An Adaptive Shared/Private NUCA Cache Partitioning Scheme for Chip Multiprocessors*

Haakon Dybdahl
Norwegian University of Science and Technology
NO-7491 Trondheim, Norway
dybdahl@idi.ntnu.no

Per Stenström
Chalmers University of Technology
SE-412 96 Goteborg, Sweden
pers@ce.chalmers.se

Abstract

The significant speed-gap between processor and memory and the limited chip memory bandwidth make last-level cache performance crucial for future chip multiprocessors. To use the capacity of shared last-level caches efficiently and to allow for a short access time, proposed non-uniform cache architectures (NUCAs) are organized into per-core partitions. If a core runs out of cache space, blocks are typically relocated to nearby partitions, thus managing the cache as a shared cache. *This uncontrolled sharing of all resources may unfortunately result in pollution that degrades performance.*

We propose a novel non-uniform cache architecture in which the amount of cache space that can be shared among the cores is controlled dynamically. The adaptive scheme estimates, continuously, the effect of increasing/decreasing the shared partition size on the overall performance. We show that our scheme outperforms a private and shared cache organization as well as a hybrid NUCA organization in which blocks in a local partition can spill over to neighbor core partitions.

1 Introduction

Two important challenges for next generation microprocessors are the slow main memory and the limited off-chip bandwidth. Efficient management of the last-level on-chip cache is therefore important in order to accommodate a larger number of cores in future multi-core architectures.

A last-level multi-core cache can be organized as private partitions for each core or having all cores sharing the entire cache. The shared cache organization can be utilized more flexibly by sharing data between cores. However, it is slower than a private cache organization. In addition, private caches do not suffer from being polluted by accesses from other cores by which we mean that other cores displace blocks without contributing to a higher hit rate.

Non-uniform cache architectures (NUCA) are a proposed hybrid private/shared cache organization that aims at combining the best of the two extreme organizations [2, 3, 5, 8, 9, 16] by combining the low latency of small (private) caches with the capacity of a larger (shared) cache. Typically, frequently used data is moved to the shared cache portion that is closest to the requesting core (processor); hence it can be accessed faster. Recently, NUCA organizations have been studied in the context of multi-core systems as a replacement for a private last-level cache organization [3, 6]. The cache is statically organized into private partitions but a partition attached to one core can also keep blocks requested by other cores. When a block is installed in a certain partition, a replaced block from that partition will be installed in a neighbor's partition, picked by random. As a result, on a miss in one partition, all other partitions are first checked before accessing main memory. While this hybrid scheme provides fast access to most blocks, it can suffer from *pollution* because of the uncontrolled way by which partitions are shared among cores.

Our contribution in this paper is a novel NUCA design for multi-cores based on private partitioning in which the sizes of the core-local partitions that are shared are chosen adaptively to maximize the overall performance. We show in the paper that our adaptive scheme outperforms the uncontrolled sharing of private partitions in [3] which is prone to pollution effects.

Our work is inspired by earlier work on dynamic partitioning of the resources in *shared caches* among

*Per Stenstrom is a member of the HiPEAC Network of Excellence funded by EU under FP6.

cores [7, 10, 13]. In the NUCA setting, the new issue becomes how to select the *size of the shared partition* which is not addressed in the earlier work. We combine and extend several existing mechanisms intended for solving problems in other contexts. The contribution of this paper is the unique combination and usage of these mechanisms and the novel architecture for the last-level cache.

In our organization, hits to the private partitions are fast, while hits to neighboring partitions are slower. The size of the private partition is dynamically controlled and balanced against the other cores. Cores that can best utilize the cache get more private cache space, but the private cache space is never larger than the local last-level cache. The cache usage in the shared partition is controlled as well. The size of the private partitions for each core is dynamically adjusted to minimize the total number of cache misses by determining the change in the total miss rate.

We compare the performance of the new scheme with the performance of private and shared cache organizations. Additionally, we also compare it with the NUCA scheme proposed by Chang and Sohi [3] in which essentially all last-level cache resources can be shared. The new scheme outperforms all these schemes. Our simulations show that we can improve the performance by more than 20% for the memory-intensive applications in the SPEC2000 suite compared to private caches.

We describe the new scheme in Section 2. Sections 3 and 4 then present the evaluation methodology and the results, respectively. Related work is discussed in Section 5 and we conclude in Section 6.

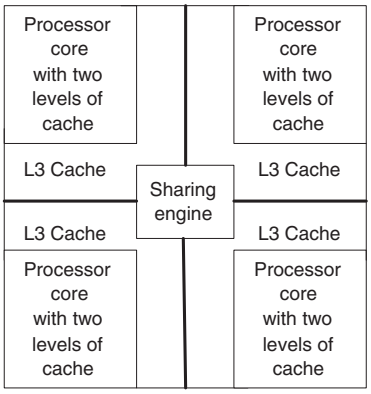

Figure 1. Conceptual view of the on-chip architectural framework (not floorplan).

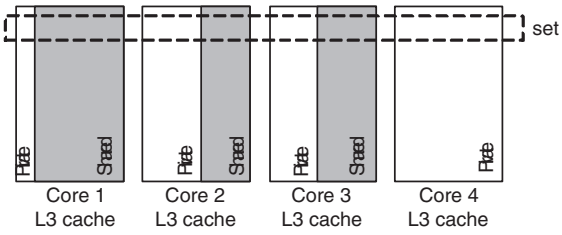

Figure 2. Sharing of the L3 cache. Each local cache has a private and potentially a shared partition.

2 The Adaptive Partitioning Scheme

An architectural framework for a four-core system with the proposed scheme is shown in Figure 1. Note that this is a conceptual view rather than implying a physical layout. Each core has three levels of cache, where the L3 cache is the last-level cache. The *sharing engine* implements sharing of the last-level cache among the cores. Hits in the core-local L3 cache partition are faster than the hits in the neighboring last-level cache partitions.

The L3 cache usage is controlled in two ways: (a) a part of the cache is private and inaccessible by the other cores, (b) the size of the private cache partition and the number of cache blocks in the shared partition of the cache are controlled on a per-core basis in order to minimize the *total* number of cache misses. The division of private and shared partitions is conceptually shown in Figure 2. Each core has its own local cache with two partitions: A private partition for its own cache blocks and a shared cache partition intended for all the cores. Each set is divided among the four cores. The most recently used cache blocks for each core are stored in the private (and fast) partition of the set. Looking for a tag match in the set is a two phase process. First the tags in the private cache partition are checked then, if there is no match, the rest of the set is checked.

The sharing engine depicted in Figure 1 consists of several components: (a) a method for estimation of the best private/shared partitioning of caches, (b) a method for sharing the cache and (c) a replacement policy for the shared cache space. These components and the necessary structures are described in the following subsections.

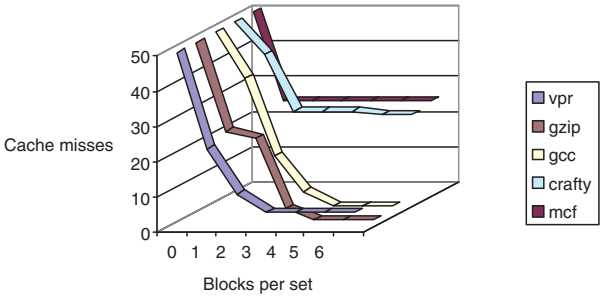

Figure 3. Number of misses as a function of number of blocks per set.

2.1 Estimation of Partition Sizes

Our scheme adapts the size of the private partitions by essentially increasing or decreasing the number of blocks per set for each private partition but keeping the number of sets fixed. Obviously, the potential benefit of balancing the cache partitions depends on the cache-size sensitivity of the applications that are running. For illustrative purposes, we show the cache-size sensitivity for five applications in Figure 3 for a fixed snapshot (in processor cycles) of the entire execution. The graph shows the number of cache misses per application as a function of the number of cache blocks per set but with a fixed number of sets for the cache.

For *mcf*, the innermost graph, only a single block per set is required. There is no benefit from increasing the number of blocks per set as the remaining misses are likely cold misses. On the other hand, *gzip* requires four blocks per set to avoid most misses.

Our scheme adjusts the shared partition size if the total number of misses is expected to go down if one core gives up a block to the benefit of another. For example if *mcf* and *gzip* both had three blocks per set it would make sense to change the partition so that *mcf* had two blocks and *gzip* had four blocks per set since the total number of cache misses is then reduced (see Figure 3). Adjusting partition sizes dynamically requires two methods: one for estimating the gain of increasing the cache size and one for estimating the loss of decreasing the cache size per application/core. We present the methods chosen next.

The method for estimating the number of cache misses that would have been avoided if the cache size was increased with one block per set is implemented as follows. Each set has a register for each core that is referred to as *shadow tag* according to Figure 4(b). When a cache block is evicted from the L3 cache, the tag of the block is stored in the *shadow tag* associated with the core that fetched the block into cache. On a cache miss, the tag of the miss is compared to the *shadow tag* for that set. If there is a match, the counter *hits in shadow tags* for the requesting core is increased. This counter is shown in Figure 4(c).

The method for estimating the number of increased cache misses as a result of decreasing the cache size with one block per set is derived from Suh et al. [13] and works as follows. If there is a hit in the LRU block for the requesting core, a counter is increased for that core. This counter (see Figure 4(c)) represents the number of cache misses that would have occurred if the cache size is reduced by one block per set.

The algorithm re-evaluates the private partition sizes per core on a regular basis. In our experiments, we use 2000 cache misses in the last-level cache to trigger a re-evaluation. This period is long enough to measure cache sensitivity and short enough to make the scheme dynamic. The core with the highest gain for increasing the cache size, i.e. the core with most hits to its shadow tags, is compared to the core with the lowest loss of decreasing the cache size, i.e. the core with the fewest hits to its LRU blocks. If the gain is higher than the loss, one cache block (per set) is provided to the core with the highest gain. The counters are reset after each re-evaluation period.

In the initial partitioning, 75% of the local cache partition acts as a private cache whereas 25% is a contribution to the shared partition.

2.2 The Structures

The hardware structures needed for the new scheme for a four core multi-core chip are shown in Figure 4. Even though the structures are shown as tables, the physical layout can be divided and distributed among the cores. Each cache block is extended with a *core identification* as shown in Figure 4(a). This field is updated with the value from the requesting core every time a cache block is installed in the cache. When a cache block is evicted from the last-level cache, the tag of the block is stored in the shadow tag table for the core that fetched the block, see Figure 4(b). The last block that was evicted for core 2 in set 1 is the block with tag *f*. Accesses that miss in the cache, but have a tag match in the shadow tag table, would hit in the cache if the private partition had one more block in this set. This event is counted by the shadow tag's hit counter according to Figure 4(c). For example if core 1 requests the block with tag *a* in set 0, the counter for the shadow tag's hits will increase from 10 to 11. The other counter, *hits in the*

Index	Tag	LRU data	Cache line data	Core ID
..	..	..	..	..

(a) This is an example of a cache block. Each cache block is extended to include the core ID.

Set number	Core 1	Core 2	Core 3	Core 4
0	*a*	*b*	*c*	*d*
1	*e*	*f*	*g*	*h*
..	..	..	..	..

(b) This is the structure with the shadow tags. Each cache set has one shadow tag per core.

Counter	Core 1	Core 2	Core 3	Core 4
Hits in the LRU blocks	2	3	2	3
Hits in the shadow tags	10	11	9	2

(c) There are two global counters per core.

Description	Core 1	Core 2	Core 3	Core 4
Max. no. of blocks in set	2	3	2	3

(d) The partitioning parameters (one per core) used by the replacement policy.

Figure 4. The extra storage requirements for the new scheme.

LRU blocks, is increased when a request hits in the LRU block. This number represents the increase in number of misses as a result of reducing the cache size by one block per set.

A constraint (*max. no. of blocks per set*) is associated with each core that limits the maximum number of blocks that can be in each set as shown in Figure 4(d). This value reflects the total maximum number of blocks in each set for both the private partition and the shared partition. If the value is larger than the maximum size of the private partition (i.e. the associativity of the local cache) the core can use the shared space in addition to the private partition. If the value is smaller than the maximum size of the private partition, some cache blocks in the private partition are shared. In this case the core is still allowed to allocate *one* block in the shared partition. This increases the cache space flexibility and utilization. In such cases, the allocated block will be a candidate for eviction by the replacement policy for the shared partition.

2.3 Management of the Partitions

Each L3 cache is divided into a private and a shared partition as shown in Figure 2. The private partition is not shared and is managed by a least recently used (LRU) replacement policy. To locate a block in the cache, the partitioning does not matter. However, to find a victim using LRU, the private partition is only considered and blocks belonging to the shared partition are not involved. Compared to a conventional LRU algorithm this requires that only a part of the blocks in a set is affected by an eviction. In our evaluation we use a 4-way private cache so there are only four different cases to consider and the amount of extra logic should hence be small.

In order to describe the behavior of the entire cache scheme, let us consider the key events:

- **Cache hit in private portion of L3 cache.** The block that is hit is moved to the top of the LRU stack, and the others are moved down. No access to the shared partition of the cache set is required.

- **Cache hit in neighboring L3 cache.** Before this happens, a miss occurred in the private partition of the last-level cache. Then all the neighboring caches are checked in parallel since the cache block can be in any of these caches. The cache block with the hit is moved to the local cache. The LRU cache block in the private cache replaces the block with a hit in the shared cache, and the block is set as MRU in the shared cache.

- **Cache miss.** The block is requested from main memory and inserted into the private cache. The LRU block in the private partition of the cache is inserted into the shared partition of the cache. The block that is evicted is found according to Algorithm 1. We describe the algorithm in the next subsection.

2.4 The Replacement Policy

On cache misses, data loaded from main memory is always allocated in the private partition of the cache as most recently used. This normally requires that one block is evicted from the private partition of the cache. The evicted block is allocated in the shared cache partition. Each core has a minimum of 1 cache block per set in the shared block partition, so space is guaranteed for this block. The algorithm for finding which block to evict in the shared cache partition in order to allocate the evicted cache block from the private cache is shown in Algorithm 1. The search for a victim block starts at the bottom of the LRU stack (step 2) and steps the LRU stack towards the MRU block. If the core that owns the cache block has too many cache blocks within the set (step 4), this block is chosen for eviction (step 5). If no block is found, the LRU cache block is evicted (step 8).

2.5 Repartitioning of the Cache

Repartitioning the cache might sound expensive. However, the only changes that are needed are in the partition parameters for the private and shared cache. This influences the replacement policy of the cache, and the eviction process on cache misses that actually repartition the cache later on. When the partition size for one core is decreased, the "extra" cache blocks are not invalidated, but they are valid until they are evicted. When a private set is controlled for a hit, the tags for all blocks in the set are compared to the tag of the requested block including the blocks that are not in the partition of the private set. This lazy repartitioning, i.e. repartitioning only involves changing the parameters for the replacement policy, requires virtually no effort and can therefore be done at any pace.

Algorithm 1 Pseudo code for finding a block for eviction. The function returns the position of the block to evict.

```
1: function FIND BLOCK TO EVICT
2:     for LRU_stack_pos ←no blocks per set, 1 do
3:         proc_id ← core owns block(LRU_pos)
4:         if max no of blocks in set[proc_id] <
   count no of blocks in set(proc_id) then
5:             return LRU_stack_pos
6:         end if
7:     end for
8:     return number of blocks per set
9: end function
```

2.6 Discussion

One implication of using the new scheme is that some applications do not get the cache space they require because some other applications can utilize the cache more efficiently in the sense that the total miss rate will be lower. This means that an application that frequently accesses the last-level cache and which benefits from a large cache space is likely to get more cache capacity. An application that infrequently accesses the last-level cache, but that would also experience a lower miss rate from a larger cache space will less likely get a larger cache space since the number of misses removed by increasing the cache space for that application is lower. Consequently, applications that access the cache rarely or that do not benefit from a larger cache space will receive modest cache capacity if other more demanding cores (on the same chip) benefit from a large cache space.

The speed of the application (instructions per clock cycle) will depend on the number of accesses to main memory since these take several hundred clock cycles. The new scheme will prioritize these slow running applications if increasing the cache size helps. This is different from maximizing the average speed of the cores in a multi-core chip. In fact, the objective is to maximize the harmonic mean performance of the cores. As Smith points out [12]: This is more important than optimizing the average performance since most systems are often bound by the slowest running application. The result is that performance might be sacrificed for fast running applications to speed up slower running applications compared to a conventional shared cache.

2.7 Implementation Cost

The implementation cost can be divided into the extra storage required by the new scheme and the extra logic required.

The shadow tags require $s * p * t$ bits where s is the number of sets, p is the number of cores, t is the number of bits per tag. However, shadow tags are not needed for all sets as shown later in Section 4.6. We have found that monitoring only 6% of the sets is sufficient for estimating cache sizes with no degradation of performance. The field for the ID of the core that fetched the block requires $\log_2 p*b$ bits where b is number of cache blocks. Finally, the storage for the two counters and one register per core is $p*3*w$ bits for w-bits registers and counters. The total storage cost is then $0.06*s*p*t+\log_2 p*b+p*3*w$ bits. For the baseline architecture used in the evaluation section the storage requirements are increased with 152 Kbits. 16% of this is used for shadow tags and 84% is used for core IDs in the blocks. This is an increase of 0.5% of the storage requirement for a 4-MByte last-level cache.

The logic and communication in the sharing engine require some extra cache logic. However, a conventional multi-core chip with private caches also requires logic for connecting the caches as they usually share the off-chip bus. Most of the logic in the sharing engine can be rather slow since its latency is overlapped by the slow memory access latency. We therefore believe that the area and power budget for the sharing engine are modest, however more work is required to quantify this.

3 Methodology

Simulation is used to compare the efficiency of the new scheme with a conventional LRU-based shared

Table 1. The baseline configuration used for the experiments.

Parameter	Value
Register Update Unit Size	128 instructions
Load Store Queue	64 instructions
Fetch queue size	4 instructions
Fetch, Decode, Issue and Commit width	4 instructions/cycle
Functional Units	4 INT ALUs, 4 FP ALUs, 1 INT Multiply/Divide, 1 FP Multiply/Divide
Branch Predictor	Combined, Bimodal 4K table, 2-Level 1K table, 10-bit history table, 4K Chooser
Branch Target Buffer	512-entry, 4 way
Mispredict Penalty	7 cycles
L1 Instruction/Data Cache	64K, 2-way (LRU), 64 B Blocks, 2/3 cycle latency
L2 Instruction/Data Cache	128/256K, 4-way (LRU), 64 B Blocks, 9/9 cycle latency
Shared L3 Cache	4 MByte unified, 16-way (LRU), 64 B Blocks, 19 cycle latency
Private L3 Cache	1 MByte per processor, 4-way (LRU), 64 B Blocks, 14 cycle latency for private cache, 19 cycle latency for neighboring cache
Main Memory	260 cycles first chunk (258 for private cache), 4 cycles inter chunk. Chunk size 8 bytes. 9 GBytes/s theoretical limit for 4.5 GHz processor
I-TLB/D-TLB (Translation Lookaside Buffers)	128-entry, fully associative, 30 cycles miss penalty
Processor cores	4 independent cores

cache, with private caches and with recently proposed NUCA schemes. The simulated architecture is shown in Figure 1. The baseline chip multiprocessor (CMP) architecture has four cores per chip with private L3 cache partitions that can also be shared. We use a detailed pipeline-level, out-of-order execution model simulator with non-blocking caches to get statistics on improvements using the instructions-per-cycle (IPC) metric. The model is based on *SimpleScalar* version 3 [1], but is extended to simulate the new schemes and CMP configurations including congestion to main memory.

The baseline parameters for the simulator are shown in Table 1. The latency for the L3 cache is 19 cycles for a shared cache and 14 cycles for a smaller private cache. A hit in a neighboring cache is assumed to take 19 cycles. The increase from 14 to 19 cycles is caused by extra communication latency and the time to handle the cache miss in the local cache. We assume serial last-level caches where tag and data lookup are separated. The latency introduced by the cache itself is lower for a miss than for a hit since data lookup is not performed for misses. The numbers are based on recent processors from AMD and Intel, other papers [3, 16] and CACTI 4.0[15]. These numbers do not only depend on the architecture but also on the implementation. Therefore we only show relative performance in the evaluation section.

All of the *SPEC2000* benchmark applications were used with the reference data sets except for two: The simulator had compatibility problems with *vortex* and *sixtrack*, and they are not included in the experiments. We create multiprogrammed workloads for our CMP architecture as follows. In each experiment, four randomly picked applications are run in parallel. Each application is randomly forwarded between 0.5 and 1.5 billion instructions and then we simulate two hundred million cycles.

We do not consider sharing of cache blocks in this paper as would be the case had we used parallel workloads. However we hypothesize that the new scheme will be effective also for such workloads and will address it in our future work.

4 Results

Several of the SPEC2000 applications have a small working set which more or less fits into the L1 and L2 cache. These applications are not sensitive to enhancements of the last-level (L3) cache and hence not relevant for the evaluation of the proposed scheme. Nevertheless, it is important that our proposed scheme is robust also for non-memory-insensitive applications. We compare the performance of the new scheme with a pure private cache organization because the performance of such an organization is quite predictable and well understood. We first classify the applications with respect to their sensitivity to last-level cache performance. We then consider the speedup of our scheme, the effects of larger caches and technology scaling and show the results of reducing the number of shadow tags. Finally we compare the performance of our scheme to an earlier proposed NUCA scheme.

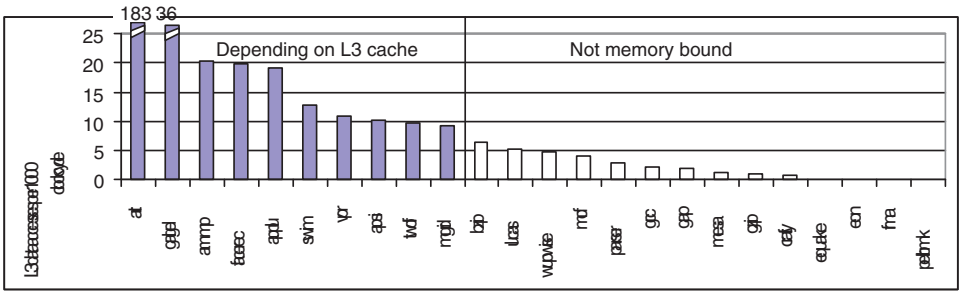

Figure 5. Classification of applications based on number of misses in L2 data cache.

4.1 Classification of Workloads

The number of last-level data cache accesses is shown in Figure 5 for different SPEC2000 applications. We classify the applications as (a) either being last-level cache intensive or (b) not depending on the last-level cache. The applications with more than nine last-level data cache accesses per thousand clock cycles are classified as last-level cache intensive. The rationale behind this is that there could potentially be a last-level cache miss every hundred clock cycles which add another few hundred cycles to the execution time. The number of last-level cache misses per clock cycle could have been used to classify the applications as well, but then the applications that work well for a conventional LRU scheme would not have been included in the evaluation of the proposed scheme. If the new scheme degraded performance for these applications it might not have been detected.

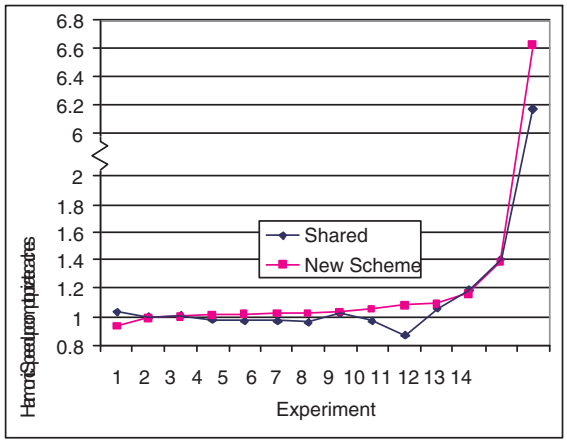

Figure 6. Harmonic mean of IPC for the last-level cache intensive benchmarks per experiment.

4.2 Speedup of the Proposed Scheme

The harmonic mean of the instructions per clock cycle (IPC) for all four cores per experiment is shown in Figure 6 for the last-level cache-intensive applications. Each experiment consists of four randomly picked applications run in parallel as described in the methodology section. The experiments are sorted by the performance of the new scheme relative to a private cache organization with the experiments with the highest speedup to the right. Except from a single experiment, the new scheme has equal or higher performance than both the private cache and shared cache schemes. The shared cache does a good job of speeding up the performance for the last-level cache intensive applications which is why the new scheme only has 2% higher harmonic speedup while the average speedup is 5%. Compared to private caches the harmonic mean of the new scheme is 21% higher while the average speedup is 13% higher.

The performance of the proposed scheme is shown for each last-level cache intensive application in Figure 7. The performance is compared against that of private caches, shared caches and private caches where each cache is the size of the shared cache (4 x size private). By considering the speedup with the 4 x larger private cache we see which applications that could benefit from larger caches. These are *ammp*, *art*, *twolf* and *vpr*. The proposed scheme works well for these four applications while a shared cache degrades the performance for two of them compared to private caches.

The new scheme degrades performance for some of the applications. This is expected since these applications are not given the same amount of cache space as with the private and shared schemes because some other application are believed to benefit more from using the cache space.

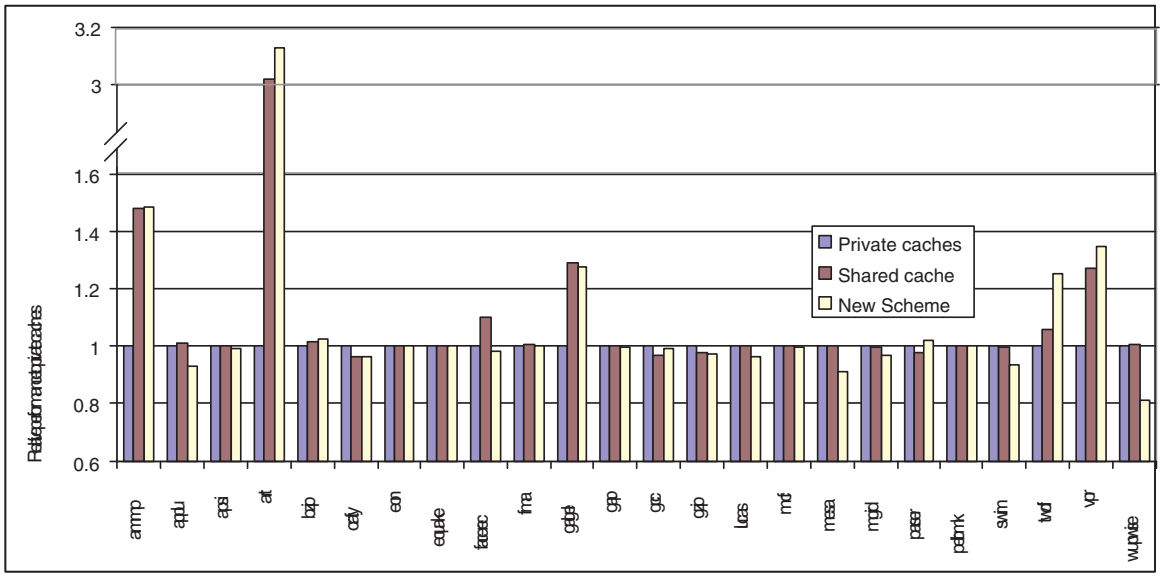

Figure 8. Speedup for all benchmarks.

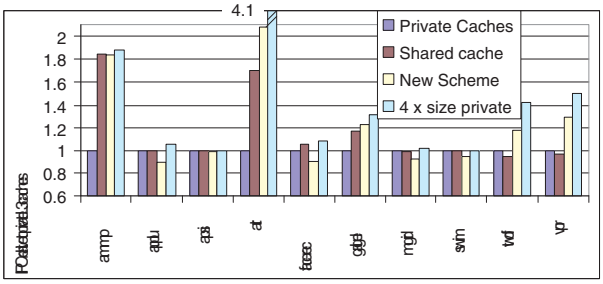

Figure 7. Speedup for the last-level cache intensive benchmarks.

4.3 Speedup for All SPEC2000 Benchmarks

The speedup per application compared to private caches for all SPEC2000 applications is shown in Figure 8 with both application categories; last-level cache intensive and last-level cache non-intensive applications. The poor performance of *wupwise* is caused by an experiment that contains three instances of *ammp* and a single instance of *wupwise*. *ammp* can utilize the cache space better than *wupwise* and therefore the performance is degraded to speedup the slow *ammp*. The actual numbers for the IPC for this case is for the proposed scheme: *wupwise*: 1.326 and *ammp*: 0.0323, 0.0322 and 0.0319, and for the other schemes *wupwise*: 1.7974 and for *ammp*: 0.0319 x 3. Looking at the harmonic mean, the new scheme improves the performance (although very little) while the other scheme degrades the performance (also very little). Even though the performance of *wupwise* is degraded, the new scheme makes the correct decision since the goal is to increase the harmonic mean.

4.4 Effects of Larger Caches

The SPEC2000 benchmark applications do not require very large caches. To illustrate this, Figure 9 shows the performance for an 8-MByte L3 cache. Most of the applications do not run faster with the larger cache as seen by the performance of the four times larger (4 x 8 MByte) private cache. For a simple comparison, the timing model is the same as used with the 4-MByte cache. The proposed scheme actually degrades the performance for many applications. This is because the proposed scheme infers constraints in a system that really does not need any restrictions because the cache size is so large compared to the requirement.

4.5 Impact of Technology Scaling

As technology gets denser in the future, the clock cycle will become slightly shorter while the communication latency is expected to stay the same. We have run experiments to find the impact of future technology scaling. The cycle time is assumed to be reduced by 30% for the core, which causes the number of cycles to be increased for the different caches since much of

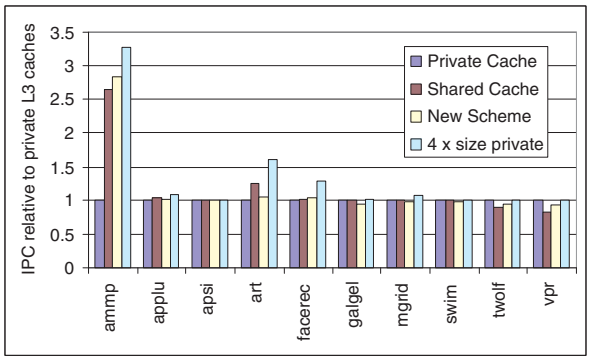

Figure 9. 8 MByte l3 cache.

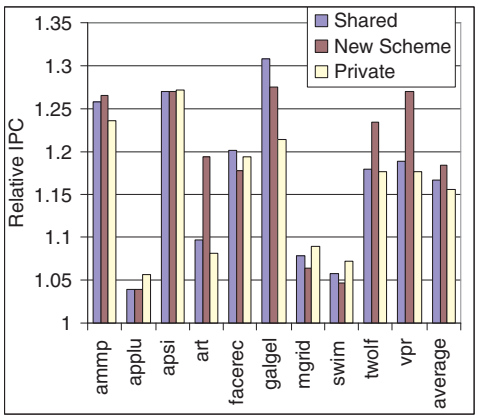

Figure 10. The impact of scaling the technology.

the cache cycle time is due to communication latency: The latency of the L2 cache is increased from 9 to 11 clock cycles, the latency of the L3 private/shared cache is increased from 14/19 to 16/24 and the main memory access time is increased from 258/260 to 330/338 for pure private/shared. The increase in clock cycles is based on that the cache partitions that are close to the cores have less increase than the last-level cache since a shorter communication distance is involved. As the technology scales, the access to main memory becomes slower and the shared cache becomes slower. The results are shown in Figure 10. On average (shown to the right in the graph) the new scheme has the highest gain. This is because the new scheme removes most memory accesses to main memory which becomes increasingly important.

4.6 Reducing the Number of Shadow Tags

The experiments in the previous sections were done with four shadow tags for every set to predict marginal gains of increasing cache size for the cores. However, it is not necessary to implement shadow tags in all sets. Our previous work has revealed that monitoring the sets with the lowest index works well and better than randomly generated subsets or subsets based on prime numbers [7]. We have run simulations with shadow tags in only 1/16 of the sets with the lowest index and found that the average IPC was increased with 0.1% while the harmonic IPC was decreased by 0.1%. This means that the tags with the lowest index represent the whole cache very well for the new scheme. LRU hits are counted for in all sets, but the numbers are normalized when compared to shadow tag hits. The cost of monitoring only $1/16 \approx 6\%$ of the sets is not very high and is discussed in Section 2.7.

4.7 Comparison with another NUCA Scheme

We implemented a hybrid scheme based on Chang and Sohi's work [3]. Their scheme is based on a chip multiprocessor with private caches. However, when a cache block is evicted from a private cache it might get installed in the neighboring cache. To illustrate the scheme through an example, consider a core *a* that has cache *a* and another core *b* that has cache *b*. When core *a* loads new data into its private cache, one block is evicted to make space for the new data. If this block was loaded by the core that owns the private cache (*a*), and it is evicted due to an access by the same core (*a*), this block is installed into a random neighboring cache as the most recently used (MRU). In this case this is cache *b* since there are only two caches. If the block that was evicted from cache *a* belonged to cache *b*, it must earlier have been evicted from cache *b*, and therefore it is not allocated again. When spilling a cache block from cache *a* into cache *b*, a block from cache *b* has to be evicted as well. This block is not allocated in some other caches to avoid ripple effects.

We call this scheme for "random replacement" and compare it against the proposed scheme for memory-intensive applications in Figure 11. The proposed scheme in general works better than the random-replacement scheme. This is not surprising as the random-replacement scheme works best when not all cores are competing for the cache resources. Figure 12 shows an experiment with both benchmark categories. In this case the proposed scheme is not that superior

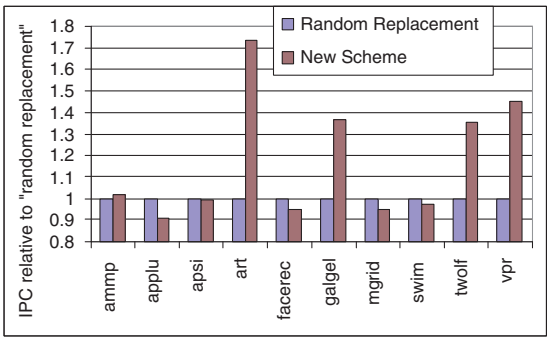

Figure 11. Performance of the new scheme relative to "random replacement".

compared to the random replacement scheme. This is due to the many applications that do not use the last-level cache space.

5 Related Work

Recently there has been a significant interest on NUCA schemes for chip multiprocessors with papers addressing the sharing of data, migration of cache blocks and reducing the hit time [2, 3, 6, 9, 16]. Important aspects of these works are cache coherence protocols and duplication of shared cache blocks to reduce access latencies for several cores. Our work is concerned with optimizing the cache usage per core and is complementary to these works.

Even though this is the first paper to consider adaptive cache partitioning for NUCAs, several of the components used are a combination or extension of earlier described methods. We acknowledge this prior art next. Suh et al. described a way of partitioning a cache for multithreaded systems by estimating the best partition sizes [13, 14]. They counted the hits in the LRU position of the cache to predict the number of extra misses that would occur if the cache size was decreased. A heuristic used this number combined with the number of hits in the second LRU position to estimate the number of cache misses that are avoided if the cache size is increased. We extended this scheme with shadow tags and counters for estimating the number of hits that would be avoided by increasing the cache size[7]. The method presented in [7] increases the precision of the predictions.

In this paper we use the same mechanism, but apply it to a NUCA organization to control the size of the private partitions. The previous work considered adjusting the size of the cache partitions within a shared cache while we in this paper adjust the shared partition size of the local last-level caches as well as controlling the number of blocks per core in the shared partition. The previous works did not consider a shared partition with variable size, nor did they look at combining private and shared caches. Evaluation of cache partitioning in shared chip multiprocessor caches has also been studied earlier by Kim et al. [10] for a two-core CMP where a trial and fail algorithm was applied. Trial and fail as a partitioning method does not scale well with increasing number of cores since the solution space grows fast.

Spilling evicted cache blocks to a neighboring cache was described by [3, 6]. They did not consider putting constraints on the sharing or methods for protection from pollution. No mechanism was described for optimizing partition sizes. We extend their work by insertion of the constraints on cache usage, divide the cache space in private and shared partitions and mechanisms for finding the best partition sizes. As our results show, the extensions improve performance significantly.

A mechanism for protecting cache blocks within a set was described by Chiou et al. [4]. Their proposal was to control which blocks that can be replaced in a set by software in order to reduce conflicts and pollution. The scheme was intended for a multi-threaded core with a single cache. We use the same mechanism to divide the blocks within a set into two partitions, a private partition and a shared partition. We then combine the shared partitions from several caches into a single shared cache space.

Qureshi et al. independently developed a mechanism very similar to shadow tags [11]. The only significant difference is in the selection criteria for the tags and they have included a formal proof of the concept based on the assumption that all sets affect performance equally.

6 Conclusion

The performance of multi-core systems relies on an effective cache system. We have considered improving the cache system by adapting the cache usage per core to its needs by protecting its most recently used data in the last-level cache.

Simulations show that the new scheme has higher performance in terms of instructions per clock cycle than private cache organizations because of a higher utilization and potentially larger caches for applications that benefit from an increased cache size. Compared to shared caches performance is in general increased due to (a) lower access time and (b) improved sharing. Hits

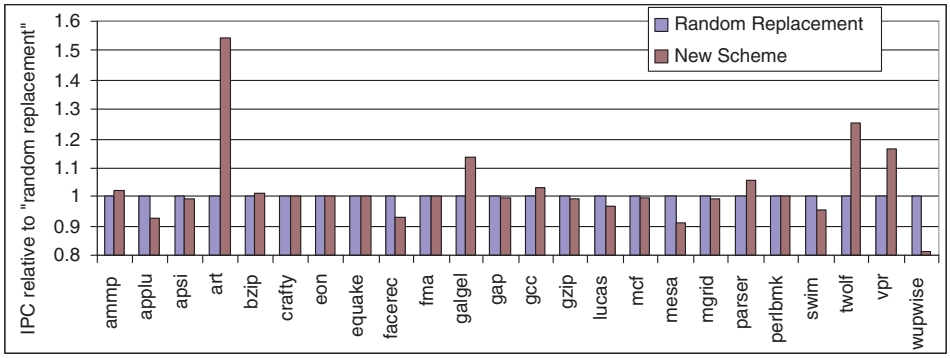

Figure 12. Performance of the new scheme relative to "random replacement".

to the local cache are faster than in a shared cache due to its smaller size and lower associativity. Each core is protected from pollution by the other cores and is given a cache size which minimizes the total number of cache misses for all cores. Earlier NUCA schemes do not consider partitioning and protection of cache blocks in the last-level cache. As technology scales and latency becomes more dominant, the proposed scheme is predicted to be even more advantageous because it reduces the number of cache misses which will be an increasing bottleneck. Even though we have only simulated a four-core processor, we believe the scheme will scale to systems with a higher processor count. Additionally, the proposed scheme only requires modest hardware for implementation.

References

[1] T. Austin, E. Larson, and D. Ernst. SimpleScalar: An infrastructure for computer system modeling. IEEE Comp., V35I2, 2002.

[2] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. MICRO 37, 2004.

[3] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. ISCA, 2006.

[4] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. Proceedings of Design Automation Conference, Los Angeles, 2000.

[5] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. MICRO 36, 2003.

[6] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. ISCA, 2005.

[7] H. Dybdahl, P. Stenström, and L. Natvig. A cache-partitioning aware replacement policy for chip multiprocessors. HiPC, 2006.

[8] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler. A NUCA substrate for flexible CMP cache sharing. ICS, 2005.

[9] C. Kim, D. Burger, and S. W. Keckler. Nonuniform cache architectures for wire-delay dominated on-chip caches. IEEE Micro, vol. 23, no. 6, pp. 99-107, 2003.

[10] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning on a chip multiprocessor architecture. Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT), 2004.

[11] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A case for mlp-aware cache replacement. ISCA, 2006.

[12] J. E. Smith. Characterizing computer performance with a single number. *Communications of the ACM*, 31(10), 1988.

[13] G. Suh, S. Devadas, and L. Rudolph. Dynamic cache partitioning for simultaneous multithreading systems. IASTED Int. Conf. on Parallel and Distributed Computing Systems, 2001.

[14] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. HPCA, 2002.

[15] D. Tarjan, S. Thoziyoor, and N. P. Jouppi. Cacti 4.0. Technical report HP Laboratories Palo Alto, HPL-2006-86, 2006.

[16] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. ISCA, 2005.