UPPSALA
UNIVERSITET

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 217

Towards Low-Complexity Scalable Shared-Memory Architectures

HÅKAN ZEFFER

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2006

ISSN 1651-6214
ISBN 91-554-6647-8
urn:nbn:se:uu:diva-7135

Dissertation presented at Uppsala University to be publicly examined in Auditorium Minus, Museum Gustavianum, Uppsala, Friday, October 13, 2006 at 14:15 for the degree of Doctor of Philosophy. The examination will be conducted in English.

Abstract
Zeffer, H. 2006. Towards Low-Complexity Scalable Shared-Memory Architectures. Acta Universitatis Upsaliensis. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 217. 48 pp. Uppsala. ISBN 91-554-6647-8.

Plentiful research has addressed low-complexity software-based shared-memory systems since the idea was first introduced more than two decades ago. However, software-coherent systems have not been very successful in the commercial marketplace. We believe there are two main reasons for this: lack of performance and/or lack of binary compatibility.

This thesis studies multiple aspects of how to design future binary-compatible high-performance scalable shared-memory servers while keeping the hardware complexity at a minimum. It starts with a software-based distributed shared-memory system relying on no specific hardware support and gradually moves towards architectures with simple hardware support.

The evaluation is made in a modern chip-multiprocessor environment with both high-performance compute workloads and commercial applications. It shows that implementing the coherence-violation detection in hardware while solving the interchip coherence in software allows for high-performing binary-compatible systems with very low hardware complexity. Our second-generation hardware-software hybrid performs on par with, and often better than, traditional hardware-only designs.

Based on our results, we conclude that it is not only possible to design simple systems while maintaining performance and the binary-compatibility envelope, it is often possible to get better performance than in traditional and more complex designs.

We also explore two new techniques for evaluating a new shared-memory design throughout this work: adjustable simulation fidelity and statistical multiprocessor cache modeling.

Keywords: shared memory, distributed shared memory, hardware-software trade-off, software coherence, coherence profiling, remote access cache, chip multiprocessor, simultaneous multi threading, simulation, workload characterization, statistical cache model

Håkan Zeffer, Department of Information Technology, Box 337, Uppsala University, SE-75105 Uppsala, Sweden

ISSN 1651-6214
ISBN 91-554-6647-8
urn:nbn:se:uu:diva-7135 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-7135)

To the people that inspire me

List of Papers

This dissertation is based on the following five papers, which are referred to by the capital letters A to E.

A. Håkan Zeffer, Zoran Radović and Erik Hagersten, **Exploiting Locality: A Flexible DSM Approach**, In Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, Rhodes, Greece, April 2006.

B. Håkan Zeffer, Zoran Radović, Martin Karlsson and Erik Hagersten, **TMA: A Trap-Based Memory Architecture**, In Proceedings of the 20th ACM International Conference on Supercomputing, Cairns, Queensland, Australia, June 2006.

C. Håkan Zeffer and Erik Hagersten, **A Case For Low-Complexity Multi-CMP Architectures**, Technical Report 2006-031, Department of Information Technology, Uppsala University, Uppsala, Sweden, June 2006. Submitted for review.

D. Erik Berg, Håkan Zeffer and Erik Hagersten, **A Statistical Multiprocessor Cache Model**, In Proceedings of the 2006 IEEE International Symposium on Performance Analysis of System and Software, Austin, Texas, USA, March 2006.

E. Dan Wallin, Håkan Zeffer, Martin Karlsson and Erik Hagersten, **Vasa: A Simulator Infrastructure with Adjustable Fidelity**, In Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems, Phoenix, Arizona, USA, November 2005.

Reprints were made with permission from the publishers. All papers are verbatim copies of the original publications but are reformatted to the one column format of this book. (Some pictures are split and the text changed accordingly.)

Here follows a list of other publications by the author. These papers are not part of this thesis.

- Håkan Zeffer, Zoran Radović, Oscar Greenholm and Erik Hagersten, **Exploiting Spatial Store Locality through Permission Caching in Software DSMs**, In Proceedings of the 10th International Euro-Par Conference, Pisa, Italy, August 2004.

- Håkan Zeffer, Zoran Radović and Erik Hagersten, **Flexibility Implies Performance**, Technical Report 2006-013, Department of Information Technology, Uppsala University, Uppsala, Sweden, April 2005.

- Håkan Zeffer and Erik Hagersten, **Adaptive Coherence Batching for Trap-Based Memory Architectures**, Technical report 2005-016, Department of Information Technology, Uppsala University, Uppsala, Sweden, May 2005.

- Håkan Zeffer, **Hardware-Software Tradeoffs in Shared-Memory Implementations**, Licentiate thesis 2005-002, Department of Information Technology, Uppsala University, Uppsala, Sweden, May 2005.

- Pavlos Petoumenos, Georgios Keramidas, Håkan Zeffer, Stefanos Kaxiras and Erik Hagersten, **Modeling Cache Sharing on Chip Multiprocessor Architectures**, In Proceedings of the 2006 IEEE International Symposium on Workload Characterization, San Jose, California, USA, October 2006.

Contents

Introduction

Database, web and high-performance computing servers are increasingly part of our daily lives. For example, we use high-performance computing for weather forecasts and when new cars, airplanes, boats and trains are designed. We rely on commercial database servers when we book an airplane or movie ticket, or when we use an ATM machine to deposit or withdraw money from a banking account. Many of these commercial parallel-computer systems contain elaborate support for coherent shared memory [26, 38, 42, 79].

Recently, the continued decrease in transistor size and the increasing delay of wires have lead to the development of chip multiprocessors (CMPs) [7, 38, 65, 81]. A CMP implements multiple processor cores on a single die and introduces new resource tradeoffs for chip designers and architects. Today, all the big computer companies have announced chip multiprocessor designs and most of them are selling CMP-based systems already [43, 38, 79].

Large-scale shared-memory systems have been successfully built for many years. However, the cost in terms of design and verification is very high and tends to increase with each new generation. Much of the complexity comes from the fact that most such systems rely on two layers of coherence, traditionally called intradomain and interdomain coherence.

This thesis studies multiple aspects of how to design future scalable shared-memory servers with the main focus on hardware-software tradeoffs. While it has been possible to design low-complexity systems based on software coherence for many years, this thesis shows that it is possible to get rid of two of their major limitations: their lack of binary compatibility and their lack of high and predictable performance.

Our results show that the addition of simple hardware primitives for fine-grained coherence-violation detection makes it possible to implement low-complexity software-coherent designs that provide both binary compatibility and high performance. Hence, we believe that a software-coherent system with simple hardware support is a much better platform for multichip support than the traditional approach, in terms of cost, complexity and flexibility, in the chip-multiprocessor era.

The thesis also addresses techniques for evaluating a new shared-memory system through adjustable simulation fidelity and statistical multiprocessor cache modeling.

1.1 Computer Development and Trends

Processor performance improvements have been driven by advances in silicon fabrication technology and by architectural improvements. The number of transistors available for each new technology generation is following Gordon Moore's famous "law" from 1965 [57] in which he predicted that the number of transistors would double every 18th month[1]. The increased transistor budget has traditionally been used to improve single-thread processor performance. For example, processor structures such as register files, branch predictors and issue windows have grown with each new generation. As a consequence of the technology- and the architectural improvements today's processors can operate at a much higher frequency than they could just a few years ago. However, memory latency has not improved at the same fast pace and a latency gap has evolved between the processor core and memory [88]. In the early 80's, a memory reference took about the same time as an arithmetic operation such as an addition or a subtraction. It is today possible to do hundreds or even thousands of arithmetic operations while waiting for a single memory reference.

In order to hide the long memory latency cache memories have been proposed and heavily used. A cache memory is a small and fast memory located close to the processor core. They are typically implemented in a faster technology, such as high-speed SRAM instead of DRAM, but use more transistors to hold the state of a single bit. The idea with a cache memory is that it can store part of an applications footprint, and hence, speed up the average memory access time.

Semiconductor Industry Association (SIA) and Agarwal et al. have predicted that while transistors will continue to get smaller the relative speed of wires will decrease [5, 73]. It will, for example, not be possible to propagate a signal across an entire chip in one clock cycle in the near future [5]. This development has led to larger caches [42], less complex processor cores and the introduction of chip multiprocessors [7, 38, 65, 81]. It is also common today to use multiple hardware threads per processor core (simultaneous multithreading (SMT)) in order to increase on-chip parallelism and throughput [38, 40, 41, 83]. The new CMP/SMT approach offers performance increases mainly through multiprocessing and not through increased single thread performance. CMPs/SMTs come in both small and large system configurations and are predicted to increase rapidly on the market. For example, where the Sun T1 processor (code name Niagara) [38] uses a single CMP do the IBM designs Power4 [79] and Power5 [41] combine multiple CMPs to form larger systems.

[1] This is a reformulation of his first prediction which stated that the number of transistors per area that are economically achievable will continue to double every year. At least for ten more years [57].

Intel researchers predict that 75 percent of all their processor products will be CMPs by the end of 2006 [20].

1.2 Caches and Locality

One key to fast-running programs is fast access to data. In order to lower the memory latency, modern computer systems implement multiple levels of caches. The idea behind cache memories comes from two important observations: most programs do not access all code or data uniformly, and smaller hardware can be made faster. The first observation is based on the principle of locality, which can be divided into two sub groups: spatial locality and temporal locality. Spatial locality (locality in space) states that items whose addresses are near one another tend to be referenced close together in time. Temporal locality (locality in time), on the other hand, states that recently accessed items are likely to be accessed in the near future.

Cache memories are typically implemented in a faster technology than main memory. However, the high cost of fast memory typically leads to a hierarchical design with multiple levels of cache. The first-level cache is smaller, faster and more expensive per byte than the second-level cache. The second-level cache is smaller, faster and more expensive per byte than the third-level cache, and so on.

A cache typically operates at a data unit called a cache line or a cache block in order to exploit spatial locality and to reduce storage overhead. The idea is to bring an entire cache line (more data than needed) into the cache on a miss and hope that other parts of the cache line will be accessed soon causing cache hits instead of misses. It is very important that programmers who want high performing software understand not only runtime complexity of algorithms but also how to write code that works well with caches. Cache misses can be divided into four categories [18, 34].

Compulsory misses The first time a processor accesses a cache line a mandatory miss occurs. These cache misses are often called *compulsory misses* or *cold misses*. They typically occur once for each cache line in an application's footprint during the run of the application. Hence, they can be seen as a startup cost, but can be avoided with *prefetching* (explained soon).

Capacity misses Since caches are small when compared to memory they can often not hold the entire footprint of an application. Hence, a *capacity miss* occurs when the requested cache line has been replaced from the cache because of the cache's limited size and is later accessed again. They can be reduced or avoided if the cache is made bigger or if the

footprint of the application can be compacted to fit in the cache. However, as stated above, a larger cache typically comes with higher access latency.

Conflict misses It is typical that not all cache lines can be stored in all places of a cache. One reason for this is that limiting the number of places where a cache line can be placed can lower the access time of the cache. A *conflict miss* occurs when a cache line was replaced not because of the size of the cache but because of this placement limitation and is later accessed again. Conflict misses can be avoided by increasing this limit. However, doing so typically increases the access latency and the dynamic power consumption.

Coherence misses *Coherence misses* can only occur in multiprocessors and are caused by the coherence protocol and the parallel application running on the system. One example of a coherence miss is when a cache line has been invalidated by another processor and is referenced again. Coherence misses can further be divided into *true-sharing* misses and *false-sharing* misses. True sharing occurs when another processor uses the data that is referenced. False sharing, on the other hand, occurs when no other processor is working on the referenced data but other data located on the same cache line.

Prefetching is a technique that brings data into a certain level of cache, from a lower cache level or from memory, in advance. There are both hardware-based schemes [6, 16, 17, 39, 74, 78, 79, 85] and programmer or compiler guided software-prefetching schemes [59, 60, 61, 82]. Hardware prefetching schemes typically require dedicated hardware that does the prediction and makes sure to fetch data. While these schemes come with hardware complexity they keep the application binary unmodified and does not require recompilation. Software prefetching schemes move most of the complexity to the compiler or the programmer.

1.3 Shared-Memory Architectures

Many servers in the commercial and technical market support shared memory. Plentiful of research have been put into these architectures, their coherence protocols and memory-consistency models. This section gives a short background to shared-memory architectures. Cache coherence and memory consistency are briefly discussed in Section 1.3.1 and Section 1.3.2 respectively.

Maybe the most classical shared-memory design is the *symmetric (shared-memory) multiprocessor* (SMP) where all processors have the same access

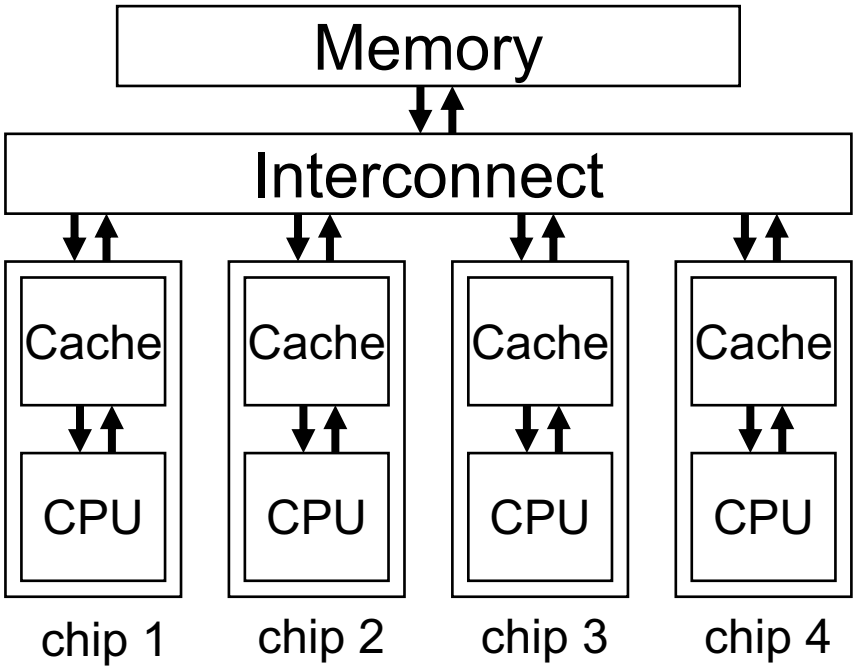

Figure 1.1: An example of a symmetric multiprocessor (SMP) with four processors and shared memory.

time to all parts of the shared memory[2]. These systems often rely on a shared bus and implement a snoop-based cache-coherence protocol [29]. Figure 1.1 shows an example SMP with four processors, each with private caches, connected with each other and the memory over a bus. A shared bus makes it possible to implement a global order between different memory operations and to implement efficient snoop-based cache coherence without any extra indirections [29]. However, since all processors can see all transactions snoop-based coherence schemes have scalability problems. This is mainly because of the increased traffic and contention when adding more processors to the bus.

SMPs are often used as building blocks when forming larger machines. Examples are Stanford's DASH [51] and Sun's WildFire [31] that both connect multiple snoop-based SMPs together with a directory-based coherence protocol. Since each SMP has its own memory these machines are often called distributed shared-memory (DSM) servers and typically have non-uniform access time to the different portions of the shared memory. The non-uniform memory access time classifies these machines as non-uniform memory-access

[2]We refer to the uniform memory access time when talking about a symmetric multiprocessor in this thesis.

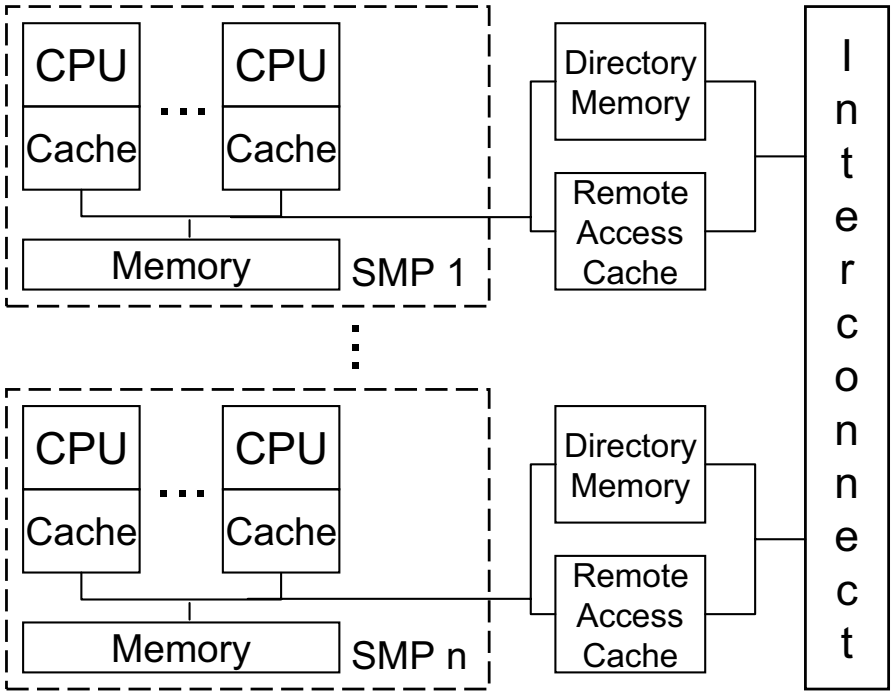

Figure 1.2: An example of a DASH-like [51] distributed shared memory server connecting *n* SMPs together.

architectures or NUMA machines for short. Directory-based coherence protocols can utilize unordered interconnects and scale much better than their snoop-based counterparts. The extra indirection through the directory removes the need for all processors to see all transactions. However, it typically increases the latency of coherence actions.

Figure 1.2 shows how the Stanford DASH machine is designed [50, 51]. Each SMP is extended with a directory memory, a remote-access cache and state machines implementing the interSMP directory-based coherence protocol. A remote-access cache is a node-private cache allowing a node to cache portions of memory located on other nodes in the system. A large remote access cache can reduce the number of remote coherence misses, and hence, improve performance of the system [13, 50, 80].

Improvements in technology have enabled multiple processor cores on a single die, so-called chip multiprocessors. A chip multiprocessor has much in common with an old SMP. However, the entire system, except memory and I/O, is moved onto a single die. The interconnect connecting the processor cores is typically located between private level-one caches and a shared second-level cache [7, 38, 79]. However, future designs may have more levels

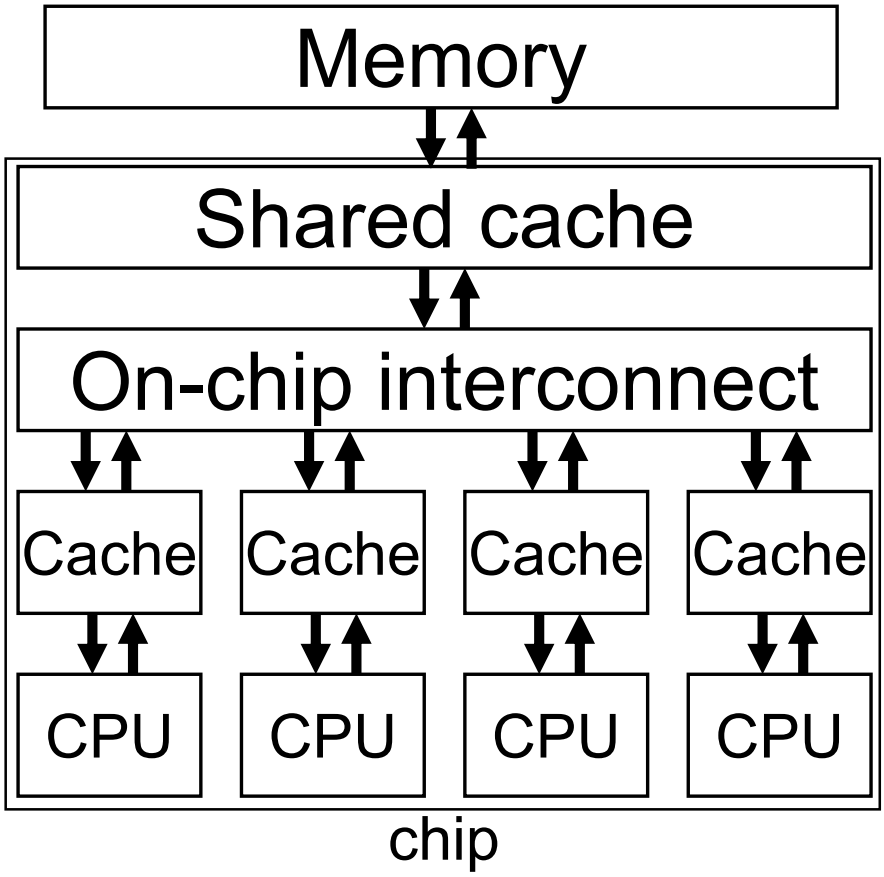

Figure 1.3: An example of a chip multiprocessor (CMP) with four processor cores, private level-one caches and a shared second-level cache.

of shared/private cache. Figure 1.3 shows an example of a chip multiprocessor with four processor cores and private first-level caches. The processor cores are connected to each other with an on-die interconnect located between the private caches and the shared second-level cache.

CMPs can be connected together to form larger configurations, similarly to SMPs connected together to form DSM machines. Piranha [7], IBM Power4 [79] and IBM Power5 [41] are examples of such machines. Figure 1.4 shows an example of how multiple chip multiprocessors can be interconnected.

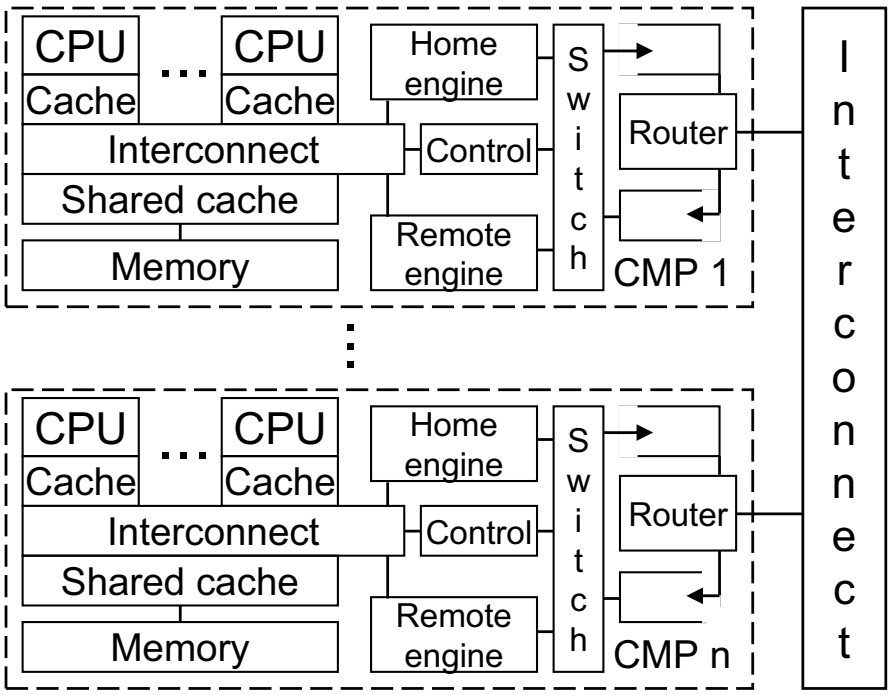

Figure 1.4: An example of a Piranha-like [7] multi-CMP. This system connects *n* CMPs together forming a distributed shared-memory system.

1.3.1 Cache Coherence

A key challenge with shared-memory systems is implementing high-performance cache-coherence protocols. These protocols keep caches transparent to software, usually by maintaining a coherence invariant including that each block may have either one writer or multiple readers. The coherence protocol is a very important part of a shared-memory multiprocessor since it deals with both correctness and performance.

There has been plentiful of coherence optimizations targeting different kinds of sharing patterns and certain kinds of coherence misses, such as, migratory sharing [15, 76] and producer consumer (here in form of self invalidation) [46, 49]. Speculative non-binding prefetch mechanisms, based on both address-based predictors [45, 63] and instruction-based predictors [36, 39], tries to reduce the coherence miss latency by fetching data and permission in advance. Owner [1] and destination set [54] prediction are other proposals that try to eliminate the extra indirection in 3-hop directory-protocol misses. There are also some new and very interesting coherence proposals: token coherence [55, 56] and coherence decoupling [35] both make it possible to decouple performance and correctness.

1.3.2 Memory Consistency

Another key to multiprocessor performance is the memory-consistency model. It defines the ordering rules between accesses to different cache lines within the system. They range from sequential consistency [47] on one end, allowing very limited buffering, to release consistency [25] on the other end, allowing extensive buffering and pipelining. Total store order [75], processor consistency [25, 30] and weak ordering [2, 19] all fall in between (in the order strong to weak). From a programmer's point of view, a multiprocessor system that implements a weak memory-consistency model is harder to program. Hence, it has been argued that shared-memory multiprocessors should support simple memory-consistency models [33]. On the other hand, it has been shown that sequential consistency performs poorly relative to all other models, while processor consistency like models provide most of the benefits of the weak and release consistency models [27].[3] Many speculation-based optimizations work for both strict and loose models [11, 28].

1.3.3 Hardware-Software Tradeoffs

There is a wide range of options for hardware-software tradeoffs for implementing coherent shared memory, ranging from all-hardware coherence to implementations relying on no specific hardware support. It is fairly common, however, that systems rely on both hardware as well as run-time software to improve performance.

Most cache-coherent NUMA machines rely on hardware performance counters to guide the page migration software [31, 48]. The Sun WildFire system has hardware support for detecting pages in need of replication [31]. Replicated copies of each page can be instantiated by software, but the coherence between the copies is maintained by hardware on a cache-line 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 has efficient support for trapping to a software handler on the rare event of massive sharing [3].

The hardware-software tradeoff is even more apparent in systems with programmable coherence engines or dedicated coherence processors [14, 44, 64, 68]. Here, the entire coherence protocol is controlled by specialized code, which increases the possibilities for flexible protocol adoptions as well as correcting protocol bugs.

The idea of implementing shared memory with software across a cluster of non-coherent machines was first proposed by Li and Hudak [52]. They called the new approach shared virtual memory (SVM) because the virtual-

[3] Simulation study with blocking processor reads.

memory subsystem was used for access control and message passing for internode communication. The first SVM prototype, running on an Apollo ring, was implemented in the mid 80s. However, several factors limit the performance of SVM systems: the large coherence unit size (a page) can cause false sharing, and the cost of each communication or synchronization operation is large since it is performed with software messages. Two main research directions have evolved to improve the performance of software shared memory implementations: relaxing consistency models [12, 37, 70, 92] and providing fine-grained access control [67, 70, 71, 90].

Page-based systems often rely on weak memory consistency models and multiple writer protocols to manage the false sharing introduced by their large coherence unit [10, 23, 77]. Carter et al [12] introduce the release consistency model in shared virtual memory. Lazy release consistency was introduced by Keleher et al [37] and home-based lazy released consistency by Zhou et al [92]. The majority of systems implement numerous coherence strategies/protocols. For example, Munin implements both invalidate- and update-based protocols [12].

Fine-grained software DSMs maintain coherence by instrumenting memory operations in programs [67, 69, 70, 72, 90]. These systems usually provide stable and predictable performance for the majority of parallel benchmarks originally developed for hardware multiprocessors. However, 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 [21].

Software-based DSM proposals were given much attention in the 90s [12, 69, 72, 92]. While a software implementation can provide a cheaper and faster time to market, hardware implementations have thus far been able to offer much better and more predictable performance.

1.4 Evaluation of New Architectures

It is common to use simulators together with benchmarks that represent the workload of the system when designing a new computer. This section briefly discusses simulators and statistical models.

1.4.1 Full-System Simulation

Simulation is extensively used, both in academia and industry, to evaluate future computer designs and the software running on them. A major benefit

of simulation is that new architectures can be studied and that software can be ported to and optimized for machines before the actual hardware is available.

Many independent simulations are typically started in parallel, for example, with different parameters or with different workloads. While there is plenty of application and parameter level parallelism the simulation time is usually very long. The reason for this is that simulators are typically designed to resemble the proposed design as closely as possible, leading to extremely detailed simulators. It is not uncommon with a turn-around time of multiple days. Many different ways to speed up a simulation have been proposed over the years. Identification of phases [66] and sampling [86, 87, 89] are just two of them.

The full-system simulator used in this thesis is Virtutech's Simics [53]. Simics has the ability to model both single-processor and multiprocessor systems. It can boot commercial operating systems and run advanced user level programs such as high-performance compute applications and commercial databases. We run the Simics version that implements the SPARC v9 instructions set architecture and run the Solaris operating system on our simulated machines. Simics makes it possible to extend a machine with timing-models and/or new hardware. **Paper E** describes the models used in this thesis. The infrastructure, called Vasa [84] contains models of processor cores, caches, store-buffers, memory and interconnects. Vasa can run in three different fidelity modes. Hence, early investigations and tests can use a lower fidelity and run with a short turn-around time while final results can be verified and generated with a more detailed model.

1.4.2 Statistical Modeling

Statistical modeling can also be used to speedup simulation time or to help programmers optimize code for a certain platform. There are two classes of statistical models. The first uses the distribution of input data to generate a shorter synthetic trace representing the workload. The synthetic trace is then run through a traditional simulator in order to get results [22, 66]. The second class or approach uses a mathematical model of the system; hence, a simulator is not needed [4, 8, 9, 32]. The mathematical model may be solved analytically or numerically to produce results. The research in statistical modeling done in this thesis is of the second class and is described in **Paper D**.

Paper Summaries

This section contains summaries of the included papers.

2.1 Exploiting Locality: A Flexible DSM Approach

The idea of implementing shared memory with software across a non-coherent cluster of machines is more than two decades old [52]. Plentiful of research has been put into these systems, especially during the 90's. However, the promise of low cost and short time to market could not make up for their unpredictable and often poor performance. Hence, they are very rare in the commercial market place.

It has been shown that per-application tailor-made coherence strategies [24], different coherence protocols for different kinds of data [12], coherence granularity and memory-consistency model adjustments [91] can be used to enhance software-DSM performance. While these systems can be tweaked to perform well through various tricks it is often quite cumbersome to do so.

Paper A proposes multiple new software-DSM optimizations in DSZOOM's synchronous directory-protocol environment. The *write-permission cache* is used in both invalidation- and update-based systems. It is extended to work as a bandwidth filter in the update-based coherence protocol and is evaluated together with multiple coherence unit sizes in both the update- and the invalidation-based protocol. Moreover, the paper shows that the virtual memory system can be used to enhance performance in fine-grained software-coherent systems. **Paper A** also simplifies the usage of flexible systems with multiple coherence protocols and optimizations by introducing *coherence profiling* and *coherence flags*. A user can profile an application and set corresponding coherence flags at startup the next time the program is to be run.

The paper further shows that software flexibility can be used to narrow the gap between hardware and software coherent systems. DSZOOM is compared to a hardware DSM built with identical interconnect and node hardware and its performance is shown to be on average 11 percent lower than that of the hardware system. DSZOOM is faster than the hardware system on four of the applications run.

2.2 TMA: A Trap-Based Memory Architecture

The traditional way of designing large-scale coherent shared memory is by modifying the memory system leaving most of the processor core unchanged. **Paper B** proposes 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 machines. The essence of the proposal is to detect coherence violations in hardware, trigger a *coherence trap* when one occurs and maintain coherence by software in coherence trap handlers. We call the proposal a *trap-based memory architecture* or TMA for short.

The detection and handling of coherence violations can be implemented with a minimal amount of extra hardware support by exploiting the trap mechanism already existing in modern microprocessors. While many schemes are possible, **Paper B** evaluates a minimalistic TMA implementation, TMA Lite, based on a hardware *magic-value* comparator for read permission checks and a hardware write-permission cache implementation for writes. The system architecture presents a binary transparent view to the application and all necessary software support is contained in system software. While a software coherence scheme introduces some extra overhead when compared to a hardwired system, it also comes with 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 switch coherence schemes in a trivial manner.

Paper B argues that the cost of these modifications in terms of area, power and engineer year is small enough to justify incorporating them in designs spanning multiple market segments. The goal being a system design cost and a system time-to-market that is equal to the processor design time. The performance evaluation, based on detailed simulation of dynamically scheduled, multithreaded chip multiprocessors, shows that the system performs on par with a highly optimized hardware-only distributed shared-memory machine when some of the flexibility introduced by software is taken into account.

2.3 A Case For Low-Complexity Multi-CMP Architectures

Paper C presents CRASH (Complexity-Reduced Architecture for SHared memory), which is a second-generation trap-based memory architecture.

While the TMA Lite system has its advantages it also has its flaws. For example, it replicates all shared data and maintains coherence between those

replicated copies. This makes the design simple but comes with high overhead in memory, especially when many nodes are used. CRASH removes the need for replication and contains an improved coherence check design.

CRASH's coherence checks are based on two node permission bits per cache line. The node read bit indicates if a node has read permission and the node write bit indicates if a node has write permission. As in the TMA Lite design, a coherence violation leads to a coherence trap making it possible to implement an efficient interchip coherence protocol in software. CRASH's design removes all false positives from the coherence checks and moves coherence detection from the virtual- to the physical address space. Checks on physical addresses makes it possible to implement the coherence layer below the operating system and removes the aliasing problem that is introduced when multiple virtual addresses maps to the same physical address.

The CRASH design also contains a small memory controller modification. It is introduced in order to remove the need of replication and makes sure that local and remote data is handled correctly when evicted from a chip's cache subsystem. On a write-back, local data has to be written to its location in local memory and remote data has to be written back to the corresponding node in a traditional design. The modified memory controller handles local data the traditional way. However, remote data is simply written to a dedicated area in the local memory. This area is indexed as a direct-mapped cache, has a configurable size, but does not contain any hardware for hit/miss detection. It is simply left to the software coherence protocol to avoid collisions and to handle remote write backs from this particular area. The memory controller modification can be seen as a way to decouple the intrachip coherence protocol run in hardware and the interchip coherence protocol run in software when on-die memory controllers are used[1].

The CRASH scheme with the modified memory controller forms a natural base for remote access caching in the local DRAM subsystem. Hence, the coherence traffic due to capacity misses can greatly be reduced for many applications. We run both high-performance compute applications and commercial workloads in **Paper C** while evaluating CRASH. The results show that the second-generation TMA system is able to perform on par with, and often better than, highly optimized hardware solutions.

We would like to add some comments in retrospect to this paper. First, the permission checks proposed in **Paper C** makes it possible to decouple the permission check from the actual coherence state. Since the bits only indicate read and write permission all combinations of MOESI protocols are possible. Second, the forward progress solution provided in the paper is quite detailed and might add some complexity (at least in the store case). However, it is possible to solve this at a much higher level, for example, in the internode

[1]The design also works with off-chip memory controllers.

software-coherence layer. Third, it is of course also possible to solve internode write backs directly on second-level cache evictions as in SMTp [14]. We believe this is harder to implement, but possible to do. Finally, we know it is possible and believe it might be simpler to add support for remote memory "put" operations instead of implementing efficient remote interrupts as in the paper. The remote put operation together with polling can be used to implement efficient messages between nodes. This might be a good idea since it would reduce the complexity and because future CMPs will have a lot of threads. Hence, one or a few threads may be dedicated for home-node coherence requests.

2.4 A Statistical Multiprocessor Cache Model

The introduction of general-purpose microprocessors running multiple threads will put a focus on methods and tools helping a programmer to write efficient parallel applications. Such a tool should be fast enough to meet a software developer's need for short turn-around time, but also be accurate and flexible enough to provide trend-correct and intuitive feedback.

Berg and Hagersten [8, 9] have shown that a mathematical cache model can be used for estimating the miss ratio of fully-associative caches based on the reuse-distance distribution of the memory references of an application. The miss ratio for an application and all possible cache sizes can be calculated based on data from a single run since the number of cache lines in the system is a parameter to the post processing part of the mathematical model.

Paper D extends the cache model presented by Berg and Hagersten [8] to model multiprocessor memory systems and to estimate cold misses. Very sparse data is collected during a single execution of the studied application. The data is passed to the mathematical post-processing system which makes it possible to show how the number of capacity and coherence misses, caused by invalidations, varies when the cache size is changed and when the different processors are connected in different ways. The model is evaluated with a traditional simulator and the results show this scheme to be very accurate.

2.5 Vasa: A Simulator Infrastructure with Adjustable Fidelity

The arrival of chip-multiprocessor designs and multithreaded processor cores has lead to an increased need for efficient multiprocessor simulations. **Paper E** presents the Vasa simulation framework which can be run with adjustable

fidelity, and hence, speed. Vasa is a set of timing models that are highly integrated with the commercial full-system simulator Simics [53] and that in great detail can model store buffers, caches, memory controllers and networks.

Vasa can be run in three different fidelity modes. The most detailed mode contains a detailed multi-threaded processor-core model enabling detailed CMP/SMT studies. However, in order to let a user test early designs and to prototype systems, Vasa contains two faster modes as well. The fastest being about 300 times faster than the most detailed one. The same memory system is used in all three modes and is capable of modeling write-back, write-through, shared and private caches, and both snoop-based and directory-based coherence schemes.

Paper E shows that memory-system studies targeting the lower levels of cache, coherence and memory can use faster modes and still yield the same result as with the detailed and much slower mode. Note that faster modes make it possible to run longer, and hence, test bigger and more realistic applications. Vasa is used in both **Paper B** and **Paper C**.

Contributions of this Thesis

The focus of this thesis is on low-complexity scalable shared memory for the chip-multiprocessor era. However, it also contains contributions in the area of simulation and statistical modeling. The main contributions of the thesis are:

We introduce *coherence flags* and *coherence profiling* in **Paper A** that makes it simpler to get good performance from the all-software distributed shared memory system DSZOOM. We further show that flexibility can be used to enhance software-only coherence and sometimes even outperform hardware coherent systems. **Paper A** also contains a detailed study of the *write permission cache* in both update- and invalidation-based coherence protocols. **Paper A** is the first paper (known to the author) describing a fine-grained software distributed shared-memory system that uses the virtual-memory system to increase performance.

Paper B introduces the concept of *trap-based memory architectures* in which a *coherence trap* is handled on the hardware thread that detected the corresponding miss. We show how fine-grained software coherence can get rid of the instrumentation by adding minimal hardware support to a modern processor design. We present detailed simulation analysis of the TMA Lite system and show that the trap overhead involved in a coherence-trap driven system is not a major performance problem.

Paper C is the first simulation study of a hardware-software coherent system running commercial workloads. However, the main contributions of this paper are the coherence trap mechanisms and the memory controller modifications that greatly reduce the complexity while decoupling the two layers of coherence in multi-CMP designs. **Paper C** also demonstrates that the flexibility of software coherence can be used to tune servers resulting in substantial performance improvements, for example, by using an in-DRAM remote-access cache and different prefetching strategies for different workloads.

Paper D extends StatCache [8, 9] to model multiprocessor memory systems, including shared caches and invalidation-based coherence misses. It also shows how the cold miss ratio of an application can be estimated.

A key contribution of **Paper E** is that it shows that running with a less detailed mode often produces as good results as running with a very detailed and much slower mode. This is especially true when investigating the lower levels of the cache hierarchy, the memory or the traffic over an interconnection network. Vasa is used by multiple universities worldwide[1] and there is a user forum named "VASA Users" available at `www.simics.net`.

[1] Vasa is free software and is available under LGPL (GNU Lesser General Public License).

Comments on my Participation

A. Exploiting Locality: A Flexible DSM Approach
The paper was jointly written with Zoran Radović. I designed and evaluated most of the experiments and optimizations. Zoran and I jointly implemented and optimized them in DSZOOM.

B. TMA: A Trap-Based Memory Architecture
The paper was jointly written with Zoran Radović and Martin Karlsson. I designed and evaluated all the experiments and optimizations. Zoran and I jointly implemented the coherence protocols. Martin helped with simulations and discussions on the trap mechanism.

C. A Case For Low-Complexity Multi-CMP Architectures
I am the principal author and have implemented and evaluated all the experiments and architectural features.

D. A Statistical Multiprocessor Cache Model
Erik Berg is the principal author of this paper, however, I edited the final version. Erik Berg and I did most of the research and programming jointly.

E. Vasa: A Simulator Infrastructure with Adjustable Fidelity
All authors have contributed to all aspects of the project. I am the main contributor of the Vasa implementation. Dan Wallin and Martin Karlsson are the principal authors of the article. Dan Wallin designed and performed all the experiments.

In all papers, all authors have contributed with valuable input, feedback and ideas.

Conclusion

Researchers have tried to design simple software-based shared-memory systems for several decades. Still, such systems are not common in the commercial marketplace. We believe the reason for this is their lack of binary compatibility and/or their often poor and unpredictable performance.

Results presented in this thesis and summarized in Figure 5.1 showed the performance of CRASH, our second-generation hardware-software hybrid presented in **Paper C**, when compared to three hardware coherent systems. All systems were run with four nodes and one four-threaded processor core each. `HWperf`, `HW256` and `HW64` corresponded to hardware coherent systems with a perfect directory cache, a 256k direct-mapped directory cache and a 64k direct-mapped directory cache respectively. The `CRASH` bars corresponded to the CRASH system that performed coherence-violation detection in hardware but ran the internode coherence protocol in software. The `CRASH OPT` bars corresponded to the CRASH system when tuned for each particular workload. (More details can be found in **Paper C**.)

The results clearly demonstrated that CRASH provided both binary compatibility and hardware competitive performance. CRASH ran multiple complex commercial database workloads and its performance was comparable to, and often better than, hardware coherent systems. We conclude, based on these results, that a hardware-software hybrid can provide both binary transparency and high performance while providing very attractive properties such as low complexity, low cost and flexibility.

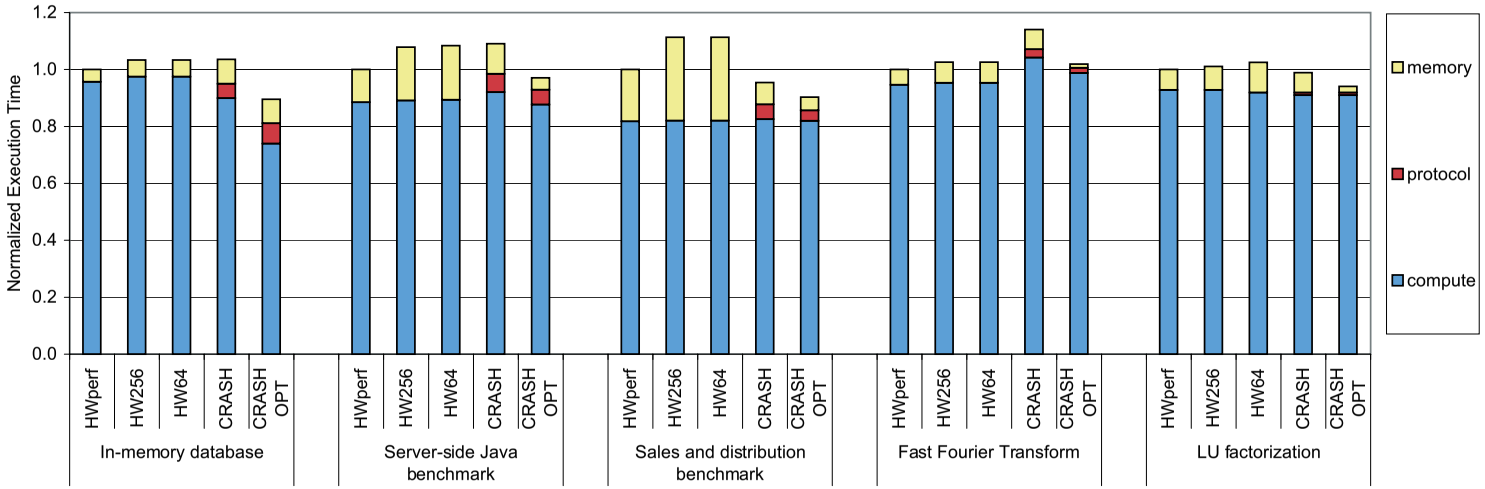

Figure 5.1: CRASH performance when compared to three hardware coherent systems. This figure shows four-node configurations where each node contains a single four-threaded processor core. `HWperf`, `HW256` and `HW64` corresponds to hardware coherent systems with a perfect directory cache, a 256k direct-mapped directory cache and a 64k direct-mapped directory cache respectively. `CRASH` corresponds to the basic software coherent system and `CRASH OPT` corresponds to the system when optimized for each particular workload.

Reflections and Outlook

In this section I share some of my thoughts gathered during years of academic research and six months in industry.

Paper A shows that using software coherence might not be a bad idea after all. I believe fine-grained all-software distributed shared memory might be feasible if implemented in a high performing compiler and runtime infrastructure. For example, one might get even better performance at a very low cost if an OpenMP compiler's optimizing backend implements the "instrumentation".

An OpenMP runtime system might also be a good environment for the TMA Lite system presented in **Paper B**. All deadlocks related to the write-permission cache are for example removed (all synchronization has to be visible to the runtime system). However, I do not recommend that either DSZOOM or TMA Lite are used to run commercial applications.

Both **Paper A** and **Paper B** relies on remote atomic operations. A finding Zoran Radović and I did while working on DSZOOM is that the interaction between the remote- and the local atomic operations are of great importance for efficiency of such systems. That is, it must be possible to mix the two in order to get DSZOOM's synchronous directory protocol to perform well. I further believe that it might be interesting to redo the poll evaluation, but this time with chip multiprocessors and with dedicated protocol threads.

I consider the trap-based memory architecture CRASH much more mature than DSZOOM and TMA Lite. Its complexity is a bit higher, but I find the added value very high. I believe that CRASH will perform well for both HPC and commercial workloads and that it comes with very attractive properties such as low complexity, flexibility and binary compatibility.

We have so far only looked at software coherence between distinct nodes consisting of SMPs and chip multiprocessors. However, I find it very appealing to move the software-coherence layer onto chip multiprocessors in the near future. For example, it might soon be interesting to implement many small hardware-coherent domains on a single chip, connected by software coherence. This both reduces the complexity of the chip and enables multiple optimizations within coherence domains.

While we have added minimal hardware support for an efficient software-coherence protocol providing binary compatibility and performance in this work, an argument can certainly be made for slightly more hardware. I can see two distinct purposes for such hardware: even higher performance and support for new areas such as transactional memory. However, I find it very important that such hardware is kept simple, small and does not limit the flexibility of

the design. For example, I do not think implementing a remote-access cache in hardware is a good idea.

I believe our coherence-trap support and permission detection mechanisms can be used in many areas. Our primitives could simplify hardware-fault containment between chips and future coherence domains. They enable fast simulation à la Wisconsin Wind Tunnel [62], they can be used by debuggers and profiling tools, and can be used to implement a LogTM-like [58] transactional memory system if some additional hardware is added to the design.

To conclude, I see no reason why to implement future multi-CMPs with hardware-only coherence. I base that conclusion on the stable and competitive performance of CRASH, its lower complexity and its ability to boot single-image operating systems and to run complex applications.

Acknowledgements

I first meet Professor Erik Hagersten when I took the course Parallel Computer Systems at KTH (Royal Institute of Technology) in Stockholm where he held a guest lecture named *From SunFire to SunFire*. I got very inspired by that particular lecture and by Erik's knowledge and experience from industry. Hence, I decided to apply for a position in his research team. Erik; thanks for offering me a PhD student position at Uppsala University and for all the help and knowledge that you have provided me with. I have had a great time and learned a lot!

I would also like to thank all the members (most of them former members) of *Uppsala Architecture Research Team* (UART) for all the time and fun we have had together. Thanks, Lars Albertsson, Erik Berg, Martin Karlsson, Henrik Löf, Zoran Radović and Dan Wallin/Ekblom.

I spent six months on a very rewarding internship with Sun Microsystems, Inc. in Sunnyvale, California. I would like to thank all the people I worked with and learned from. I had a very good time and really enjoyed our interaction. Special thanks to Anders Landin, Shailender Chaudry, Bob Cypher, Magnus Ekman, Paul Loewenstein, Marc Tremblay, Mike Vildibill, Padmaja Nandula, Larry and Jerry.

Special thanks to my co-authors and collaborators Zoran Radović, Erik Berg, Martin Karlsson, Dan Wallin/Ekblom, Oscar Greenholm, Pavlos Petoumenos, Georgios Keramidas and Professor Stefanos Kaxiras. It has been a great experience to work with you and I think we did put together a few really good papers! Special thanks to Karin Hagersten and Marianne Ahrne who have proofread part of this thesis! Thanks to Björn Victor for LaTex help.

I would also like to thank all the people I have submitted and/or have (pending) patents with: Anders Landin, Erik Hagersten, Bob Cypher, Shailender Chaudry, Paul Loewenstein and Zoran Radović at Sun Microsystems, Inc and Erik Berg, Erik Hagersten, Mats Nilsson and Mikael Pettersson at Acumem AB.

The people that brought me into the world of computer architecture, compilers, simulators and operating systems also deserve special thanks. Big thanks to Sven Karlsson, Peter Drakenberg and Mats Brorsson at KTH. Thanks also to my former colleagues at Virtutech, Inc. Peter Magnusson, Bengt Werner, Jakob Engblom, `mch`, `fla`, `am`, Gustav, Tomas, Daniel, `mve`, Guillaume and all the others.

I would also like to thank David Wood and Mark Hill at University of Wisconsin for providing the `apache` and the `jbb` benchmarks. Anders Landin

and Sun also deserves special thanks (again) for providing me with the `mdb` and the `sap` benchmarks.

I have had many very interesting conversations with good researchers all over the world throughout these years. Thanks to all of you! Special thanks goes to Angelos Bilas for all comments and the very interesting discussion we had on CRASH. Most of the retrospect comments in Section 2.3 are based on our discussion! Thanks for the time; it was a lot of fun!

I have been very successful in the gentle art of applying for traveling grants. I would like to thank ARTES for the HPCA trip Martin Karlsson and I did. I would also like to thank Smålands Nation in Uppsala that has provided me with quite a few traveling grants. For example, my first U.S. visit was possible because of a grant from Smålands. This was the trip that made it possible for me to present my research at Sun Microsystems and that lead to my internship. Big thanks!

Thanks to the administrative staff and all the other people at the department of Information Technology. The stay here would not have been such a pleasure experience without you! Thanks also to Sverker Holmgren and all the people at the Division on Scientific Computing at Uppsala University for the use of their Sun WildFire and 15K servers. Jukka and Joel for admin help and time.

This work is supported by Sun Microsystems, Inc., and the Parallel and Scientific Computing Institute (PSCI), Sweden. UART is a member of the HiPEAC network of excellence, which also has supported my research.

I would also like to thank my family and all of my friends. Mamma, pappa, Elisabeth och Annika; det är väldigt skönt att ha en så fin familj att kunna luta sig mot. Tack!

Last, but definitely not least, I would like to thank my fiancé and soon to be wife, Anna-Lena, for all the support and love during these years. I love you!

Summary in Swedish

Utvecklingen inom datorområdet har, som de flesta av oss vet, gått väldigt snabbt. Datorernas ökade snabbhet kan härledas till teknologiframsteg inom halvledarindustrin och innovationer av datorarkitekter. Halvledarindustrin gör det möjligt att använda fler och snabbare transistorer i och med varje ny teknologigeneration medan datorarkitekterna använder det ökade antalet transistorer till att bygga så effektiva datorer som möjligt. Datorarkitektur är vetenskapen om hur komponenter ska konstrueras, väljas och kopplas samman för att skapa en dator med hög prestanda, som drar lite energi och som är billig.

Snabba datorer används bland annat till att förutspå väder, vid konstruktion av farkoster samt i banker och företag världen över för att hålla reda på pengar och andra typer av data. Många av dessa, för oss dolda datorer, innehåller flera processorer (en multiprocessor) och så kallat delat minne. Med dagens teknik är det möjligt att bygga chipmultiprocessorer, vilka, precis som namnet antyder, innehåller flera processorkärnor på ett och samma chip. I och med deras införande behöver man inte koppla samman flera chip för att skapa multiprocessorsystem.

Alla delar av en dator har tyvärr inte blivit snabbare med samma höga hastighet. På 80-talet tog det ungefär lika lång tid att hämta data från minnet i en dator som det tog att utföra en beräkning, såsom en addition eller en subtraktion. I dagens datorer hinner man göra flera hundra, om inte tusen, beräkningar under den tid det tar att hämta data från minnet. Eftersom snabb tillgång till data är viktigt för datorernas prestanda har man infört mindre och snabbare minnen mellan processorn och det långsamma men stora huvudminnet. Dessa minnen kallas för cacheminnen och är ofta byggda i en snabbare men dyrare teknologi än huvudminnet. Ett cacheminne kan inte innehålla all data som finns i huvudminnet (eftersom de är små) men hjälper till genom att snabba upp hämtningen av ofta förekommande data. Även datorer med flera processorer innehåller cacheminnen vilket i kombination med delat minne leder till ett problem som kallas för cachekoherens.

Avhandlingen beskriver och utvärderar hur man med låg komplexitet kan konstruera servrar bestående av flera chipmultiprocessorer. Idén är att använda hårdvara för att upptäcka när systemet behöver lösa koherensproblem men att implementera protokollet som löser dessa i mjukvara.

Artikel A handlar om DSZOOM, ett system som både upptäcker koherens problem och löser dessa i mjukvara. Den stora fördelen med system av den här typen är deras enkelhet och låga kostnad medan den stora nackdelen är deras ofta låga och oberäkneliga prestanda. Artikeln jämför DSZOOM med

Suns WildFire prototyp, ett system där både upptäckande och lösande av koherens sker i hårdvara. I artikeln visas att flexibiliteten som mjukvara medför avsevärt kan förbättra DSZOOMs prestanda. Faktum är att med hjälp av sin flexibilitet är DSZOOM endast 11 procent långsammare än WildFire på de program som testats i artikeln. Då ska man komma ihåg att DSZOOM är utvecklat av två doktorander medan WildFire är ett kommersiellt projekt som involverade en stor projektgrupp och kostade miljontals med amerikanska dollar att konstruera.

Artikel B och **C** innehåller kärnan av avhandlingen och handlar om hur man kan förbättra både prestanda och kompatibilitet hos mjukvarukoherenta system genom att flytta detektionen av koherensbehov till hårdvara medan man behåller protokollet som löser koherens i mjukvara. **Artikel B** kallar detta koncept *trap-based memory architecture* (TMA) och visar hur en minimal design kan se ut. Den enkla hårdvaran som lagts till i den minimala TMA konstruktionen, kallad TMA Lite, genomför de rättighetstester som i DSZOOM gjordes i mjukvara. Artikeln visar att TMA Lite medför avsevärt högre prestanda och mycket bättre transparens för en användare än vad DSZOOM gör.

I **Artikel C** presenteras en andra generationens TMA system. Något mer hårdvara har nu lagts till, men i gengäld är både prestanda och skalbarhet bättre. Även i detta system löses koherensen mellan datorerna (chipmultiprocessorerna) i mjukvara vilket medför att hårdvarukomplexiteten är låg i jämförelse med en hårdvarulösning. Mjukvaruprotokollet tillsammans med de effektivare koherens testerna, gör att andra generationens TMA system ofta till och med är snabbare än de mer klassiska hårdvarulösningarna.

Avhandlingen handlar också om hur framtida datorsystem ska kunna utvärderas i konstruktionsfasen och hur program för sådana framtida datorer ska kunna snabbas upp och förbättras redan innan hårdvara finns tillgänglig. Man utvärderar ofta framtida datorer i en simulator innan man bygger dem. Den stora fördelen med detta är att man kan se hur en dator beter sig utan att behöva bygga systemet i fråga. Den stora nackdelen är att dagens simulatorer är väldigt långsamma och att det är en konst i sig att skriva en representativ simulator. Dessutom är det både tidsödande och mycket svårt att få interaktionen mellan dagens komplexa mjukvara och simulatorn att representera morgondagens system, vilket ju var målet med hela övningen.

Artikel E beskriver Vasa, den simulatorinfrastruktur som har använts i den här avhandlingen. Vasa gör det möjligt att simulera moderna chipmultiprocessorer, minnessystem, bussar och nätverk i tre olika noggrannhetsnivåer. De olika noggrannhetsnivåerna gör det möjligt för användaren att bestämma noggrannhet beroende på problem och prestandakrav. En lägre noggrannhet ger nämligen mycket högre prestanda vilket är ett krav för vissa studier.

Artikel D beskriver en statistisk modell för hur program som körs på en dator uppför sig med avseende på cacheprestanda. Modellen bygger på Erik

Bergs[1] tidigare forskning men är utökad för att kunna hantera multiprocessorsystem. Artikeln inför även ett nytt sätt att uppskatta antalet kallmissar som ett program råkar ut för under sin exekvering. Den statistiska modellen kan användas av datorarkitekter som behöver köra stora komplexa program eller behöver extremt hög simuleringsprestanda, samt av programmerare som optimerar kod skriven för multiprocessorsystem.

[1]Dr. Erik Berg är en tidigare medarbetare i våran forskningsgrupp och är en av författarna till Artikel D.

References

[1] M. E. Acacio et al. Owner Prediction for Accelerating Cache-to-Cache Transfer Misses in a cc-NUMA Architecture. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, November 2002.

[2] S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[3] A. Agarwal et al. The MIT Alewife Machine. *IEEE Proceedings*, 87(3), 1999.

[4] A. Agarwal, J. Hennessy, and M. Horowitz. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2), 1989.

[5] V. Agarwal et al. Clock Rate versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[6] J.-L. Baer and T.-F. Chen. An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, November 1991.

[7] L. Barroso et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[8] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2004.

[9] E. Berg and E. Hagersten. Fast Data-Locality Profiling of Native Execution. In *Proceedings of the 2005 ACM/SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2005.

[10] A. Bilas, C. Liao, and J. P. Singh. Using Network Interface Support to Avoid Asynchronous Protocol Processing in Shared Virtual Memory Systems. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[11] H. W. Cain and M. H. Lipasti. Memory Ordering: A Value-Based Approach. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[12] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[13] J. B. Carter et al. Design Alternatives for Shared Memory Multiprocessors. In *Proceedings of the 5th International Conference on High Performance Computing*, December 1998.

[14] M. Chaudhuri and M. Heinrich. SMTp: An Architecture for Next-generation Scalable Multi-threading. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[15] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[16] F. Dahlgren, M. Dubois, and P. Stenström. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7), 1995.

[17] F. Dahlgren and P. Stenström. Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(4), 1996.

[18] M. Dubois et al. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[19] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, June 1986.

[20] D. Dunn. Intel Expects 75% Of Its Processors To Be Dual-Core Next Year. In *Information Week*, March 2005.

[21] S. Dwarkadas et al. Comparative Evaluation of Fine- and Coarse-Grain Approaches for Software Distributed Shared Memory. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, January 1999.

[22] L. Eeckhout et al. Control Flow Modeling in Statistical Simulation for Accurate and Efficient Processor Design Studies. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, June 2004.

[23] A. Erlichson et al. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. In *Proceedings of the 7th International*

Conference on Architectural Support for Programming Languages and Operating Systems, October 1996.

[24] B. Falsafi et al. Application-Specific Protocols for User-Level Shared Memory. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, November 1994.

[25] K. Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[26] K. Gharachorloo et al. Architecture and Design of AlphaServer GS320. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2000.

[27] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, April 1991.

[28] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991.

[29] J. R. Goodman. Using Cache Memory to Reduce Processor-Memory Traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, June 1983.

[30] J. R. Goodman. Cache Consistency and Sequential Consistency. Technical Report 61, SCI Committee, 1989.

[31] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th International Symposium on High-Performance Computer Architeture*, January 1999.

[32] J. S. Harper, D. J. Kerbyson, and G. R. Nudd. Analytical Modeling of Set-Associative Cache Behavior. *IEEE Transactions on Computers*, 48(10), 1999.

[33] M. D. Hill. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer*, 31(8), 1998.

[34] M. D. Hill and A. J. Smith. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers*, 38(12), 1989.

[35] J. Huh et al. Coherence Decoupling: Making Use of Incoherence. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2004.

[36] S. Kaxiras and J. R. Goodman. Improving CC-NUMA Performance Using Instruction-Based Prediction. In *Proceedings of the 5th International Symposium on High-Performance Computer Architeture*, January 1999.

[37] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, May 1992.

[38] P. Kongetira, K. Aingaran, and K. Olukutun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2), 2005.

[39] D. M. Koppelman. Neighborhood Prefetching on Multiprocessors Using Instruction History. In *Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, October 2000.

[40] D. Koufaty and D. T. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 23(2), 2003.

[41] K. Krewell. Power5 Tops on Bandwidth. In *Microprocessor Report*, December 2003.

[42] K. Krewell. Best Servers of 2004: Where Multicore Is the Norm. In *Microprocessor Report*, January 2005.

[43] K. Krewell. Sun's Niagara Begins CMT Flood: The Sun UltraSPARC T1 Processor Released. In *Microprocessor Report*, January 2006.

[44] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[45] A.-C. Lai and B. Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, May 1999.

[46] A.-C. Lai and B. Falsafi. Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, June 2000.

[47] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9), 1979.

[48] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

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

[50] D. Lenoski et al. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[51] D. Lenoski et al. The Stanford Dash Multiprocessor. *IEEE Computer*, 25(3), 1992.

[52] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. In *Proceedings of the 5th ACM Annual Symposium on Principles of Distributed Computing*, August 1986.

[53] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2), 2002.

[54] M. M. K. Martin et al. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[55] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Correctness and Performance. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[56] M. R. Marty et al. Improving Multiple-CMP Systems Using Token Coherence. In *Proceedings of the 11th International Symposium on High-Performance Computer Architeture*, February 2005.

[57] G. E. Moore. Cramming more Components onto Integrated Circuits. In *Electronics Magazine*, April 1965.

[58] K. E. Moore et al. LogTM: Log-based Transactional Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architeture*, February 2006.

[59] T. C. Mowry. Tolerating Latency in Multiprocessors through Compiler-Inserted Prefetching. *ACM Transactions on Computer Systems*, 16(1), 1998.

[60] T. C. Mowry and A. Gupta. Tolerating Latency through Software-Controlled Prefetching in Shared-Memory Multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2), 1991.

[61] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[62] S. S. Mukherjee et al. Wisconsin Wind Tunnel II: A Fast, Portable Parallel Architecture Simulator. *IEEE Concurrency*, 8(4), 2000.

[63] S. S. Mukherjee and M. D. Hill. Using Prediction To Accelerate Coherence Protocol. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, June 1998.

[64] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, August 1995.

[65] K. Olukotun et al. The Case for a Single-Chip Multiprocessor. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[66] E. Perelman et al. Using SimPoint for Accurate and Efficient Simulation. In *Proceedings of the 2003 ACM/SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2003.

[67] Z. Radović and E. Hagersten. Removing the Overhead from Software-Based Shared Memory. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, November 2001.

[68] S. K. Reinhardt, R. W. Pfile, and D. A. Wood. Decoupled Hardware Support for Distributed Shared Memory. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, May 1996.

[69] D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-Grain Software Distributed Shared Memory on SMP Clusters. In *Proceedings of the 4th International Symposium on High-Performance Computer Architeture*, February 1998.

[70] D. J. Scales, K. Gharachorloo, and C. A. Thekkath. Shasta: A Low-Overhead Software-Only Approach to Fine-Grain Shared Memory. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996.

[71] I. Schoinas et al. Fine-grain Access Control for Distributed Shared Memory. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994.

[72] I. Schoinas et al. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, October 1998.

[73] Semiconductor Industry Association (SIA). *International Technology Roadmap for Semiconductors 1999*.

[74] S. Somogyi et al. Spatial Memory Streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, June 2006.

[75] SPARC International, Inc. *The SPARC Architecture Manual, Version 9*.

[76] P. Stenström, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, May 1993.

[77] R. Stets et al. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, October 1997.

[78] M. K. Tcheun, H. Yoon, and S. R. Maeng. An Adaptive Sequential Prefetching Scheme in Shared-Memory Multiprocessors. In *Proceedings of the 1997 International Conference on Parallel Processing*, August 1997.

[79] J. M. Tendler et al. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

[80] R. Thekkath et al. An Evaluation of a Commercial CC-NUMA Architecture: The CONVEX Exemplar SPP1200. In *Proceedings of the 11th International Symposium on Parallel Processing*, April 1997.

[81] M. Tremblay et al. The MAJC Architecture: A Synthesis of Parallelism and Scalability. *IEEE Micro*, 20(6), 2000.

[82] D. M. Tullsen and S. J. Eggers. Effective Cache Prefetching on Bus-Based Multiprocessors. *ACM Transactions on Computer Systems*, 13(1), 1995.

[83] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.

[84] D. Wallin et al. Vasa: A Simulator Infrastructure with Adjustable Fidelity. In *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems*, November 2005.

[85] T. F. Wenisch et al. Temporal Streaming of Shared Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[86] T. F. Wenisch et al. TurboSMARTS: Accurate Microarchitecture Simulation Sampling in Minutes. In *Proceedings of the 2005 ACM/SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, June 2005.

[87] T. F. Wenisch et al. Simulation Sampling with Live-Points. In *Proceedings of the 2006 IEEE International Symposium on Performance Analysis of Systems and Software*, March 2006.

[88] Wm. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *SIGARCH Computer Architecture News*, 23(1), 1995.

[89] R. E. Wunderlich et al. SMARTS: Accelerating Microarchitecture Simulation via Rigorous Statistical Sampling. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, June 2003.

[90] H. Zeffer, Z. Radović, and E. Hagersten. Exploiting Locality: A Flexible DSM Approach. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*, April 2006.

[91] Y. Zhou et al. Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation. In *Proceedings of the 6th ACM/SIGPLAN Symposium on Principles and Practice of Parallel Programming*, June 1997.

[92] Y. Zhou, L. Iftode, and K. Li. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory. In *Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, October 1996.

Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 217

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title "Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology".)

Distribution: publications.uu.se
urn:nbn:se:uu:diva-7135

ACTA
UNIVERSITATIS
UPSALIENSIS
UPPSALA
2006