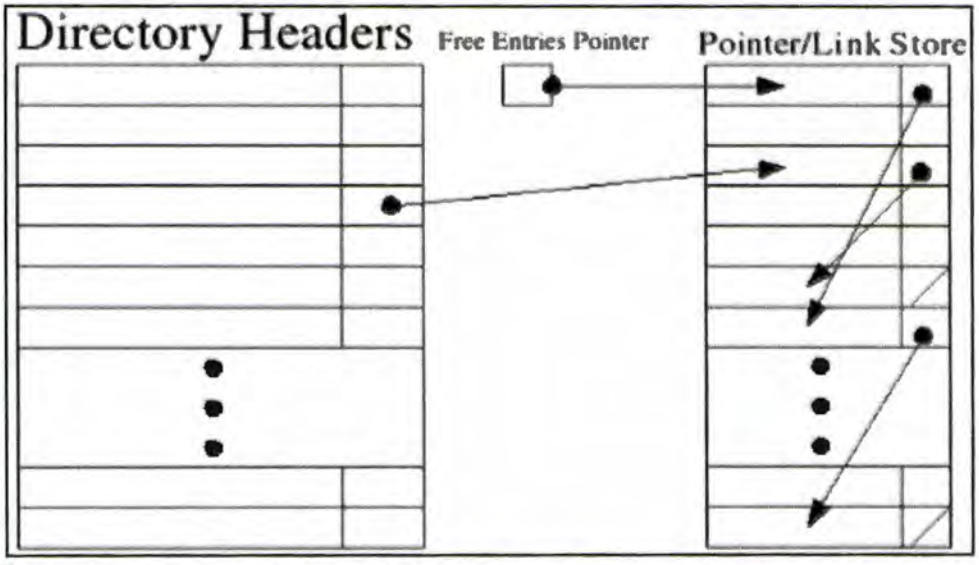

Figure 11: Dynamic Pointer Allocation Directory Layout

Dynamic pointer allocation uses the linked list of entries from the pointer/link store just as a bit-vector or coarse-vector scheme would use the presence bits in a directory entry. When the time comes in the protocol to send invalidations due to a write request, instead of sending out invalidations based on presence bits, the dynamic pointer allocation scheme traverses the linked list structure (however long it may be) and sends invalidations before reclaiming the entire list of entries back into the free entries pointer list. The advantage of the dynamic pointer allocation scheme is that it allows the protocol to scale well to a larger number of processors without resorting to using imprecise sharing information as is the case for the coarse-vector scheme. Though the linked list structure lends itself well to scaling, it unfortunately does not lend itself well to an efficient hardware implementation, especially with the needed consideration for cases in which the limited amount of pointer/link store entries are all taken [20].

Other pointer schemes have been presented such as the limited pointer scheme [9]. This scheme allocates a limited number of elements which point to the nodes caching the memory block for that entry instead of allowing a dynamic number by way of a linked list. The problem with this scheme when compared to dynamic pointer allocation is two-fold. First, although in the average case no more than one processor is caching a block of memory, the scheme must account for the infrequent cases in which more are needed; this means that limited pointer suffers from the same inefficient use of space as the full bit vector protocol does. The second problem is the case in which the limited number of pointers has been used up. In this case, the protocol makes use of an overflow broadcast bit. When more pointers are needed than the limited number available the protocol degrades into a broadcast protocol, no longer keeping precise

sharing information, the protocol assumes all caches are currently sharing the data. In the case where there are consistently more sharers than there are limited pointers we have a very inefficient use of network bandwidth because a large number of invalidations are being sent to nodes that are not caching a particular line [9].

Chapter 5: Problem Definition

Here we will present the main problems with directory based cache coherence protocols, particularly in the SMTp architecture and what can be done to alleviate these problems. The overall problem area of SMTp and of directory-based protocols in general is the high memory overhead dedicated to maintaining the directory structure. I plan to explain this problem and describe a solution.

5.1 Problems with Directory-based Protocols and SMTp

Section 2.2 presented the SMTp (simultaneous multithreading with a protocol thread) architecture. In it we presented the protocol handlers and the small architectural additions to facilitate deadlock avoidance and also showed results proving the viability of SMTp. However, that is not to say that SMTp is without problems. SMTp suffers from a problem that distributed shared memory machines with dedicated hardware cache controllers such as FLASH [13] do not have, which is data cache pollution. Because the cache controller in SMTp is a thread in an SMT processor, it shares the data cache with the application threads. This can lead to destructive interference that is detrimental to performance.

The second problem with SMTp is actually inherent to most directory-based cache-coherent distributed shared memory machines. Because most systems' directory stores a full bit-vector representing the shared state of the local memory blocks, the size of this directory grows linearly with the amount of memory in each node. The size of this directory can be upwards of 15% of system memory in a typical system. This problem can be partially alleviated by using a larger block size which results in fewer blocks that

need a directory entry. However, making the block size larger leads to a new problem called false sharing that can lead to an increase in the number of invalidations sent in a system. Below is an example of the false sharing problem.

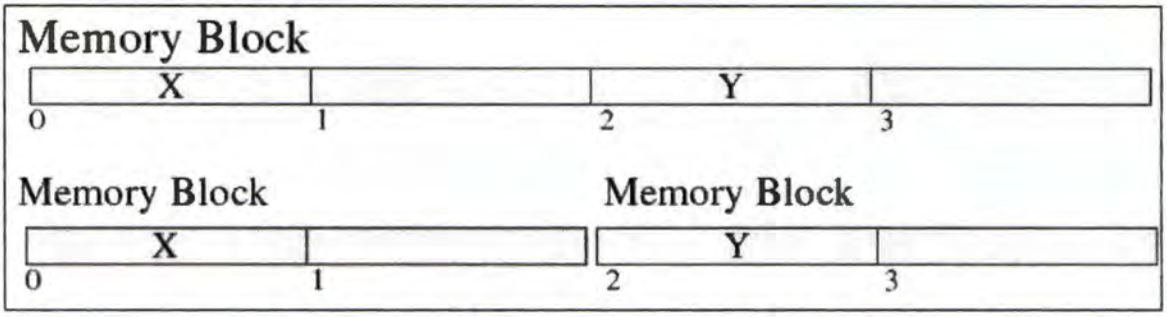

Figure 12: Different memory block size in false sharing

In the top image of Figure 12 we see a simplified example with a block size of 4 and two variables used by the system, X and Y at locations 0 and 2 respectively. In the lower image, we see a block size of 2 with the variables X and Y still in the same locations.

Table 7: Example Situation with False Sharing

Situation 1	
Action	Result
CPU A reads X	A is set as a sharer on the memory block
CPU B reads Y	B is set as a sharer on the memory block
CPU A writes X	A gets exclusive access to the memory block and CPU B is sent an invalidation

Table 8: Example Situation without False Sharing

Situation 2	
Action	Result
CPU A reads X	A is set as a sharer on memory block 1.
CPU B reads Y	B is set as a sharer on memory block 2
CPU A writes X	A gets exclusive access to memory block 1 – But since memory block 1 has no other sharers, no invalidations are sent.

The tables above show a simple example of the false sharing problem. Situation 1 reflects a memory block size of 4 while situation 2 reflects a memory block size of 2. In each situation, after CPU A and CPU B read variables X and Y, respectively (memory locations 0 and 2) we have the same results. The difference comes when CPU A writes variable X. In a directory-based cache-coherent system, this requires a request to the directory to make CPU A the exclusive owner of that block. Once this is acknowledged, invalidations are sent out to each sharer. In this case, the only sharer was CPU B and CPU B's cached copy of Y is now invalidated even though CPU A never modified Y. In situation 2, with smaller block sizes we see that although CPU A modifies variable X, no invalidations are sent because variable Y is in a different memory block. Situation 1 exhibits the false sharing problem.

Despite increasing the number of invalidations because of false sharing, increasing the block size does not completely get rid of the large amounts of memory that a directory based cache coherence system requires. Because of this, we cannot consider increasing the block size a full solution, but only a partial solution.

Chapter 6: Solution Overview

The chapters above clearly show that the work presented here is but a small modification of the work started years ago. In this chapter, we present the solution to the directory store problems show in section 5.1 by implementing the ideas of sparse directories in a novel way.

6.1 Sparse Directory

In [9], Weber, et al. proposed two solutions for solving the large backing store needed for directory-based cache coherence. One of these methods is called Sparse Directories. The idea behind sparse directories is instead of using a full directory stored in main memory, we implement a directory cache. The directory cache needs no further backing store (such as a hard drive) because we can invalidate the directory entries being stored in a given block before replacing it.

The size of the directory cache should be at least as large as the number of cache blocks. Weber found that a size of 2 or 4 times the number of cache blocks would greatly reduce the probability of contention. This possibility of contention arose from multiple directory entries mapping to the same entry in the sparse directory. This led to the implementation of a set-associative sparse directory. The ratio of main memory blocks to directory entries is called the sparsity of the directory. For example, if a directory contains 1/16 as many entries as there are main memory blocks it has a sparsity of 16. For his evaluation, Weber started with a DASH [14] multiprocessor with 16 nodes using a full bit-vector scheme which thus had a sparsity of 1 (no sparse directories). This

machine had a directory overhead of 12.5% of memory. The machine was scaled to a 256 nodes and the overhead kept constant by making the sparsity 4.

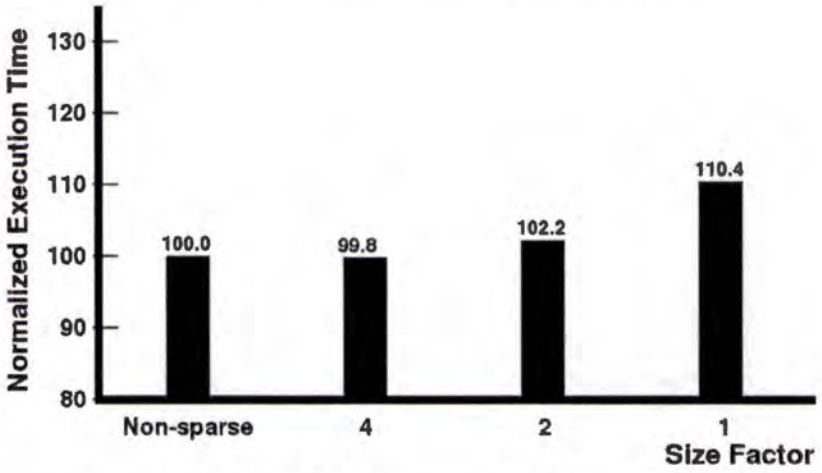

Figure 13 : Weber, et al.'s Sparse directory performance for LU

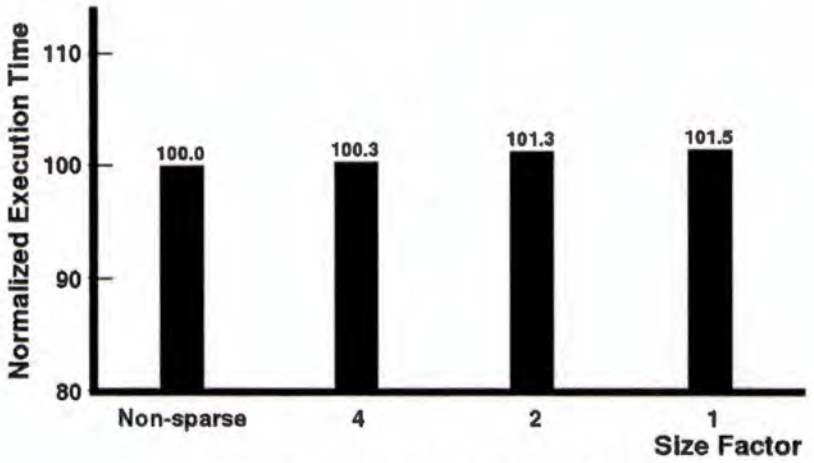

Figure 14: Sparse directory perfornance for DWF

Figure 13 and Figure 14 are adapted from [9] and show that the execution times for LU and DWF remain very similar as the sparsity of the directory is varied. The results showed that even if the directory is the same size as the processor cache, the system still performs well. This is quite important in implementing SMTp with sparse directories. The worse case, LU, still only shows a 10% increase in execution times

between a non-sparse directory and a directory the same size as the system cache. These are very promising results for SMTp with sparse directories.

6.2 Removing the Directory's Backing Store

While the work of Weber, et al. on sparse directories implements a kind of cache of the directory instead of using the entire directory, thus making the total directory overhead a fraction of what it normally is, our goal is to instead remove the entire directory altogether. The idea behind this is that while Weber, et al. created a cache of the original directory, we already have a cache for the directory, which is the level 2 CPU cache. Because SMTp already allows us to trigger protocol handlers on a cache miss we can modify the existing directory-based cache coherence protocol to use the level 2 cache as a sparse directory.

6.3 Sparse Protocol

The sparse protocol is an adaptation of the cache coherence protocol used in SMTp and explained in section 2.2. The differences include returning a blank directory entry upon level 2 cache miss in the directory space, utilizing a victim buffer in the cases where a directory entry is evicted from the level 2 cache, and finally sending invalidations when a directory entry is evicted from the level 2 cache. These three modifications and additions are explained in more detail below.

Section 6.4 explains the necessary hardware modifications for sparse directories in SMTp. One of these changes is the addition of a victim buffer for use in the level 1 and level 2 caches. The victim buffer is small enough to be looked up in parallel with the level 1 cache without increasing cycle time. In the sparse protocol, when a directory

entry is displaced from the level 2 cache, it is placed into the victim buffer. At this same time, invalidations are sent out to all nodes currently kept track of by that directory entry. The directory entry must then stay in the victim buffer until all acknowledgements come back to the home node.

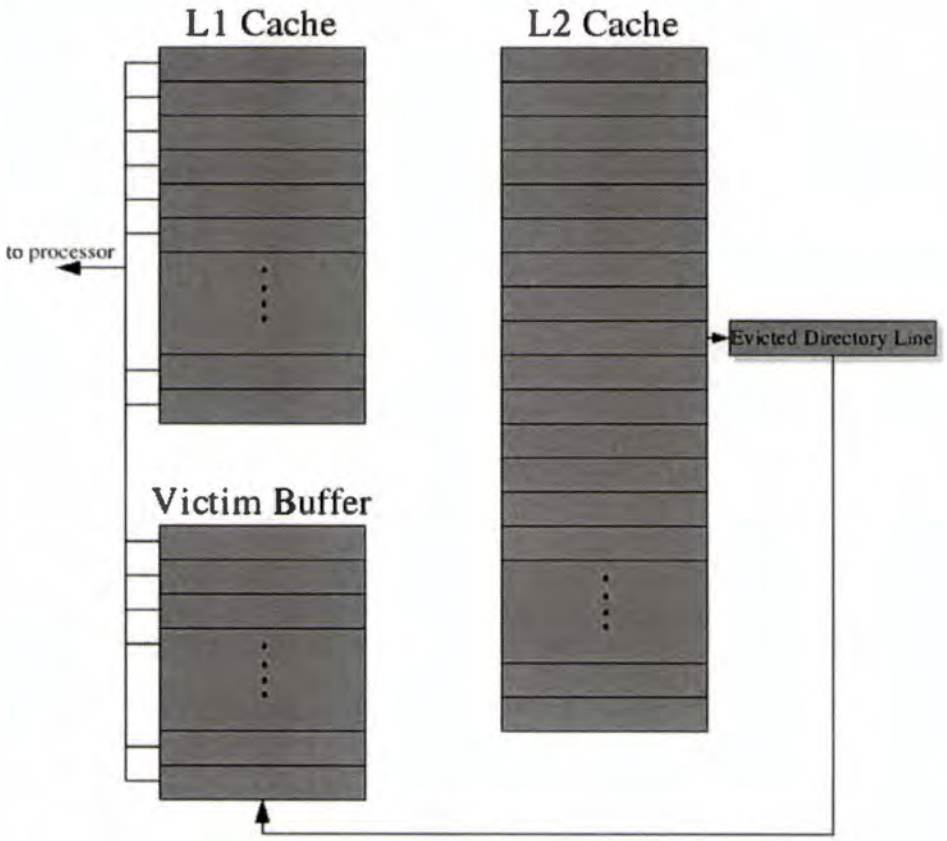

Figure 15: Victim buffer relation to L1 and L2 cache

Since the victim buffer is finite in size, if a line caching a block from the directory space is displaced and must be put in the victim buffer, but the victim buffer is full, the victim buffer stalls the processor until the needed acknowledgements come back to reclaim a victim buffer entry.

One of the features that makes the sparse directory transparent to the application programs is that if the protocol thread accesses a memory location in the directory space

(which has no actual backing store, yet is still a valid memory space), the line is in one of three places. If it is in the cache as normal, then the system continues running, if it has been evicted to the victim buffer, the parallel level 1 lookup will find it and the system will continue running. If the address is a miss in both level 1 and level 2 caches, then the level 2 cache fills the line with zeros. This effectively makes all level 2 cache lookups in the directory space automatic hits. This is consistent with what is expected since instead of being written back to main memory, the directory entry is "zeroed out" by issuing invalidations.

Table 9: Example transactions with sparse directories protocol

Action	Response
A Node Requests Memory Location X	Directory Entry for X is uncached Protocol fills cached with 0s for directory entry (Normal Protocol Transactions)
--Sequence of Normal Transactions--	--Sequence of Normal Reponses--
Local Memory Access (same index as Directory Entry)	Handler places Directory Entry in Victim Buffer Pending bit is set on directory entry Invalidations issued to all sharers Normal L2 Refill
A Node Requests Memory Location X+1	Directory Entry for X+1 is in victim buffer Because of pending bit, handler sends negative acknowledgement
Invalidation Reponses Received	Tracking bits in directory entry in victim buffer all 0 Victim Buffer entry reclaimed

The above table shows a typical exchange for the sparse protocol. When a node B makes a request for some memory location X on node A, the protocol on node A does a memory lookup for the directory entry tracking the block that contains memory location X. This is a miss in the cache which triggers another protocol handler. The level 2 cache sees that it is a directory miss and does not attempt to retrieve the data from main memory. Instead it fills the cache line with 0s. As we will see in a moment, this is the correct thing to do since we are guaranteed that there are no sharers for that cache block currently. Once the 0s are filled in the cache line is set to valid and there is now a hit in

the cache. After this, sequences of normal operations happen. This is also typical since many applications follow a pattern of memory retrieves, computation, and then memory stores. After this, the local application code has a cache miss and the cache refill chooses to victimize the cache line that currently contains our directory entry for the memory at location X. Instead of being written back to main memory, this directory entry is put into a special victim buffer whose size is small enough to allow for parallel L1 lookups. The handler then proceeds to send invalidations to all sharers registered in that directory entry. Now that the directory entry is now longer in the cache, a normal L2 cache refill can occur to service the applications L2 miss. At this point, if a node tries to make a memory request to node A's memory location X+1, node A triggers a handler that looks up the directory entry for X+1, which is the same as the directory entry for X (since we are assuming they are in the same cache block). The directory entry is found in the victim buffer and the request is denied by sending a negative acknowledgement back to the requesting node since the pending bit of the directory entry is set. This guarantees that no new sharers appear after the initial round of invalidations are sent. Once the invalidations are acknowledged from the remote sharers, the bits are all set to 0 in the directory entry and a message is sent to the victim buffer to reclaim that entry. Details on the handler are discussed below.

To implement the sparse protocol in the SMTp machine, new protocol handlers are needed. For the single-processor case we need only one handler for a working system. We call this handler PILocalDirectoryPutX. This handler is triggered when a protocol directory cache line is evicted from the level 2 cache and placed into the victim buffer. The victim buffer is explained further in section 6.1. After the directory cache

line is evicted into the victim buffer, the handler then loops for each directory entry in the cache line. After loading the directory entry, we take an action depending on what the state of that directory entry is. If the state is pending, then we leave the pending bit set, set the reclaim bit and write the directory entry with a normal store. If the directory entry is marked dirty, we allocate a buffer and send a PI_DP_GETX_REQ message to get the latest value, since the dirty bit indicates that we cannot invalidate whoever is currently caching this data since they have the latest copy. Once the data is returned, we issue a write back. We are guaranteed this will be a hit in the victim buffer since the PIDirectoryPutX handler can only be triggered if the cache line is in the victim buffer. In the event that the data is not returned properly, the directory entry is marked as pending and reclaim and the handler continues. Once the data is returned, it will see the directory entry marked as pending and reclaim and do the normal operation. If the pending counter in the entry is now zero, we issue a command to the victim buffer to reclaim that entry. In the case where the line is clean, we loop through each of the sharers marked for that directory entry. For each node that is sharing we issue an invalidation. The directory entry is then zeroed out, written back (which is again a hit in the victim buffer) and we continue to the next directory entry in the cache line.

Once the handler has looped through all the directory entries in the cache line, we check to see if there were any pending entries. If so, then we set the pending bit of the first entry of the cache line and exit. If there are no pending entries, then we issue a command to the victim buffer to reclaim that buffer number. The command is issued to the victim buffer through what we have called a misc. bus write. This is a write to a memory mapped register on the processor.

6.4 Necessary Hardware Modifications

The sparse directory protocol for SMTp will run with very few changes to the underlying hardware other than the changes already made to facilitate SMTp. One of these changes is in the addition of a specialized register to hold the starting address of the directory space. Although our design removes the actual directory space from main memory, we still require a starting address for "directory space" so that the caching system can trigger certain actions required only for addresses in the directory space. This register is referred to as DirectoryStart and is set during the boot process using the same misc. bus write mechanism used to reclaim entries in the victim buffer. The rest of the hardware changes necessary for SMTp with sparse directories are changes to the caching system. We have modified the level 2 cache to check the DirectoryStart register and compare that with the starting address of a cache line during a refill. If the address is greater than DirectoryStart, the level 2 cache fills the line with zeros. The reason for this explained further in section 6.3.

The second modification is the victim buffer shared between the level 1 and level 2 caches. Although technically all work on the victim buffer is done in the level 1 cache. The use of the victim buffer within the sparse protocol is also explained further in section 6.3. The victim buffer is a small, fully associative victim cache for directory entries evicted from the level 2 cache. However, because both level 1 and level 2 caches are write-back caches, the entry is actually added to the victim buffer from the level 1 cache since it must be queried for the most up-to-date value for the given cache line. The victim buffer is also looked up in parallel with the level 1 cache. This piece of hardware also compares the starting address of the cache line with the DirectoryStart register. If

the address is greater than the register value (placing the address within the directory space), then the level 2 cache allocated an entry in the victim buffer and places the line within it.

Chapter 7: Contributions

Table 10: Cache miss rates in 4 SPLASH-2 Benchmarks on SMTp

SMTp	Data Cache	L2 Cache
FFT 16p	1.46%	14.1%
LU 16p	1.24%	2.80%
Radix 16p	3.56%	14.0%
Ocean 16p	8.57%	7.0%

Table 11: Cache miss rates in 4 SPLASH-2 Benchmarks on IntPerfect

IntPerfect	Data Cache	L2 Cache
FFT 16p	1.40%	15.0%
LU 16p	1.25%	2.85%
Radix 16p	3.52%	14.6%
Ocean 16p	8.58%	7.28%

The tables above show the data cache of the level 1 cache miss rate and the overall level 2 cache miss rate for SMTp architecture for the four SPLASH-2 [22] benchmarks discussed in section 3.2. In this case, IntPerfect represents an aggressive hardware distributed shared memory machine with the memory controller running at the same speed as the main processor. Section 2.2 of this paper shows the overall results of SMTp when compared to IntPerfect, Int512KB, and Int64KB. Those results show that IntPerfect still outperforms SMTp in overall benchmark running time. This is to be expected. However, in this case we are more interested in the cache miss rates since the increase in invalidations and thus the increase in application misses is the short coming of SMTp with sparse directories. The fact that the miss rate of SMTp is favorable compared to IntPerfect tells us that the slight increase in cache misses is acceptable especially when taking into consideration that we have reclaimed the entire directory memory overhead and still maintained precise sharing information.

Table 12: SMTp/IntPerfect L2 total cache miss comparison

Benchmark	SMTp	IntPerfect	Percentage Difference
FFT 16p	1,080,301	1,048,365	2.96
LU 16p	90,734	90,680	0.06
Radix 16p	475,142	469,058	1.28
Ocean 16p	3,709,071	3,568,826	3.78

The tables above show the total level 2 cache misses for both SMTp and IntPerfect. These results give a more realistic indication of cache miss tendencies in SMTp to give us a better idea of SMTp with sparse directories performance. As you can see, the difference between SMTp and IntPerfect for FFT is only 2.96%, 0.06% for LU, 1.28% for Radix, and 3.78% for Ocean. Although we see that the total number is greater in SMTp, the percentage difference is very small, indicating we are already performing close to the performance that can be expected from an all-hardware cache controller implementation.

Chapter 8: Conclusions, Related and Future Work

8.1 Conclusions

The work done with SMTp and its extension with sparse directory shows that with only minor changes to the hardware, coupled with intelligent protocol design, we can build large-scale distributed shared memory machines from commodity parts. We also have found that with the ability to trigger handlers upon cache misses we have the ability to a context in an SMT-based processing node to do many of the things that previously required expensive and non-standardized hardware to do. One example of possible future work is using a protocol context to do rollback and recovery in systems that demand high uptimes and reliability. In [18], Prvulovic, et al. present a design for a rollback recovery system that is implemented by modifying only the directory controller. This lends itself easily to SMTp in which the directory controller is a fully programmable second context. In the future we see SMTp possibly being transformed into SMTrb (simultaneous multithreading with a rollback recovery thread).

8.2 Comparison with Alternatives

SMTp with sparse directories is not the only research that has been done to alleviate the problem of large directory overhead. In [12] a cache coherence scheme that removes the memory overhead in a unique way at the expensive of a sometimes large increase in network traffic is presented. As Figure 16

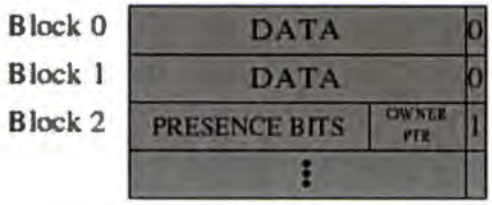

Figure 16: In-Memory Directory Memory Layout with data/dir bit

shows, data and directory entries share space in the memory. Each block in memory contains a bit signifying whether that block is data or directory data; this bit is called the data/dir bit. If the bit is 0, the block is data, if the bit is 0, the block is currently being cache and the memory now contains sharing information. A cached block is in one of four states. It is either Uncached (U) which specifies that no cache has a valid copy, Shared (S) which specifies that other processor(s) have the block cached and are read only, Shared-Owner (S,O) which specifies that this cache has a read only copy and is the owner, or Exclusive-Owner (E,O) which finally signifies that the cache has the only copy of the data and the right to modify it.

In this scheme a block starts out in main memory, of node A, as data, if a node B requests the block, the data is sent to the requesting node B and the block that previously held the data in main memory of node A now has its data/dir bit set to 1 (signifying dir) and the presence and owner pointer bits are set to signify the block is now owned by node B. This has a network traffic cost of 80 bytes; 8 for the request and 72 for data and ownership information. This is called a Read Remote Transaction. If another node C makes a request for the same block, node A forwards the request to node B. Node B then sends data and ownership to node C. The presence bits and ownership bits are now changed to reflect the new owner, node C. The most recent requester always claims

ownership to exploit temporal locality. With two requests, the total network traffic cost is 88 bytes. This transaction is called a Read Remote – Forward Remote. The third transaction is a Read Remote – Forward Local. In this case, node A caches a block from its own main memory. This is a simple case where node B requests the block from node A. Node A goes to its main memory sees that it is the owner and instead of forwarding the request, node A retrieves the data from its own cache and replies with data and ownership. This case has the same traffic cost as the Read Remote, 80 bytes. The final read case is the Read Local – Forward Remote. In this case, node A makes a request to a block in its own memory and sees that the data/dir bit is set to dir. The request must then be forwarded to the remote node caching the data. The remote node B receives the request and replies back with the data and ownership to node A. Writes in this scheme follow a similar pattern of forwarding along requests while maintaining a single copy of the data between the nodes [12].

Unfortunately, though the scheme did achieve its goal of eliminating the high overhead of the directory it did so at a high cost of network traffic. In their simulations, for 64 Kbyte caches and 32 processors the network traffic for Ocean, Radix and Raytrace SPLASH-2 benchmarks was 31%, 16% and 82% higher than the base case. When raised to 1 Mbyte, the numbers are 3%, 11% and 26% higher. Results with hardware modifications show better results. However, even with these newer, more promising results, the study still show the difficulty in removing the directory overhead with minimal impact on performance.

8.3 Future Work

A major drawback of the SMTp with sparse directories design is it is not simply a new protocol, it also has the necessity of additional hardware for the victim buffer. This victim buffer is necessary for the system to be able to continue while waiting for the invalidation acknowledgments or data requests from the sharing or exclusive nodes. However, we feel it may be possible to remove the victim buffer and use the same space that is used for the bypass buffer that is necessitated by the SMTp design.

Another piece of future work also being considered is not quite as obvious as simply removing the victim buffer. Through our research with SMTp, we have found that one of the largest hits to application performance is level 1 cache pollution due to directory entries. In an all-hardware based directory controller, there is no cache pollution due to directory entries since the directory entries stay strictly in the directory controller. Even in a programmable directory controller such as MAGIC in the FLASH architecture [13], the directory controller processor has its own cache system separate from the main processor's which prevents this kind of pollution. To help this problem of cache pollution we plan to leverage the fact that all directory space addresses can be thought of as "hits" in the level 2 cache. In SMTp with sparse directories a directory entry is either in the level 2 cache, or the cache controller fills the line with zeros and continuing, functioning as a hit. We plan to modify the cache controller to prevent directory entries from entering the level 1 cache. This will prevent the cache pollution we suffer from right now, while the impact on the protocol thread should be minimal since all directory accesses are now uniform access time based on the level 1 cache miss penalty (we can never incur a level 2 cache miss penalty).

8.4 Final Thoughts

This thesis has presented a brief history of multiprocessors and their reason for existing. It has also discussed the many topics that need to be considered in a multiprocessor such as shared-bus versus message passing versus distributed shared-memory systems. We have also looked at the technologies that made SMTp with sparse directories possible, such as the SMT processors currently making it to market. My hope is to further develop the technologies needed for SMTp with sparse directories and to explore different issues associated with it, such as scaling issues, invalidation rates and cache miss rates as well as fine-tune to the architecture to its simplest and most cost effective form.

Overall, SMTp has provided us with an opportunity to explore a variety of cache coherence protocol variations as well as other systems such as the rollback recovery system mentioned earlier in this chapter. SMTp with sparse directories will hopefully only be the tip of the iceberg for the possible uses of this new architecture.

References

[1] "A Brief History of Message Passing". EPPC Training and Education Centre http://www.epcc.ed.ac.uk/overview/publications/training_material/tech_watch/96_tw/techwatch-scalable/scalable-tw.book_8.html, 1996.

[2] Bailey, D., "FFTs in external or hierarchical memory," *J. Supercomputing.* 4(1) p.23–35, 1990.

[3] Baxter, J., "Stanford FLASH Multiprocessor". *Stanford FLASH Multiprocessor*, http://www-flash.stanford.edu/, October 18, 2004.

[4] BBN Laboratories, "Butterfly Parallel Processor." *Tech Rep*, Cambridge, MA, 1986

[5] Blelloch, G.E., et.al., "A Comparison of Sorting Algorithms for the Connection Machine CM-2", *3th Symp. On Parallel Algorithms and Architectures*, p.3-16, July 1991.

[6] Chaudhuri, M., and M. Heinrich. "SMTp: An Architecture for Next-generation Scalable Multi-threading". Proceedings of the 31st International Symposium on Computer Architecture (ISCA), p.24-135, June 2004.

[7] Eggers, S., Joel Emer, Henry Levy, Jack Lo, Rebecca Stamm, and Dean Tullsen. "Simultaneous Multithreading: A Platform for Next-generation Processors." *IEEE Micro*, p.12-18, September/October 1997.

[8] Emma, P. "Understanding some simple processor performance limits," *IBM Journal of Res. & Develop.*, 41,(3)p. 215-232, May 1997.

[9] Gupta, A., W. D. Weber, and T. Mowry, "Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes," Proceedings of International Conference on Parallel Processing, vol.I, p.312-321, 1990.

[10] Heinrich, M., "The Performance and Scalability of Distributed Shared Memory Cache Coherence Protocols". Ph.D. Dissertation, Stanford University, October 1998.

[11] Hennessy, John L., and David A. Patterson. *Computer Architecture A Quantitative Approach.* 3rd ed. San Francisco, CA: Morgan Kaufmann, p.1-833, 2003.

[12] Ho, C., H. Ziegler and M. Dubois, "In-memory Directories: Eliminating the Cost of Directories in CC-NUMAs," Symposium on Parallel Algorithms and Architectures, SPAA\x{2019}98, p. 222-230, June 1998.

[13] Kuskin, J., D. Ofelt, M. Heinrich, et al., "The Stanford FLASH Multiprocessor". Proceedings of the 21st International Symposium on Computer Architecture (ISCA), p.302-313, April 1994.

[14] Lenoski, D., J. Laudon, K. Garachorloo, W.-D. Weber, A. Gupta, J. Henessy, M. Horowitz, and M.S. Lam. "The stanford dash multiprocessor." *IEEE Computer*, 25(3) p.63-79, 1992.

[15] Li, K. and Hudak, P. "Memory Coherence in Shared Virtual Memory Systems." *ACM Trans. on Computer Systems*, 7(4) p.321-359, November 1989.

[16] "Message Passing Interface (MPI) Standard." *Message Passing Interface (MPI) Standard*. http://www-unix.mcs.anl.gov/mpi/. 1999.

[17] Pfister, G.F., et al. "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture." 1985 Conference on Parallel Processing, p. 764-771, 1985

[18] Prvulovic, M., Z. Zhang, and J. Torrellas. "Revive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors." Proceedings of the 29th Int'l Symposium on Computer Architecture, May 2002.

[19] Raina, Sanjay, "Virtual Shared Memory: A Survey of Techniques and Systems. Technical Report CSTR-92-36", Department of Computer Science, University of Bristol, December 1992.

[20] Simoni, R. and Mark Horowitz. "Dynamic Pointer Allocation for Scalable Cache Coherence Directories." Proceedings of the International Symposium on Shared Memory Multiprocessing, p.72-81, April 1991.

[21] Swan, R.J., et al "The Implementation of the Cm* Multi-microprocessor." Proceedings AFIPS NCC, p.645-654, 1977.

[22] Woo, S. C., M. Ohara, et al., "The SPLASH-2 Programs: Characterization and Methodical Considerations". Proceedings of the 22nd International Symposium on Computer Architecture, p.24-36, June 1995.

[23] Wulf, W.A., and S.A. McKee. "Hitting the Memory Wall: Implication of the Obvious." *ACM Computer Architecture News*, 23(1) p20-24, March 1995.

[24] Yeager, K., "The MIPS R10000 Superscalar Microprocessor." *IEEE Micro*, 16(2) p.28-40, April 1996.