Victim Migration: Dynamically Adapting Between Private and Shared CMP Caches

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

Future CMPs will have more cores and greater on-chip cache capacity. The on-chip cache can either be divided into separate private L2 caches for each core, or treated as a large shared L2 cache. Private caches provide low hit latency but low capacity, while shared caches have higher hit latencies but greater capacity. Victim replication was previously introduced as a way of reducing the average hit latency of a shared cache by allowing a processor to make a replica of a primary cache victim in its local slice of the global L2 cache. Although victim replication performs well on multithreaded and single-threaded codes, it performs worse than the private scheme for multiprogrammed workloads where there is little sharing between the different programs running at the same time.

In this paper, we propose victim migration, which improves on victim replication by adding an additional set of migration tags on each node which are used to implement an exclusive cache policy for replicas. When a replica has been created on a remote node, it is not also cached on the home node, but only recorded in the migration tags. This frees up space on the home node to store shared global lines or replicas for the local processor.

We show that victim migration performs better than private, shared, and victim replication schemes across a range of single threaded, multithreaded, and multiprogrammed workloads, while using less area than a private cache design. Victim migration provides a reduction in average memory access latency of up to 10% over victim replication.

1. Introduction

Future chip-scale multiprocessors (CMPs) are likely to continue to increase both the number of cores and the total cache capacity on chip. Off-chip misses will remain expensive, but increases in clock frequency together with worsening relative wire delays will also increase latencies for cross-chip communication, reaching tens of clock cycles in future technologies [1, 12]. Effective use of on-chip cache must therefore consider both the cost of off-chip misses and the cost of cross-chip communications.

Two baseline outer-level cache designs, *private* and *shared*, illustrate the trade-offs between these two components of effective memory access latency. A private design dedicates a slice of the on-chip L2 cache storage as a private L2 cache for each processor core. The shared design aggregates all the L2 cache capacity to form a single L2 cache shared by all the cores.

The private design has low L2 hit latency, as the private L2 is physically co-located with the processor core and has much smaller area than a shared cache. This provides good performance provided the working set fits within the local L2 slice. The disadvantage of the private L2 scheme is that effective on-chip cache capacity is reduced for shared data, as each core must retain its own copy of any shared data block. Also, the fixed partitioning of resources does not allow a thread with a larger working set to "borrow" L2 cache capacity from the private caches of other processors hosting threads with smaller working sets.

The shared design reduces the off-chip miss rate for large shared working sets, as only a single on-chip copy is required for any shared data. However, large shared L2 caches have worse access latency than a small private L2 cache and can suffer from inter-thread cache conflicts.

Many studies have shown that either private or shared caches can considerably outperform the other, depending on the specific characteristics of the workload [27, 15, 8, 15]. This observation has motivated several proposals to develop hybrid architectures that retain the advantages of both private and shared designs.

A number of proposals seek to reduce the effective access latency of a large shared cache by adopting a non-uniform cache access (NUCA) architecture. Current shared L2 caches [4, 18, 25] are constructed using a "dancehall" configuration, as shown in the left of Figure 1, where processors with private L1 caches are on one side of an interconnect crossbar and a banked shared L2 cache is on the other. To reduce access latency and energy consumption, large caches are physically divided into many small banks or slices. Current designs use a fixed worst-case latency to access any slice, but as cross-chip latencies grow, this will result in unacceptable hit times. NUCA [17] designs allow access latency to vary depending on the relative placement of the processor and L2 slice containing the data. Dynamic NUCA schemes have been proposed for uniprocessors [17, 7], where frequently-accessed cache blocks gradually migrate closer to the processor. These schemes are considerably more complicated when applied to multiprocessors with dancehall configurations [5, 8, 15]. These schemes require some form of duplicated L2 tag kept local to each processor to reduce the number of slices that must be searched to locate an on-chip block. Further, all such local tags must be kept consistent with any block migration triggered by a remote processor, imposing additional serialization constraints on otherwise independent cache accesses [5, 8, 15].

An alternative base structure is a tiled CMP as shown on the right side of Figure 1, where the CMP is organized as an array of replicated tiles connected over a switched network. Each tile contains a processor with L1 caches, a slice of the L2 cache, and a connection to the on-chip network. Tiled CMPs scale well to larger processor counts and can easily support families of products with varying numbers of tiles, including the option of connecting multiple separately tested and speed-binned die within a single package. A tiled CMP is a natural structure for private L2 caches, but can also be used to implement a shared L2 cache [27]. Recent work has shown how a victim replication (VR) scheme can reduce the effective hit latency of a tiled shared L2 cache by placing a copy of an evicted L1 cache block in the local slice of the L2 instead of sending it back to the home tile [27]. If the processor requests the same block again, it can now access the local copy instead of the remote original. This approach provides many of the same benefits as a dynamic NUCA scheme, but with much less complexity.

This earlier study shows that victim replication has better overall performance than either private or shared schemes for multi-threaded and single-threaded workloads [27]. However, as we show in the evaluation section, the victim replication scheme does not perform as well as private caches when running a multiprogrammed workload, where each processor is running an independent program. The problem in this case is that private data is placed on chip twice, once at the home tile and once as a replica at the processor using the data. This reduces effective on-chip capacity and causes conflicts between independent threads.

In this paper, we introduce *victim migration (VM)*, which improves on victim replication by providing additional migration tags at the home tile. These additional tags are used to implement an exclusive cache policy for replicas, where the regular tag and data storage of the home tile's L2 slice is freed when a replica is present on a remote tile. The newly vacated home block can then be reused to hold a different shared block or a replica for the processor local to the home tile, while the migration tag points to the remote replica data. We show how the total area overhead of this scheme is less than for a private cache scheme, while providing the best performance over all workloads (single-threaded, multithreaded, and multi-programmed) as compared to private, shared, or victim replication schemes.

2. Baseline Tiled CMP Designs

In this section, we describe three baseline L2 cache configurations: private, shared, and victim replication. The tiled CMP model used is shown in Figure 1. Additional assumptions about the model are as follows: 1) The primary instruction and data caches 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 Design

In the private design (Figure 2), the processor uses the local L2 slice as a private L2 cache, as is the case for several commercial CMPs [6, 16]. This is equivalent to simply shrinking a traditional multi-chip multiprocessor onto a single chip. Snoopy protocols scale poorly to future technologies and large numbers of nodes, and so we employ a high-bandwidth on-chip directory to maintain coherence among the private L2 caches.

The directory is held as a duplicate set of L2 tags distributed across tiles by address [4]. For each processor accessing a particular cache block, a copy of the block must be resident in its private L2 cache, such as *block i* shown in Figure 2. In addition, an on-chip directory holding an entry for *block i* is stored at *blk i*'s home tile.

A 8-Node "Dance-Hall" CMP *A 4 x 4 Tiled CMP*

Figure 1. Dancehall versus tiled chip multiprocessor designs.

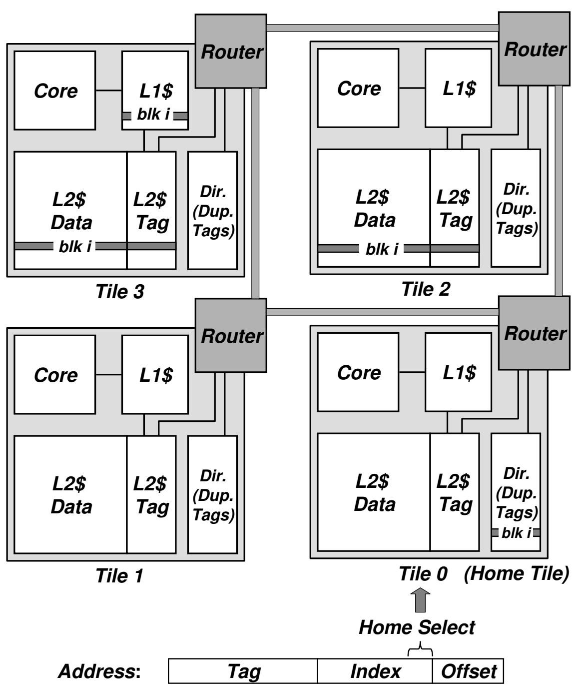

Figure 2. The private L2 design treats each L2 slice as a private cache.

Figure 3 shows a simplified two-tile example of how this scheme works. Each tile has a direct-mapped L2 cache with four cache blocks. To determine address `A`'s status on-chip, we use the lower bits of the index, the `home select` bits, to find `A`'s home tile. The remaining bits of the index are used to find the duplicate tag entry corresponding to `A` in the directory on the home tile. This entry stores the duplicates of all L2 tags in the cache set that `A` maps to from all the tiles. In this example, `A` maps to set 3, and the duplicated tag entry has the L2 tags of set 3 from both tile 0 and tile 1. With these tags, we can easily deduce the directory information of `A`. Therefore, we have a perfect directory for all of the cache blocks on chip. The main drawback of this approach is the area overhead, which we discuss later.

Cache-to-cache transfers are used to reduce off-chip requests for local L2 misses, but these operations require three-way communication between the requester tile, the directory tile, and the owner tile. This operation is more costly than hits to global locations in a shared design, where a three-way cache-to-cache transfer only occurs if the block is held exclusive.

2.2. Shared Scheme

In the shared design, all of the L2 slices are managed as a single shared L2 cache with addresses interleaved across tiles. The shared design is used by a number of existing CMP designs [4, 18, 25, 21], where several processor cores share a banked L2 cache. Figure 4 shows the implementation in detail. Each address is statically mapped to a home tile (identical to the private directory mapping scheme). On an L1 cache miss, the fetch request is processed by the L2 slice at the requested block's home tile. Latency to the L2 slice varies according to network congestion and the number of network hops between the requesting processor and the home tile. All the L1 caches are kept coherent using additional directory bits on each L2 block, which track which tiles have remote copies. For the 8-node implementation used in our evaluation, this adds 3 state bits and an 8-bit sharing vector to each L2 cache line. The overhead of the sharing vector will grow as the processor count grows, but a number of previously proposed techniques could be used to reduce directory overhead. Some requests are satisfied using cache-to-cache transfers between L1 caches using reply-forwarding.

2.3. Victim Replication

Victim replication (VR) [27] is a simple hybrid scheme that tries to combine the large capacity of the shared design with the low hit latency of the private design. Victim replication is based on the shared design, but in addition it tries to capture L1 evictions in the local L2 slice. Each retained victim is a local L2 replica of a block that already exists in the L2 cache of the remote home tile.

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

While a replica can be created for for all primary cache victims, VR never evicts a global block with remote sharers in favor of a local replica, as an actively shared global block is likely to be in use. VR also 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 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.

3. Victim Migration

Victim Migration (VM) is a flexible hybrid cache architecture that can dynamically adapt to mimic the behavior of pure shared or pure private designs. Figure 5 shows the cache hierarchy arrangement for VM. Each L2 consists of a tag array, a data array, and directory bits, similar to the shared design. In addition, each L2 also has an extra set of tags, we refer to them as the *VM tag array*. For now, we assume that the size and associativity of the the VM tag array is identical to the regular L2 tags.

Similar to previous designs, each cache block is statically mapped to a home tile in VM, which holds the block in one of two forms. First, it can be managed exactly like the shared design Second, if the block is being actively shared by another tile, either as an regular L1 cache block or an L2 replica, the L2 cache may only store the tag of the block in the VM tag array. By using the VM tag array, victim migration removes the unnecessary duplication of data at the home tile, freeing up space to hold more replicas or other global blocks. The only added complexity is that both the regular tags and the VM tag array must be searched during a data fetch. If a hit is found in the VM tags, the request is satisfied

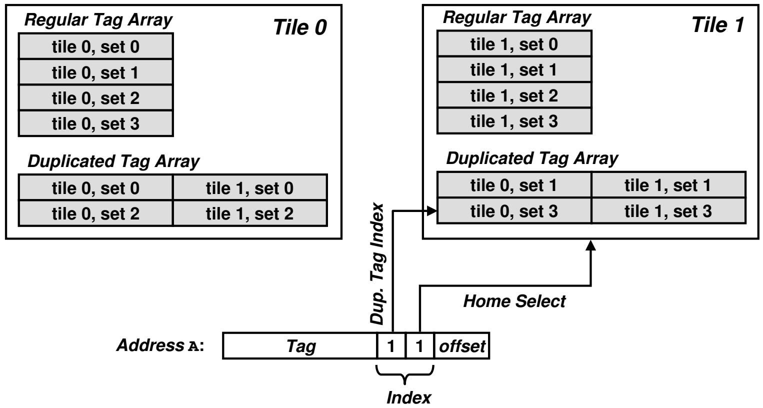

Figure 3. Example of using duplicated L2 cache tags to maintain L2 data coherence.

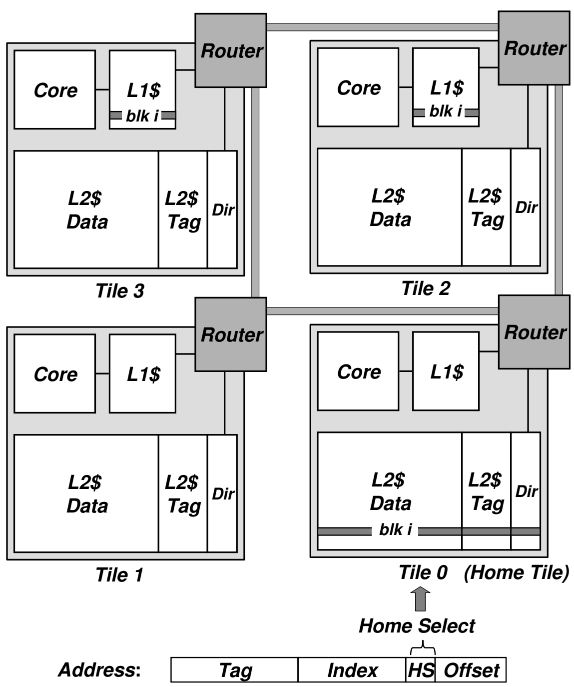

Figure 4. The shared L2 design treats all L2 slices as part of a global shared cache.

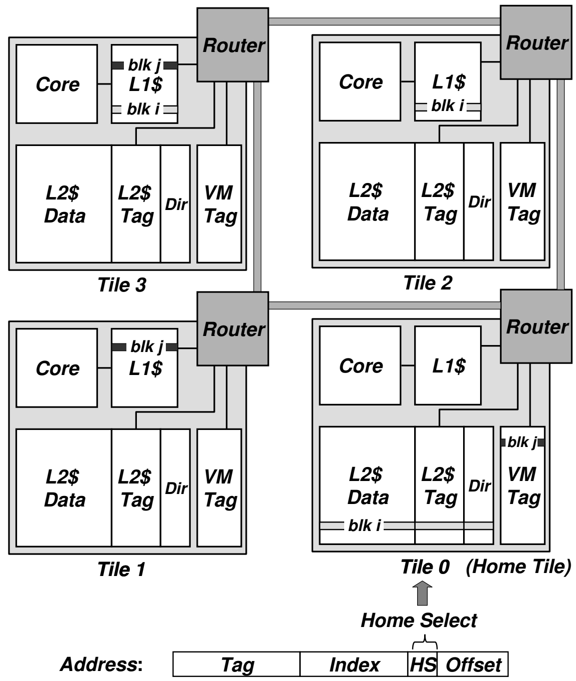

Figure 5. Illustration of Victim Migration.

through three-way cache-to-cache transfers using reply-forwarding.

3.1. Management Policies

In this section, we provide a set of heuristics to efficiently manage the cache capacity on-chip. Specifically, we discuss three policies. First, the *L2 refill policy* determines where to place a cache block when the L2 receives a memory reply for an L2 miss. Second, the *L1 eviction policy* determines whether to replicate (and if so, where to keep) an L1 victim in the local L2 slice. Third, if the local L2 slice decides not to replicate an L1 victim and sends it back to the home tile, the *remote tile writeback policy* determines where to place the data if it is held in the duplicated tags. We assume that all three policies first look for invalid blocks in the main data and tag arrays as these spaces can be used at no cost. The policies describe below will only be followed if there are no invalid blocks.

L2 Refill Policy

The L2 refill policy is rather simple. We first look for an invalid VM tag to hold the tag of the address only, and forward the data to the requesting tile. If all the VM tags are used, we randomly evict a block in the main data array and write back any dirty data to memory if needed.

L1 Eviction Policy

The L1 eviction policy determines whether to replicate an L1 victim, and if so, where to hold it in the local L2 slice. We first simultaneously search for an invalid VM tag and an actively shared block in the regular tags. If both exist, the tag of the actively shared block can be moved to the invalid VM tag entry without losing information. The L1 victim can safely overwrite the shared block's local data. As no data is evicted from the local L2 cache, this operation should not cause performance degradation. The only minor effect may come from the possibly longer hit latency required to perform a three-way cache-to-cache transfer when a remote request hits in the VM tags when the line was previously stored in the regular tags.

If the above scenario is not possible and we must evict a valid block, we look to replace the following two classes of block in descending priority order: 1) a global block with no L1 sharers; 2) an existing replica block. If neither of the two classes of block exist, we do not replicate the L1 victim. This approach is similar to the one used in [27].

Remote Tile Writeback Policy

This policy is used whenever a tile has to evict a line back to the home node, either from its primary cache when no replica can be created, or from the victim replicas when they are themselves evicted. If the line is already held in the regular tag and data array, we perform a conventional update. If the tag is held in VM tag array and another tile still has a copy of the data, we simply update the directory information in the VM tag. However, if the last on-chip copy of a cache block is sent home and its tag is kept in the VM tag array, we must decide if and where to keep this unique copy.

We first look for an actively shared global block, which currently does not need the data array space to store its data. This global block can be swapped with the remote writeback. If we can find such a swap, no data is evicted from the chip.

If this scenario is not possible, we use the approach outlined in the L1 eviction policy to look for unowned blocks, then replica blocks to replace. If a replica is replaced, there can be a ripple effect as the evicted replica is written back to its own home node.

If no unowned blocks or replicas are found, we again choose not to evict actively shared blocks as they are likely to be in the active working set. In this case, the remote tile writeback is evicted from the chip and written back to memory if necessary.

4. Experimental Methodology

In this section, we describe the simulation framework we used to evaluate the alternative cache designs. 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. Naturally, 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 migration scheme, and cannot provide the same benefit of reducing cross-chip traffic.

4.1. 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.

Four cache configurations are simulated for each different design, as summarized in Table 1. To simplify result reporting, all latencies are scaled to the access time of the primary cache, which can be reached within a single clock cycle. We assume a 70 nm technology based on BPTM [10], thus use a a 16 FO4 clock cycle for configuration 1, which has a smaller 16KB L1 cache, and a 24 FO4 clock cycle for configurations 2 through 4. Both of these cycle times represent modern power-performance balanced pipeline designs [11, 24]. High-frequency designs might target a cycle time of 8–12 FO4 delays [13, 23], in which case cycle latencies can be multiplied appropriately. Five cycle access latency is used for a 256KB L2 cache and six cycles for 512KB and 1MB caches. We also scale all latencies appropriately for configuration 4, in which the L1 cache is smaller. We model each hop in the network as taking 3 cycles, including the router latency and an optimally-buffered inter-tile copper wire on a high metal layer. Note that the worst case contention-free L2 hit latency is between 29 to 32 cycles for these configurations, 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 workloads had significant inter-thread conflicts, reflected by high off-chip miss rates.

4.2. Workloads

Table 2 summarizes the single-threaded, multi-threaded, and multi-programmed workloads 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-programmed workloads are also created using a mixture of the single-threaded SpecINT2000 benchmarks. Each workload consists of 8 different programs, chosen at random.

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 multiprogrammed workloads are also created using a mixture of the single-threaded SpecINT2000 benchmarks. Each workload consists of 8 different programs, chosen at random.

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

Due to the long running nature of the workloads, we used a sampling technique to reduce simulation time. 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 for 10,000 cycles to warm up the pipelines for the cache, DRAM, and network models. After this warming phase, we gather detailed statistics for one million instructions, before reentering fast-forward mode. Detailed samples are taken at random intervals during execution and include 33% of all instructions executed, i.e., fast-forward intervals average around five million instructions. The number of samples taken for each workload ranges from around 150 to 1,000. To minimize the bias in the results introduced by system variability [2], we ran multiple runs of each workload with varying sample lengths and frequencies. Results show that the variability is insignificant for our workloads.

5. Simulation Results

In this section, we present the results of all four designs for multi-threaded, single-threaded, and multiprogrammed workloads.

5.1. Multi-Threaded Workloads

Figures 6 to 9 show the key result, the average memory access latency seen by a processor. The minimum latency is one cycle, when all accesses hit in the L1 cache. In the following, we take Configuration 1 (8+8/256/16FO4), and give a simple analysis of how each cache design alternative worked. Figure 6 shows the access latency, and Figure 10 shows the memory accesses breakdown. In Figure 10, a cache access is categorized as either an L1 hit, an L2 hit, or a miss that must access off-chip memory. An L2 hit can be either a fast local L2 hit or a slower non-local hit to an L2 slice on a different tile. Hits through cache-to-cache transfers are considered non-local and hits in the replicated victims are considered local.

Analysis

From Figure 6, we observe that for the majority of the workloads, the difference in performance between private and shared designs is significant. One workload, `IS`, has a working set that fits in the L1 cache, thus L2 policies do not matter as most accesses are L1 hits. Average access latency in this case is roughly one cycle for all four designs.

Compared to the shared design (second bar in Figure 10), the private design (first bar in Figure 10) has higher off-chip miss rates but also many more local hits across the workloads. We expect the private design to win if the difference in the off-chip miss rate is small compare to the extra number of local hits it has compared with the shared design. This is the case for workloads `BT`, `CG`, `EP`, `FT`, `LU`, and `apache`. On the other hand, if the difference in off-chip miss rate is significant, we expect the shared design to win even though it has many fewer local hits, as it minimizes expensive off-chip misses. This is the case for workloads `MG`, `SP`, `dbench`, and `checkers`.

Both victim replication (third bar in Figure 10) and victim migration (fourth bar in Figure 10) create replicas for reduced hit latency (shown by the increase in local L2 hits from the shared design) at the expense of slightly increased off-chip miss rate.

Victim migration works better than victim replication for all of these workloads. VM stores the tags of actively shared cache blocks in the VM tag array, thus vacates some of the actual data storage space. This space is split between replicas and unshared global blocks. Having more replicas is likely to increase the number of local L2 hits, and having more global blocks is likely to reduce the off-chip miss rate. Both of the scenarios can be observed by comparing the third and fourth bars in Figure 10.

Other Configurations

As we increase the on-chip capacity, whether private or shared works better changes even for the same workload depending on whether its working set fits into the given cache size. For `EP`, the shared design does better in larger caches as more of its working set fits on-chip. For `SP`, the private design starts to win with larger caches when more of its working set fits into the 1MB local L2 slice. VM manages to be the best policy for most of these workloads. A performance summary is shown in Table 3.

5.2. Single-Threaded Workloads

Figures 11 to 14 show the average access latency for the single-threaded workloads. Because they have working sets that are generally smaller than even the smallest configuration simulated (2MB), VM did not provide noticeable improvement over VR. However, VM is either

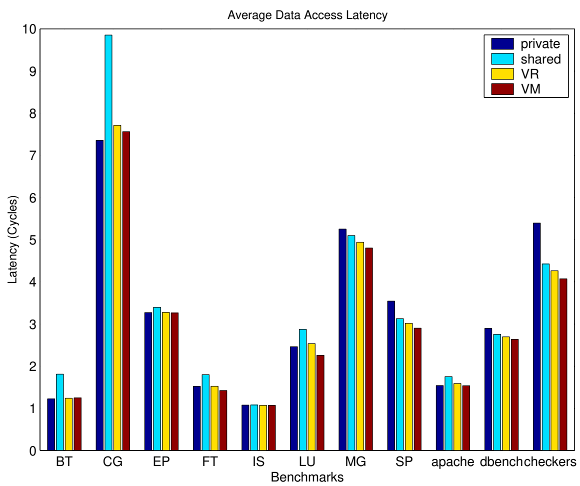

Figure 6. Access latencies of multi-threaded workloads. Configuration 1: 8+8/256/16FO4.

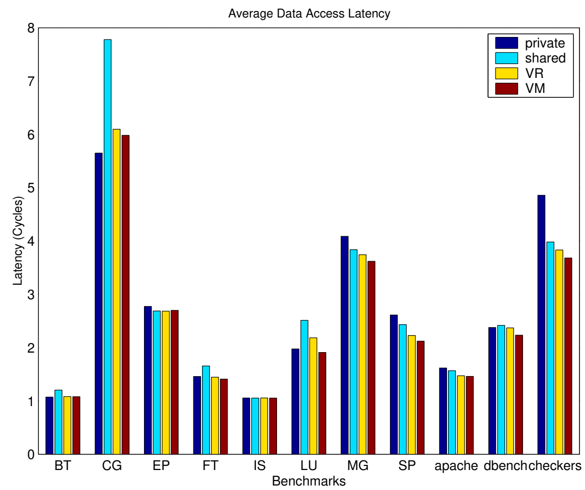

Figure 7. Access latencies of multi-threaded workloads. Configuration 2: 16+16/256K/24FO4.

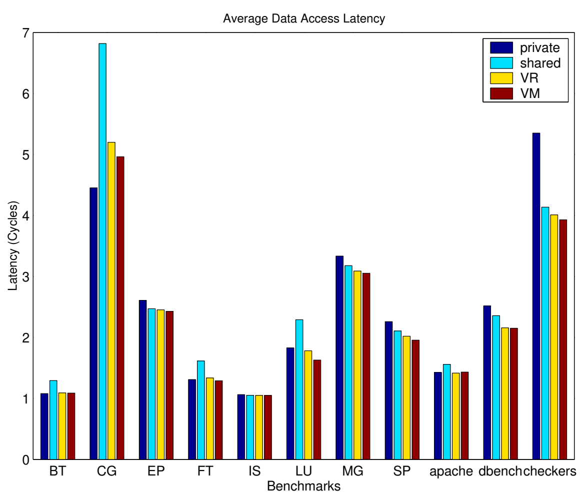

Figure 8. Access latencies of multi-threaded workloads. Configuration 3: 16+16/512K/24FO4.

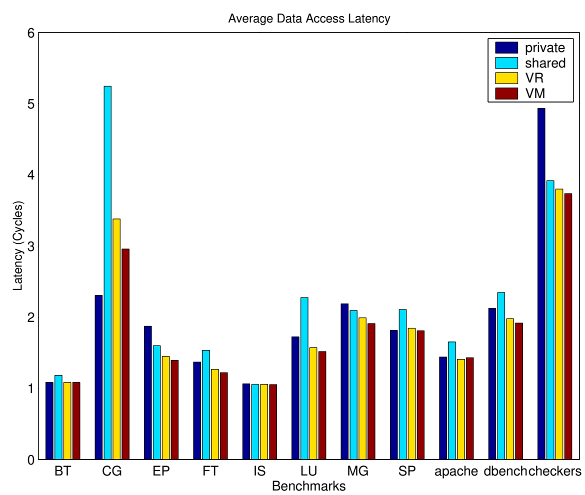

Figure 9. Access latencies of multi-threaded workloads. Configuration 4: 16+16/1024/24FO4.

the best policy or a very close second across all benchmarks.

5.3. Multi-Programmed Workloads

Multiprogrammed workloads tend to have very little sharing among the different threads, thus the private design is likely to do significantly better than the shared design. Figures 15 to 18 confirm this intuition, where the shared design is always the worst by a large margin.

Victim replication's performance is close to the private design, usually within 5%. Because there is very little sharing, each home block is generally used by only one tile, meaning that the cache block is stored twice on the chip. This duplication significantly reduces the effective capacity, making victim replication unlikely to win over the private design. This effect is better demonstrated in the smaller cache sizes, where capacity is at a higher premium.

Compared to victim replication, victim migration eliminates the need to keep a duplicate copy at the home tile, behaving just like the private design when necessary. In addition, VM allows data to be stored at a global location, stealing limited capacity from other threads when their working sets do not saturate their local L2 slice. While more flexible capacity stealing techniques have been proposed in [8, 22], they are based on snooping coherence protocols, in which locating a cache block on-chip is relatively easy. In a directory-based design, however, this global search is likely to be very complex and expensive. Overall, victim migration is the best policy for almost all workloads.

5.4. Reducing VM Tag Array Area Overhead

The main drawback of victim migration is the area overhead caused by the VM tag array. For simplicity, we have so far assumed that the VM tag array size is identical to the regular L2 tags. However, the VM tag array can be of any size and associativity. We selected configuration 3 (16+16/512/24FO4) and simulated the performance of two additional VM tag array sizes: at 50%, and 25% of the regular tag array size. The 50% case caused no performance degradation. The 25% case lost about 15% of the latency reduction achieved by the full VM tag array over victim replication. We also experimented with higher VM tag array associativities and observed no noticeable gains.

5.5. Area Comparison of Designs

The vast majority of the area in caches are occupied by data arrays, peripheral circuitry, and interconnects, which is the same for all four designs described in this paper. However, the tag bits, status bits, and in our case, directory bits, all take up non-negligible space. In this section, we provide a simple quantified comparison of the area occupies by the tags and directories for each of these designs. We use the parameters in configuration 1 (8+8/256/16FO4) in the comparison. We also assume a 40-bit physical address width and 64 byte cache block size, which are modest representatives of modern CMP machines [25].

Table 4 shows the tag and directory area estimates in bits/block used for each design. It also shows the total bits overhead compared to the shared design, which requires the least area. The actual overall cache area overhead is likely to be much smaller than the ones in Table 4 when the area of peripheral circuitry and interconnects are taken into consideration.

In the shared design, the address is used to index a single large shared cache, the width of the tag is smaller than that of the private design. In the 8-tile configuration, three bits are used to select a home tile, making the shared tag 3 bits shorter than that of the private design. The directory uses an 8-bit wide sharing vector. It also leverages the existing valid and dirty status bits to represent state, adding only one extra state bit in our design, for a total of a 9-bit directory. The private design uses the largest area, by having a wider tag and a fully duplicated tag array to maintain the on-chip directory.

For victim replication, the L2 tag must be wide enough to hold physical addresses from any home tile, thus the tag width becomes the same as the private design. Global L2 blocks redundantly set these bits to the address index of the home tile. Replicas of remote blocks can be distinguished from regular L2 blocks as their additional tag bits do not match the local tile index. The full version of victim migration incurs the largest area overhead of all four designs. It consists of all of the components used in victim replication, as well as the VM tag array. However, the overhead can be reduced to less than that of the private design by halving the size of the VM tag array with no performance degradation.

6. Related Work

Data migration techniques [17, 7] discussed in the introduction could have poor performance when applied to tiled CMPs. A given L2 block may be repeatedly accesses by cores at opposite corners of the die. A recent study [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.

Several proposals advocate data replication [8, 27, 22], which allows sharers to replicate local copies of shared data for fast access. Victim replication has been already discussed in detail in this paper. CMP-NuRAPID [8] extends NuRAPID to support data replication for CMPs based on a snooping coherence protocol. However, the actual implementation is complex and incurs a large area overhead. In the baseline IBM Power4 scheme [25], each node has a non-inclusive L3 cache that stores the local L2 victims. However, while L3s can be snooped by other nodes, the local L2 victim always overwrite the local L3, causing considerable pressure on the L3 cache and reduces the effective L3 capacity. In [22], this baseline scheme is improved by using a small history table to selectively remove some clean writebacks with data present in the L3.

Data replication also bears resemblance to earlier work on remote data caching in conventional CC-NUMA and COMA architectures [20, 9, 28], 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 a CMP, 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 blocks that will have to be fetched from slow off-chip memory.

Several related studies include [14], which discusses the design space for future CMPs with private L2 caches. In [15], a hardware hybrid of shared and private designs is proposed, in which a statically determined sharing degree partitions the CMP into private regions, while each region itself maybe contain multiple processor core that share all of the cache resource within the region.

7. Conclusion

As wire delay continue to worsen in future microprocessors, both off-chip accesses and on-chip communications must be carefully managed to efficiently utilize the on-chip cache capacity. Typical private and shared designs either have low hit latency or low off-chip miss rate, but not both, prompting hybrid techniques trying to combine the advantages of both. This paper introduces victim migration, a hybrid technique that extends the previously proposed victim replication scheme. Victim migration uses an extra migration tag array to remove the need to duplicate shared data at the home tile, freeing significant space for storing additional replicas and global data. We show that victim migration is the best overall policy across a wide range of workloads.

8. Acknowledgments

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*, Austin, TX, March 2005.

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

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

[6] M. Cameron and B. Rohit. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, March/April 2005.

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

[8] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. In *ISCA-32*, Madison, WI, June 2005.

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

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

[11] A. Hartstein and T. Puzak. Optimum power/performance pipeline depth. In *MICRO-36*, San Diego, CA, 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*, Anchorage, AK, May 2002.

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

[15] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler. A NUCA substrate for flexible CMP cache sharing. In *ICS05*, Cambridge, MA, June 2005.

[16] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, March/April 2003.

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

[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] Raza Microelectronics, Inc. XLR processor product overview, May 2005.

[22] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive mechanisms and policies for managing cache heirarchies in chip multiprocessors. In *ISCA-32*, Madison, WI, June 2005.

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

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

[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] M. Zhang and K. Asanović. Victim Replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA-32*, Madison, WI, June 2005.

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

Component	Parameter			
	Configuration 1 8+8/256/16FO4	Configuration 2 16+16/256/24FO4	Configuration 3 16+16/512/24FO4	Configuration 4 16+16/1024/24FO4
L1 I-Cache Size/Associativity	8 KB/16-way	16 KB/16-way	16 KB/16-way	16 KB/16-way
L1 D-Cache Size/Associativity	8 KB/16-way	16 KB/16-way	16 KB/16-way	16 KB/16-way
L1 Load-to-Use Latency	1 cycle	1 cycle	1 cycle	1 cycle
L1 Replacement Policy	Psuedo-LRU	Psuedo-LRU	Psuedo-LRU	Psuedo-LRU
L2 Cache Slice Size/Associativity	256 KB/16-way	256 KB/16-way	512 KB/16-way	1 MB/16-way
L2 Load-to-Use Latency (per slice)	8 cycles	5 cycles	6 cycles	6 cycles
L2 Replacement Policy	Random	Random	Random	Random
External memory latency	192 cycles	128 cycles	128 cycles	128 cycles
One-hop latency	3 cycles	3 cycles	3 cycles	3 cycles
Worst case L2 hit latency (contention-free)	32 cycles	29 cycles	30 cycles	30 cycles
CMP Configuration	4×2 Mesh			
Processor Model	in-order			
Cache Line Size	64 B			

Table 1. Simulation parameters. The numbers for each configuration represent the cache sizes and cycle times. For example, 8+8/256/16F04 reads 8K L1I cache, 8K L1D cache, 256K L2 cache, with 16 FO4-delay cycle time.

Benchmark (Instruction Count in Billions)		Description
Multi-Threaded Benchmarks		
BT	(1.7)	class S. block-tridiagonal CFD application
CG	(5.0)	class W. conjugate gradient kernel
EP	(6.8)	class W. embarassingly parallel kernel
FT	(6.6)	class S. 3X 1D fast fourier transform (-O0)
IS	(5.5)	class W. integer sort. (icc-v8)
LU	(6.2)	class R. LU decomposition. with SSOR CFD application
MG	(5.1)	class W. multigrid kernel
SP	(6.7)	class R. scalar pentagonal CFD application
apache	(3.3)	Apache's 'ab' worker threading model, 2000 requests, 3 at a time (gcc 2.96)
dbench	(3.3)	executes Samba-like syscalls, 3 clients, 10000 requests (gcc 2.96)
checkers	(2.9)	Cilk checkers (parallel $\alpha-\beta$ search), Black 6, White 5 (Cilk 5.3.2, gcc 2.96)
Single-Threaded Benchmarks		
bzip	(3.8)	`bzip2` compression algorithm version 0.1
crafty	(1.2)	A high-performance chess program designed around a 64-bit word
eon	(2.9)	A probabilistic ray tracer
gap	(1.1)	A language and library designed mostly for computing in groups
gcc	(6.4)	`gcc` compiler version 2.7.2.2 for Motorola 88100 processor
gzip	(1.0)	Data compression program using LZ77 for GNU
mcf	(1.7)	Single-depot vehicle scheduling algorithm in public mass transportation
parser	(5.6)	A word processing parsing tool
perlbmk	(1.8)	A cut-down version of Perl v5.005_03 without most OS-specific features
twolf	(1.5)	The TimberWolfSC place and route tool
vortex	(1.5)	An single-user object-oriented database transaction program
vpr	(5.3)	A place and route tool for FPGAs
Multi-Programmed Benchmarks		
mix0	(23.9)	bzip, crafty, eon, gap, gcc, gzip, mcf, parser
mix1	(24.8)	gcc, gzip, mcf, parser, perlbmk, twolf, vortex, vpr
mix2	(19.1)	bzip, crafty, eon, gap, perlbmk, twolf, vortex, vpr
mix3	(22.8)	bzip, gap, mcf, twolf, crafty, gcc, parser, vortex
mix4	(19.1)	bzip, gap, mcf, twolf, eon, gzip, perlbmk, vpr
mix5	(25.7)	crafty, gcc, parser, vortex, eon, gzip, perlbmk, vpr
mix6	(12.7)	crafty, eon, gap, gzip, mcf, perlbmk, twolf, vortex
mix7	(21.5)	bzip, gap, gzip, mcf, parser, twolf, vortex, vpr
mix8	(28.0)	bzip, crafty, eon, gap, gcc, mcf, parser, vpr

Table 2. Benchmarks Descriptions. Three different categories of benchmarks are used: single-threaded benchmarks, multi-threaded benchmarks, and multi-programmed benchmarks.

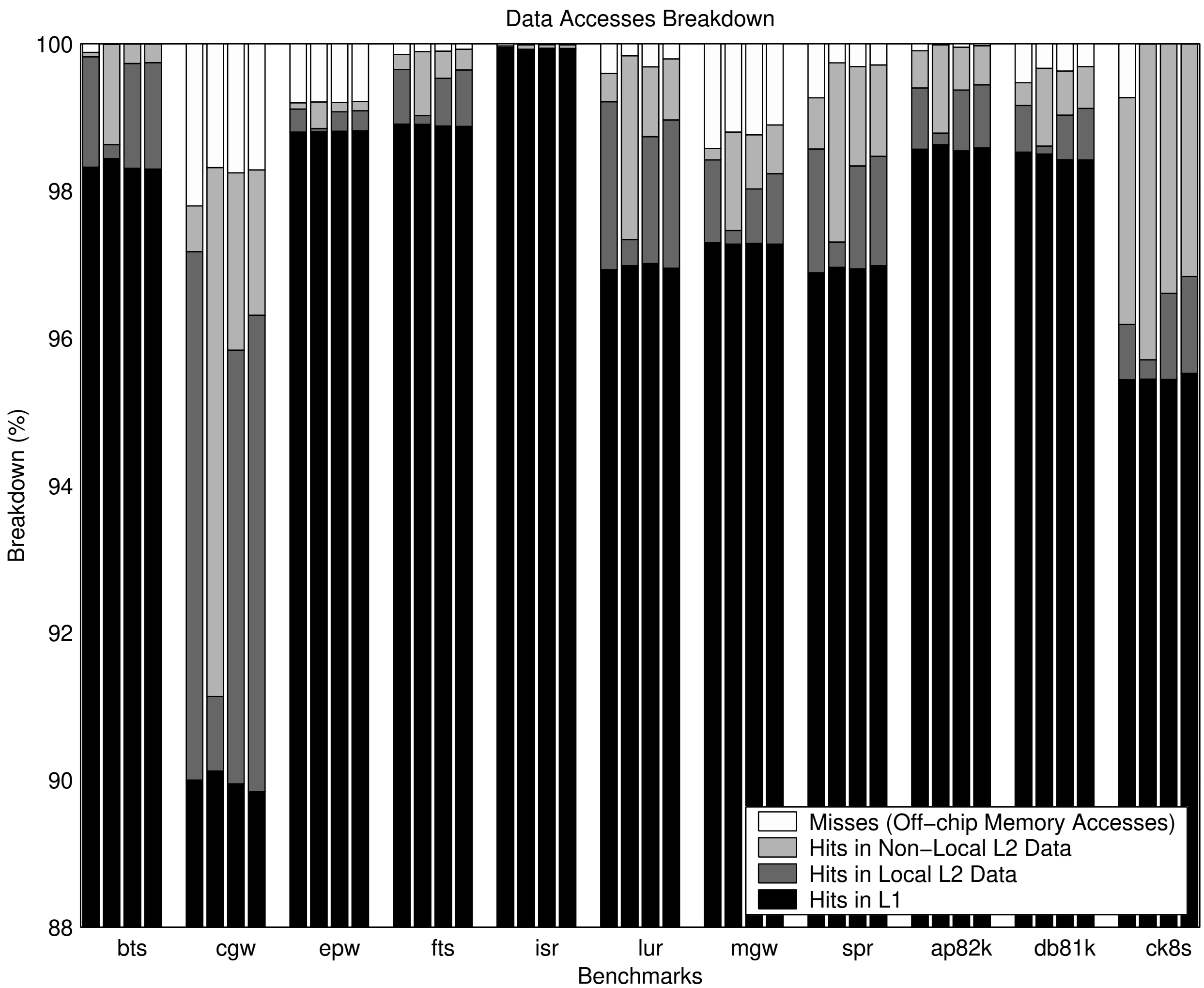

Figure 10. Memory access breakdown of multi-threaded programs. Moving from left to right, the four bars for each workload are for the private design, shared design, victim replication, and victim migration, respectively. Hits from cache-to-cache transfers are considered non-local and hits from replicas are considered local.

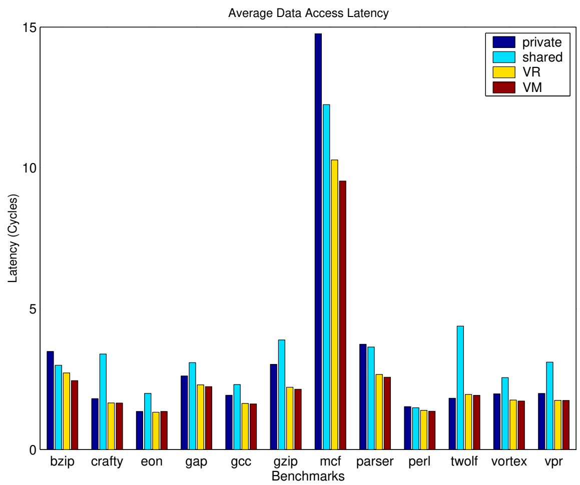

Figure 11. Access latencies of single-threaded workloads. Configuration 1: 8+8/256/16FO4.

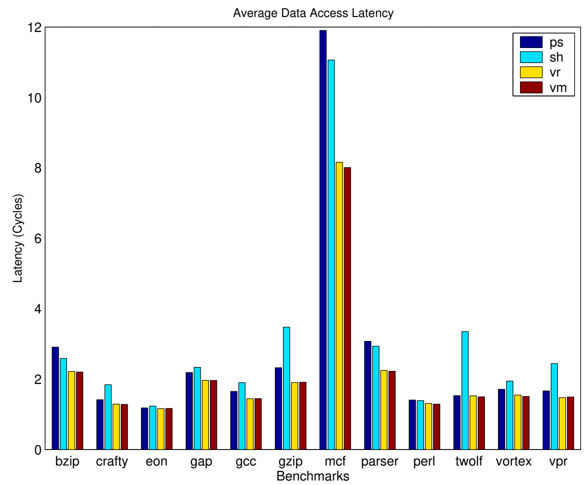

Figure 12. Access latencies of single-threaded workloads. Configuration 2: 16+16/256K/24FO4.

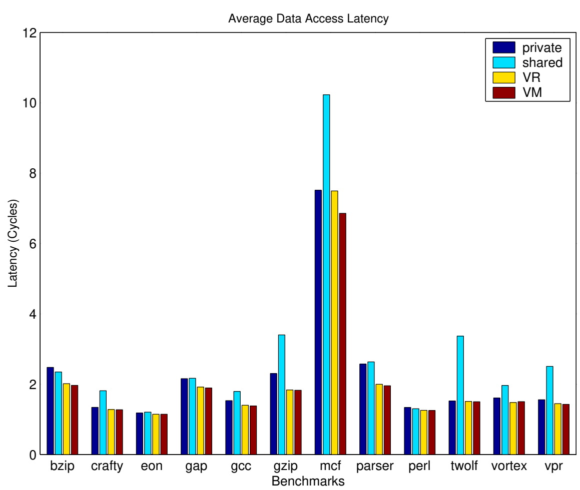

Figure 13. Access latencies of single-threaded workloads. Configuration 3: 16+16/512/24FO4.

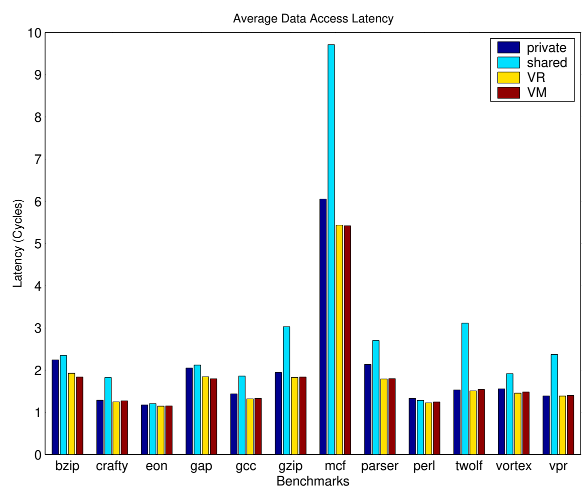

Figure 14. Access latencies of single-threaded workloads. Configuration 4: 16+16/1024/24FO4.

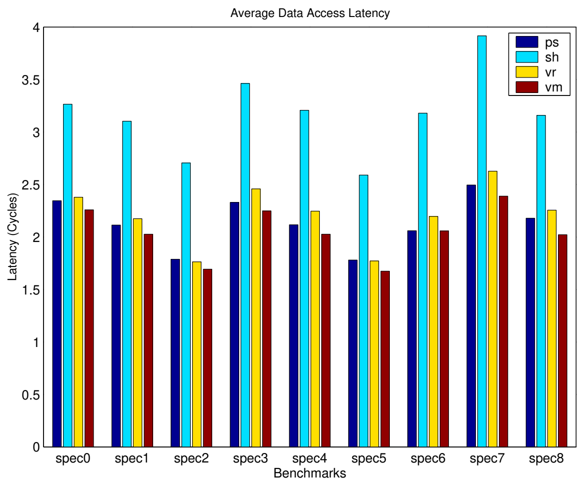

Figure 15. Access latencies of multi-programmed workloads. Configuration 1: 8+8/256/16FO4.

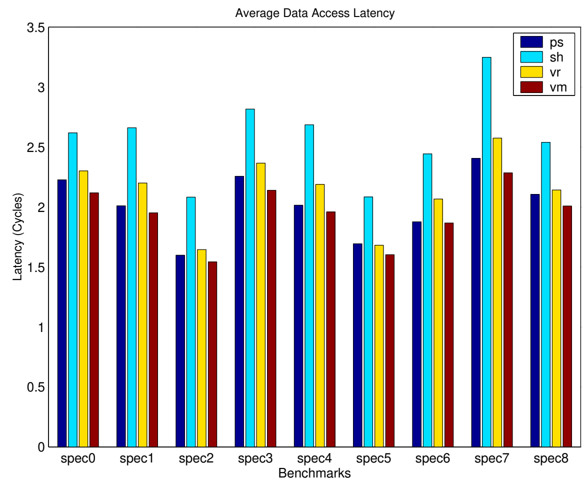

Figure 16. Access latencies of multi-programmed workloads. Configuration 2: 16+16/256/24FO4.

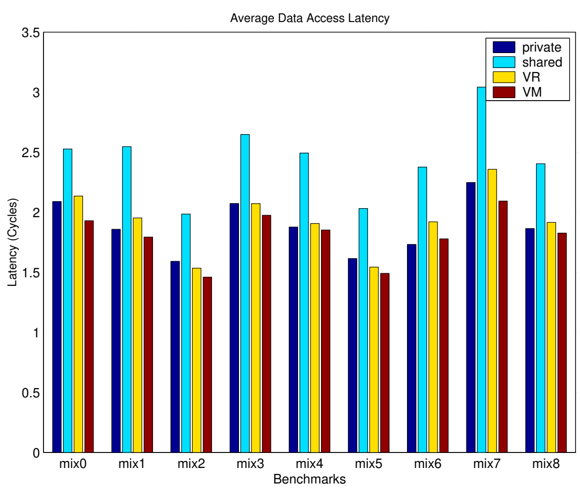

Figure 17. Access latencies of multi-programmed workloads. Configuration 3: 16+16/512/24FO4.

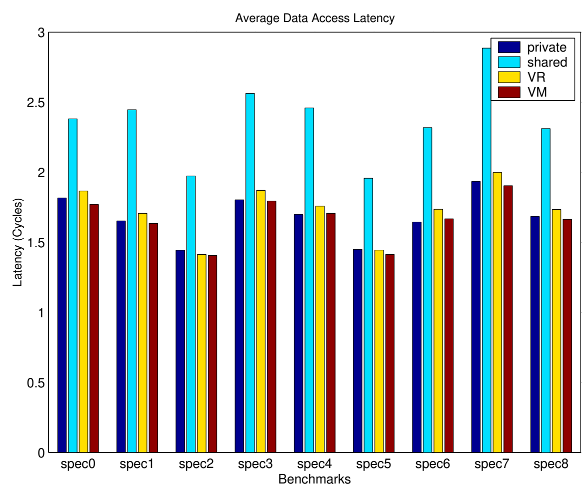

Figure 18. Access latencies of multi-programmed workloads. Configuration 4: 16+16/1024/24FO4.

	Configuration 1 8+8/256/16FO4			Configuration 2 16+16/256/24FO4			Configuration 3 16+16/512/24FO4			Configuration 4 16+16/1024/24FO4		
	Reductin (%)			Reductin (%)			Reductin (%)			Reductin (%)		
Workload	VM/S	VM/P	VM/VR	VM/S	VM/P	VM/VR	VM/S	VM/P	VM/VR	VM/S	VM/P	VM/VR
Multi-Threaded Workloads												
BT	45.3	-1.9	-0.6	11.5	-0.6	0.2	18.9	-0.6	0.4	9.4	0.0	0.0
CG	30.3	-2.7	2.0	30.0	-5.6	1.9	37.4	-10.3	4.8	77.3	-22.1	14.2
EP	4.1	0.3	0.3	-0.4	2.7	-0.5	1.8	7.5	1.1	14.8	34.5	3.9
FT	26.4	5.6	7.1	17.5	3.5	2.5	25.2	1.6	3.6	25.8	12.3	3.8
IS	0.8	0.6	0.1	-0.0	0.1	0.0	0.3	1.3	0.1	0.2	0.9	0.5
LU	27.3	9.0	12.3	31.6	3.5	14.4	40.5	12.2	9.4	50.1	13.7	3.7
MG	6.1	9.3	2.8	6.0	12.9	3.4	4.1	9.2	1.2	9.6	14.5	4.3
SP	7.7	22.1	4.0	14.6	23.1	5.0	7.7	15.5	3.3	16.5	0.2	2.0
apache	14.0	-0.2	3.6	7.3	10.9	0.9	8.8	-0.4	-1.1	15.4	0.7	-1.6
dbench	4.5	9.9	2.2	8.2	6.4	6.0	9.6	17.0	0.3	22.3	10.7	3.1
checkers	8.7	32.5	4.7	8.1	31.9	4.0	5.3	36.2	2.1	4.9	32.0	1.7
Avg	15.9	7.6	3.5	12.2	8.1	3.4	14.5	8.1	2.3	22.4	8.9	3.2
Single-Threaded Workloads												
bzip	22.3	42.3	11.2	17.5	32.0	0.8	19.4	26.0	2.4	27.6	22.1	4.9
crafty	105.0	9.4	0.2	43.6	10.4	0.7	42.3	5.4	0.5	43.4	1.1	-1.7
eon	47.2	0.1	-2.1	5.7	1.1	-0.5	5.1	2.9	-0.1	4.5	1.8	-0.5
gap	38.0	16.9	2.8	18.9	11.5	0.2	14.5	13.9	1.3	18.0	14.0	2.6
gcc	42.6	19.2	1.1	31.3	14.0	-0.4	29.8	10.8	1.3	39.5	8.0	-0.9
gzip	81.6	41.1	3.3	81.6	21.5	-0.5	86.2	26.0	0.5	65.0	5.8	-0.4
mcf	28.5	54.9	7.9	38.1	48.6	1.8	49.2	9.6	9.3	79.2	11.7	0.3
parser	41.7	45.7	3.9	31.8	38.1	0.9	34.7	31.5	2.2	50.1	18.6	-0.4
perl	9.1	11.8	2.2	7.8	9.1	1.4	4.2	6.8	0.4	3.1	6.9	-1.6
twolf	127.4	-5.3	1.6	123.6	2.3	1.9	124.2	1.6	0.5	101.9	-0.7	-2.0
vortex	48.2	14.8	2.2	28.8	13.3	2.6	30.7	6.9	-1.6	29.5	5.1	-1.8
vpr	78.0	14.2	0.2	63.8	11.5	-1.1	75.6	9.1	1.3	69.2	-0.8	-1.1
Avg	55.8	22.1	2.9	41.0	17.8	0.7	43.0	12.5	1.5	44.3	7.8	-0.2
Multi-Programmed Workloads												
mix0	44.6	4.7	5.3	23.6	5.1	9.1	30.9	8.2	10.6	34.6	2.7	5.5
mix1	53.1	6.3	7.3	36.3	3.5	12.7	42.0	3.6	8.9	49.6	1.1	4.4
mix2	59.7	7.4	4.1	35.1	4.3	6.7	36.0	9.0	5.1	40.3	2.7	0.5
mix3	54.0	5.4	9.4	31.7	5.5	10.6	34.0	5.0	4.9	42.8	0.5	4.3
mix4	58.2	5.9	10.8	37.1	2.9	11.7	34.6	1.3	2.9	44.1	-0.5	3.1
mix5	54.7	6.4	5.9	30.0	5.7	4.9	36.2	8.3	3.5	38.6	2.6	2.3
mix6	54.5	0.1	6.7	30.8	0.5	10.7	33.6	-2.6	8.0	39.0	-1.4	4.1
mix7	63.9	4.4	10.0	42.2	5.3	12.7	45.3	7.3	12.7	51.6	1.5	4.9
mix8	56.2	7.8	11.6	26.4	4.8	6.6	31.6	2.1	4.9	38.9	1.1	4.2
Avg	55.4	5.4	7.9	32.6	4.2	9.5	36.0	4.7	6.8	42.2	1.1	3.7

Table 3. Access latency reduction achieved by victim migration. The three numbers for each workload indicate the percentage reduction with respect to the shared design (VM/S), the private design (VM/P), and victim replication (VM/VR).

Design	Tag width	Dir. entry width	Total Width	Bit/Block Overhead vs. Shared
Shared	25	9	34	0.0%
Private	28	28	56	4.0%
Victim Replication	28	9	37	0.6%
Victim Migration (1/1)	28	43	71	6.8%
Victim Migration (1/2)	28	26	54	3.7%
Victim Migration (1/4)	28	20	48	2.6%

Table 4. Cache area overhead of different designs.