Victim Replication: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors

Michael Zhang and Krste Asanović

MIT Computer Science and Artificial Intelligence Laboratory
The Stata Center, 32 Vassar Street, Cambridge, Massachusetts
{*rzhang, krste*}*@csail.mit.edu*

Abstract

In this paper, we consider tiled chip multiprocessors (CMP) where each tile contains a slice of the total on-chip L2 cache storage and tiles are connected by an on-chip network. The L2 slices can be managed using two basic schemes: 1) each slice is treated as a private L2 cache for the tile 2) all slices are treated as a single large L2 cache shared by all tiles. Private L2 caches provide the lowest hit latency but reduce the total effective cache capacity, as each tile creates local copies of any line it touches. A shared L2 cache increases the effective cache capacity for shared data, but incurs long hit latencies when L2 data is on a remote tile.

We present a new cache management policy, victim replication, which combines the advantages of private and shared schemes. Victim replication is a variant of the shared scheme which attempts to keep copies of local primary cache victims within the local L2 cache slice. Hits to these replicated copies reduce the effective latency of the shared L2 cache, while retaining the benefits of a higher effective capacity for shared data. We evaluate the various schemes using full-system simulation of both single-threaded and multi-threaded benchmarks running on an 8-processor tiled CMP. We show that victim replication reduces the average memory access latency of the shared L2 cache by an average of 16% for multi-threaded benchmarks and 24% for single-threaded benchmarks, providing better overall performance than either private or shared schemes.

1. Introduction

Chip multiprocessors (CMPs) exploit increasing transistor counts by placing multiple processors on a single die [21]. CMPs are attractive for applications with significant thread-level parallelism where they can provide higher throughput and consume less energy per operation than a wider-issue uniprocessor. Existing CMP designs [4, 17, 18, 25] adopt a "dancehall" configuration, where processors with private primary caches are on one side of an interconnect crossbar and a shared L2 cache is on the other. In future CMPs, both the processor count and the L2 cache size are likely to increase, with the wire delay reaching tens of clock cycles for cross-chip communication latencies [1, 12]. Most current L2 caches have a simple fixed-latency pipeline to access all L2 cache slices. Using the worst-case latency, however, will result in unacceptable hit times for the larger caches expected in future processors.

We believe future CMP designs will naturally evolve toward arrays of replicated tiles connected over a switched network. Tiled CMPs scale well to larger processor counts and can easily support families of products with varying numbers of tiles. In this paper, we study a class of tiled CMPs where each tile contains a processor with primary caches, a slice of the L2 cache, and a connection to the on-chip network, as shown in Figure 1. This structure resembles a shrunken version of a conventional mesh-connected multi-chip multiprocessor.

There are two basic schemes to manage the on-chip L2 storage in these tiled CMPs: 1) each L2 slice is treated as a *private* L2 cache for the local processor, or 2) the distributed L2 slices are aggregated to form a single high-capacity *shared* L2 cache for all tiles. We refer to these two schemes as L2P and L2S, respectively.

The L2P scheme has low L2 hit latency, providing good performance when the working set fits in the lo-

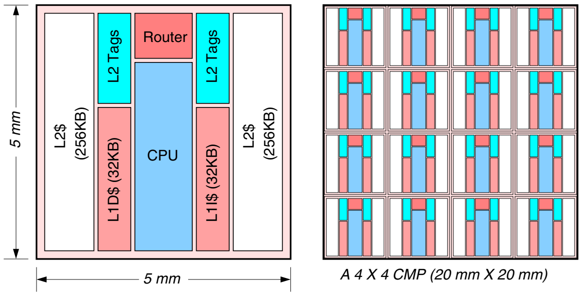

Figure 1. A tiled CMP design in a 70nm process.

Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05)
1063-6897/05 $20.00 © 2005 **IEEE**

cal L2 slice. But each tile must keep a local copy of any data it touches, reducing the total effective on-chip cache capacity for shared data. Also, the fixed partitioning of resources does not allow a thread with a larger working set to "borrow" L2 cache capacity from other tiles hosting threads with smaller working sets.

The L2S scheme reduces the off-chip miss rate for large shared working sets, but the network latency between a processor and an L2 cache slice varies widely depending on their relative placement on the die and on network congestion. As we will show in the results, the higher L2 hit latency can sometimes outweigh the capacity advantage of the shared L2 cache, and cross-chip latencies will continue to grow in future technologies.

L2S is an example of a non-uniform cache access (NUCA) design [16]. There have been several recent proposals [5, 6, 16] to reduce access latency for NUCA caches. Both D-NUCA [16] and NuRAPID [6] attempt to reduce average latency for a uniprocessor NUCA cache by migrating frequently-accessed data to cache slices closer to the processor. However, these two protocols could yield poor performance when applied to CMPs, where a given L2 cache line may be repeatedly accessed by multiple cores located at opposite corners of a chip. A recent study by Beckmann and Wood [5] investigates the behavior of block migration in CMPs using a variant of D-NUCA, but the proposed protocol is complex and relies on a "smart search" algorithm for which no practical implementation is given. The benefits are also limited by the tendency for shared data to migrate to the center of the die.

In this paper, we introduce *victim replication*, a hybrid cache management policy which combines the advantages of both private and shared L2 schemes. Victim replication (L2VR) is based on the shared L2 scheme, but reduces hit latency over L2S by allowing multiple copies of a cache line to co-exist in different L2 slices of the shared L2 cache. Each copy of an L2 cache line residing on a tile other than its home tile is a *replica*. In effect, replicas form dynamically-created private L2 caches to reduce hit latency for the local processor. We show that victim replication provides better overall performance than either private or shared cache designs, while using a much simpler protocol than previous NUCA designs.

2. Tiled CMP Designs

In this section, we describe the four L2 cache configurations we considered for tiled CMPs. All CMPs are based on a unit tile replicated in a 2-D mesh configuration as shown in Figure 1. Each tile contains a processor, primary instruction and data caches, an L2 cache slice with any associated directory, and a network switch. The tile in Figure 1 is approximately drawn to scale based on the floorplan of the Motorola MPC7447A [7] scaled to 70 nm technology, with the Altivec units removed and the L2 cache doubled to 1 MB capacity. Additional assumptions are as follows:

1. The primary instruction and data caches are not the focus of this paper, and are kept small and private to give the lowest access latency.
2. The local L2 cache slice is tightly coupled to the rest of the tile and is accessed with a fixed latency pipeline. The tag, status, and directory bits are kept separate from the data arrays and close to the processor and router for quick tag resolution.
3. Accesses to L2 cache slices on other tiles travel over the on-chip network and experience varying access latencies depending on inter-tile distance and network congestion.
4. A generic four-state MESI protocol with reply-forwarding is used as the baseline protocol for on-chip data coherence. Each CMP design uses minor variants of this protocol.

2.1. Private L2 Scheme

The L2P scheme, shown in Figure 2(a), uses the local L2 slice as a private L2 cache for the tile. This is equivalent to simply shrinking a traditional multi-chip multiprocessor onto a single chip. We employ a high-bandwidth on-chip directory scheme to keep the multiple L2 caches coherent, with the directory held as a duplicate set of L2 tags distributed across tiles by set index [4]. Cache-to-cache transfers are used to reduce off-chip requests for local L2 misses, but these operations require three-way communication between the requestor tile, the directory tile, and the owner tile. This operation is significantly more costly than remote hits in the L2S case, where a cache-to-cache transfer happens only if the line is held exclusive.

2.2. Shared L2 Scheme

The L2S scheme manages the L2 slices as a single shared L2 cache with addresses interleaved across tiles, as shown in Figure 2(b). L2S is similar to existing CMP designs, where several processor cores share a banked L2 cache [4, 18, 25]. Processor to L2 latency varies according to the number of network hops to the L2 slice and network congestion. We maintain coherence among all primary caches by adding additional directory bits to each L2 line to track which tiles have copies in their primary caches. Requests are satisfied by primary cache-to-cache transfers using reply-forwarding when appropriate.

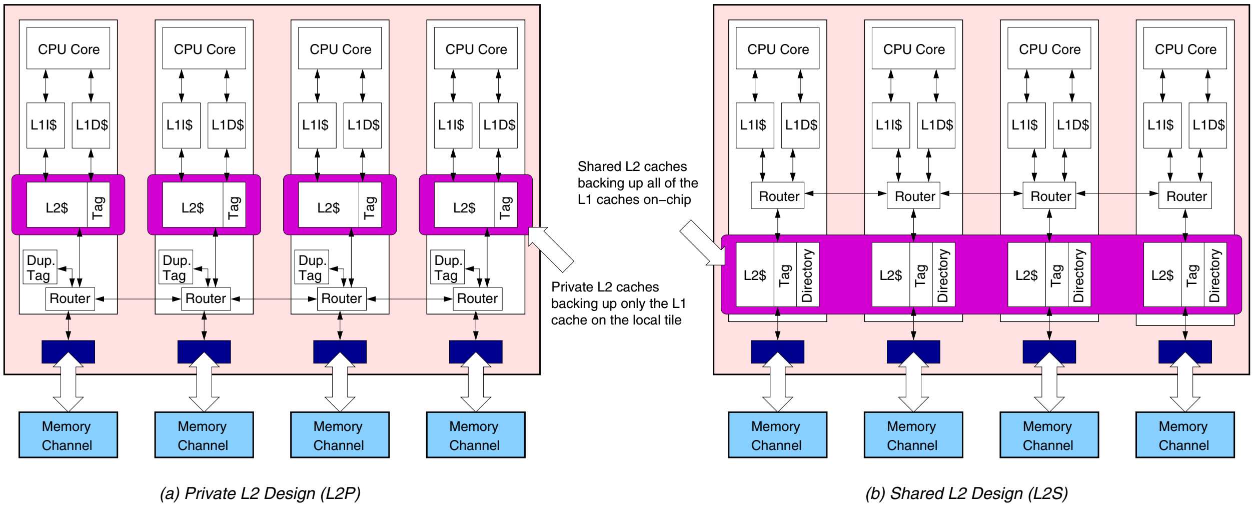
CPU Core
L1I$
L1D$
L2$
Tag
Dup. Tag
Router
Memory Channel
Private L2 caches backing up only the L1 cache on the local tile
Shared L2 caches backing up all of the L1 caches on−chip
Directory
(a) Private L2 Design (L2P)
(b) Shared L2 Design (L2S)

Figure 2. The two baseline L2 designs. The private L2 design treats each L2 slice as a private cache. The shared L2 design treats all L2 slices as part of a global shared cache.

2.3. Victim Replication

Victim replication (L2VR) is a simple hybrid scheme that tries to combine the large capacity of L2S with the low hit latency of L2P. L2VR is based on L2S, but in addition L2VR tries to capture evictions from the local primary cache in the local L2 slice. Each retained victim is a local L2 replica of a line that already exists in the L2 of the remote home tile.

When a processor misses in the shared L2 cache, a line is brought in from memory and placed in the on-chip L2 at a home tile determined by a subset of the physical address bits, as in L2S. The requested line is also directly forwarded to the primary cache of the requesting processor. If the line's residency in the primary cache is terminated because of an incoming invalidation or writeback request, we simply follow the usual L2S protocol. If a primary cache line is evicted because of a conflict or capacity miss, we *attempt* to keep a copy of the victim line in the local slice to reduce subsequent access latency to the same line.

We could create a replica for all primary cache victims, but L2 slice capacity is shared between victim replicas and global L2 lines (each set can contain any combination of replicas and global lines). We never evict a global line with remote sharers in favor of a local replica, as an actively cached global line is likely to be in use. The L2VR replication policy will replace the following classes of cache lines in the target set in descending priority order: (1) An invalid line; (2) A global line with no sharers; (3) An existing replica. If no lines belong to any of these three categories, no replica is made and the victim is evicted from the tile as in L2S. If there is more than one line in the selected category, L2VR picks at random. Finally, L2VR never replicates a victim whose home tile happens to be local.

All primary cache misses must now first check the local L2 tags in case there's a valid local replica. On a replica miss, the request is forwarded to the home tile. On a replica hit, the replica is invalidated in the local L2 slice and moved into the primary cache. When a downgrade or invalidation request is received from the home tile, the L2 tags must also be checked in addition to the primary cache tags.

L2VR has a small area overhead over L2S, because the the L2 tags must be wide enough to hold physical addresses from any home tile, thus the tag width becomes the same as L2P. Global L2 lines redundantly set these bits to the address index of the home tile. Replicas of remote lines can be distinguished from regular L2 lines as their additional tag bits do not match the local tile index.

2.4. Small Victim Cache

In effect, L2VR dynamically builds a private victim cache in the local L2 slice. For comparison, we also consider adding a conventional private victim cache [15] to the primary caches of the L2S scheme (L2VC). The size of the victim cache is chosen to approximately match the area increase for L2VR, which needs additional tag bits for replicas in the L2 caches. We optimistically assume that the L1 victim cache access can be completed in one cycle after a primary cache miss.

3. Experimental Methodology

In this section, we describe the simulation techniques we used to evaluate the alternative caching schemes. To present a clearer picture of memory system behavior, we use a simple in-order processor model and focus on the average raw memory latency seen by each memory request. Clearly, overall system performance can only be determined by co-simulation with a detailed processor model, though we expect the overall performance trend to follow average memory access latency. Prefetching, decoupling, non-blocking caches, and out-of-order execution are well-known microarchitectural techniques which overlap memory latencies to reduce their impact on performance. However, machines using these techniques complete instructions faster, and are therefore relatively more sensitive to any latencies that cannot be hidden. Also, these techniques are complementary to the victim replication scheme, and cannot provide the same benefit of reducing cross-chip traffic.

3.1. Simulator Setup and Parameters

We have implemented a full-system execution-driven simulator based on the Bochs [19] system emulator. We added a cycle-accurate cache and memory simulator with detailed models of the primary caches, the L2 caches, the 2D mesh network, and the DRAM. Both instruction and data memory reference streams are extracted from Bochs and fed into the detailed memory simulator at run time. The combined limitations of Bochs and our Linux port restricts our simulations to 8 processors. Results are obtained by running Linux 2.4.24 compiled for an x86 processor on an 8-way tiled CMP arranged in a 4×2 grid.

To simplify result reporting, all latencies are scaled to the access time of the primary cache, which takes a single clock cycle. The 1 MB local L2 cache slice has a 6 cycle hit latency. We assume a 70 nm technology based on BPTM [9], and model each hop in the network as taking 3 cycles, including the router latency and an optimally-buffered 5 mm inter-tile copper wire on a high metal layer. DRAM accesses have a minimum of 256 cycles of latency. Table 1 lists the important system parameters used in the experiments.

A 24 FO4 processor clock cycle is assumed in this paper, representing a modern power-performance balanced pipeline design [11, 23]. High-frequency designs might target a cycle time of 8–12 FO4 delays [13, 22], in which case cycle latencies should be doubled or tripled. Note that the worst case contention-free L2 hit latency is 30 cycles, hinting that even a small reduction in cross-chip accesses could lead to significant performance gains.

The L2 set-associativity (16-way) was chosen to be larger than the number of tiles to reduce cache conflicts between threads. For L2 associativities of 8 or less, we found several benchmarks had significant inter-thread conflicts, reflected by high off-chip miss rates. Inter-thread interference is a concern for any large scalable outer-level shared cache that maintains inclusion with inner-level private caches.

Table 1. Simulation parameters. All latencies are measured in 24 FO4-delay cycles.

Component	*Parameter*
Processor Model	in-order
Cache Line Size	64 B
L1 I-Cache Size/Associativity	16 KB/16-way
L1 D-Cache Size/Associativity	16 KB/16-way
L1 Load-to-Use Latency	1 cycle
L1 Replacement Policy	Psuedo-LRU
L2 Cache Size/Associativity	1 MB/16-way
L2 Load-to-Use Latency (per slice)	6 cycles
L2 Replacement Policy	Random
L1 Victim Cache Size/Associativity	8 KB/16-way
L1 Victim Cache Load-to-Use Latency	1 cycle
Network Configuration	4×2 Mesh
One-hop latency	3 cycles
Worst case L2 hit latency (contention-free)	30 cycles
External memory latency	256 cycles

3.2. Benchmark Suite

Table 2 summarizes the mix of single-threaded and multi-threaded benchmarks used to evaluate the designs. All 12 SpecINT2000 benchmarks are used as single-threaded workloads. They are compiled with the Intel C compiler (version 8.0.055) using `-O3 -static -ipo -mp1 +FDO` and use the MinneSPEC large-reduced dataset as input.

The multi-threaded workloads include all 8 of the OpenMP NAS Parallel Benchmarks (NPB) mostly written in FORTRAN, Classes S and W are standard input sizes, and class R is custom-sized to have manageable runtimes that fall between the S and W classes. In addition to NPB, we use two OS-intensive server benchmarks, `apache` and `dbench`, written in C. We also use one AI benchmark, `checkers`, written in Cilk [10], which uses a dynamic work-stealing thread scheduler. All of the multi-threaded benchmarks are compiled with `ifort-v8 -g -O2 -openmp` unless otherwise noted in Table 2.

All benchmarks were invoked in a runlevel without superfluous processes/daemons to prevent non-essential processes from interfering with the benchmark. Each simulation begins with the Linux boot sequence, but results are only gathered after the benchmark begins execution until completion.

Due to the long running nature of the benchmarks, we used a sampling technique to reduce simulation time.

Table 2. Benchmarks Description.

Single-Threaded Benchmarks		Multi-Threaded Benchmarks	
Benchmark (Instruction Count in Billions)	Description	Benchmark (Instruction Count in Billions)	Description
bzip (3.8)	`bzip2` compression algorithm	BT (1.7)	class S. block-tridiagonal CFD
crafty (1.2)	High-performance chess program	CG (5.0)	class W. conjugate gradient kernel
eon (2.9)	Probabilistic ray tracer	EP (6.8)	class W. embarassingly parallel kernel
gap (1.1)	A language used for computing in groups	FT (6.6)	class S. 3X 1D fast fourier transform (-O0)
gcc (6.4)	`gcc` compiler version 2.7.2.2	IS (5.5)	class W. integer sort. (icc-v8)
gzip (1.0)	Data compression using LZ77	LU (6.2)	class R. LU decomp. with SSOR CFD
mcf (1.7)	Single-depot vehicle scheduling algorithm	MG (5.1)	class W. multigrid kernel
parser (5.6)	Word processing parser	SP (6.7)	class R. scalar pentagonal CFD application
perlbmk (1.8)	Cut-down version of Perl v5.005_03	apache (3.3)	Apache's 'ab' worker threading model (gcc 2.96)
twolf (1.5)	The TimberWolfSC place/route package	dbench (3.3)	executes Samba-like syscalls (gcc 2.96)
vortex (1.5)	An object-oriented database program	checkers (2.9)	Cilk checkers (Cilk 5.3.2, gcc 2.96)
vpr (5.3)	A FPGA place/route package		

We extend the functional warming method for superscalars [26] to an SMP system, and fast-forward through periods of execution while maintaining cache and directory state [3]. At the start of each measurement sample, we run the detailed timing model to warm up the cache, memory and network pipelines. After this warming phase, we gather detailed statistics for one million instructions, before re-entering fast-forward mode. Detailed samples are taken at random intervals during execution and include 20% of all instructions executed, i.e., fast-forward intervals average around five million instructions. The number of samples taken for each benchmark ranges from around 150 to 1,000. Simulations show that the fastforwarding results match up with detailed runs to within 5% of error. To minimize the bias in the results introduced by system variability [2], we ran multiple runs of each benchmark with varying sample length and frequency. Results show that the variability is insignificant for our benchmark suite.

4. Simulation Results

In this section, we report on our simulation results. We first report on the multi-threaded benchmarks, and then the single-threaded benchmarks.

4.1. Multi-threaded Benchmark Performance

The primary results of our evaluation are shown in Figure 3, which shows the average memory access latency experienced by a memory access from a processor. The minimum possible access latency is one cycle, when all accesses hit in the primary cache.

The average memory access latency is composed of many complex components that differ between schemes. In the following, we provide a detailed analysis of the behavior of each benchmark using three metrics: the average memory access latency seen by the processors (Figure 3), the off-chip miss rate of the benchmarks (Figure 4), and a breakdown of the memory accesses (Figure 5).

There are different types of hit in the L2 cache across the different schemes. For L2P (first bar in Figure 5), hits in remote slices are from cache-to-cache transfers, which take longer than a remote L2 hit in L2S. For L2S (second bar in Figure 5), a hit in L2 can be in the local L2 slice or a remote L2 slice. For L2VR (third bar in Figure 5), the hits to local L2 cache also include hits to replicas.

As shown in Figure 3, the small victim cache (L2VC) improves access latency a little by catching some primary cache conflict misses. However, by reusing the far larger data capacity of the local L2 slice, L2VR provides a much greater reduction in average memory access latency. Thus, we omit L2VC in the following discussion.

We divide the analysis of the 11 benchmarks into three groups: 1) those with equal performance between L2P and L2S; 2) better performance with L2P; 3) better performance with L2S.

Equal Performance for L2P and L2S

Benchmark `IS` is the sole benchmark in this category. This benchmark has a working set which fits in the primary cache, with an average memory access latency of just over one cycle, and is unaffected by the L2 policy.

L2P Better Than L2S

Seven of the benchmarks perform better with L2P than L2S. Benchmarks `BT`, `FT`, `LU`, `SP`, and `apache` perform better with L2P because their working sets fit into the one megabyte private L2 slice. The lower latency of L2P dominates performance for these benchmarks. Benchmark `dbench` has a similar profile except it has a working set much larger than even the shared cache, as shown by the high off-chip miss rate for all three

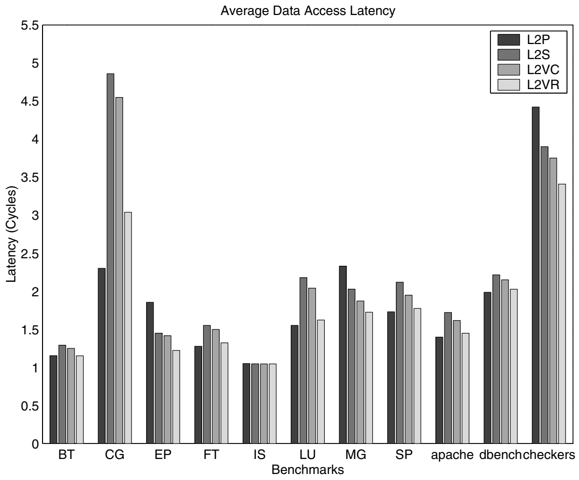

Figure 3. Access latencies of multi-threaded programs.

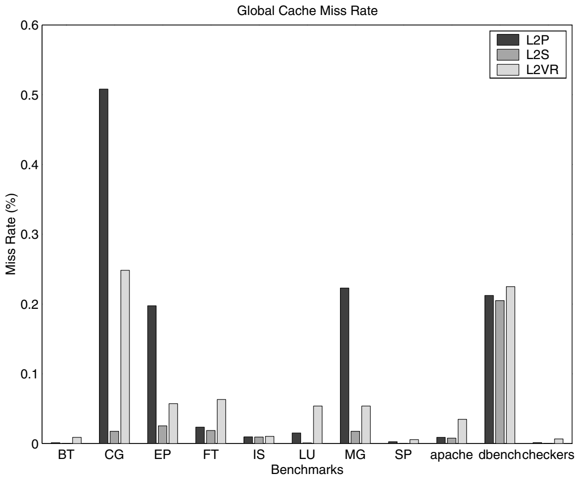

Figure 4. Off-chip miss rates of multi-threaded programs.

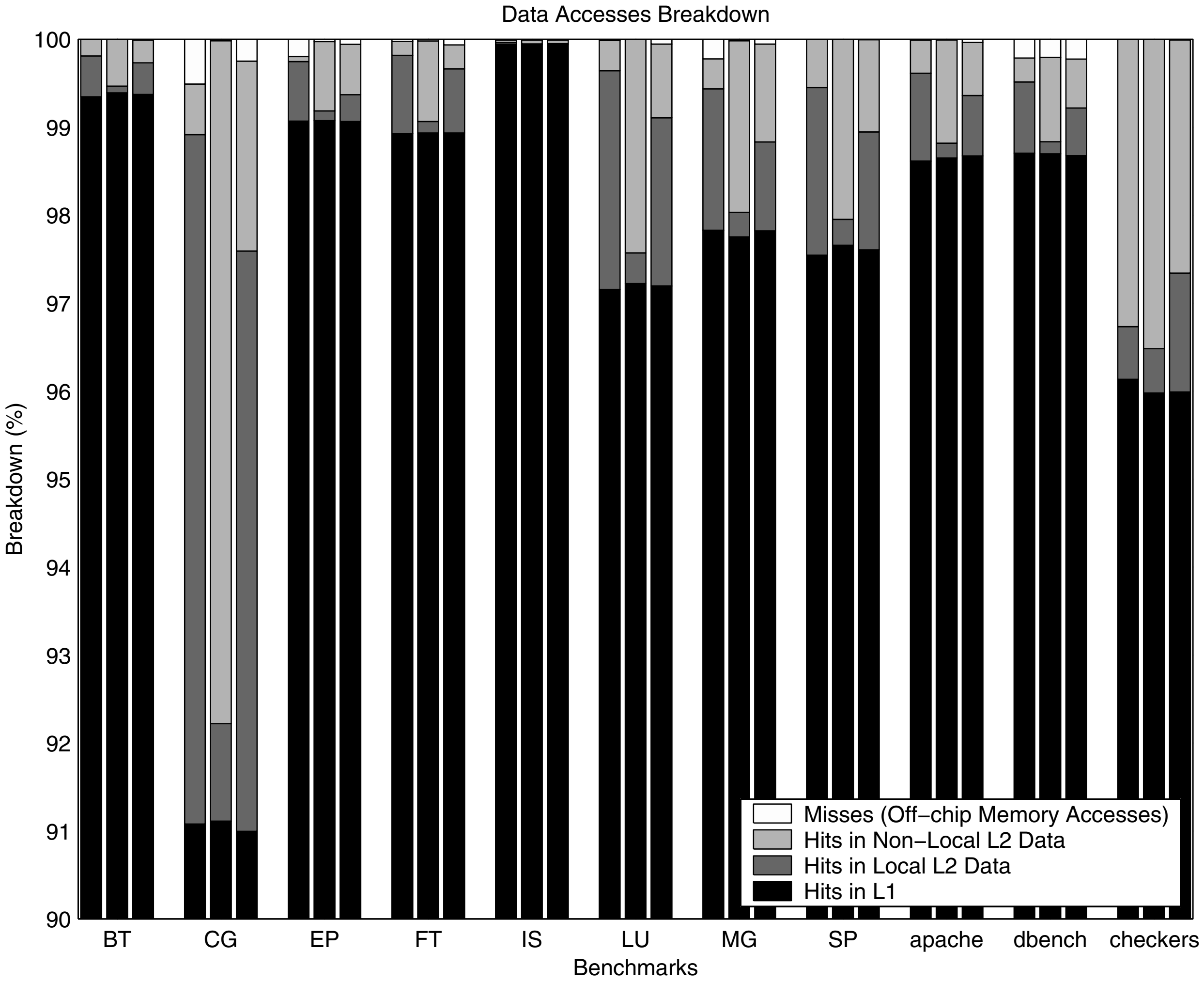

Figure 5. Memory access breakdown of multi-threaded programs. Moving from left to right, the three bars for each benchmark are for the L2P, L2S, and L2VR schemes, respectively. For L2P, hits to non-local data are from cache-to-cache transfers. For L2S, hits to shared data are in either the local tile or a non-local tile. For L2VR, hits to local data also include replica hits.

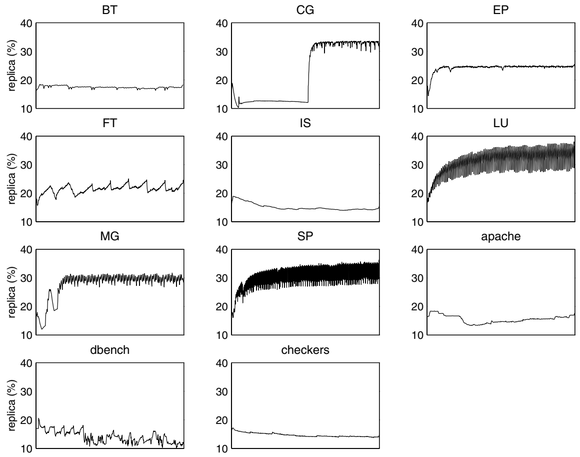

Figure 6. Time-varying graph showing percentage of L2 allocated to replicas in multi-threaded programs. Average of all caches is shown.

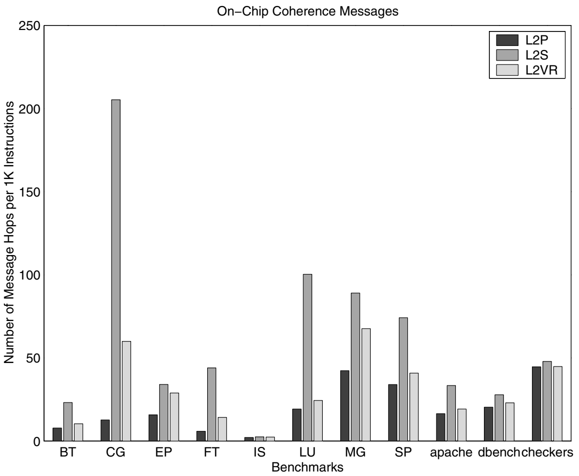

Figure 7. Network traffic per 1K instructions for multi-threaded programs.

schemes. Thus, L2S holds no effective capacity advantage over L2P.

Benchmark CG performs better with L2P as it has a working set that does not fit in the primary cache (9% primary cache miss rate) but does mostly fit into the local L2 cache slice. Even though CG has a significantly higher off-chip miss rate for L2P than L2S, the latency benefit of L2P, magnified by the large number of L2 accesses, makes L2P the winner by a wide margin.

For all but CG, L2VR is able to create enough replicas so that its performance is usually within 5% of L2P. L2VR also out-performs L2S significantly. Benchmark CG is the only case where L2P notably outperforms both L2S and L2VR.

L2S Better Than L2P

Three of the benchmarks perform better with L2S. Benchmarks MG and EP work better with L2S because they experience significantly less off-chip cache misses than with L2P, (first two bars in Figure 4). This demonstrates the usefulness of the additional on-chip cache capacity over L2P. However, L2S does not outperform L2P by as much as one might expect, (first two bars in Figure 3), because L2S has a higher average L2 hit latency.

Benchmark checkers also performs better with L2S, but for a different reason. This benchmark has a small working set that fits in the local L2 slice, causing practically no off-chip misses for all three schemes. However, it uses a dynamic work-stealing scheduler that incurs a lot of thread migration during execution, shown by the large number of cache-to-cache transfers. With L2P, shared data is migrating between different tiles along with the threads. Most of L2's hits are to remote caches using expensive L2 cache-to-cache transfers.

For these three benchmarks, L2VR performs best overall. Although L2VR has slightly more off-chip cache accesses than L2S, they are offset by the reduction in inter-tile accesses from replica hits (second and third bar in Figure 5).

Multi-Threaded Performance Summary

Out of the eleven benchmarks, one is unaffected by L2 policy. Three benchmarks perform better with L2S than L2P, but perform best with L2VR. Six benchmarks perform better with L2P than L2S, but L2VR has similar performance to L2P. For only one benchmark, CG, does L2VR fall significantly behind L2P.

Table 3 shows the per-benchmark savings achieved by L2VR over L2S, on which it is based. L2VR reduces memory latency by around 16%, with 7 out of the 11 benchmarks reaching over 14%. While all benchmarks experiences a moderate increase in off-chip misses for L2VR over L2S, none is large enough to offset the benefit of the replicas.

4.2. On-chip Network Traffic Reduction

An additional benefit of L2VR is the reduction of on-chip coherence traffic. Figure 7 shows the number of coherence message per thousand instructions executed, weighed by the number of hops each message traversed. As expected, L2P has by far the lowest traffic for the all benchmarks except checkers, which has a larger

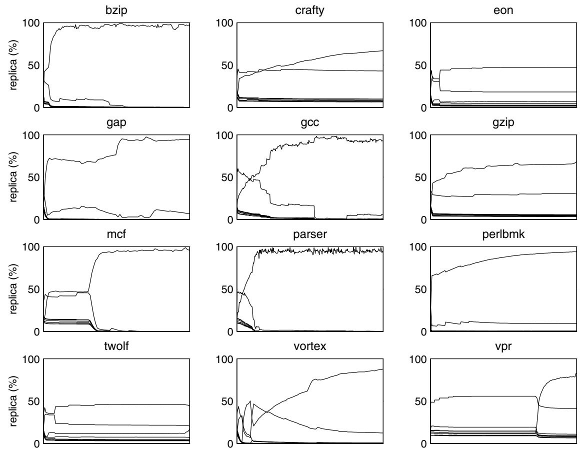

Figure 8. Time-varying graph showing percentage of L2 allocated to replicas in single-threaded programs. Individual caches are shown.

Table 3. Reduction in memory access latency for L2VR over L2S.

Single-Threaded Benchmarks		Multi-Threaded Benchmarks	
Benchmark	Reduction (%)	Benchmark	Reduction (%)
bzip	18.5	BT	10.9
crafty	27.9	CG	37.5
eon	2.0	EP	15.7
gap	13.5	FT	14.8
gcc	21.1	IS	0.0
gzip	46.4	LU	25.6
mcf	41.4	MG	15.0
parser	26.2	SP	16.3
perl	5.1	apache	15.8
twolf	36.1	dbench	8.6
vortex	15.3	checkers	12.7
vpr	30.1		
Average	23.7	*Average*	15.8

percentage of the L2 hits from cache-to-cache transfers. L2VR eliminates some inter-tile messages when accesses can be resolved in local replicas. The reductions range from 4% (`dbench`) to over 70% (`LU`) with 6 out of the 11 benchmarks reaching over 40%.

4.3. Adaptive Replication Policy

Figure 6 plots the percentage of total L2 cache capacity allocated to replicas for the eleven multi-threaded benchmarks in our benchmark suite against execution time. We plot the final result from each of the detailed sample points. This graph shows two important features of L2VR. First, it is an adaptive process that adjusts to the execution phases of the program. Execution

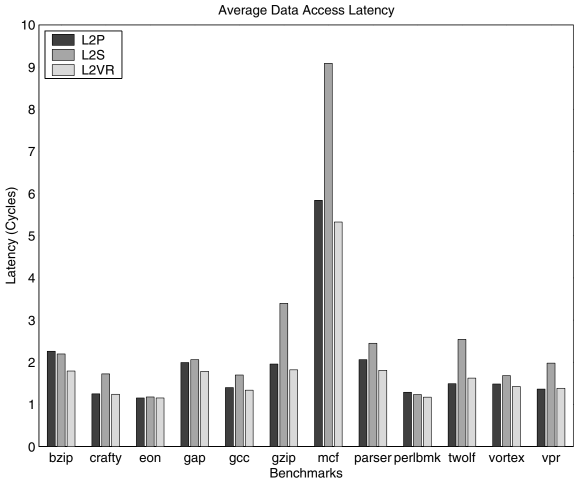

Figure 9. Access latency comparison of single-threaded programs.

phases can be clearly observed in `CG`, `FT`, and `dbench`. Second, the victim storage capacity offered by L2VR is much larger than a comparable dedicated victim cache, as modeled in L2VC. Five out of the eleven benchmarks reached over 30% replicas, equal to a 300 KB victim cache in our case.

It should be noted that for an N-tile CMP, the average percentage of victims over all caches cannot exceed $(N-1)/N$, and in our case, 87.5%. This is because at most $N-1$ replicas can exist for each global line. This limit does not hold for each individual cache.

4.4. Single-Threaded Benchmarks

Single-threaded code is an important special case for a tiled CMP. The L2VR policy dynamically adapts to a single thread by forming a three-level cache hierarchy: the primary cache, the local L2 slice full of replicas, and the remote L2 slices acting as an L3 cache.

This behavior is shown in Figure 8, which plots the time-varying graph of the percentage of replicas in each of the eight individual L2 caches. Because we are performing a full-system simulation, we sometimes observe the single thread moving between CPUs under the control of the operating system scheduler. For benchmarks `bzip`, `gap`, `parser`, and `perlbmk`, one tile held the thread most of the time. For benchmarks `crafty`, `eon`, `gzip`, `twolf`, and `vpr`, the thread seems to bounce between two tiles. We did not attempt to optimize the kernel scheduler to pin the thread on one tile throughout the run. Across the benchmarks, the L2 slice of the tile on which the thread is running holds a very high percentage of replicas, usually over 50%.

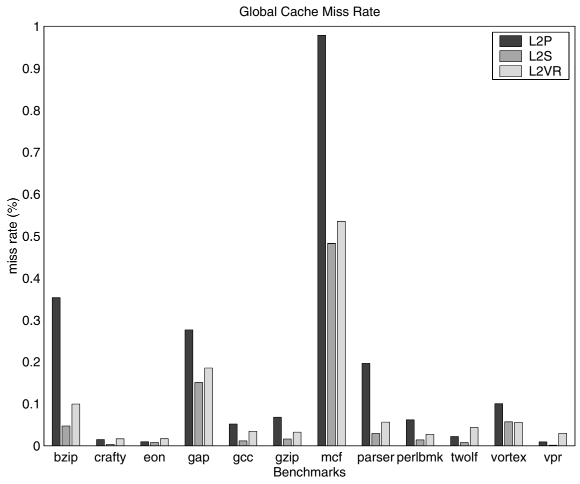

Figure 10. Off-chip miss rate of single-threaded programs.

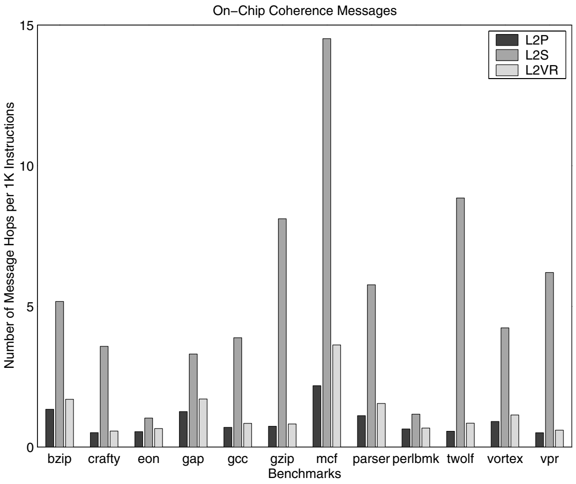

Figure 11. Network traffic per 1K instructions for single-threaded programs.

Figure 9 shows the memory access latencies for the single-threaded benchmarks, and Figure 10 shows the off-chip miss rates. In many cases, L2S performs significantly worse than the other schemes, because L2 hit latency is a critical component of performance for these codes. Table 3 summarizes the savings achieved by L2VR over L2S. Nine out of the twelve benchmarks achieved 15% or more savings, with six of them over 25%, and an average of 24%. L2VR is usually comparable or better than the L2P scheme, combining the advantages of fast accesses to the local L2 slice with a large on-chip backup cache.

Figure 11 shows the number of coherence message per thousand instructions executed, weighed by the number of hops each message traversed, for single-threaded benchmarks. On average, L2VR reduces the network traffic by 71% compared to L2S, and approaches the level of traffic generated by L2P.

5. Related Work

Earlier NUCA proposals [5, 6, 16] were briefly discussed in the introduction. The tiled CMP architecture we introduce here has a much simpler and regular structure yet provides similar benefits in avoiding worst-case access latency to a distributed L2 cache, with replication to further reduce latency of frequently accessed data. Huh et. al. discussed the design space for future CMPs in [14], but only evaluated private L2 caches. Oi and Ranganathan studied the effect of remote victim caching [20], though the the size of the remote victim cache is fixed and cannot dynamically adapt to individual benchmarks or the current phase of execution.

The cache and memory system of several academic and commercial architectures also share similarity with victim replication. The IBM Power4 architecture [25] has a private non-inclusive L3 cache on each node (a node may have 2-8 processors), where entries are allocated either for local L3 misses or local L2 victims. However, while L3s can be snooped by other nodes, local L2 victims always overwrite the local L3, thus in practice a shared working set larger than one L3 cannot remain resident. In contrast, L2VR does not overwrite shared global L2 lines to help maintain a large working set on-chip. In addition, victim replication avoids global snooping by allocating global L2 misses in a known home node.

The Pirahna [4] architecture uses a shared L2 cache among all processors. The L2 cache is non-inclusive of the primary caches and essentially caches all of the primary cache victims. However, Pirahna has a "dance-hall" structure and coherence among primary caches is kept by a snoopy bus protocol.

The MIT Raw architecture [24] is a tiled multiprocessor that resembles the CMP model presented in this paper. It sports 16 simple RISC-like cores with 2 MB total cache on-chip, distributed evenly among all tiles. However, there is no hardware mechanism to keep the caches coherent.

The L2VR scheme share some similarity with earlier work on remote data caching in conventional CC-NUMA and COMA architectures [8, 27], which also try to retain local copies of data that would otherwise require a remote access. There are two major differences in the CMP structure, however, that limit the applicability of prior remote caching work. First, in CC-NUMAs, all the local cache on a node is private so the allocation between local and remote data only affects the local node. In the L2VR scheme, on-chip L2 capacity is shared by all nodes, and so a local node's replacement policy affects cache performance of all nodes. Second, in both CC-NUMA and COMA systems, remote data is further away than local DRAM, thus it is beneficial to use a large remote cache held in local DRAM. In addition, the cost of adding a remote cache is low and does not diminish the performance of existing L2 caches. In the CMP structure, the remote caches are closer to the local node than any DRAM, and any replication reduces the effective cache capacity for lines that will have to be fetched from slow off-chip memory.

6. Conclusion

Tiled CMPs are an attractive evolution from existing "dancehall" CMP structures, providing a simpler replicated physical design. As CMP processor counts and cache sizes grow, and cross-chip relative wire delay increases, reducing data movement to off-chip memory and across chip will become the dominant architectural design challenge. In this paper, we have shown that while distributed shared L2 caches usually reduce off-chip traffic, they can do so at the cost of adding significant on-chip communication delays. The victim replication scheme proposed in this paper helps reduce on-chip communication delay by replicating cache lines within the same level of the cache, local to processors actively accessing the line. The result is a dynamically self-tuning hybrid between private and shared caches.

7. Acknowledgments

We thank the anonymous reviewers and members of the MIT SCALE group, especially Chris Batten and Ronny Krashinsky, for their comments. Special thanks to Ken Barr for the benchmarks used in the paper. This work is partly funded by the DARPA HPCS/IBM PERCS project and NSF CAREER Award CCR-0093354.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *ISCA-27*, May 2000.

[2] A. Alameldeen and D. Wood. Addressing workload variability in architectural simulations. In *HPCA-9*, 2003.

[3] K. Barr, H. Pan, M. Zhang, and K. Asanović. Accelerating multiprocessor simulation with a memory timestamp record. In *ISPASS-2005*, March 2005.

[4] L. Barroso et al. Piranha: a scalable architecture based on single-chip multiprocessing. In *ISCA-27*, May 2000.

[5] B. Beckmann and D. Wood. Managing wire delay in large chip-multiprocessor caches. In *MICRO-37*, 2004.

[6] Z. Chishti, M. Powell, and T. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *MICRO-36*, December 2003.

[7] C. Corley. PowerPC benchmarking update. In *Smart Networks Developer Forum*, April 2004.

[8] F. Dahlgren and J. Torrellas. Cache-only memory architectures. *IEEE Computer*, 32(6), 1999.

[9] Device Group at UC Berkeley. Predictive technology model. Technical report, UC Berkeley, 2001.

[10] Supercomputing Technologies Group. Cilk 5.3.2. http://supertech.lcs.mit.edu/cilk, June 2000.

[11] A. Hartstein and T. Puzak. Optimum power/performance pipeline depth. In *MICRO-36*, 2003.

[12] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of IEEE*, 89(4), April 2001.

[13] M. Hrishikesh et al. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *ISCA-29*, May 2002.

[14] J. Huh, D. Burger, and S. Keckler. Exploring the design space of future CMPs. In *PACT*, September 2001.

[15] N. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *ISCA-27*, June 1990.

[16] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *ASPLOS-X*, October 2002.

[17] K. Krewell. Intel's PC roadmap sees double. *Microprocessor Report*, 18(5):41–43, May 2004.

[18] K. Krewell. Sun's Niagara pours on the cores. *Microprocessor Report*, 18(9):11–13, September 2004.

[19] K. Lawton. Bochs. http://bochs.sourceforge.net.

[20] H. Oi and N. Ranganathan. Utilization of cache area in on-chip multiprocessor. In *HPC*, 1999.

[21] K. Olukotun et al. The case for a single-chip multiprocessor. In *ASPLOS-VII*, Cambridge, MA, October 1996.

[22] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *ISCA-29*, May 2002.

[23] V. Srinivasan et al. Optimizing pipelines for power and performance. In *MICRO-35*, November 2002.

[24] M. B. Taylor et al. Evaluation of the Raw microprocessor: An exposed-wire-delay architecture for ILP and Streams. In *ISCA-31*, June 2004.

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

[26] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA-30*, June 2003.

[27] Z. Zhang and J. Torrellas. Reducing remote conflict misses: NUMA with remote cache versus COMA. In *HPCA-3*, January 1997.