An OS-Based Alternative to Full Hardware Coherence on Tiled CMPs*

Christian Fensch and Marcelo Cintra
School of Informatics
University of Edinburgh
c.fensch@ed.ac.uk, mc@inf.ed.ac.uk

ABSTRACT

The interconnect mechanisms (shared bus or crossbar) used in current chip-multiprocessors (CMPs) are expected to become a bottleneck that prevents these architectures from scaling to a larger number of cores. Tiled CMPs offer better scalability by integrating relatively simple cores with a lightweight point-to-point interconnect. However, such interconnects make snooping impractical and, thus, require alternative solutions to cache coherence.

This paper proposes a novel, cost-effective mechanism to support shared-memory parallel applications that forgoes hardware maintained cache coherence. The proposed mechanism is based on the key ideas that mapping of lines to physical caches is done at the page level with OS support and that hardware supports remote cache accesses. It allows only some *controlled* migration and replication of data and provides a sufficient degree of flexibility in the mapping through an extra level of indirection between virtual pages and physical tiles.

We evaluate the proposed tiled CMP architecture on the Splash-2 scientific benchmarks and ALPBench multimedia benchmarks against one with private caches and a distributed directory cache coherence mechanism. Experimental results show that the performance degradation is as little as 0%, and 16% on average, compared to the cache coherent architecture across all benchmarks for 16 and 32 processors.

1 INTRODUCTION

Chip-multiprocessors (CMPs) have now replaced very wide-issue out-of-order superscalar processors as they provide higher aggregate computational power, multiple clock domains, better power efficiency and simpler design through replicated building blocks. Current CMPs are commonly built around a relatively small number of cores (2 to 8), each with its own L1, and possibly L2, cache, connected through an on-chip interconnect that is either a shared bus or crossbar. Supporting shared-memory, parallel applications requires cache coherence, which is greatly facilitated by the use of buses and crossbars in current CMPs. Such interconnects allow for straightforward hardware cache coherence mechanisms based on snooping [26] and directories [14, 17].

However, such types of interconnect are expected to become a bottleneck as the number of cores increases [20]. Either access latencies have to be significantly stretched or the area required by the interconnects has to be increased to the point of becoming impractical. Tiled CMPs [4, 6, 19, 28, 29] have been advocated as a possible alternative. Such systems are built from a relatively large number (≥ 32) of relatively simple cores plus a tightly integrated and lightweight point-to-point interconnect. Unfortunately, such scalable interconnects complicate the implementation of snooping and directory protocols. In fact, the existing hardware solution to cache coherence on such interconnects is to use fully distributed directory coherence protocols [22], which are notoriously hard to implement and verify (e.g., [1]).

In this paper, we propose an alternative, cost-effective software/hardware mechanism to support shared-memory parallel applications that forgoes hardware maintained cache coherence. The mechanism is based on the key ideas that mapping of lines to physical caches is done at the page level with OS support and that the hardware efficiently supports remote cache accesses. An extension of the basic scheme only allows some *controlled* migration and replication of data. Data is migrated by refreshing the page mappings at barriers. Read-only sharing is done with the help of the existing write-protection mechanism in the TLB/OS. Overall, the mechanisms allow a sufficient degree of flexibility in the mapping and sharing. This paper also addresses in depth some issues that arise from the implementation of the technique, such as the implementation of memory locks.

By moving the key coherence handling and decision making to software (in our case the OS), the proposed scheme, like software-managed coherence mechanisms [8, 21], benefits from the possibility of modifying the protocol after hard-

*This work was supported in part by EPSRC under grant GR/S79572/01 and by the EC under grants IP 27648 (FP6) and HiPEAC IST-004408.

978-1-4244-2070-4/08/$25.00 ©2008 IEEE

ware shipping, which may allow for customizing the protocol to application behavior and for more easily fixing bugs. Like other recent attempts to divide coherence labor between OS/software and hardware [33, 34], the mechanism is likely to be more cost-effective and easier to verify and validate than distributed directory schemes. Unlike such previous trap-based schemes, however, the small hardware extensions to support an extra level of indirection between virtual pages and tiles, as well as to support remote cache accesses, minimizes the need for OS and trap handler activity. In the proposed scheme, only the processor's first load or store to data in a page requires trap handler intervention and only the system's first load or store to data in a page requires full OS intervention. Also, unlike recent hardware-only schemes for co-operative distributed caching [2, 9, 16, 35] the proposed scheme does not rely on broadcasts, centralized tag stores or large redundant tag stores in order to map, locate and access data cached remotely.

We evaluate the proposed tiled CMP architecture on benchmarks from two very different domains – the Splash-2 scientific benchmarks and the ALPBench multimedia benchmarks. We compare the system against one with an SGI Origin like distributed directory cache coherence mechanism. Experimental results show that the proposed scheme performs very close to this system with a performance gap as close as 0% (no gap), and 16% on average, across all benchmarks for 16 and 32 processors.

The rest of the paper is organized as follows: Section 2 describes the tiled CMP architecture that we assume; Section 3 describes our proposed scheme to support shared-memory parallel programs; Section 4 describes our simulation infrastructure and our evaluation methodology; Section 5 presents the experimental results; Section 6 discusses related work; and Section 7 concludes the paper.

2 TILED ARCHITECTURES

2.1 Current CMPs and Coherence Mechanisms

Current chip-multiprocessors are currently commonly built around a relatively small number of cores (2 to 8), each with its own L1, and possibly L2, cache, and are connected through an on-chip interconnect to a lower level shared cache. So far, the choice of on-chip interconnect has followed those of multi-chip symmetric multiprocessor (SMP) systems: shared bus fabrics and crossbars. The main reason for this choice is that such interconnects allow a straightforward implementation of coherence via snooping (bus) or directory at the shared cache level (crossbar). Unfortunately, as pointed out in [20], future technology scaling will lead to on-chip interconnects having different sets of tradeoffs and design issues than traditional off-chip interconnects. In particular, wire widths and the area required by connectors do not scale down at the same rate as other features shrink, which means that either the delay or the area overheads, or both, of buses and crossbars increase as process scales. In fact, the detailed study in [20] clearly shows that the area and delay overheads of buses and crossbars will become prohibitively high in CMPs with more than 16 cores in $65nm$ and smaller processes.

In order to scale the number of cores in a CMP above this barrier, and into the numbers of cores proposed for tiled architectures [4, 6, 19, 28, 29], it is necessary to resort to scalable (i.e., point-to-point) interconnect types. Such interconnects are suitable not only because their peak bandwidth naturally scales with the number of cores, but also because, due to the short-length wires and low radix, their area overhead is a fixed, independent fraction of the number of cores. However, they do not lend themselves well to the implementation of snooping cache coherence protocols (although recent research attempts to address this limitation [25]). The alternative to continue enforcing cache coherence in such systems is to employ distributed directory schemes, which have been used in multi-chip multiprocessors in the past (e.g., [1, 22]). These have proven fairly scalable, reaching up to hundreds of processors. Snooping protocols are already somewhat difficult to completely debug and verify due to subtle corner cases and state transitions [11][1], and distributed directories, with even more states, races, and corner cases, are notoriously even harder to debug and verify (e.g., [1]). Most of this complexity stems from the fact that requests cannot always be resolved at the home directory, but must in some cases generate further requests, such as forwarding and invalidation requests, which lead to complex protocols with subtle race conditions and several pending states. All this complexity is of serious concern and designing and verifying the directory coherence protocol for each new generation of the CMP architecture may likely become an expensive bottleneck.

An alternative to enforce coherence in a distributed memory system is to use the OS' virtual memory (VM) system to handle the copies of virtual pages, as was done on software DSM systems (e.g., [5, 15, 18, 23]). In this scheme, all caches are private and it is the responsibility of software to maintain coherence. As with distributed directories, such schemes have only been tested in multi-chip systems and must be adapted to operate on a CMP. A major drawback of directly porting software DSM schemes to the CMP environment is that such schemes require moving, comparing (*"diff"*), and copying data in physical memory pages to enforce coherence. This is because creating multiple physical copies of the same virtual page is the only way to cope with

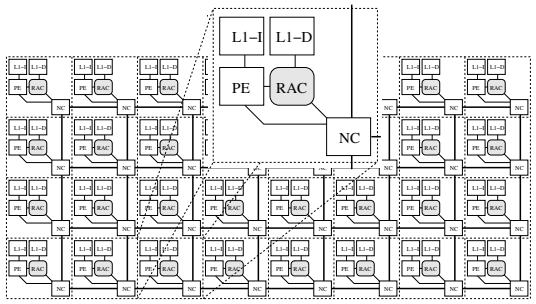
[1]Further suggestion to the difficulty of complete verification is the recent Core 2 Duo Errata AI39: "Cache Data Access Request from One Core Hitting a Modified Line in the L1 Data Cache of the Other Core May Cause Unpredictable System Behavior" [13].

false sharing and the inability of the hardware to identify which parts of a cache line have been modified. In this way, at communication points, such as lock transfers and barriers, the individual copies must be compared against the previous stable copy of the page and the modifications must be merged into a single new stable copy of the page. These operations are likely to be extremely costly in a CMP, will consume precious off-chip memory bandwidth, and generate much pollution in the relatively small on-chip caches.

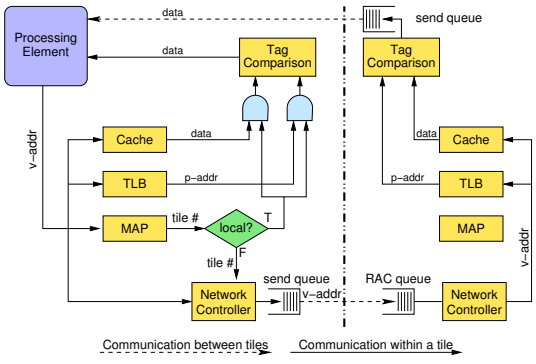
Overall, the potentially complex hardware solution of distributed directories and the potentially high-overhead software-only solution of a VM-based scheme are two extremes in the spectrum of solutions for the cache coherence problem in tiled CMPs. In Section 3 we describe our alternative to such cache coherence mechanisms, after we define the baseline tiled architecture in the next section.

2.2 A Baseline Architecture

In this paper, we are concerned with tiled CMPs consisting of 32 or more processors. Such systems are built by replicating regular building blocks, which are usually simple and small enough that the maximum intra-tile wire delay is small (1 to 2 cycles). As discussed in the previous section, snooping cache coherence approaches are unlikely to be suitable at such a scale due to the area overheads of the broadcast interconnects they require, and the only currently established alternative, namely distributed directory coherence, could prove to be prohibitively complex.



(a) Proposed tiled CMP overview.



(b) Remote cache access mechanism.

Figure 1. Proposed architecture overview.

Before we describe our design, we first present the baseline tiled CMP that we assume. We assume a fairly generic tile that consists of a compute processor (PE) that is a simple single-issue RISC processor with separate and private instruction and data caches. These first level caches are virtually indexed and physically tagged.

The on-chip interconnect fabric consists of a point-to-point network with a mesh topology where each tile is connected to its four neighbors. Each tile contains a very simple network controller (NC) that performs simple dimension-ordered routing. The number of message buffers in the NC is enough to guarantee maximum throughput, which corresponds to four non-conflicting transfers per cycle. Figure 1(a) gives a high-level overview of the architecture (the shaded gray components are explained in Section 3).

3 A HARDWARE/OS SCHEME TO AVOID CACHE INCOHERENCE

As described in Section 2.2, the baseline architecture does not support shared-memory parallel applications because it suffers from the cache coherence problem. One option is to enforce cache coherence in hardware with a distributed directory protocol. For this purpose one would add directory tags and a directory controller to each node next to the L1 data cache in Figure 1(a). In this section, we present an alternative solution. The scheme divides the work between the hardware and the OS and, in reality, does not enforce coherence across cached copies of data, but rather avoids the possibility of incoherence by not allowing multiple modifiable shared copies of data. The key ideas are to map data to tiles at the granularity of pages under OS control and to support remote cache accesses in hardware.

3.1 Caches and Coherence

This section describes the mechanism we propose to avoid incoherence. The basic idea is to treat all L1s as a single logical cache and, thus, avoid replication of data, which can lead to data incoherence. This initial architecture is extended later in Sections 3.2 and 3.3 to allow some *controlled* migration and replication.

3.1.1 Data Placement and Remote Cache Accesses

Instead of trying to keep the L1 caches coherent, the proposed scheme avoids duplicate copies of a single cache line. To achieve this, every memory line can only reside in one L1 cache (the home cache or tile) and processors in other tiles must perform remote cache reads and writes to access

the data. Thus, instead of a directory controller we add a remote cache access controller (RAC) to each tile, as shown in Figure 1(a). To receive and service remote data requests the RAC is given access to the network and it uses the dedicated ports to the cache's data and tag arrays that would be otherwise used by the snooping or directory controller.

The simplest way to place and locate data in the L1 caches whilst enforcing a single copy of each line would be to statically map lines to L1 caches based on the address. This, however, is too restrictive and takes no account of the data access patterns. At the other end of the spectrum, each line would be dynamically mapped to any one L1 cache and it would be located through broadcasts, centralized tag stores, or redundant tag stores, as has been previously proposed [2, 9, 16, 35]. What we propose is to map whole memory pages to L1 caches through extensions to the OS page table and the hardware TLB mechanisms.

We expose the internal chip structure to the OS and extend the traditional page table with a new table that maps virtual pages to architectural tiles. This is matched with a new TLB-like hardware table that caches these translations and allows for fast identification of the home L1 cache where data in the page can be found. Each tile is given one of such hardware structures, which we call a MAP. The default policy for the OS to map virtual pages to tiles is first-touch. Note that the proposed mechanism is different from simply mapping memory pages to L1 caches based on the physical address and using the virtual-to-physical page translation mechanism to provide the run-time mapping. The problem with the latter is that physical addresses are bound to specific L1 caches, which limits the OS flexibility in allocating physical memory and may lead to fragmentation and inefficient use of physical memory. Additionally, it makes any changes to the mappings much more involved, as the physical pages have to be moved in memory. It is for these reasons that we decided to add this extra level of indirection.

One important design decision at this point is where to provide virtual-to-physical address translation. Traditional CMPs keep all the translations of the local processor in the local TLB and ship only physical addresses to access lower level caches. A problem with using physical addresses for the remote cache accesses in our architecture appears when virtually indexed L1 caches are used, which is often the case in order to speed up accesses from the local processor. Thus, performing the virtual-to-physical address translations locally in the case of remote L1 accesses would require some (impractical) inverse translation at the remote tile. Our solution is to keep the virtual-to-physical address translations only in the TLB next to the home L1 cache and to ship virtual addresses over the network for remote cache accesses. Note that this is not intrinsic to our scheme, but a solution in case one wants to use virtually indexed L1 caches; with physically indexed L1 caches our proposed scheme would work as usual with physical addresses on the network.

In this scheme a processor request proceeds as follows (Figure 1(b)). Firstly, the virtual address is simultaneously used to index the local L1 cache, to perform a local TLB lookup to obtain the physical address, and to perform a local MAP lookup to obtain the identity of the home tile. If the result of the MAP translation points to a remote L1 cache, the local cache access is aborted. In this case, the result of the local TLB lookup is also ignored, including a possible TLB miss. The virtual address is then shipped to the RAC in the remote tile over the network. At the remote tile, the virtual address is simultaneously used to index the L1 cache and to perform a TLB lookup. To avoid delaying local cache requests due to remote cache requests we provide an extra port to the TLBs. Since our L1 caches are virtually indexed, the cache lookup can proceed in parallel with the TLB lookup and the extra TLB latency due to the extra port is unlikely to have any impact on the overall L1 access latency. If the TLB lookup succeeds then a tag comparison follows, using the physical tag. A cache or TLB miss is handled as usual. If the result of the MAP translation points to the local L1 cache then the local cache access proceeds as usual.

The above discussion only applies to data caches. Each tile has its own read-only instruction cache.

3.2 Data Migration

The proposed first-touch data mapping strategy, combined with the fact that mapping is done at the granularity of pages, may lead to poor performance when data migrates across threads. Mechanisms have been proposed to allow migration and replication of memory pages in CC-NUMA machines [31]. These are tailored to much larger systems with larger latencies, and, thus, we borrow some of their ideas but adapt the mechanisms and policies.

We propose a simple mechanism that allows for some degree of migration by invalidating the mappings of virtual memory pages to L1 caches. This is done by invalidating the MAP table in all tiles. After an invalidation, a first-touch policy is again used for the new mappings. Thus, invalidating the mappings does not in itself migrate pages, but it creates an opportunity for this to happen. The invalidation is more easily implemented at a quiescent state where there are no pending memory requests on chip. A natural point to perform such invalidation is at barriers. In many well-designed applications barriers are used to signal change in the data access pattern and communication across threads. Thus, barriers are also naturally good points for re-mapping and migration. Finally, to effect the migration of the data, all dirty lines in the L1 caches must be written back at a mapping invalidation such that the modified data may be reachable after the re-mapping.

The actual invalidation is done in two phases. First, each

processor invalidates the local MAP table just before joining the barrier. This is done with a new instruction that is very similar to the existing `tlbia` instruction in the PowerPC IS. At this point, the local cache controller starts writing back dirty cache lines to main memory with the goal of hiding the write-back overhead with the idle synchronization time.

In the second phase, before releasing the barrier, one processor invokes a special system call to invalidate the OS' internal MAP table. Also at this point (before barrier release) all tiles write back all remaining dirty cache lines. When the writebacks are completed, the contents of the caches are invalidated and the barrier is released.

3.3 Read-Only Data Sharing

The proposed baseline scheme, coupled with the extension to refresh mappings to allow migration, is likely to work well as long as there is not much sharing of data at the granularity of pages. Whilst full-blown sharing requires line-based hardware coherence, some degree of sharing can be easily enforced by the OS with minimal hardware support. What we propose is a simple mechanism that allows sharing of pages across multiple readers and a single writer at any given time. The mechanism works as follows. The first processor to touch a given page for reading obtains a local mapping for it whilst the OS marks the page as read-only in the page table and in the processor's TLB and MAP. Other processors touching the same page for reading are allowed to create local mappings for it, also in read-only mode. At this point, the OS *does not* need to keep track of which processors are sharing the page. The first write by a processor to a page is intercepted by the OS, which then marks the page as modified and makes this processor the owner of the page. Subsequent reads by other processors with existing local mappings can continue to use these mappings and access local data. However, subsequent writes by other processors when intercepted by the OS will not be allowed to proceed locally, but will generate a MAP entry (or change it if one already exists) that points to the owner node. Similarly, reads by processors without a local mapping for the page will generate an entry pointing to the owner node. Figure 2 shows a state diagram for the complete protocol. Note that most of the state transitions occur only at the OS level and the hardware state machine (corresponding to the MAP state in the figure) is fairly simple.

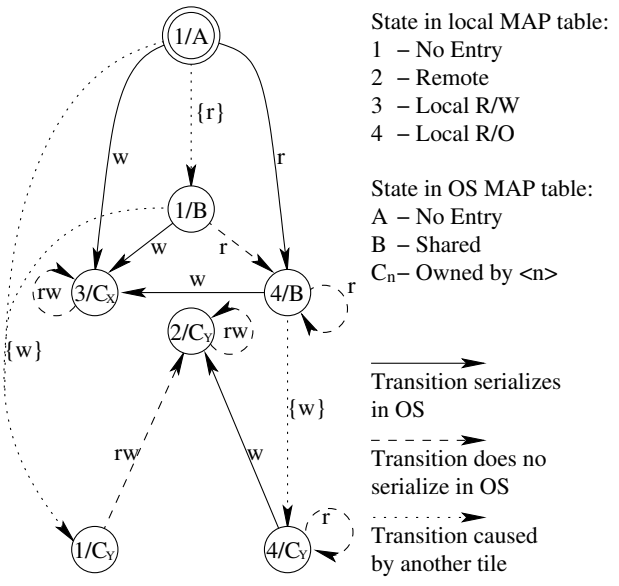


Figure 2. Sharing protocol for node X. The state of a page depends on its local (numbers) and OS (letters) state. Actions shown in braces are those taken by some other node Y.

The mechanism just described allows processors to continue using local mappings and locally cached data even after other processors write to data in the page. To prevent stale data from being used we assume a release consistency memory model and invalidate the MAP entries for shared pages on lock acquire operations. This is done by a special instruction that clears the valid bit of a MAP entry if the shared bit is set. By doing so we guarantee that all accesses to data modified by other processors will use a new remote mapping and will become remote. Entries that point to non-shared data don't have to be invalidated, since no migration happens at lock acquires and, thus, they do not change. It is also necessary to extend the barrier actions used with the migration mechanism of Section 3.2 to include a full cache flush in addition to the writebacks and the refresh of the mapping. No special action is required on lock releases.

While this mechanism may seem very similar to previous software cache coherence mechanisms (e.g., [18]), it differs from these in one crucial way. Namely, it does not allow multiple writers, reverts to a single up-to-date copy of every page upon a write, and enforces remote cache accesses in such cases. The key benefit of this is that in our scheme, no multiple modified copies of physical pages exist at any time and, thus, there is no need to perform expensive diff operations and copy data in memory.

3.4 Synchronization

Memory locks have been implemented in the past using either *compare&swap*-style atomic instructions or *load-link store-conditional* (LL-SC) pairs. The latter approach has been favored recently because it is easier to implement in hardware with cache coherence.

In our proposed architecture, *compare&swap*-style primitives can be more easily implemented than in current multiprocessors. This is because there is no replication of the lock variable in multiple caches and it is, thus, easier to enforce the atomicity of the primitive. Implementing this prim-

itive then only requires adding the compare logic to the cache controller and blocking subsequent requests from other processors until the swap is performed.

On the other hand, implementing *load-link store-conditional* pairs in our proposed architecture is more difficult than in current multiprocessors with cache coherence. In current CMPs, these are easily implemented by keeping a RESERVE register in the local L1 and relying on the hardware coherence mechanism to detect conflicting stores (Figure 3(a)). Keeping the RESERVE in the local L1 of the requesting processor will not work, however, without cache coherence. Instead, to implement the LL-SC primitive we place the RESERVE register in the home L1, and this register is then shared by all processors attempting to obtain any locks that map to this L1 cache. However, now a livelock is possible when processors attempting to lock *different lock variables* displace each other's LL from the RESERVE register (Figure 3(b)). Our solution to this is to change the operation of the RESERVE register such that once set it cannot be overwritten by LL requests to other lock addresses.

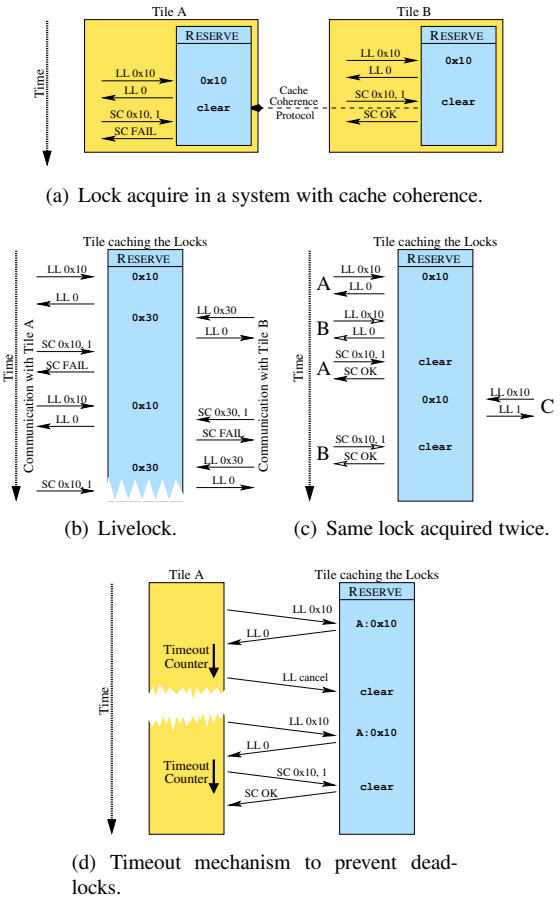
Another problem with this approach is shown in Figure 3(c), where more than one processor obtains the same lock simultaneously. This can happen when the LL and SC operations of three processors are interleaved in such a way that a second SC incorrectly succeeds because it is matched with the third LL. Our solution to this problem is to extend the RESERVE register with the ID of the tile that successfully sets it, and to only consider successful SCs that match the value in the register and come from the same tile.

The solutions proposed so far lead to another problem when the thread holding the RESERVE register fails to issue the matching SC, either accidentally or maliciously. To handle this, we introduce a timeout mechanism to clear the RESERVE register. To account for variabilities in latencies in the network, we place this timeout mechanism not in the tile holding the RESERVE, but in the requesting tile, which is then responsible for sending a special *reservation cancel* message to the tile holding the lock (Figure 3(d)).

One final side effect of using the mechanism described in Section 3.3 is that the LL instruction has to be treated as a write when it comes to replication. This guarantees that any updates to the lock variable will become visible to processors issuing LL instructions even if they previously had a read-only copy of the page.

3.5 Multi-Level On-Chip Cache Hierarchies

The design proposed so far assumes only a single level of cache per tile and no other level of cache on chip. In some cases, a higher storage capacity per tile may be required. Our architecture can be extended to work with L2 caches in each tile and our key ideas can still be applied. In this case, the L2s take the roles of the L1s in the architecture described so



(a) Lock acquire in a system with cache coherence.

(b) Livelock.

(c) Same lock acquired twice.

(d) Timeout mechanism to prevent deadlocks.

Figure 3. Problems of locks implemented with *load-link* (LL)/*store-conditional* (SC).

far and constitute a single logical shared cache. Again, mapping of memory lines to L2 caches is done at the granularity of pages with both OS and hardware support. The migration and shared-only replication mechanisms can still be applied. The only requirement is that the L1 caches must only be allowed to cache lines that are mapped to the local L2 cache. The RAC is still connected to the L1. Coherence between the L1 and L2 in the same tile can be easily maintained by making the L1 write-through.

3.6 Cost Comparison with Directory Coherence

Since we are proposing to replace a directory controller and its protocol with our RAC+MAP and a combined hardware/OS protocol, it is relevant to compare both schemes' area and complexity overheads. In particular, our main goal is to provide a less complex alternative. A comprehensive comparison between the two competing approaches would require the full design of the controllers and their circuit implementation. This is a highly involved task and, instead, we

attempt to provide some intuition into why we believe our scheme is less complex.

Like a directory controller, the RAC has to handle remote read and write requests. Unlike a directory controller, it does not have to deal with forwarded transactions and multiple invalidations, which lead to complex protocols with subtle race conditions and several pending states. The RAC can directly handle requests and generate responses for all transactions in our protocol. Thus, the RAC has fewer states and a much simpler finite state machine, which means that it has simpler logic than a directory controller does. This means that the resulting protocol is simpler to verify and validate.

As far as state storage is concerned, there is probably no significant difference. For instance, for a 32 tile system a MAP table with 128 entries, each with 22 bits (15 bits for the virtual address tag, 5 bits for the tile ID, 1 shared bit, and 1 valid bit) would have a total of 352 bytes. A directory for 32Kbytes L1 caches and 32bytes lines would have 34 bits per entry (32 bits for the sharing vector and 2 bits for line state), for a total of about 4Kbytes.

On the negative side, our system requires an additional port to the 4-way associative TLB to handle remote accesses independently from the CPU. As we mentioned earlier this is unlikely to impact the overall L1 access latency with our virtually indexed caches.

4 EVALUATION SETUP

4.1 Applications

For our performance analysis, we use the Splash-2 benchmarks [32] and three ALPBench benchmarks [24]. The Splash-2 benchmarks are representative of scientific and engineering workloads and the ALPBench benchmarks are representative of multimedia workloads. Both benchmark suites use explicit locks and barriers, assume the release consistency memory model, and rely on hardware maintained cache coherence if caches are used. We use the reference inputs for the Splash-2 benchmarks except *radiosity* for which we use a reduced input set to keep simulation time manageable. Similarly, we reduced the input for *mpegdec* to only 20 frames. Because the input sets for the ALPBench benchmarks were not intended to be used with more than 16 processors, we do not simulate larger systems for these benchmarks.

Speedups are reported with respect to the execution time of the sequential programs on a single processor after initialization. Table 1 lists the benchmarks we used.

The benchmarks were compiled with gcc 3.4.4 and glibc 2.3.5 for PowerPC. Compiler and library were modified such that they use synchronization primitives that have been adapted to our architecture.

	Benchmark	*Input*	*Instr.*	*Lock*	*Barr.*
Splash2 Ker.	cholesky	tk29.O	1,234M	72,075	3
	FFT	65,536 points	58M	32	7
	LU	512x512 matrix 16x16 block	389M	32	67
	radix	262,144 keys	54M	406	12
Splash2 App.	barnes	16,384 particles	4,361M	69,360	18
	fmm	16,384 particles	2,903M	47,074	34
	ocean	258x258 grid	412M	6,656	900
	radiosity	demo	646M	281,217	19
	raytrace	car	2,006M	95,528	2
	volrend	head	1,344M	38,604	20
	water-nsq	512 molecules	652M	35,360	19
	water-spa	512 molecules	664M	609	19
ALP	facerec	ALP Training	2,826M	30	3
	mpegdec	525_tens_040.m2v	1,049M	29	41
	mpegenc	Output of mpegdec	9,477M	29	40

Table 1. Characteristics of the applications used. The number of instructions refers to the total number for a sequential execution of the benchmark. The number of locks refers to those encountered by all 32/16 tiles (Splash-2/ALPBench) within the application (not library) code.

4.2 Simulation Environment

L1 D-cache	size	32K
	hit latency	3 cycl.
	miss latency	200+16 cycl.
	line size	32 bytes
	associativity	4-way
	writeback buf.	8

TLB/MAP	entries	128
	page size	4K
	associativity	4-way
	hit latency	1 cycl.
	miss latency	200 cycl.
RAC input queue		32 entry

Remote cache access latency without any congestion:
$2 * (h + w) + t + 1$, where h is the number of hops, w is the number of words in the message (2 or 3), and t is the access time at the remote cache.

Table 2. Memory system configuration.

We implemented a simulator using the Liberty Simulation Environment (LSE) [30]. A tile consists of a PowerPC core, a network controller, a data cache module, and a private instruction cache. The single-issue CPU is implemented as an 8-stage pipeline running at 2GHz and is simulated in detail. The cache has been implemented with the cache module from SimpleScalar [3]. The details for the memory system are shown in Table 2. We also implemented a detailed wormhole routed interconnect, where contention is accurately simulated at the network end points as well intermediate nodes. System calls and interrupts to the OS are assumed to take 2000 cycles.

4.3 Systems Evaluated

We compare our architecture against a similarly configured one where the L1 caches are kept coherent on a cache line basis through an SGI-Origin-like distributed directory protocol [22]. We note that developing this protocol was

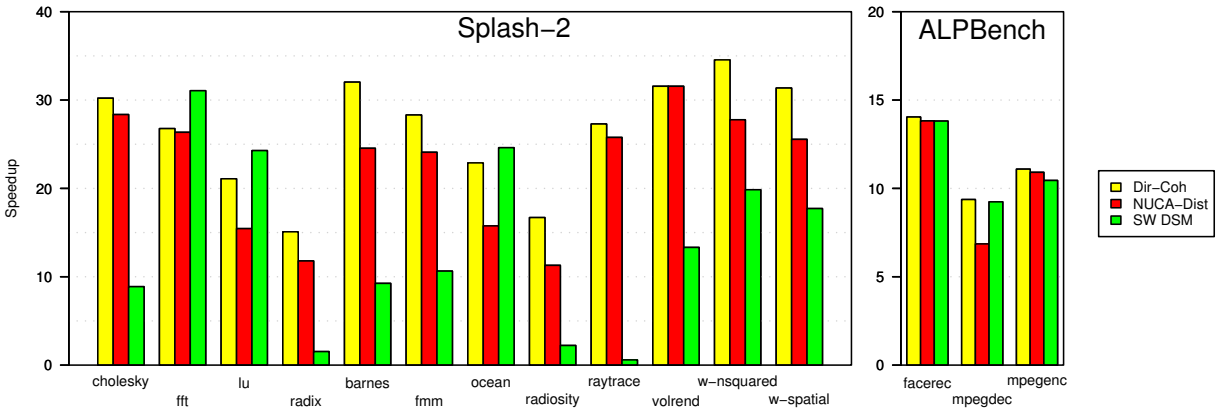


Figure 4. Speedups for 32 (Splash-2) and 16 (ALPBench) tiles compared to the execution time of a single tile.

greatly simplified by the use of common simulation artifices and the complexity we found is a far cry from the complexity we expect from a real implementation. For fairness of comparison, we augment the directory scheme with migration of pages at barriers, which minimizes any negative effects from the first-touch home-allocation policy in the initialization phase. The cost of migration is the same as in our system: 2000 cycles plus the cost of flushing the caches. For the directory controller, we assume an aggressive hardware implementation that requires only 5 cycles to process each request. We also compare our architecture against one that maintains cache coherence through a TreadMarks-like software DSM protocol [15]. Our implementation is much simplified in that it only takes into consideration the overhead of creating diffs and cache pollution by twin and diff creation. To estimate the cost of the diff we wrote a highly optimized kernel that compares the contents of two physical pages in memory and writes back one of the values if they differ. The cost was measured to be about 50K cycles.

We refer to the systems as *NUCA-Dist* for a system that implements our architecture with both re-mapping and read-only sharing of pages, *Dir-Coh* for the system with directory coherence, and *SW DSM* for the system with software DSM coherence.

5 EXPERIMENTAL RESULTS

5.1 Overall Performance

We start by comparing the overall performance of our architecture against the hardware distributed directory system. Figure 4 presents the speedups for 32 (Splash-2) and 16 (ALPBench) tile systems of *Dir-Coh*, *NUCA-Dist*, and *SW DSM*. We can see that *Dir-Coh* scales well for most benchmarks, with an efficiency (speedup divided by number of processors) of 81% on average. These results are somewhat better than those in [22] mainly due to the lower communication latencies observed in a single chip multiprocessor.

Looking at the performance of our scheme (*NUCA-Dist*) we can see that it performs fairly close to the hardware directory coherence system, with a performance gap for 32 processors ranging from 0% (no gap) to 32% (for *radiosity*), and 16% on average. Moreover, the performance gap is less than 10% for 6 out of 15 benchmarks, which is an impressive result considering that the directory coherence system uses a very aggressive hardware implementation and that our architecture requires only simple hardware support.

Finally, *SW DSM* performs, with few exceptions, significantly worse than the other systems. While the system performs very well on benchmarks that mainly use barriers for synchronization (the good results are possibly due to our simplifications), the results show that it is not able to provide sufficient scalability for most applications. The gap to our system is on average 27%, ranging from -57% to 98%. These results are in line with those reported in [12].

5.2 Memory Access Breakdown

To better understand the behavior of the proposed architecture, we track the outcome of each processor memory request. Figure 5 shows the breakdown of memory requests for each benchmark and for configurations with 32 (Splash-2) and 16 (ALPBench) processors. For each benchmark and configuration, the bar is normalized to the total number of processor memory requests, which does not vary noticeably across the different systems. The bars are broken down into the following components: accesses that hit in the local L1 cache (*local hits*); accesses that hit in a remote L1 cache (*remote hits*); accesses that go off-chip following a miss in the local cache (*local miss*); and accesses that go off-chip following a miss in a remote cache (*remote miss*).

The figure shows that the fraction of off-chip accesses is fairly small in most cases, with the exception being *ocean*, where the off-chip accesses for all systems account for about 12% of all requests. Another exception is *facerec*, where sequential execution shows only a small number of off-chip

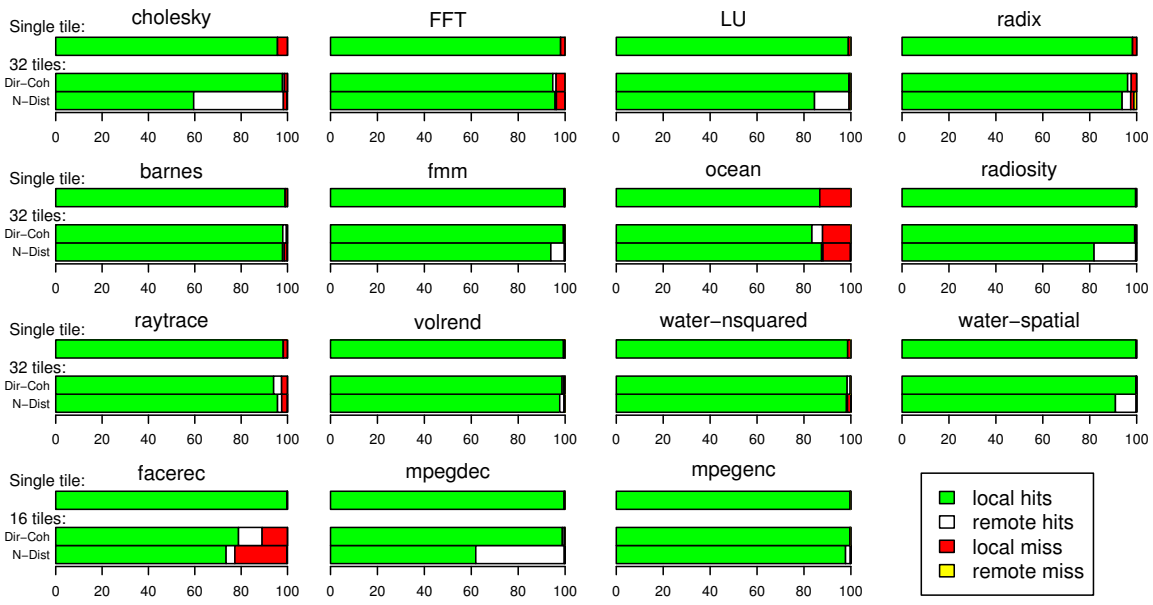


Figure 5. Distribution of memory accesses into local and remote, further divided into cache hits and misses.

accesses, but both parallel systems show a large fraction of off-chip accesses. Given this generally small number of off-chip accesses, we expect the main differentiating factor to be the ratio of local to remote cache accesses.

The results for *NUCA-Dist* show that the fraction of remote cache accesses is fairly small for most benchmarks, except *cholesky* and *mpegdec*, and, to a lesser extent, *lu* and *radiosity*. Such a relatively small number of remote cache accesses partially explains the good performance of our architecture for many benchmarks. An interesting case is *cholesky* where the fraction of remote cache accesses is high compared to most benchmarks, but its performance with *NUCA-Dist* is good. On the other hand, some benchmarks, such as *ocean* and *barnes*, show a small fraction of remote cache accesses, but their performance with *NUCA-Dist* is not as good as some of the other benchmarks. The results for *Dir-Coh*, on the other hand, show that it incurs very few remote accesses (i.e., cache-to-cache transfers), which mainly explains its very good performance.

To try to further reduce the amount of remote accesses in *NUCA-Dist* we experimented with 1KByte pages. The results (not shown) were, however, not much different from those with 4KByte pages and the small gains from the reduction in remote accesses were negated by the increase in cold and capacity misses in the TLB and MAP tables.

The impact of local, remote, and off-chip accesses can be further seen in Figure 6, which shows the average load latencies, in cycles, for the different types of loads for *NUCA-Dist* and for the average load for *Dir-Coh*. While the latencies for remote loads in *NUCA-Dist* are significantly larger than those of local loads, the average latencies are fairly close to the local ones and, thus, very close to those in *Dir-Coh*.

Migration and replication not only improve the average load latency by converting remote accesses to local ones, but also reduce the average load latency of the remote loads themselves (results not shown). This is because reducing remote accesses reduces the contention that occurs when multiple requests target the same tile.

5.3 Network and Contention Effects

One important effect of our proposed mechanism is a potential increase in the number of messages in the network, due to the remote accesses used in the scheme. To properly account for this effect, we modeled the network in detail including congestion both at intermediate nodes and at the end points. Congestion at the end points does lead to some performance degradation and is one of the main reasons for the relatively large remote cache access latency shown in Figure 6 (note that for a 32 tile system the uncontended remote cache access latency should be around 18 processor cycles). On the other hand, our results show that congestion inside the network is small and leads to negligible performance impact. One of the reasons for this is the relatively small number of messages in-flight in the network at any given time.

5.4 Impact of Flushing and Invalidations

Our read-only sharing scheme (Section 3.3) involves the potentially very expensive operations of flushing caches on barriers and invalidating the MAP table on lock acquires. To assess the actual impact of these operations' overheads on

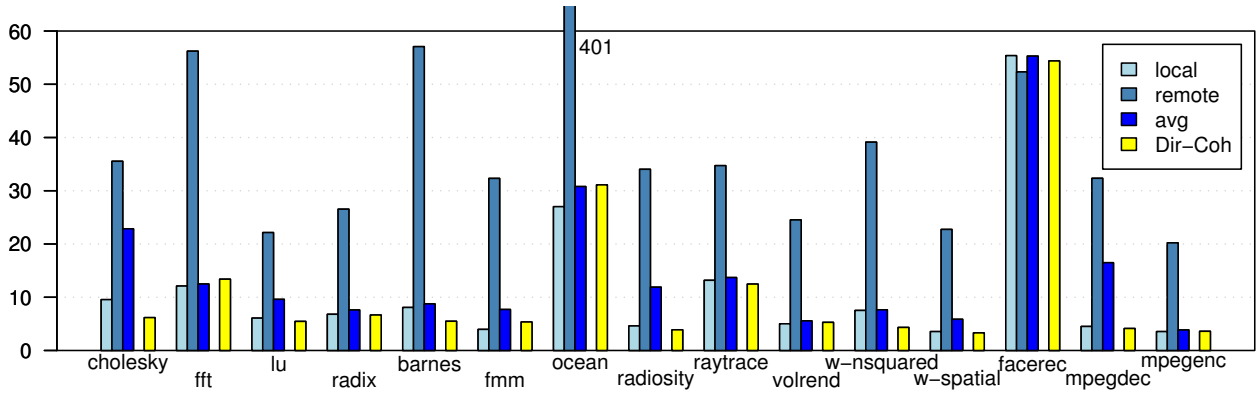


Figure 6. Average latencies for local, remote, and all loads for *NUCA-Dist*. The average latencies for *Dir-Coh* are shown as a comparison. The y-axis shows the latency in processor cycles.

Benchm.	*Barrier*	*Lock*
cholesky	<0.01	0.77
fft	0.23	0.04
lu	8.01	<0.01
radix	8.75	0.17
barnes	<0.01	<0.01
fmm	1.67	0.21

Benchm.	*Barrier*	*Lock*
ocean	10.56	<0.01
radiosity	3.74	<0.01
raytrace	<0.01	5.57
volrend	0.00	0.00
water-nsq	0.18	0.43
water-spa	0.47	0.04

Table 3. Overhead for *NUCA-Dist* with 32 processors in % caused by flushing the cache at barriers and invalidating the MAP table on a lock acquire.

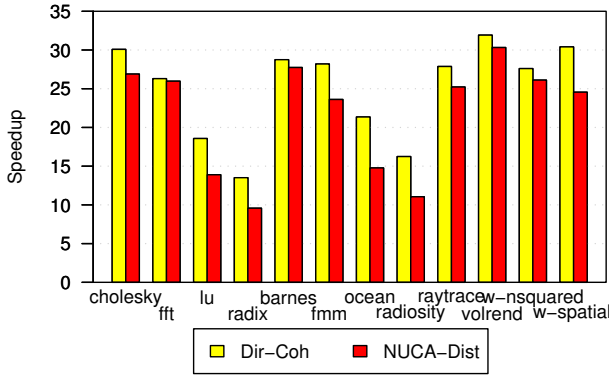


Figure 7. Speedups for 32 tiles with L2 caches compared to the execution time of a single tile also with L2 cache.

our benchmarks, we run a modified version of our scheme that does not suffer from these overheads.

The results of this analysis for a 32 core system are shown in Table 3 for the Splash-2 benchmarks. For most benchmarks, the overhead stays well below 1%. Exceptions are *ocean*, which experiences close to 11% overhead at barriers, and *raytrace*, which experiences 6% overhead at locks. The overhead for *ocean* was expected considering that this benchmark has 900 barriers. Similarly, since *raytrace* has a high number of locks, it is not much of a surprise that it suffers from invalidating the MAP table. Still, other benchmarks have similar numbers of locks and do not suffer as much. In these benchmarks a significant number of pages are not mapped as shared on a lock acquire, and thus are not invalidated. While these overheads are a non-negligible cause of some performance loss in three benchmarks, they do not affect the others as badly as one might expect.

5.5 Multi-level On-Chip Cache Hierarchies

The design evaluated so far assumes only a single level of cache per tile. We also evaluated systems with a 128KByte L2 cache per tile and a write-through L1 cache with the same size as before. The total L2 capacity on chip of 4MBytes and the relatively small capacity per core is in line with what could be expected from a CMP with 32 cores. Each L2 has 20 cycle access time.

Figure 7 shows the speedup results of such a system for *Dir-Coh* and *NUCA-Dist* for the Splash-2 benchmarks. Note that these speedup numbers are not directly comparable to those in Figure 4, because they are normalized to different sequential execution times. The figure shows that the performance gap between *NUCA-Dist* and *Dir-Coh* remains mostly the same as for systems without the second-level cache (the gap range is now 1%-32% and the average gap is 15%), demonstrating that our scheme also works with the addition of a second level cache.

6 RELATED WORK

The work in [2] extended the original uniprocessor NUCA proposal of [16] for CMPs. Unlike our work, that work focused on a large shared L2 and assumed that L1 coherence is maintained through directories.

Closer to our work, [7, 9, 35] considered the tradeoffs in organizing the L2 caches in a tiled CMP where L2 is physically distributed along with each tile. Similarly to ours, those works considered the option of organizing these distributed L2 caches as a logically single L2 cache. They differ from ours in the following ways: firstly, the L1 caches are private to each tile and allow replication of data, such that coherence is always required; secondly, those works propose techniques that allow replication of data in the L2 caches that is at the line level and is controlled by the hardware. Our work emphasizes simplicity and only allows a very restricted degree of replication that is totally controlled by the OS and, thus, forgoes hardware coherence mechanisms.

Our work is also similar in spirit to attempts to migrate most of the cache coherence management to software [8, 21]. Like those systems, our proposal benefits from the possibility to modify, and fix, the protocol with software modification and without any hardware changes. Those systems, however, run a full-blown coherence protocol in a dedicated protocol processor or a dedicated processor context. Even closer to ours are recent works that attempt to transfer some of the coherence burden to the OS/software [33, 34]. Unlike such previous trap-based schemes, however, the small hardware extensions that we propose minimize the need for OS and trap handler activity. In our proposed scheme, only the processor's first load or store to data in a page requires trap handler intervention and only the system's first load or store to data in a page requires full OS intervention. Another important difference is that all those schemes focused on coherence mechanisms for multi-chip systems.

There have been several proposals for tiled CMP architectures [4, 6, 19, 28, 29]. Most of these have focused on novel execution paradigms to exploit ILP and DLP in single-threaded applications. In the few studies with parallel applications, it is assumed that there is some hardware mechanism for cache coherence, but no details are given. Closer to our architecture, [6] does not provide hardware cache coherence, but, unlike ours, relies on the programmer/compiler to maintain coherence.

Our work is related to previous work on OS directed page migration and replication in CC-NUMA environments, such as [31]. Those differ from ours in that the hardware cache coherence mechanism of CC-NUMA machines supports fine-grain caching of memory lines, so that the page-level migration and replication is only necessary when the workloads overflow the private caches.

While most past shared-memory systems offered cache coherence in hardware, the Cray T3D and T3E are notable exceptions [27]. Unlike our proposed system, those machines did not support remote cache accesses and did not offer OS control of caching. Thus, avoiding incorrect local caching of shared data was left to the responsibility of the programmer/compiler. Hardware supported remote memory accesses were also proposed in the M-Machine [10]. However, that system also allowed indiscriminate private caching of data and no details are given on how coherence would be maintained, and it leaves the decision of caching versus remote accesses to the programmer/compiler.

Finally, our work is also related to previous work on software DSM systems, such as [5, 15, 18, 23]. Similarly to our proposal, those also tried to avoid the costs of hardware coherence by using the OS page mechanism to enforce coherence, but unlike ours, the majority of those systems supported full-blown coherence in software with full replication and multiple readers and writers. Our proposal, on the other hand, allows only a single writer at a time and relies on the relatively short communication delays on chip to perform efficient remote cache accesses. While [23] enforced a single-writer policy, it allowed ownership to move across nodes instead of enforcing remote accesses, which can lead to significant traffic. In addition, while [18] supported remote writes, it did not support remote reads, which had to be implemented by a tortuous mechanism by which the remote node performs remote writes on request. Those works also differ from ours in that they were tailored to multi-computer systems, where no hardware-supported single address exists.

7 CONCLUSION

In this paper, we proposed and evaluated a novel cost-effective software/hardware mechanism to support shared-memory parallel applications that forgoes hardware maintained cache coherence. The proposed mechanism treats all caches in the tiled CMP as a single logical cache and is based on the key idea that mapping of lines to physical caches is done at the page level with OS support. We extend a tiled CMP architecture with this mechanism and evaluate it on the Splash-2 and ALPBench benchmarks against an SGI-Origin-like cache coherent system. We propose two simple mechanisms to perform migration of pages and sharing of read-only data. These mechanisms bring the performance of the proposed system within 16% on average for 16 and 32 processors of the directory coherent system across all benchmarks. This is an impressive result considering that the directory coherence system uses a very aggressive hardware implementation and that our architecture requires only simple hardware support.

REFERENCES

[1] D. Abts, S. Scott, and D. J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proceedings of IPDPS 17*, Apr. 2003.

[2] B. M. Beckmann and D. A. Wood. Managing Wire Delay in Large Chip-Multiprocessor Caches. In *Proceedings of MICRO 37*, pages 319–330, Dec. 2004.

[3] D. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors: The SimpleScalar Tool Set. Tech. Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.

[4] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE Architectures. *Computer*, 37(7):44–55, July 2004.

[5] J. B. Carter, J. K. Bennett, and W. Zwaenepoel. Implementation and Performance of Munin. In *Proceedings of SOSP 13*, pages 152–164, Oct. 1991.

[6] C. Caşcaval, J. G. Castaños, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H. S. Warren, Jr. Evaluation of a Multithreaded Architecture for Cellular Computing. In *Proceedings of HPCA 8*, pages 311–322, Feb. 2002.

[7] J. Chang and G. S. Sohi. Cooperative Caching for Chip Multiprocessors. In *Proceedings of ISCA 33*, pages 264–276, June 2006.

[8] M. Chaudhuri and M. Heinrich. SMTp: An Architecture for Next-generation Scalable Multi-threading. In *Proceedings of ISCA 31*, pages 124–137, June 2004.

[9] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *Proceedings of ISCA 32*, pages 357–368, June 2005.

[10] M. Fillo, S. W. Keckler, W. J. Dally, N. P. Carter, A. Chang, Y. Gurevich, and W. S. Lee. The M-Machine Multicomputer. *International Journal of Parallel Programming*, 25(3):183–212, June 1997.

[11] E. Hagersten. Personal Communication regarding the verification of the coherence protocol of Sun Microsystems' Enterprise Servers E3000, E4000, E5000 and E6000. July 2007.

[12] L. Iftode, J. P. Singh, and K. Li. Understanding Applications Performance on Shared Virtual Memory Systems. In *Proceedings of ISCA 23*, pages 122–133, May 1996.

[13] Intel. *Intel Core2 Extreme Processor X6800 and Intel Core2 Duo Desktop Processor E6000 and E4000 Sequence Specification Update*, July 2007. Document No: 313279-016.

[14] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: A Dual-Core Multithreaded Processor. *IEEE Micro*, 24(2):40–47, March-April 2004.

[15] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *USENIX Winter 1994 Technical Conference Proceedings*, pages 115–131, Jan. 1994.

[16] C. Kim, D. Burger, and S. W. Keckler. An Adaptive, Non-Uniform Cache Structure for Wire-Delay Dominated On-Chip Caches. In *Proceedings of ASPLOS 10*, pages 211–222, Oct. 2002.

[17] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way Multithreaded Sparc Processor. *IEEE Micro*, 25(2):21–29, March-April 2005.

[18] L. I. Kontothanassis, G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, Jr., S. Dwarkadas, and M. L. Scott. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. In *Proceedings of ISCA 24*, pages 157–169, June 1997.

[19] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanović. The Vector-Thread Architecture. In *Proceedings of ISCA 31*, pages 52–64, June 2004.

[20] R. Kumar, V. Zyuban, and D. M. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. In *Proceedings of ISCA 32*, pages 408–419, June 2005.

[21] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of ISCA 21*, pages 325–337, Apr. 1994.

[22] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of ISCA 24*, pages 241–251, June 1997.

[23] K. Li. IVY: A Shared Virtual Memory System for Parallel Computing. In *Proceedings of ICPP 1988*, volume 2, pages 94–101. Pennsylvania State University Press, Aug. 1988.

[24] M. Li, R. Sasanka, S. V. Adve, Y.-K. Chen, and E. Debes. The ALPBench Benchmark Suite for Complex Multimedia Applications. In *Proceedings of IISWC 2005*, pages 34–45, Oct. 2005.

[25] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of ISCA 30*, pages 182–193, June 2003.

[26] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2):10–20, March-April 2005.

[27] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of ASPLOS 7*, pages 26–36, Oct. 1996.

[28] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of MICRO 36*, pages 291–203, Dec. 2003.

[29] M. B. Taylor, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, A. Agarwal, and S. Amarasinghe. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proceedings of ISCA 31*, pages 2–13, June 2004.

[30] M. Vachharajani, N. Vachharajani, and D. I. August. The Liberty Structural Specification Language: A High-Level Modeling Language for Component Reuse. In *Proceedings of PLDI 2004*, pages 195–206, June 2004.

[31] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *Proceedings of ASPLOS 7*, pages 279–289, Oct. 1996.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of ISCA 22*, pages 24–36, June 1995.

[33] H. Zeffer and E. Hagersten. A Case For Low-Complexity MP Architectures. In *Proceedings of the Conference on Supercomputing*, Nov. 2007.

[34] H. Zeffer, Z. Radović, M. Karlsson, and E. Hagersten. TMA: A Trap-Based Memory Architecture. In *Proceedings of ICS 20*, pages 259–268, June 2006.

[35] M. Zhang and K. Asanović. Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proceedings of ISCA 32*, pages 336–345, June 2005.