A Case For Low-Complexity Multi-CMP Architectures

Håkan Zeffer and Erik Hagersten

Department of Information Technology, Uppsala University
P.O. Box 337, SE-751 05, Uppsala, Sweden
{hakan.zeffer, erik.hagersten}@it.uu.se

Abstract

The advances in semiconductor technology have set the shared memory server trend towards processors with multiple cores per die and multiple threads per core. This paper presents simple hardware primitives enabling flexible and low complexity multi-chip designs supporting an efficient inter-node coherence protocol run in software. The design is based on two node permission bits per cache line and a new way to decouple the intra-chip coherence protocol from the inter-node coherence protocol. The protocol implementation enables the system to cache remote data in the local memory system with no additional hardware support.

Our evaluation is based on detailed full system simulation of both commercial and HPC workloads. We compare a low-complexity system based on the proposed primitives with aggressive hardware multi-chip shared-memory systems and show that the performance is competitive, and often better, across a large design space.

1 Introduction

The continued decrease in transistor size and the increasing delay of wires have lead to the development of chip-multiprocessors (CMPs) [1, 10, 17, 24], which implement multiple processor cores on a chip. CMPs come both in small and large system configurations. The Sun UltraSPARC T1 processor (code name Niagara) [8, 18] uses a single CMP while Piranha [1], IBM Power4 [24] and IBM Power5 [10] combine multiple CMPs to form larger systems (multi-CMPs). It is also common to use multiple hardware threads (SMT) per core in order to increase on-chip parallelism for both high-performance computing and commercial computer systems [8, 9, 10, 26].

With today's single chip systems implementing 32 hardware threads [8, 18] a traditional mid-range server actually fits on a single die. One very important question, with these trends in mind, is: Is it worth paying the extra development cost to enable even larger systems?

This paper argues that architects should focus on commercial single-chip performance and let small and simple hardware primitives, together with a software layer, provide support for shared-memory multi-chip scalability. Our system design is inspired by SMTp [3] and TMA Lite [30] and uses hardware inter-node coherence checks, but run the inter-node coherence protocol in software on the hardware thread that caused the coherence miss.

We use two per cache line node permission bits for our coherence checks and an innovative way to decouple the intra-chip coherence protocol and its on-die memory controller from the inter-node coherence protocol. The later scheme also let us use ordinary DRAM memory as a remote access cache. This DRAM remote access cache (DRAC) makes it possible to cache large amounts of remote data (gigabytes) locally and is applicable on a large spectrum of coherence designs, including traditional hardware coherent NUMA systems.

This paper contributes with the following:

- We propose new hardware primitives for inter-node coherence that greatly simplify large-scale shared memory implementations. Our hardware primitives require much less chip area than traditional hardware mechanisms.

- We introduce a new way to handle the interaction between the intra-node- and the inter-node coherence protocol in modern chip-multiprocessors. The logic is simple and greatly reduces the complexity of inter-node write backs. It also enables in DRAM remote access caching.

- This is the first paper that in great detail simulates a system with hardware coherence detection

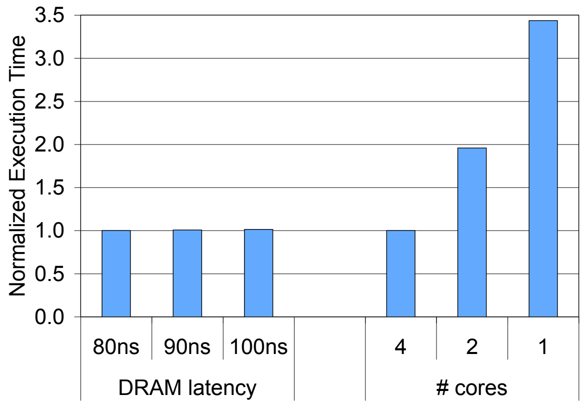

Figure 1. Normalized execution time for SPECjbb2000 while varying the DRAM access latency (left) and number of processor cores (right) for a Niagara like processor.

and a software protocol running commercial applications. We show that our system matches and often outperforms three aggressive hardware coherent distributed shared-memory machines.

The rest of this paper is outlined as follows: we start out with a short motivation in Section 2. Section 3 presents hardware modifications and the DRAC. Section 4 and 5 describe our methodology and our results respectively. We then present related work and conclude.

2 Motivation

While the big and lucrative market for small and mid-range servers is targeted by today's single chip systems [8, 18] it still adds value to support larger system configurations. However, extending a single-chip CMP system to support multi-CMP configurations comes at a high cost, both in terms of design and verification. The coherence mechanism, for example, has to be extended to support the additional inter-CMP coherence protocol. The current trend is that the memory controller and these coherence engines are moved on-die in order to reduce both the local- and the remote memory latency [1, 10, 24].

Figure 1 shows normalized execution time for SPECjbb2000 running on a single Niagara-like chip [8, 18] with four hardware threads per processor core (details in Section 4). The left part of the figure shows normalized execution time when the DRAM access latency is increased from 80ns to 100ns for four processor cores. The right part of the figure shows normalized execution time when the number of processor cores on the chip is varied from four to one. The results indicate that modern chip designs often can overlap small increases in DRAM latency but that removing cores from the chip drastically reduces chip performance.

We are convinced that support for inter-chip coherence, such as state machines and directory caches, will require chip area, and hence, degrade commercial single chip performance. One way to get around the problem is to design two separate chips: one for single-chip and one for multi-chip systems. However, this is a costly solution and we do not believe all companies can afford this approach.

The techniques presented in this paper makes it possible to design a single chip that support scalable configurations without the sacrifice of chip area, and hence, single-chip performance. Throughout this paper, we compare our system with three traditional hardware coherent machines with both on die coherence engine and directory cache. We do not reduce their performance based on the additional area used for inter-node coherence.

3 Architecture and Mechanisms

This section discusses the simple hardware primitives that have to be added to a processor to enable an efficient inter-node coherence protocol run in software. While the mechanisms proposed can be incorporated in any kind of processor, our baseline processor closely follows the Niagara design [8, 18]. It has a six-stage pipeline, separate level-one data and level-one instruction caches where the data cache is write-through to a shared second-level cache. A more detailed description and system parameters can be found in Section 4.

Our system, named CRASH (Complexity-Reduced Architecture for SHared memory), uses hardware support for detecting permission violations but runs the coherence protocol in software on the hardware thread that caused the miss. The directory is maintained by the software protocol and is stored in normal cacheable memory. Hence, the directory layout can easily be changed and optimized for each particular system configuration. On the other hand, it does occupy space in the memory hierarchy and might lead to slightly more cache misses.

Figure 2 shows a complete node diagram of the proposed architecture. All parts other than the physical DRAM and the network router are on the die. The shaded areas correspond to CRASH modifications and will be explained in detail in this section.

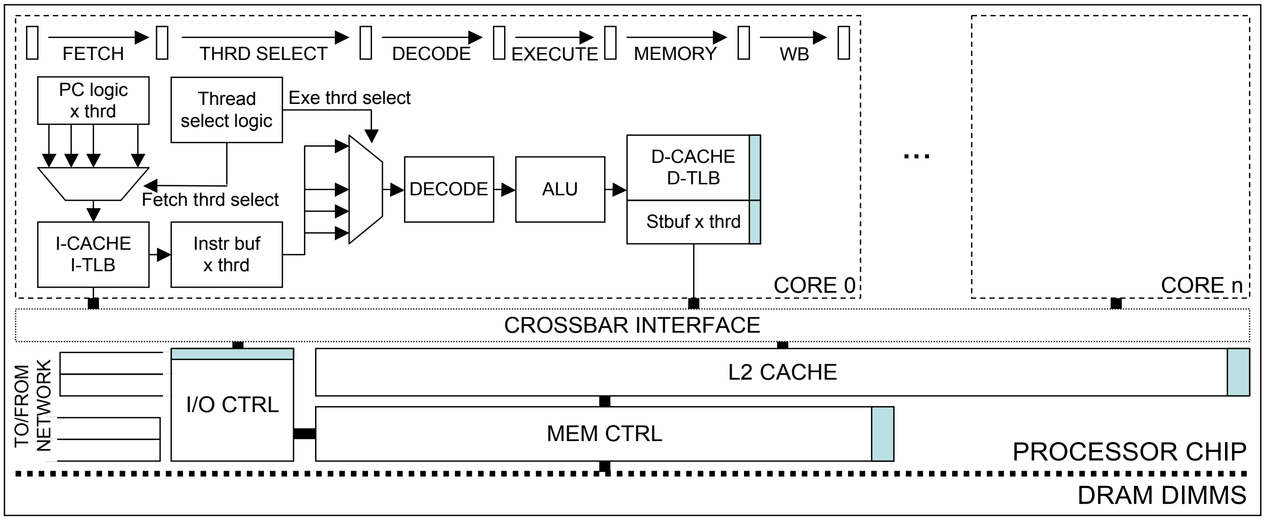

Figure 2. Node architecture of CRASH. The shaded areas represent the logic and storage added for CRASH support.

3.1 Global Address Support

When adding more nodes to a system one typically wants to add extra memory as well. In order to support the extra memory and to be able to cache remote memory in the local memory system all cache tags, TLBs and the corresponding data paths have to be extended to support extra physical address bits. These bits are not needed for single-chip configurations but for our and traditional multi-chip designs.

3.2 Node Permission Checks

In CRASH, the inter-node coherence checks are done in hardware but the inter-node coherence protocol is run in software on the hardware thread that detected the coherence violation. We use two bits of meta data per cache line to indicate a cache line's inter-node coherence state. The node read (NR) bit indicates if a node has read permission and the node write (NW) bit indicates if a node has write permission. This section discusses how the coherence checks are implemented, how the two node permission bits are maintained in all levels of the memory hierarchy, how they are updated, and how forward progress is guaranteed.

3.2.1 The Load Check

In order to guarantee that inter-node coherence violations are detected, our design checks the node read bit together with the normal state and tag check at level-one cache lookups. If the node read bit indicates read permission, the normal load behavior is performed, data is forwarded to the destination register and the load can commit. On the other hand, if the node read bit indicates that the node does not have read permission, the coherence protocol has to be invoked. Also here the normal load cache hit behavior is used but with one small exception: we assert a dedicated signal indicating an inter-node read miss. This signal is checked with a comparator at the load path of the processor core and marks the load as faulting. Hence, the load traps and the software protocol is invoked[1].

We have now covered the load check when a load hits in the level-one cache. At a level-one cache miss, on the other hand, the following procedure is followed. When data arrive from the crossbar interface it is installed in the level-one cache and is forwarded to the destination register through the load data path. However, also the node read bit is installed in the level-one cache and is forwarded on the load path (the dedicated signal). The load can commit if the bit indicates read permission. However, if the bit indicates a read coherence violation the signal is asserted and the load is marked as faulting. Hence, the coherence protocol is invoked.

[1] We rely on the SPARC V9 [28] trap- and interrupt mechanism to streamline the design and minimize additional hardware.

3.2.2 The Store Check

The inter-node store check is done through interaction between the store buffer and the level-two cache. When store data reaches the head of the store buffer, data together with the address and a mask, indicating which bytes to write, is sent over the crossbar interface to the level-two cache. The cache checks the node write bit together with the tag and the normal state. If the store is a cache hit and the node read bit indicates permission, the normal store behavior is used. Data is merged into the cache line and the store buffer is signaled that the store did succeed, and hence, can be removed from its store-buffer entry.

If the store is a cache hit but the node write bit indicates that the node does not have write permission, the update of the cache line is stopped and the store buffer is signaled that a node write coherence violation has occurred. Hence, the data has to stay in the store buffer until the coherence miss has been solved and the store can be retried. The inter-node coherence miss signal is forwarded to the hardware thread responsible for the store buffer and this particular store together with the address. Hence, a non-precise trap invokes the coherence protocol.

If the store is a second-level cache miss, on the other hand, the following procedure is followed. When data returns from the DRAM it is installed in the cache. The tag array, the status and the two node permission bits are updated. The store is then retried and the cache-hit scenario described above is used.

3.2.3 Permission Bit Maintenance

The two inter-node permission bits must be maintained in both cache and in DRAM memory. Moreover, it must be possible for the software coherence protocol to check and update them.

We have simply added the meta data bits to our cache simulator and use the same ECC trick that Nowatzyk et al. [16] and Gharachorloo et al. [5] use for the directory implementation in S3.mp and Piranha, when stored in memory. Note that the node permission bits have to be written back to memory when a dirty cache line is evicted from the second-level cache.

The coherence protocol must be able to read data without node read permission and write data without node write permission in order to do its task. We use dedicated ASIs[2] for reading and writing without permission. For example, the read without node permission ASI let data pass through the load data path to the destination register with the permission violation signal masked. In a similar way, for stores, a dedicated bit is forwarded with the store operation to the second-level cache where it is used to mask out the node write permission check. The two bits are read- and writable by the protocol.

It is also possible for the coherence protocol to read and write the two bits.

[2] Address Space Identifiers [28].

3.2.4 Store Buffer Modification

It is crucial that the coherence protocol is able to write valid data and permission bits to the local memory system even if the store buffer is blocked by an inter-node coherence miss.

We have extended the store buffer design in CRASH to incorporate a special coherence protocol entry. This entry is written by a dedicated ASI and can only be occupied by protocol stores. When occupied, it acts as if it is the head of the store buffer, and hence, takes precedence over the rest of the entries. This implies that there is no need to maintain order on this particular entry and since it can only hold protocol stores there is no need to CAM against it.

3.2.5 Guaranteeing Forward Progress

When adding a new level of coherence, in this case the inter-node coherence, one has to be careful to not loose the forward progress property of the system. Forward progress for loads is guaranteed since we do not only update the cache line with valid data but also writes the destination register of the missing load. We then execute a `done` instruction instead of a `retry` [28] when leaving the protocol. Hence, the load just occurred and its place in the global memory order is defined.

We use the store buffer mechanisms with the extra entry to guarantee store forward progress. We merge the data from the store operation that triggered the coherence protocol to start executing with the last update from the protocol code. Hence, the original store operation will take place together with the coherence protocol initiated write of permission. Also this mechanism is controlled with a dedicated ASI.

3.3 DRAM RAC

The memory controller is responsible for queues to DRAM, opening and closing of pages, RAS and CAS calculation and much more. However, we will in this paper only cover the logical CRASH modifications shown in Figure 3.

This section discusses the CRASH memory controller modifications that make it possible to cache remote data in the local memory system. The DRAC

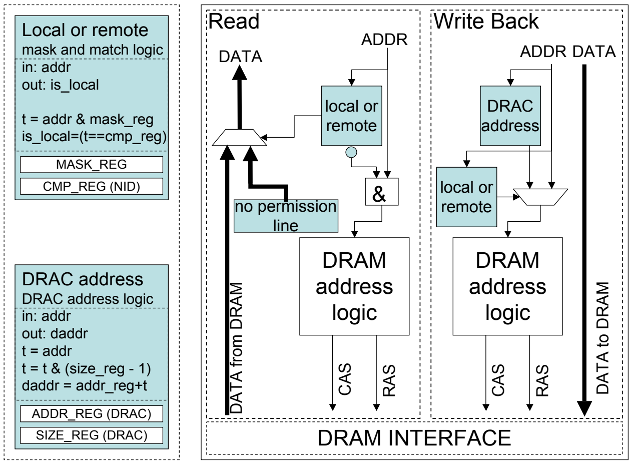

Figure 3. Logical view of the memory controller logic added for DRAC support. The boxes to the left show the four registers (`mask_reg`, `cmp_reg`, `addr_reg` and `size_reg`) and the simple logic that is needed. To the right is a high-level description of the read and the write-back logic.

can logically be seen as a direct-mapped and tag-less (in terms of hardware) cache maintained by the coherence protocol. The main task of the DRAC is to decouple the two levels of coherence.

In short the logic consists of two parts: 1) logic that classifies an address as remote or local and 2) simple DRAC address logic. The upper left part of Figure 3 shows the former, which is based on a mask register (`mask_reg`) and a match register (`cmp_reg`). The address is simply masked with the mask register and the output is compared with the content of the match register.

Also the address calculation logic is simple. It is shown in the lower left corner of Figure 3 and is based on a start address register (`addr_reg`) and a size register (`size_reg`). The physical address is masked with the size of the DRAC minus one in order to get the index. This index is then summed with the start address register in order to get the new physical address to use. Note that the size of the DRAC is controlled with a register (`size_reg`).

Data resident inside the DRAC can be reached in two different ways: 1) a remote address that maps to the corresponding index in the cache can be used, a *remote DRAC address*, and 2) the local physical address of the DRAC plus the index can be used, the *local DRAC address*.

3.3.1 Read Logic

Figure 3 shows how a read and a write-back are handled by the memory controller. The implementation shown is not optimized for performance but for clarity. Much of the logic can be run in parallel in a real implementation.

The mask&match logic is used to determine if an address is local or remote. If the address is local, the normal read handling is used and the data are supplied from DRAM. If the address is a remote address, on the other hand, the memory controller stops the DRAM access and returns a special cache line with no read and no write permission. Hence, both the NR and the NW bits indicate no permission and the data may be set to any value (we use zero). Because of the lack of permission, this cache line will trap the processor that caused the read miss.

The memory controller always returns a no permission cache line on remote read accesses since we do not want to add tag check hardware nor complexity to the system. The software coherence protocol maintains the cache and guarantees that multiple remote cache lines that maps to the same DRAC index is never present in the local cache system at the same time.

When the software coherence protocol is started by a node read permission trap, it starts out by checking the DRAC tags located in local memory. If this data indicates a DRAC hit, the local DRAC address is used to load the data from DRAM and the remote address to place it in the local cache system. In the case of a DRAC miss, the coherence protocol has to send a request to the home node in order to solve the coherence miss. Moreover, before the new data can be placed in the local cache system the remote cache line allocated at the DRAC index that the missing address maps to has to be written back to its home node. This is done by sending a write-back request to that cache line's home node and provide the data when the protocol on that node answeres to the request.

3.3.2 Write-Back Logic

A big challenge in coherent computer system design is the write-back mechanism. When data is evicted from the second-level cache a write-back request is sent to the memory-controller that is responsible for writing the data to DRAM. If this data is local data the memory controller simply uses the default handling and writes the dirty data to DRAM. If the data is remote data, the DRAC address logic is used to calculate the corresponding physical DRAM address and the data is written to the DRAC instead. Remember that our coherence protocol guarantees that this is the only remote cache line that maps to this particular DRAC index.

This logic decouples the two layers of coherence by letting all second-level cache write-backs go directly to local memory without any inter-node coherence interaction.

3.4 Deadlock Considerations

The cyclic dependencies that can occur between application instructions and protocol instructions in SMTp [3] do not apply to CRASH. This is because of the simple pipeline and the in-order memory system of the base architecture. For example, if a load misses in the data cache all instructions from this particular thread are flushed from the pipeline. This policy is implemented to let other threads use the resources, and hence, increase chip throughput. However, if the CRASH extensions are implemented in a more traditional out-of-order processor, deadlock avoidance mechanisms similar to the ones presented in SMTp have to be implemented.

3.5 I/O Controller Logic

The I/O controller has to be able to recognize certain messages from the network and be able to interrupt the responsible processor. The standard interrupt logic can be used but might be slow. In our evaluation, we have made sure that these interrupts get high priority, and hence, start executing as soon as possible.

4 Evaluation Methodology

This paper presents simulation results for multiprocessors with up to sixteen nodes where each node is an aggressive multithreaded chip-multiprocessor. This section discusses the applications and the simulator infrastructure we use to evaluate CRASH.

4.1 Simulator Infrastructure

We use the Simics full-system simulator [15] extended with the Vasa memory system extensions [27]. We have implemented a new cycle accurate chip / processor model and slightly extended the memory system simulator in order to tag cache contents etc. We simulate a SPARC-V9 system running an unmodified Solaris 9 operating system.

Core and Chip Model: We model a chip-multiprocessor where each core is a dynamically scheduled, multi-threaded throughput oriented processor designed to resemble the Sun UltraSPARC T1 processor [8, 18]. The processor model is implemented in the following way. One thread per core can fetch up to 8 instructions from the instruction cache each cycle. Round robin is used to select thread when more than one need new instructions to their eight instructions wide fetch buffers. Round robin, dependencies, branches and cache misses are used to select one thread to send an instruction down the one instruction wide pipeline. The processor does not implement branch prediction. The rest of the pipeline is a decode stage (register read), an execute stage (arithmetic operation and address calculations). The last two stages are for TLB, data cache access, trap and write-back logic. Each core is single issue and has an integer ALU, a FPU, a load/store unit and a branch unit.

SMT Capabilities	1, 2, 4 or 8 way SMT
CMP Capabilities	1, 2, 4 or 8 way CMP
Frequency	1.4 GHz
Pipeline Stages	6
Fetch/Issue/Retire Width	8/1/1
Store Buffer	16 entries per thread
L1 Data Cache	16kB, 4-way, 2 cycle hit
L1 Instruction Cache	8kB, 4-way, 2 cycle hit
L2 Shared Unified Cache	1MB, 8-way, 10 cycle hit
L1/L2 Block Size	64 bytes

Table 1. Simulated chip parameters.

SDRAM access latency	80ns
SDRAM bandwidth	Unlimited
Network Topology	Fully connected
Hop latency	40ns
NIC latency	40ns
Interconnect Bandwidth	Unlimited

Table 2. Simulated system parameters.

The memory hierarchy simulator models the latency and bandwidth of two levels of lockup-free caches per chip. The first level instruction and data caches are shared between the threads within a core. The second level cache is shared among cores and is connected to them with a crossbar. We model first-level write-through caches, while the second-level cache implement a write-back strategy. We use inclusion between the caches. We send out a prefetch to the second level cache when a store is placed in the store buffer. This idea is taken from [19] and provides memory level parallelism for stores. Table 1 shows simulated chip parameters.

System Configuration: In this paper each node consists of a single chip with an on-die memory controller. The memory controller is capable of handling local cache misses. When simulating the hardware coherent system, it also contains a hardware coherence engine making it possible to handle remote cache misses from the network interface. The protocol is a highly optimized non-blocking directory protocol. The on-chip coherence agent, responsible for node-to-node coherence, uses a fully mapped bit vector to keep track of sharers [12, 13]. The directory is located in memory but the coherence agent uses a directory cache to cache directory information. The memory controller used in the CRASH system, on the other hand, does not maintain coherence between nodes.

We evaluate our system using multiple number of nodes, number of cores per die and number of threads per core. However, we always keep the total number of simulated threads to 16 in order to avoid application differences in the results.

We have in this paper assumed unlimited bandwidth and a fully connected network. The reason for these assumptions is that we do not want interference from an unbalanced system or some "hard to find" bottleneck. It is important to note that the CRASH software coherence protocol and the hardware coherence protocol uses the same protocol algorithms so the number of packets sent and the bandwidth consumed by those packets should be equal for a particular coherence miss. Table 2 shows simulated system parameters.

Interactions with Solaris: To make the trap handling as realistic as possible, we use reserved trap types in the SPARC-V9 instruction set to implement our coherence traps. We have applied a binary patch that modifies the corresponding trap vector entry in Solaris 9 with our coherence protocol code. When a coherence trap is signaled, it is handled just as a normal trap and the protocol routines are executed just as any other trap handler instructions in the pipeline of the cycle-accurate simulator, consuming pipeline resources and polluting the caches.

Machine Configurations: This paper compares the CRASH architecture with three hardware coherent machine models as shown in Table 3. The different hardware coherent machine models vary how large the directory cache is. `HWperf` represents the most aggressive hardware system with a perfect directory cache. `HW256` and `HW64` correspond to more realistic design points with a 256kB and a 64kB direct mapped directory cache respectively. These directory caches are to be considered quite large since they are stored on the processor die. `CRASH` corresponds to the CRASH configuration and does not implement any hardware coherence nor directory cache. Instead, the DRAC address calculation shown in Figure 3 is incorporated in its memory controller and, unless stated otherwise, is configured with a DRAC size of 64MB.

4.2 Benchmarks

We use three commercial server side benchmarks and two HPC benchmarks to evaluate CRASH.

MDB: Dynamic Web Search Content Serving: We use a commercial in-memory database that responds to web search queries. The full-scale database is started and warmed. The benchmark's warm-up phase is simulated until the steady-state phase is reached. One query is approximately 1.5M instructions long.

JBB: SPECjbb2000 is a server-side Java benchmark that models a 3-tier system, focusing on the middleware server business logic and object manipulation [23]. The benchmark includes driver threads to generate transactions as well as an object tree working as a backend. Our configuration uses 24-driver threads (1.5 per processor) and 24-warehouses (with a total data size of approximately 500MB).

SAP: SAP is a server side *Sales and Distribution* benchmark. We use a commercial database server in our experiments. The simulated machine is booted, the full-scale database is started and warmed, and the benchmark's warm-up phase is simulated until the steady-state phase is reached. (Generating the steady-state checkpoint requires several months of simulation. Total data size is greater than 2 GB.) One dialog step is approximately 68M instructions long.

FFT: The FFT kernel is taken from the SPLASH-2 [29] benchmark suite and is a complex 1-D version of the radix$-\sqrt{n}$ six-step FFT algorithm. The kernel is optimized to minimize inter-processor communication.

LU: This is the contiguous version of LU from the SPLASH-2 [29] benchmark suite. It factors a dense matrix into the product of a lower triangular and an upper triangular matrix.

Data set sizes for the applications studied can be found in Table 4. The commercial workloads are all based on steady-state phase simulator checkpoints as described above. All caches are warmed before we start to collect data. For the HPC workloads the caches are warmed during the initialization phase and data is collected during the parallel region of the benchmark. Both HPC benchmarks are run to completion.

5 Simulation Results

This section shows CRASH performance when compared to the three hardware-coherent systems described in Table 3.

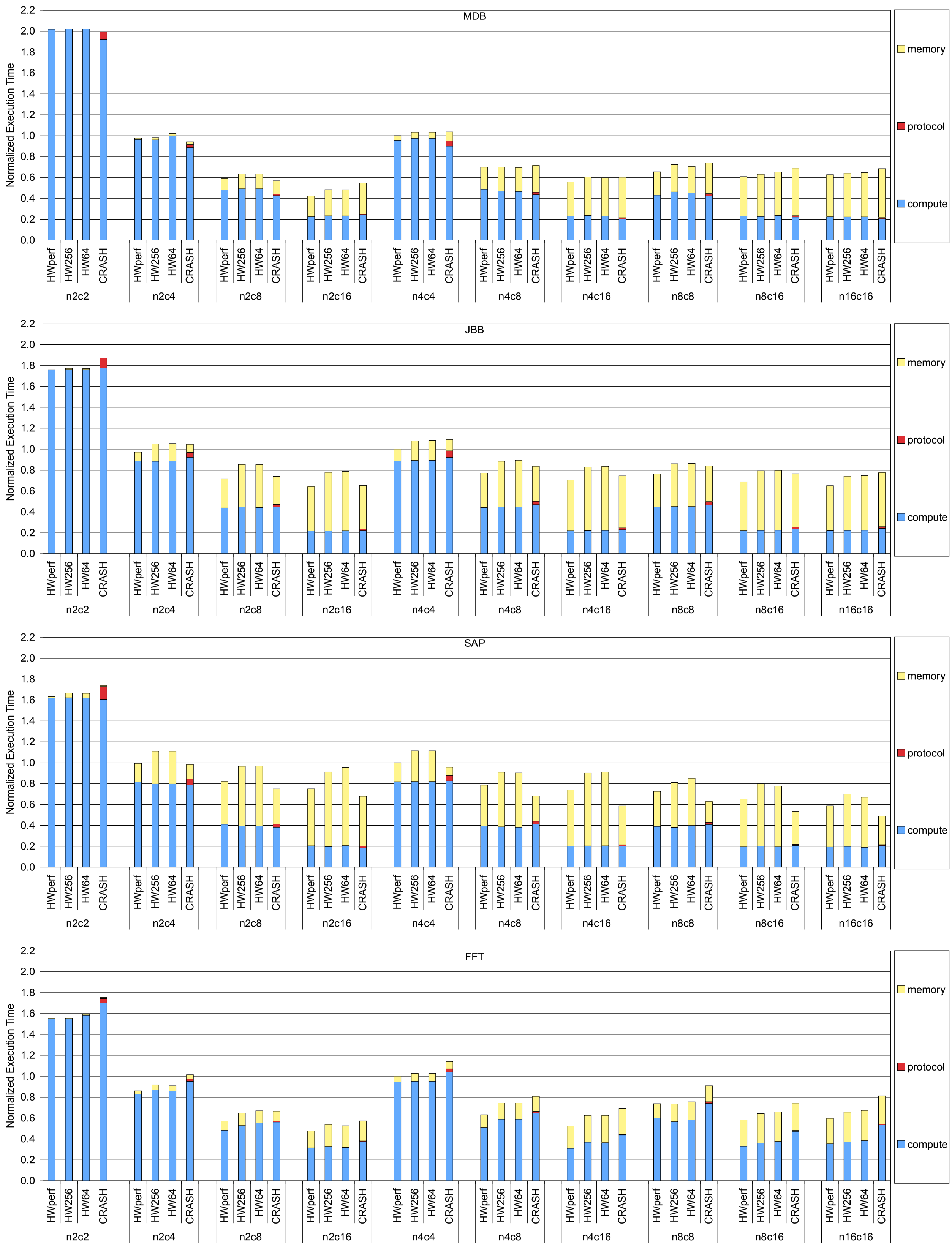

Figure 4. Normalized execution time for the different hardware and the CRASH system for `mdb`, `jbb`, `sap` and `fft`. The cycles are broken down into compute, protocol and memory cycles.

Model name	Description
`HWperf`	On-chip CA/MC, perfect directory data cache
`HW256`	On-chip CA/MC, 256kB direct mapped directory data cache
`HW64`	On-chip CA/MC, 64kB direct mapped directory data cache
`CRASH`	On-chip MC (no CA), capable of DRAC address calculation

Table 3. Machine Models (CA=Coherence Agent (HW), MC=Memory Controller)

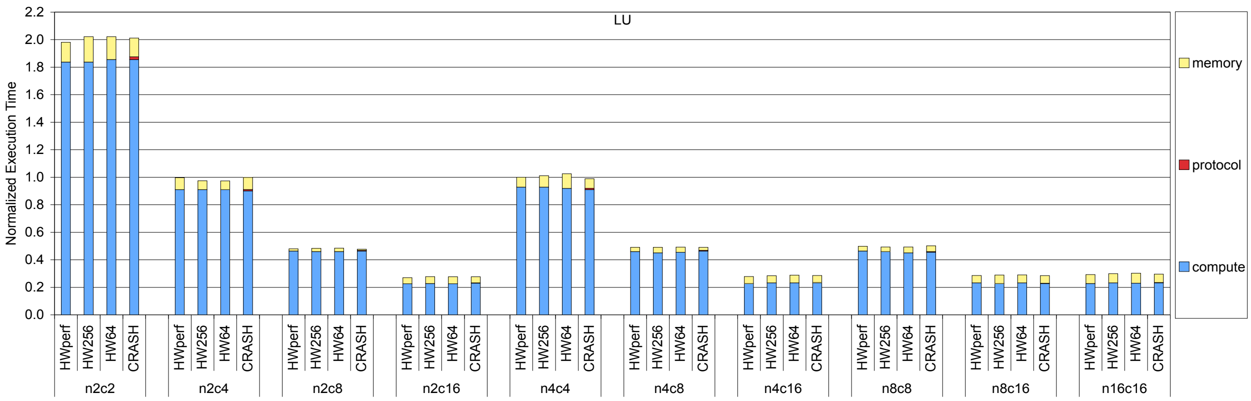

Figure 5. Normalized execution time for the different hardware and the CRASH system for `lu`. The cycles are broken down into compute, protocol and memory cycles.

Program	Problem Size
`mdb`	22,500 queries
`jbb`	90,000 transactions
`sap`	450 dialog steps
`fft`	1M points, blocked for L2
`lu`	512 x 512 matrix, 16 x 16 block

Table 4. Benchmark working set sizes.

5.1 Node, Core and Thread Scaling

Figure 4 and 5 show CRASH performance when the number of nodes, cores and threads per core are varied in different configurations. The different configurations are named after the total number of nodes and cores in the system. For example, `n2c2` corresponds to a system with two nodes and two cores (one core per node) and `n4c8` corresponds to a system with four nodes and eight cores (two cores per node). All configurations have 16 hardware threads in total. Hence, the `n2c2` (`n4c8`) configuration has eight (two) hardware threads per core.

Each bar is built up of three parts: `compute`, `protocol` and `memory`. The compute part corresponds to each cycle when the pipeline's execution units executed normal instructions. The protocol part is only valid for CRASH and corresponds to the cycles when protocol code occupied the execution units. The rest of the cycles (if any) are simply labeled memory (all threads are waiting for memory).

This paper does not argue that one configuration is better than another. Instead we investigate CRASH's performance point in a large design space. CRASH is a bit slower than the hardware coherent systems when the threads manage to hide the memory latency, as in `jbb-n2c2`. The protocol instructions has to be executed, and hence, steals the ALUs from application instructions. On the other hand, one can see that CRASH outperforms not only the reasonable hardware configurations but also the one with perfect directory cache for many node/core configurations where this is not the case. The reason for this is simply that the protocol instructions do not interfere with the application instructions and that CRASH's DRAC reduces the coherence traffic as will be shown later.

5.2 Protocol Cache Pollution

The software coherent system uses software data structures for maintaining the directory and the

Benchmark	**L1i active ratio** instr/prot instr	**L1d active ratio** data/prot data	**L2 active ratio** instr/data/prot instr/prot data
`mdb`	0.996/0.004	0.943/0.057	0.053/0.819/0.000/0.128
`jbb`	0.996/0.004	0.901/0.099	0.073/0.809/0.000/0.118
`sap`	0.867/0.133	0.914/0.086	0.273/0.609/0.035/0.083
`fft`	0.996/0.004	0.868/0.132	0.031/0.917/0.000/0.052
`lu`	0.995/0.005	0.987/0.013	0.044/0.915/0.000/0.041
Benchmark	**L1i miss ratio** with prot/without prot	**L1d miss ratio** with prot/without prot	**L2 miss ratio** with prot/without prot
`mdb`	0.013/0.013	0.093/0.087	0.038/0.038
`jbb`	0.096/0.088	0.208/0.208	0.062/0.058
`sap`	0.227/0.215	0.162/0.154	0.070/0.066
`fft`	0.006/0.006	0.059/0.059	0.028/0.027
`lu`	0.004/0.004	0.055/0.055	0.003/0.003

Table 5. Cache pollution caused by the software coherence protocol: The top rows show active ratio in the different on-chip caches and the lower rows show miss ratio for on-chip caches when the protocol (instructions and data) occupies space in the cache and when it does not.

DRAC. Hence, they occupy cache space and might degrade performance. Table 5 shows both active ratios and miss ratios for the various on-chip caches and the `n4c4` configuration. We define active ratio of a cache and a certain type of data, x, to be the average space occupied by x over time. Moreover, we define miss ratio as number of misses divided by number of accesses. Both active ratio and miss ratio are expressed as a number between zero and one.

The protocol code does not seem to be a big problem in any level of the caches for any of the applications run except for `sap`. For `sap` the average level-one instruction cache occupancy of protocol instructions is 13 percent. However, the active ratio of protocol instructions at the second-level cache is below 4 percent for all applications.

The protocol data, on the other hand, occupies between one and 13 percent of the level-one data cache and large parts of the second-level cache. This is especially true for the commercial applications.

Table 5 also shows miss ratio for all cache levels with and without interference of the protocol. The miss ratios shown are miss ratios for application data. Hence, the protocol's hits and misses are not counted for. We find it very interesting that the miss ratios are not improved that much even though protocol data occupy large portions of the second level cache. The biggest performance improvement (in terms of execution time) was seen for `mdb` and was less than five percent.

5.3 DRAM RAC Size

It might seem unfair to compare CRASH with its in-memory remote access cache to the hardware coherent systems with their fairly small (1MB per chip) second-level cache. This section argues that this is not the case and shows how the DRAC size affects CRASH performance.

The chip we use in this paper was designed to resemble a somewhat scaled-down Niagara chip [8, 18]. Where Niagara has 3MB of shared second-level cache for its 32 hardware threads (96kB per thread) we have chosen to model a 1MB cache for all our chip configurations. That means that our eight threaded chips (2-node configurations) get 128kB per thread and that our four threaded chips (4-node configurations) get 256kB per thread and so on.

Adding a third-level cache or a dedicated remote access cache to the hardware multi-CMP systems will definitely increase their performance. However, the cache level closest to memory and the number of hardware threads are typically designed to hide the local memory latency for a particular chip. Introducing a larger cache, a new cache level or a remote access cache will introduce complexity, require chip area and alter the commercial single-chip performance. The CRASH DRAC mechanism, on the other hand, simplifies the design, the interaction between the coherence protocols and uses very limited chip area while providing the system with remote access cache capabilities.

Figure 6 shows CRASH's performance while vary-

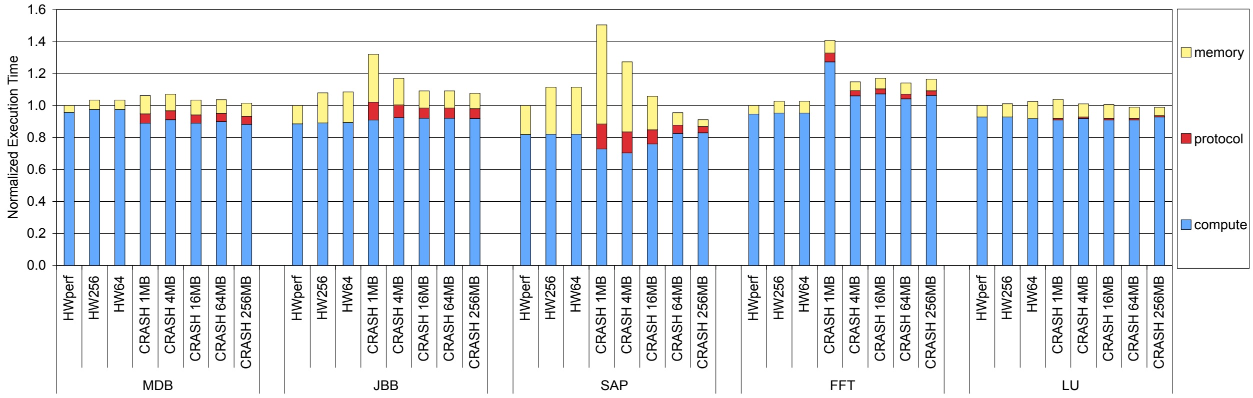

Figure 6. Normalized execution time for CRASH while varying the DRAC size. The hardware coherent systems are included for comparison. (`n4c4`: 4 nodes with one core a 4 threads each.)

ing the DRAC size for the `n4c4` configuration. The trend is very clear: having a large DRAC is important. CRASH with a 256MB DRAC is comparable to or even better than the hardware coherent systems for all applications studied except for `fft`. The performance of `fft` is improved a lot when moving from a 1MB DRAC to a 4MB DRAC. However, the improvement from an even larger DRAC is very small. The reason for this is `fft`'s communication intensive behavior where all processors communicate with all other processors. Note that the CRASH 256MB DRAC configuration is more than 15 percent faster than the reasonable hardware systems and about 10 percent faster than the system with a perfect directory cache for `sap`. Also note that the `c4n4` configuration is not one of the better configurations for CRASH when compared to the hardware systems (as can be seen in Figure 4), which indicates that much larger improvements are possible with different configurations.

5.4 Inter-Node Hop Latency

This section investigates CRASH performance when scaling the inter-node hop latency. One reason for a longer remote latency is a not fully connected network with multiple hops between nodes.

Figure 7 shows normalized execution time for the different systems while varying the hop latency from 20ns to 160ns. We find it very interesting that CRASH performs better and better compared to the hardware systems with an increasing hop latency. The reason for this is the ability to cache more remote data locally, and hence, avoiding the big penalty of inter-node hops. However, this is not the case for `fft` where the performance difference is more or less constant while varying the hop latency. The reason for `fft`'s behavior is the communication intensive behavior that a remote access cache cannot hide (see Figure 6 and 7). Note that with a 160ns inter-node hop latency CRASH is able to outperform the hardware system with a perfect directory cache for `jbb`, `sap` and `lu` and the reasonable hardware systems for all benchmarks except `fft`. Also note that CRASH outperforms the hardware systems with more than 20 percent for many of the configurations.

5.5 Fixed Sequential Prefetching

It is not only the remote access cache capabilities that are easy to change in CRASH. It is also simple to correct protocol bugs, protocol performance bugs, prefetching strategy and inter-node coherence unit size. Figure 8 shows normalized execution time for the hardware coherent systems and CRASH while the later varies the fixed sequential prefetch degree k [4]. `CRASH 0` corresponds to the CRASH system without prefetching. `CRASH n` corresponds to the CRASH system with a prefetch degree of n. That is, with a prefetch degree of 1, one additional cache line is requested at each miss. With a prefetch degree of 2, two additional cache lines are requested and so on.

CRASH performance is greatly improved by prefetching for all applications. However, the optimal prefetching degree varies between the applications. The performance of `fft`, for example, is greatly improved when a prefetch degree of three is used and is competitive with the performance of the reasonable hardware systems.

While it is possible to implement prefetching schemes in hardware coherent systems, they typically

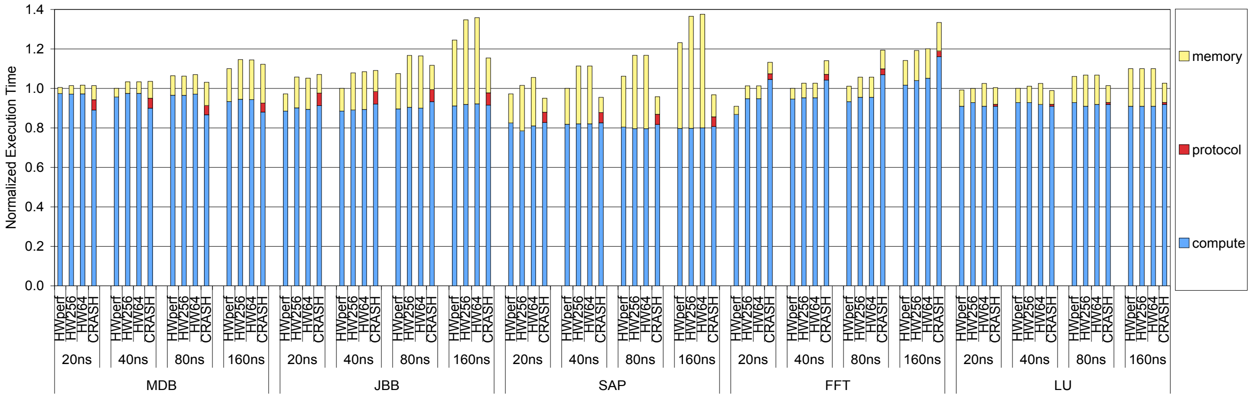

Figure 7. Normalized execution time for the hardware and the CRASH system while varying the inter-node hop-latency. (`n4c4`: 4 nodes with one core a 4 threads each.)

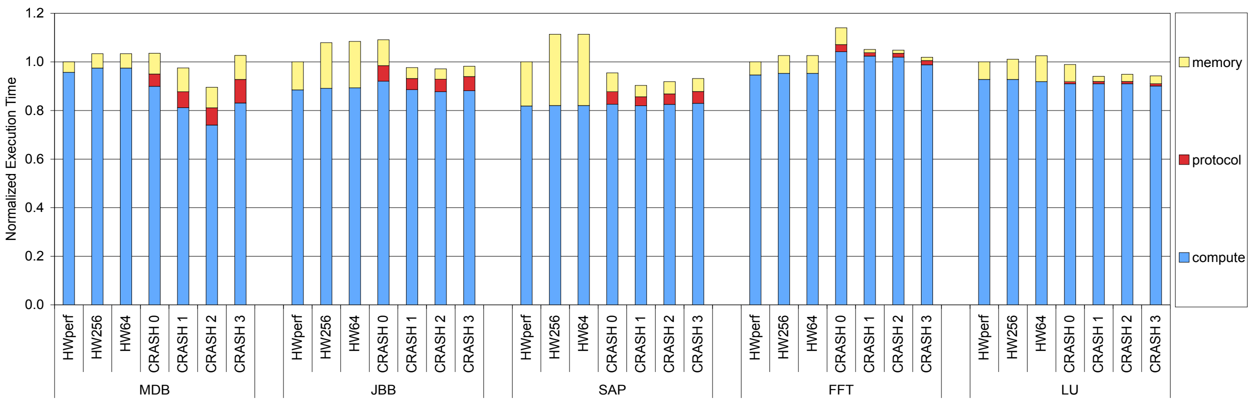

Figure 8. Performance of the CRASH system while varying the fixed sequential prefetching degree. The three hardware coherence systems are included for comparison. (`n4c4`: 4 nodes with one core a 4 threads each.)

have to perform well for all kinds of applications (or at least for commercial key applications). With CRASH it is possible to run with the optimal inter-node coherence unit size or prefetching strategy for a particular workload.

We have not taken the additional bandwidth that a prefetching protocol typically consumes into account in this investigation. However, a lot of time is typically spent on database tuning in commercial setups. Hence, the authors are convinced that testing one or two prefetching schemes or inter-node coherence unit sizes is time well spent. Such tests will of course also show if a coherence scheme saturates the available bandwidth.

6 Related Work

There is a wide range of options for hardware/software trade-offs for implementing coherent shared memory. For example, some systems run the entire coherence protocol in software on a dedicated coherence processor. Stanford's FLASH [11], Sun's S3.mp [16] and Wisconsin's Typhoon-0 [21] are examples of such machines. SMTp [3] and TMA Lite [30] are more recent proposals in which SMT threads are used to run the coherence protocol.

These designs enable flexible protocol adoptions as well as protocol bug corrections. However, FLASH, S3.mp and Typhoon-0 rely on dedicated coherence processors capable of snooping the memory bus [11, 16, 21]. This scheme is much harder on modern chip designs with on-die memory controllers. CRASH differs from SMTp [3] since it can be implemented with both on- and off-chip memory controllers. Moreover, SMTp traps on each second-level cache miss and have to solve inter-node write-backs instantly. CRASH uses the DRAC memory controller modification to simplify the interaction between the coherence protocols. For example, inter-node write backs are solved at inter-node coherence misses instead of at second-level cache evictions. TMA Lite [30] avoids the write-back complexity by replicating pages with the virtual memory system. However, the virtual memory system and entire pages have to be used for remote data caching. Moreover, CRASH do not get any false coherence misses for either loads or stores as do TMA Lite [30].

Our permission bits and the manipulation of them have much in common with the fine grained tagging of memory blocks and the nine operations defined on them as proposed in the Tempest interface for user-level shared memory [20]. However, the implementation described in [20] uses Typhoon's snooping coherence processor to implement the Tempest interface. That is, the coherence processor is needed.

It has also earlier been proposed to use the trap mechanism for fine-grained shared memory support [22, 30]. In the Blizzard-E system [22] corrupt ECCs indicate state invalid while store checks instrumented into the application protect cache lines in state shared. Our node permission bits let us detect both read and write violations with true binary transparency and without corrupting ECCs.

Another interesting approach is the "informing" memory operation proposal [7]. The idea is to let the coherence protocol assume permission for all data in the level-one data cache and trap on level-one data cache misses. However, due to control speculation and preceding exceptions that might bring unchecked data to the level-one cache, the out-of-order processor implementation is non-trivial. Shared level-one caches are likely to further complicate their scheme. The CRASH mechanism with the node permission bits is much simpler and removes the traps on level-one cache misses.

Many computer systems have implemented remote access cache support. For example, Sequent's NUMA-Q [14] and Stanford's DASH [13] both have hardware controlled fixed size remote access caches. In-memory remote access caches has been tested in Stanford's FLASH [11] and Convex's Exemplar [2, 25]. Since the FLASH design is based on a software coherence protocol run on the coherence processor the remote access cache size and its layout is trivial to change. In Exemplar, on the other hand, hardware state machines control the remote access cache, and hence, the implementation is less flexible.

The DRAC mechanism proposed in this paper is first of all a mechanism that decouples the intra-node and the inter-node coherence protocol from each other. However, the mechanism let us use part of DRAM as a direct-mapped and hardware tag-less remote access cache. The remote access cache is managed in software but the write backs from the second-level cache are performed without interaction from the inter-node coherence protocol. Note that we can implement a "second-level" n-associative remote access in software if we want to.

Other proposals, such as the Stache protocol [20] and the Simple-COMA [6], use the virtual memory system to replicate pages. This can be seen as a very large remote access cache located in DRAM memory. CRASH differs from these systems in many ways. For example, the DRAC mechanism let CRASH cache remote data at the cache line granularity without any involvement of the virtual-memory system.

7 Conclusions

This paper presents key hardware mechanisms for implementing low complexity multi-chip architectures and a new in-memory remote access cache. The remote access cache makes it possible to cache large amounts of remote data in the local memory system and greatly simplifies the inter-node write-back coherence mechanism.

Detailed full system simulation of 2-, 4-, 8- and 16-node systems with different amount of on chip resources shows that our hardware/software tradeoff is competitive with hardware-only distributed shared-memory systems across both commercial and HPC workloads and across multiple design points.

Based on the performance, the simplicity and flexibility of our system, we argue that future systems should be optimized for single chip performance and only include minor key hardware primitives for large-scale shared-memory support. We believe this will improve the commercial value of emerging chip designs since removing inter-node coherence state machines and directory caches from a chip will free up area that can be used for more compute resources.

References

[1] L. Barroso et al. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, pages 282–293, June 2000.

[2] T. Brewer and G. Astfalk. The Evolution of the HP/Convex Exemplar. In *Proceedings of COMPCON Spring'97: 42nd IEEE Computer Society International Conference*, pages 81–86, Feb. 1997.

[3] 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 (ISCA'04)*, pages 124–135, June 2004.

[4] F. Dahlgren, M. Dubois, and P. Stenström. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 6(7):733–746, July 1995.

[5] K. Gharachorloo, L. A. Barroso, and A. Nowatzyk. Efficient ECC-Based Directory Implementations for Scalable Multiprocessors. In *Proceedings of the 12th Symposium on Computer Architecture and High-Performance Computing*, Oct. 2000.

[6] E. Hagersten, A. Saulsbury, and A. Landin. Simple COMA Node Implementations. In *Proceedings of the Hawaii International Conference on System Sciences (HICSS)*, Jan. 1994.

[7] M. Horowitz et al. Informing Memory Operations: Providing Memory Performance Feedback in Modern Processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 260–270, May 1996.

[8] P. Kongetira, K. Aingaran, and K. Olukutun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 2005.

[9] D. Koufaty and D. T. Marr. Hyperthreading Technology in the Netburst Microarchitecture. *IEEE Micro*, 2003.

[10] K. Krewell. Power5 Tops on Bandwidth. In *Microprocessor Report*, Dec. 2003.

[11] J. Kuskin et al. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*, pages 302–313, Apr. 1994.

[12] J. Laudon and D. Lenoski. The SGI Origin: A cc-NUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA'97)*, pages 241–251, June 1997.

[13] 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 (ISCA'90)*, May 1990.

[14] T. Lovett and R. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*, pages 308–317, May 1996.

[15] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[16] A. Nowatzyk et al. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP'95)*, volume I, pages 1–10, Aug. 1995.

[17] 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 (ASPLOS-VII)*, pages 2–11. ACM Press, Oct. 1996.

[18] OpenSPARC.net, June 2006. Available from `http://www.opensparc.net`.

[19] R. Rajwar and J. R. Goodman. Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution. In *Proceedings of the 34th IEEE/ACM International Symposium on Microarchitecture (MICRO-34)*, pages 294–305, Nov. 2001.

[20] S. K. Reinhardt, J. Larus, and D. A. Wood. Tempest and Typhoon: User-Level Shared Memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture (ISCA'94)*, May 1994.

[21] 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 (ISCA'96)*, pages 34–43, May 1996.

[22] 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 (ASPLOS-VI)*, pages 297–306, Oct. 1994.

[23] Standard Performance Evaluation Corporation. *SPECjbb2000, A Java Business Benchmark*. White Paper.

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

[25] 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*, Apr. 1997.

[26] 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 (ISCA'95)*, pages 392–403, June 1995.

[27] D. Wallin, H. Zeffer, M. Karlsson, and E. Hagersten. Vasa: A Simulator Infrastructure with Adjustable Fidelity. In *In Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005)*, Nov. 2005.

[28] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual, Version 9*. PTR Prentice Hall, 2000.

[29] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.

[30] H. Zeffer, Z. Radović, M. Karlsson, and E. Hagersten. TMA: A Trap-Based Memory Architecture. In *Proceedings of the 20th International Conference on Supercomputing (ICS'06)*, June 2006.