MorphCache: A Reconfigurable Adaptive Multi-level Cache Hierarchy*

Shekhar Srikantaiah, Emre Kultursay, Tao Zhang, Mahmut Kandemir, Mary Jane Irwin, Yuan Xie
The Pennsylvania State University, University Park, PA - 16802
{srikanta, euk139, tzz106, kandemir, mji, yuanxie}@cse.psu.edu

Abstract

Given the diverse range of application characteristics that chip multiprocessors (CMPs) need to cater to, a "one-cache-topology-fits-all" design philosophy will clearly be inadequate. In this paper, we propose MorphCache, a Reconfigurable Adaptive Multi-level Cache hierarchy. MorphCache dynamically tunes a multi-level cache topology in a CMP to allow significantly different cache topologies to exist on the same architecture. Starting from per-core L2 and L3 cache slices as the basic design point, MorphCache alters the cache topology dynamically by merging or splitting cache slices and modifying the accessibility of different cache slice groups to different cores in a CMP. We evaluated MorphCache on a 16 core CMP on a full system simulator and found that it significantly improves both average throughput and harmonic mean of speedups of diverse multithreaded and multiprogrammed workloads. Specifically, our results show that MorphCache improves throughput of the multiprogrammed mixes by 29.9% over a topology with all-shared L2 and L3 caches and 27.9% over a topology with per core private L2 cache and shared L3 cache. In addition, we also compared MorphCache to partitioning a single shared cache at each level using promotion/insertion pseudo-partitioning (PIPP) [28] and managing per-core private cache at each level using dynamic spill receive caches (DSR) [18]. We found that MorphCache improves average throughput by 6.6% over PIPP and by 5.7% over DSR when applied to both L2 and L3 caches.

1 Introduction

Increasing number of cores on a chip, increasing diversity of workloads, and different degrees of data sharing within these workloads present unique challenges to the design of cache hierarchies in CMPs. The challenges include the organization and the policies associated with the cache hierarchy to meet performance and scalability goals [1]. Cache organization deals with the number of levels in the cache hierarchy and the size, associativity, latency, and bandwidth parameters at each level. Cache policies determine accessibility, allocation, and eviction policies to effectively utilize on-chip cache resources. The objective in designing a good cache hierarchy is to minimize the latency to frequently accessed data. At the same time, it is also important to design the cache in such a way that it optimizes the performance across a diverse range of applications.

1.1 Background

There is a general consensus in the community about the L1 cache in a CMP being private to each core in order to reduce the access latency of most frequently accessed data. However, there has been extensive research in the recent years about the organization of the L2 cache. A private L2 cache helps in reducing access latency and cache design complexity, whereas a shared L2 cache eliminates data replication, thereby increasing effective cache capacity and improving system performance. A spectrum of organizations ranging from private [4, 3, 18] to shared [27, 12, 9, 21, 20, 19, 10], including several intermediate organizations [15, 5, 26], have been proposed. Almost all are concerned about managing only the *last level of cache* (typically, the L2). The problem of managing the cache organization at a single level is simplified by the fact that it has to deal only with capacity management as opposed to managing a multi-level topology. The deepening of the cache hierarchy in recent architectures that introduces an L3 cache as the last level of cache brings out an entirely new dimension of this problem.

Consider the cache hierarchies of two of the latest Intel architectures: Dunnington [7] and Nehalem [7], shown in Figure 1. These two architectures have distinctly different cache topologies, with Dunnington having each L2 cache slice shared among a pair of cores, whereas in Nehalem, each L2 is private to a core. Given that there are six cores in a socket in Dunnington, we have three L2 caches connected to each shared L3 cache, whereas in Nehalem, four L2 caches share an L3 cache. In general, the topology of the cache hierarchy decides the capacity, associativity and access latency of each slice of cache and the degree of sharing at each level given a fixed area budget for the cache subsystem. These factors influence the system performance to a significant extent. Given the diverse range of application characteristics that CMPs needs to cater to, a "one-cache-

*This research is supported in part by NSF grants #1017882, #0963839, #0905365, #0903432, #0702617, CNS #0720645, CCF #0811687, and CCF #0702519, SRC grants, and a grant from Microsoft Corporation.

978-1-4244-9434-7/11/$26.00 ©2011 IEEE

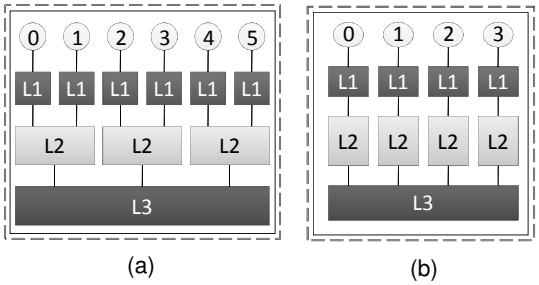


Figure 1. Multicore cache architectures: (a) Dunnington (b) Nehalem. In each figure, the circles denote processors (cores) and L1, L2 and L3 represent on-chip caches.

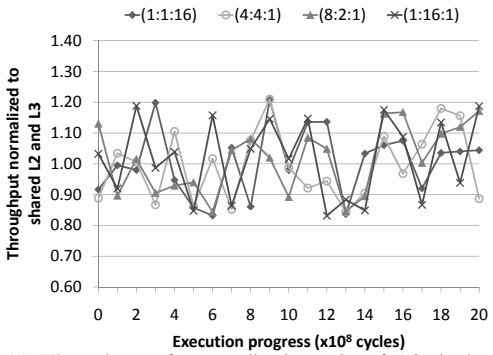
topology-fits-all" design philosophy will clearly be inadequate. Therefore, we believe that the design of the on-chip cache topology in CMPs requires a fresh look.

1.2 Motivation

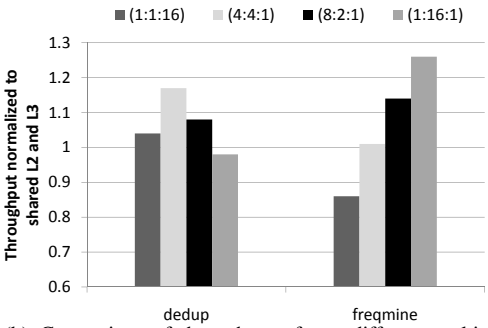
Consider the plot in Figure 2(a) showing the throughput (sum of instructions per cycle) of a mix (workload) of 16 single-threaded applications (application mixes are detailed in Section 4) from the SPEC CPU 2006 benchmark suite as execution progresses over 2 billion cycles (20 intervals of 100 million cycles of the region of interest in a warmed up cache). Each curve in the graph represents a different cache topology in a 16-core CMP with 16 slices of both L2 and L3 caches (see Section 4 for detailed configuration parameters). In all cases, there is a one-to-one mapping between applications and cores. A configuration $(x : y : z)$ represents a cache topology for a 16 core CMP, where each L2 slice is shared by x cores (and their private L1 caches), each L3 slice is shared by y L2 slices, and there are z L3 cache slices in the hierarchy. For example, (4:4:1) stands for a cache topology where 4 cores share each slice of L2 cache, 4 slices of L2 share each L3 and there is 1 slice of L3 in the system. By this definition, (1:1:16) represents a topology where each of the 16 cores has a private L2 slice and a private L3 slice, and (16:1:1) represents a topology where all the 16 cores share a single L2 and a single L3 slice. We can see from Figure 2(a) that the best configuration (one with the highest throughput) varies with time during the execution of the workload. Figure 2(b), on the other hand, plots the throughput obtained by two different *multithreaded* applications (dedup and freqmine from the PARSEC benchmark suite), when run separately with 16 threads each on a 16-core CMP with different cache topologies. We can see that, while dedup achieves its highest throughput with the (4:4:1) topology, freqmine achieves its best throughput with the (1:16:1) topology. Clearly, any one topology does not serve best for both. These behaviors can be best explained in terms of the variations in the *active cache footprints* (ACF) of the different threads over different epochs and the degree of data sharing between the threads of the application. We define the ACF of a thread in a time epoch as the set of unique cache lines referenced by the thread in that epoch (i.e., ACF is the working set of the thread in the given time epoch). The best cache topology for a multithreaded application can vary with time due to one or more of the following reasons: (i) variations in ACFs of different threads at a given level of cache; (ii) variations in data sharing among the ACFs of the different threads; and (iii) variations in ACFs at different levels of the cache within the same thread. The main drawback of a fixed cache topology (in both multiprogrammed (Figure 2(a)) and multithreaded (Figure 2(b)) workloads) is that it statically determines both the capacity sharing and accessibility to different cache slices at different cache levels by different cores. Therefore, if a particular multiprogrammed workload of single-threaded applications or a set of threads need the caches at certain levels to be shared differently to improve system performance, this cannot be achieved using a fixed (static) topology. The main objective of this paper is to realize a practical and low-overhead, dynamically reconfigurable cache topology to remedy this problem.

1.3 Contributions

In this paper, we propose a reconfigurable adaptive multilevel cache hierarchy called *MorphCache*. MorphCache dynamically tunes a multi-level cache hierarchy in a mid-sized ($\leq$16 cores) CMP that realizes the benefits of significantly different cache topologies on the same architecture. Starting from per-core L2 and L3 cache *slices* as the basic design point, MorphCache alters the cache topology dynamically by merging or splitting cache slices and modifying the accessibility of different cache slice groups to different cores. Therefore, *it can reconfigure an on-chip cache hierarchy in both horizontal and vertical directions*. MorphCache uses ACF estimation to gauge the benefits of merging/splitting cache slices at multiple levels. The cache footprint estimation process results in valuable data about active cache utilizations of multiple cores as well as the degree of constructive/destructive interferences among cores in multiple levels of the cache topology. MorphCache then uses this information to dynamically reconfigure the cache topology to improve overall system performance. The key insight in this paper as compared to previous works on capacity management in last level caches [10, 9, 15, 4, 5, 21, 18] is that, in addition to cache partitioning (capacity management) at a single level, the cache topology has a significant impact on the system performance when the cache subsystem constitutes multiple levels of caches that can potentially be shared among multiple cores. While the idea of reconfigurable caches itself is not new [23], a reconfigurable multilevel cache topology for CMPs has been relatively less explored [2, 29]. The insight here is that no single, statically determined cache topology is best from performance perspective due to dynamically varying active working set sizes of workloads and data sharing among threads. Further, ACFs of threads/applications at different levels of cache can be used as effective indicators of the best cache topology for the workload and an efficient hardware mechanism necessary to realize this objective will be presented.



(a) Throughput of an application mix of 16 single-threaded applications (Mix_01; see Section 4) over 2 billion cycles with different cache topologies.



(b) Comparison of throughput of two different multi-threaded applications on different cache topologies when executing on our default 16 core CMP (see Section 4 for details).

Figure 2. Throughput in each case is normalized to that on shared L2 and L3 topology. A configuration $(x : y : z)$ represents a cache topology for a 16 core CMP, where each L2 slice is shared by x cores, each L3 slice is shared by y L2 slices and there are z L3 cache slices in the hierarchy. The base case can then be denoted as (16:1:1).

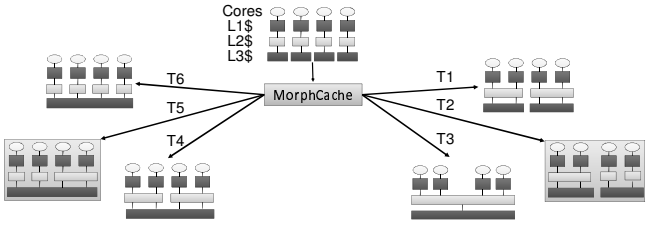


Figure 3. The essence of MorphCache is to dynamically adapt the cache topology to application/workload requirements by reconfiguring itself. Two asymmetric configurations are highlighted.

We evaluated MorphCache on a 16 core CMP on a full system simulator. We used a diverse set of multithreaded and multiprogrammed workloads comprising benchmarks from the PARSEC and SPEC CPU 2006 benchmark suites, respectively, in our evaluations, and compared MorphCache with several common commercial and hypothetical static topologies. Our results show that, on average, MorphCache improves throughput of the multiprogrammed mixes by 29.9% over a topology with all-shared L2 and L3 caches and 27.9% over a topology with per core private L2 and all-shared L3. The corresponding improvements in performance for multithreaded applications are 25.6%, and 8.5%, respectively. In addition, we also compared MorphCache to partitioning a single shared cache at each level using promotion/insertion pseudo-partitioning (PIPP) [28] and managing per-core private caches at each level using dynamic spill receive caches (DSR) [18]. We found that MorphCache improves average throughput by 6.6% over PIPP and by 5.7% over DSR when applied to both L2 and L3 caches.

2 Design of MorphCache

Figure 3 depicts a MorphCache for a CMP with four cores. Each core in MorphCache has a private L1 cache and has access to closely located L2 and L3 cache slices. MorphCache further attempts to adaptively reconfigure the cache hierarchy in such a way that it resembles the best performing configuration for the current workload. Note that all proposed cache topologies in current CMPs are symmetric in the sense that a fixed number of cache slices of L2 share an L3 cache. However, MorphCache can also morph itself into an *asymmetric topology*, as highlighted in Figure 3, if it is deemed to be the best configuration. Our goal is to determine the ideal on-chip cache hierarchy at any given point in execution. As explained later, in our approach changes to the cache hierarchy are made across epoch boundaries. At any point in time, a 16 core MorphCache can be reconfigured in such a way that the slices of cache (L2/L3) can be in one of the following modes: (i) private; (ii) dual-shared; (iii) quad-shared; (iv) oct-shared; or (v) all-shared. MorphCache manages the merging/splitting of cache slices based on an estimation of active working set sizes of different threads/applications. We first describe a simple, low-overhead architectural mechanism for estimating the ACFs of different threads/application executing on a CMP. Then, we discuss different policies for merging/splitting cache slices based on the information acquired by our ACF estimation.

2.1 Active Cache Footprint Estimation

Unlike processor and disk utilizations, cache utilization cannot be used directly as an effective metric to characterize cache requirements of different workloads as caches tend to be fully utilized. Nevertheless, the mere presence of a cache block referenced by an application in a cache does not guarantee active usage (reuse) of that data by the application. Further, most commonly used metrics like cache miss rates, average access latencies or throughput (instructions per cycle) cannot be used as direct indicators of the degree of capacity utilization or data sharing among threads/applications. Therefore, we define the *Active Cache Footprint* (ACF) of a thread in a time epoch as the set of unique cache lines referenced by that thread in that epoch. Note that this metric is different from cache utilization in that it does not count the cache lines that may have been brought into the cache in the

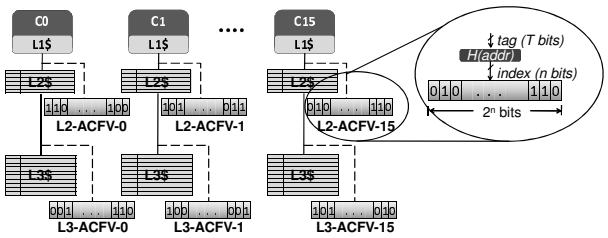


Figure 4. Active cache footprint estimation using active cache footprint vectors (ACFVs). There is an ACFV per-core, per cache slice.

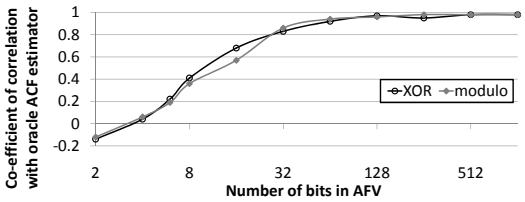


Figure 5. Correlation coefficient between estimated ACF using ACFVs of different lengths and an oracle ACF estimator for a 1 MB L2 cache slice for hmmer benchmark.

previous epochs.

We have developed a simple, novel hardware mechanism called *Active Cache Footprint Vectors* (ACFVs) for estimating the ACF of threads. ACFVs are small bit-vectors used to represent the ACF in a cache slice. Figure 4 depicts the organization of ACFVs. Whenever an eviction occurs, the tag of the new data is hashed using an efficient hardware hash function to index into the ACFV corresponding to the cache slice. The corresponding ACFV bit is set to 1. At the same time, the tag of the data being replaced is also hashed using the same hash function and the corresponding bit in the ACFV is set to 0. Therefore, at any given point in time, the contents of the ACFV are representative of the ACF of the thread. We observed that some bits in the ACFV are set due to stale data that was brought into the cache slice and was not reused for a long time. Such stale data continues to add to the estimated cache footprint. To avoid this, we reset all the bits in the ACFV once in an interval of making a cache slice merge or split decision. This ensures that the ACFVs represent the data that is actively being used.

The ACFVs have two important properties that make them useful for designing a reconfigurable cache topology like MorphCache: (i) the number of 1's in the ACFV (denoted as $|ACFV|$) is representative of the active utilization (size of the ACF) of the cache slice, and (ii) the number of common 1's in two ACFVs corresponding to different cache slices running threads sharing the same address space serves as a measure of the degree of data sharing between the two threads.

These two properties prove to be extremely valuable in making decisions about merging/splitting cache slices in MorphCache as we will show in the following sections. We can obtain accurate information about ACF by using a one-to-one mapping of the cache lines to the bits in ACFV at the cost of additional hardware. However, we observed that even a small ACFV is sufficiently representative of the relative active utilization. Figure 5 shows the value of the *correlation coefficient* between the estimated ACF using different lengths of ACFVs and the corresponding actual ACF determined using an oracle (one-to-one mapping bit-vector). The two curves represent the correlation coefficients obtained for an *XOR hash function* and a *modulo hash function* for a 1MB cache L2 slice when executing the hmmer benchmark from the SPEC CPU 2006 benchmark suite. We observe that with a relatively small number of bits in the ACFV, we obtain a high correlation coefficient. For example, the correlation coefficient is 0.94 for 64 bits and 0.96 for 128 bits. We also carried out the same set of experiments for all other benchmarks and had similar observations. Although updating the ACFVs does not add to the critical path of the cache access, it is important to have an efficient implementation of the hash function. Various efficient hardware implementations of hashing have been described in [22].

There are several key advantages of estimating the ACF in designing a reconfigurable cache topology. First, a small ACF in a cache slice neighboring a cache slice having a large ACF serves as a good indicator of a need for capacity sharing among the two cache slices. Previous schemes like cooperative caching [4] and adaptive spill-receive cache [3] attempt to address such scenarios (albeit, at a single level). However, they incur significant overheads of spilling cache lines from one cache and receiving them in the other. MorphCache avoids these overheads by merging the cache slices to avoid such evictions. Second, a significant overlap between ACFs of two or more threads (sharing the same address space) serves as a good indicator of significant data sharing among threads. It would be beneficial to merge these cache slices to reduce the overheads of repeated transfers of cache lines among different cache slices. Replication based schemes try to work around this situation by replicating such cache lines in both caches. In turn, they suffer from capacity diminishing due to replication. MorphCache explicitly detects such scenarios and merges the cache slices to avoid both repeated transfers and the overheads of data replication.

2.2 Merging Cache Slices

Merging neighboring cache slices[1] is a fundamental activity in MorphCache that facilitates the cache topology changes. Merging two slices allows the cores that use the individual slices to share the entire cache capacity of those two slices. Note that, *MorphCache is initially configured as a private L2 and private L3 hierarchy*. There are two conditions under which MorphCache favors a merger of neighboring cache slices:

[1]When a pair of cache slices are merged, the total associativity of the new slice is the sum of the associativities of the two merged slices. For instance, merging two *n-way* slices of size *S* each results in a *2n-way* shared slice of size *2S*.

(i) One of the cache slices is highly-utilized, while the other is under-utilized: This leads to reduced benefits from the under-utilized cache slice. Therefore, capacity sharing by merging the slices to have a moderately utilized merged cache would improve performance.

(ii) Both the caches are highly-utilized, the slices are shared by threads sharing the same address space (as in multithreaded workloads), and there is a significant number of common 1's the ACFVs of the corresponding cache slices: This reflects the possibility of significant data sharing among the threads.In this scenario, the benefits in performance arise from improving the utilization of aggregate shared cache capacity due to reduction in the replication of data. Additionally, when the cache slices of threads sharing data are shared, it also helps to reduce the coherence overheads.

In order to determine when a cache slice is highly utilized or under-utilized, we use a threshold that we call the *Merge/Split Aggressiveness Threshold* (MSAT). The MSAT is a pair (h, l) that indicates the high and low utilization boundaries. Therefore, whenever the size of the ACF is greater than the higher bound of MSAT, i.e., $|ACFV| > h$, it implies that the cache is being highly utilized. Similarly, $|ACFV| < l$ implies that the cache is under-utilized. Through extensive experiments, we determined that an MSAT value of (60,30) provides a reasonable aggressiveness for merging/splitting. We later describe an automatic method for throttling this threshold in Section 5.3.

Merging two neighboring slices of the last level cache (L3) is always safe. However, merging two L2 slices when the corresponding L3 slices are split may lead to the L2 cache capacity being larger than that of the L3. This scenario would lead to problems in maintaining the inclusiveness property. While non-inclusion (as in some previous works like [4]) is an alternative to gain some additional performance, we use inclusive caches in our work to avoid the design complexity of the coherence protocols. To avoid this scenario, we check the L3 cache ACFVs if the L3 caches can also be merged (or are already merged) and decide to merge the L2 cache slices *only when* it is possible to merge the corresponding L3 slices as well. Another issue with the process of merging cache slices at runtime is that it may lead to duplication of data in the merged cache when the same data is present in both the cache slices being merged.To solve this problem and to maintain coherence, we can simply flush one of the caches to eliminate the duplicates. However, this may incur a performance penalty due to loss of locality for the other non-duplicated entries. To address this problem, we perform what we call *"lazy invalidation"* by deferring the invalidation of the duplicate data until it is first accessed. In lazy invalidation, whenever a duplicated data is accessed and there is a hit from multiple cache slices, only one of the copies is retained as valid and the other is invalidated. This effectively saves us the performance penalty of flushing one of the merged caches. Another concern that needs to be addressed while merging the caches is that of maintaining the LRU information. In an ideal LRU implementation, we can merge the entries according to time-stamps. However, many practical implementations of LRU are pseudo-LRU implementations [24]. For generalized tree-LRU based pseudo LRU algorithms, we can also merge the trees of the cache slices being merged in any order and the future accesses will quickly determine a new LRU sub-tree. Further, when two slices are merged, their ACFVs are maintained separately, but logically the two ACFVs are treated as one large ACFV obtained by juxtaposition of the two ACFVs. The fraction of 1s in the resultant large ACFV is used for computing the active utilization of the new merged slice.

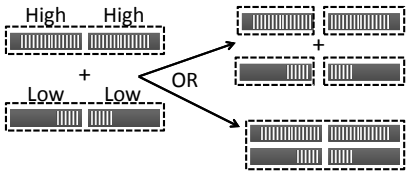


Figure 6. Split/merge conflicts.

2.3 Splitting Cache Slices

Splitting two merged cache slices may be called for when the ACFs of the caches change significantly during the course of execution and it is no longer justified to keep them in a merged mode. Note that ACF estimation using ACFVs continues on a per cache slice basis even when two or more cache slices have been merged. As a dual of the merging case, splitting L2 caches is always safe, while splitting L3 caches when the corresponding L2 caches are merged could lead to the L2 cache being larger than the corresponding L3 cache. Moreover, it may also lead to problems with maintaining the inclusion property. Therefore, similar to the merging case, we check if the corresponding L2 cache slices are already split or can be split, and decide to split the L3 cache *only if* the corresponding L2 caches can be split.

2.4 Policies for Split/Merge Conflicts

There are certain cases in which MorphCache needs to arbitrate among conflicting requirements of splitting and merging cache slices. The conflicts mentioned above, where merging of L2 caches and splitting the L3 cache require the other level cache(s) to perform the same operation, are issues of correctness as explained in the previous sections. However, there are other cases where such conflicts are not subject to correctness. For example, when two pairs of neighboring slices are in the dual-shared mode, the individual slices involved in the first pair are both highly utilized and both the slices in the second pair are under-utilized (as shown in Figure 6), MorphCache can make two possible decisions. It can split the first pair of slices as the condition to split them is satisfied. Or, it can decide to merge the two pairs of slices as they collectively indicate that the first pair is highly utilized while the second pair is under-utilized. In case of such a split/merge conflict, MorphCache, by default, favors

a merge in a merge aggressive policy. In Section 5, we also report a comparison between the default merge aggressive policy and an alternate split aggressive policy.

We found in our experiments that the number of reconfigurations (merges and splits) performed by our scheme in multiprogrammed workloads varied between 5,248 and 12,176, averaging on 9,654 throughout the execution of the workloads. The corresponding numbers with the multithreaded workloads were 263 (minimum), 1043 (maximum), and 856 (average). We also found that, in about 39% and 54% of the cases we performed a merge or split, the resulting configuration was an asymmetric one, in multiprogrammed and multithreaded workloads, respectively. These results indicate that our scheme exercised asymmetric configurations with a relatively high frequency.

3 Hardware Support

3.1 MorphCache Interconnect

At the heart of MorphCache lies its ability to be dynamically reconfigured to obtain different cache configurations. The interconnect cannot introduce high extra penalties in cache access latencies while supporting this dynamic reconfigurability. Typical on-chip interconnects are shared buses, crossbar switches [11, 14], and interconnection networks. A shared bus fabric is the simplest solution to an on-chip interconnect. In current CMPs [7], buses are mainly used for communication between the last level cache and the main memory. Since buses allow only one transaction to be active at a time, they provide limited bandwidth. Furthermore, when multiple devices that are logically isolated are connected to a single shared bus, each gets only a fraction of the available bandwidth. Crossbars and other types of interconnection networks provide higher bandwidth [13] by supporting multiple simultaneous connections. However, they are relatively more complex and difficult to implement, let alone dynamically partition.

In MorphCache, a segmented bus [8] is employed as the interconnect structure. A segmented bus is a shared bus that is composed of multiple small buses called segments. Adjacent segments can be dynamically connected/isolated by enabling/disabling intermediate switches on the bus. The left hand side of Figure 7 shows a segmented bus with 8 segments, 8 components ($C0, \ldots, C7$) connected to these segments, and 7 switches ($S0, \ldots, S7$) that are located between the segments. The configuration at the bottom of the figure shows the segmented bus configured in a (4,2,2) formation obtained by enabling all switches except $S3$ and $S5$. A segmented bus has the ability to isolate segments of the bus, and as a result, provides support for multiple parallel bus transactions. This scheme not only increases the effective bandwidth of the bus, but also improves power efficiency due to the decreased switched capacitance. Figure 8 shows the placement of the two segmented buses in our system, one that connects the L2 cache slices and another that connects the L3 cache slices. These buses are on the memory side of the L2 and L3 caches, respectively, and when there is cache sharing, a miss on a local L2 (or L3) cache is put on the segmented bus to be delivered to all shared L2 (or L3) caches.

3.2 Segmented Bus Arbitration

Compared to a traditional shared bus, arbitration in a segmented bus is a little more complex. In MorphCache, arbitration is performed hierarchically by connecting multiple arbiters as a tree structure. Figure 9 shows this arbiter tree structure connected to 8 cache slices. In this figure, there are three levels of identical arbiters, with pin connections as shown in Figure 10. Each arbiter has 2 request inputs coming from the lower level and a request output going to the higher level. An arbiter at level n generates 2 grant signals, each one being delivered to 2^{n-1} cache slices that might have generated the corresponding request. For instance, the two grant signals generated by the level-3 arbiter in Figure 9 are connected to the cache slices $C0, \ldots, C3$ and $C4, \ldots, C7$, respectively. A cache slice has $log(n)$ grant inputs and gets access to the bus only when it receives grant signals from all the arbiters that it is connected to. This operation is realized using the circuit in Figure 11. The bus-acquire signal ($BusAcq$) is generated at the cache slice to enable bus usage when all arbiters that are configured to receive requests from this cache slice grant bus access to this slice.

The internal structure of an arbiter is also shown in Figure 10. It has five input ports, including two request signals (Req_0 and Req_1), and three output ports (Gnt_0, Gnt_1, and Req_{out}). When a request arrives at the arbiter, it is first latched by the corresponding D flip-flop. Then, triggered by this request, the arbitration logic generates grant signals based on a Round-Robin policy. The $Lastgnt$ register is used to record which request was granted the last time. In addition, to support hierarchical arbitration, the arbiter needs to forward the incoming requests to the arbiter at the next level. The input signal, $Fwdreq$, indicates whether this arbiter should forward the request to the next level arbiter and is a function of the sharing degree of the cache.

Figure 12 depicts the floorplan of a 16 core CMP with MorphCache. Each core has a slice of L2 and L3 cache located physically close to it and there are L2 and L3 cache arbiters between the caches. Our hierarchical arbitration design was implemented in Verilog HDL and synthesized using the Synopsys Design Compiler. Fundamental synthesis parameters are shown in Table 1 and the results of synthesis and timing simulation are presented in Table 2. The total area requirement for the arbiters are: (i) $2 \times 160.5um^2$ for the L2 arbiters on the two sides of the chip and (ii) $343.9um^2$ for the L3 arbiters across the chip. The wire delay values are calculated using the farthest distance between any two arbiters in this floorplan and the wire delay parameter in Table 1. The largest delay value in this table is $0.89ns$ which corresponds to the case where a request signal travels from

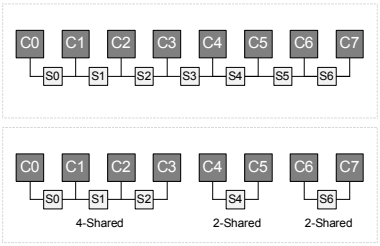


Figure 7. Segmented bus configurations.

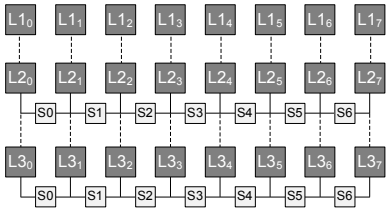


Figure 8. Three level cache hierarchy with 8 slices at each level.

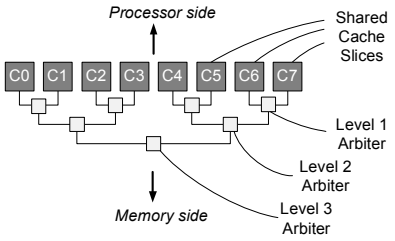


Figure 9. 3-level arbiter hierarchy comprised of 2-input arbiters.

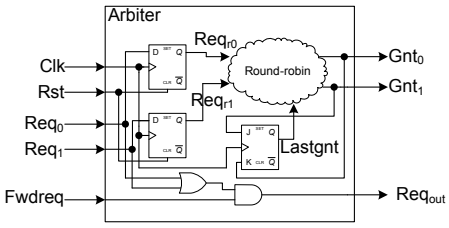


Figure 10. Segmented bus arbiter.

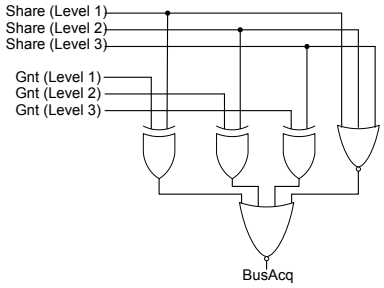


Figure 11. $BusAcq$ signal generation.

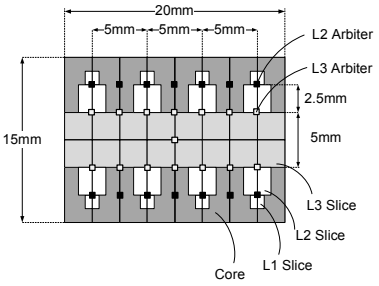


Figure 12. Floorplan.

one end of the chip to the other end, passing through 4 levels of arbiters. This sets a maximum frequency of 1.12 GHz for the arbiter. Acting conservatively, we assumed that the arbiters and the segmented bus operate at 1 GHz. Note that, the bus frequency is not limited to 1GHz. This number is obtained under the assumption that each of the bus requests, bus grants, and data transfer operations take one cycle, and they are not pipelined. Thus, a bus transaction takes 3 cycles (at 1GHz). If each operation is pipelined, a higher bus frequency can be obtained. Overlapping these operations can further hide 1 cycle data transfer latency. In the bus arbitration mechanism, the only change with respect to a fully-shared bus with hierarchical arbitration is that the arbiter inputs are compared with the Share signals (Figure 11).

A bus transaction consists of three parts, namely, request, grant, and data transfer. It takes 2 cycles between the generation of a request signal and the delivery of the corresponding grant signal. Note that, all arbiters receive their request signals and generate their grant signals simultaneously. After the arbitration process, the cache slice that owns the bus can initiate data transfer. As opposed to the transfer of the request signal, the data placed on the bus is directly delivered to the shared cache slices without going through any arbiters and suffers only wire delay. Therefore, the data transfer delay is equal to the wire delay suffered in arbitration, which is 0.4ns in the worst case and easily fits in 1 bus cycle. Assuming a bus width of 64 bytes, the whole cache block can be transferred on the segmented bus in 1 cycle. Adding 2 cycles for arbitration and 1 cycle for data transfer gives a total of 3 cycles overhead for the bus transaction (for a bus frequency of 1 GHz). For a processor running at 5 GHz, this translates into 15 cycles of overhead due to the segmented bus.[2]

Parameter	Value
Technology	45nm (Synopsys)
Wire Delay	0.038ns/mm (Cacti 6.5)
V_{CC}	1.05V

Table 1. Parameters for arbiter hierarchy synthesis.

	L2 Segmented Bus (3-level)	L3 Segmented Bus (4-level)
No. of Arbiters	7 per side of the chip	15
Total Arbiter Area	$160.5um^2$	$343.9um^2$
Total Request Delay	0.31ns (wire) + 0.38ns (logic)	0.4ns (wire) + 0.49ns (logic)
Total Grant Delay	0.32ns (logic) + 0.31ns (wire)	0.32ns (logic) + 0.4ns (wire)

Table 2. Segmented bus arbiter area and delay results.

[2]Since the segmented bus is synchronous, the duration of a cache block transfer is fixed and is exactly 5 cycles. Once a bus master gets access to the bus, all arbiters know when the bus will be released. Making use of this information, the first 5 cycles of bus arbitration can be overlapped with the data transfer of the previous bus transaction. This optimization can reduce the interconnect overhead from 15 cycles to 10 cycles.

Processor Model	4 way issue superscalar
Private L1 I & D–Caches	4-way associative, 32KB, 64 bytes block size, 3 cycle access latency
L2 Cache	16 slices, 256KB/slice, 8 way associativity per slice, 64B lines, 10 cycles for local hits, 25 cycles for merged hits
L3 Cache	16 slices, 1MB/slice, 16 way associativity per slice, 64B lines, 30 cycles for local hits, 45 cycles for merged hits
Memory	300 cycle off–chip access latency
Epoch Interval	300 Million cycles (reconfiguration interval)

Table 3. Baseline configuration.

SPEC CPU 2006 Benchmarks										PARSEC Benchmarks						
Benchmark	L2		L3		Benchmark	L2		L3		Benchmark	L2			L3		
	ACF	σ_t	ACF	σ_t		ACF	σ_t	ACF	σ_t		ACF	σ_t	σ_s	ACF	σ_t	σ_s
GemsFDTD(0)	0.34	0.14	0.46	0.25	***astar(1)***	0.42	0.06	0.56	0.02	***blackscholes***	0.23	0.04	0.07	0.18	0.02	0.05
bwaves(2)	0.56	0.05	0.43	0.17	***bzip2(2)***	0.59	0.18	0.46	0.22	***bodytrack***	0.38	0.07	0.03	0.22	0.04	0.02
cactusADM(2)	0.74	0.16	0.48	0.04	***calculix(3)***	0.62	0.02	0.56	0.02	***canneal***	0.65	0.13	0.18	0.58	0.07	0.14
dealII(3)	0.58	0.07	0.71	0.19	***gamess(0)***	0.41	0.09	0.38	0.11	***dedup***	0.47	0.05	0.08	0.74	0.16	0.12
gcc(3)	0.59	0.18	0.66	0.13	***gobmk(2)***	0.73	0.13	0.45	0.01	***facesim***	0.41	0.11	0.14	0.64	0.17	0.08
gromacs(1)	0.39	0.14	0.77	0.20	***h264ref(3)***	0.65	0.02	0.55	0.04	***ferret***	0.59	0.14	0.18	0.58	0.06	0.08
hmmer(1)	0.31	0.19	0.69	0.11	***lbm(0)***	0.44	0.19	0.42	0.08	***fluidanimate***	0.47	0.04	0.11	0.41	0.03	0.19
leslie3d(2)	0.56	0.04	0.34	0.12	***libquantum(0)***	0.26	0.14	0.18	0.11	***freqmine***	0.61	0.13	0.13	0.71	0.14	0.20
mcf(1)	0.38	0.16	0.51	0.04	***milc(1)***	0.42	0.02	0.59	0.05	***streamcluster***	0.79	0.28	0.12	0.61	0.16	0.07
namd(2)	0.55	0.04	0.48	0.12	***omnetpp(1)***	0.47	0.03	0.58	0.08	***swaptions***	0.43	0.05	0.11	0.37	0.04	0.02
perlbench(0)	0.31	0.08	0.42	0.01	***povray(2)***	0.58	0.11	0.41	0.07	***vips***	0.62	0.09	0.15	0.57	0.06	0.12
sjeng(2)	0.56	0.02	0.41	0.06	***soplex(2)***	0.53	0.07	0.47	0.07	***x264***	0.55	0.07	0.10	0.52	0.13	0.18
sphinx(1)	0.49	0.04	0.63	0.11	***tonto(3)***	0.63	0.12	0.57	0.06							
wrf(1)	0.46	0.07	0.73	0.14	***xalancbmk(3)***	0.58	0.03	0.57	0.03							
zeusmp(2)	0.54	0.05	0.44	0.17												

Table 4. The characteristics for SPEC benchmarks are collected on a cache topology with a single core with a private 256KB L2 slice and a private 1MB L3 slice. A 16 core CMP with 256KB L2 slice per core and 1MB L3 slice per core was used for PARSEC benchmarks. Average ACF and temporal standard deviation (σ_t) are shown for single threaded applications and each application's class is indicated in parentheses. Both temporal and spatial standard deviation (σ_s) are shown for multithreaded workloads.

Name	Type	Benchmarks
MIX_01	(0,0,10,6)	***calculix,bwaves,leslie,namd,sjeng,bzip2,povray,soplex,cactus,tonto,xalanc,zeusmp,dealII,gcc,gobmk,h264***
MIX_02	(0,4,6,6)	***dealII,gcc,leslie,namd,sjeng,zeusmp,bzip2,calculix,gobmk,h264,gomacs, hmmer,wrf,milc,tonto,xalanc***
MIX_03	(0,8,4,4)	***gromacs,hmmer,mcf,sphinx,wrf,astar,milc,omnetpp, namd,cactus,gobmk,soplex, gcc,calculix,h264,tonto***
MIX_04	(0,8,8,0)	***gromacs,hmmer,mcf,sphinx,wrf,astar,milc,omnetpp, bwaves,namd,leslie,sjeng,zeusmp,bzip2,povray,soplex***
MIX_05	(2,2,6,6)	***gamess,libm,sphinx,astar,bwaves,namd,sjeng,gobmk,povray,soplex, dealII,gcc,calculix,h264,tonto,xalanc***
MIX_06	(2,6,2,6)	***dealII,libq,perl,gromacs,hmmer,mcf,wrf,astar,milc,sjeng,gobmk,gcc,calculix,h264,tonto,xalanc***
MIX_07	(4,0,6,6)	***gcc,libm,libq,perl,cactus,zeusmp,bzip2,gobmk,povray,soplex,dealII,gamess,calculix,h264,tonto,xalanc***
MIX_08	(4,4,4,4)	***hmmer,mcf,libq,wrf, omnetpp,Gems,bwaves,bzip2,gobmk,perl,povray,gcc,calculix,libm,h264,xalanc***
MIX_09	(4,4,8,0)	***Gems,gamess,libm,libq,astar,gromacs,hmmer,milc,bwaves,leslie,sjeng,povray,gobmk,soplex,bzip2,zeusmp***
MIX_10	(4,6,0,6)	***perl,hmmer,mcf,wrf,astar,milc,Gems,omnetpp, dealII,libm,gcc,calculix,h264,gamess,tonto,xalanc***
MIX_11	(4,8,0,4)	***libm,libq,gromacs,hmmer,mcf,sphinx,wrf,gamess,astar,milc,omnetpp, gcc,Gems,h264,tonto,xalanc***
MIX_12	(4,8,4,0)	***gamess,libm,libq,perl,gromacs,hmmer,mcf,sphinx,wrf,astar,milc,omnetpp,sjeng,zeusmp,gobmk,soplex***

Table 5. Different multiprogrammed (SPEC) workload mixes. The type indicates the number of applications from each class of applications (class0,class1,class2,class3) in each mix.

4 Experimental Methodology

We evaluate MorphCache on a 16 core CMP with 4 issue superscalar cores. The baseline configuration of our target CMP is as shown in Table 3. We simulate the complete system using the Simics full-system simulator [16]. We have augmented Simics with accurate timing models similar to the RUBY module of GEMS [17]. For comparison with other static configurations, we assume a shared L2 and L3 cache as the *baseline cache topology*, unless otherwise specified. This is the same as (16:1:1) in the topology notation used in Section 1.2. We also compare MorphCache to a per core private configurations among other static configurations. When comparing with static cache configurations, we assumed that the static configurations have a fixed 10 cycle access latency for L2 and 30 cycle access latency for L3. In addition, the private L1 I and D caches are each 32 KB, 4-way associative, with 64B line size with 8 MSHR entries. MorphCache also has 10 and 30 cycles access latencies for local L2 and L3 cache slices, respectively. However, the remote cache slice accesses suffer an additional 15 cycles overhead due to the MorphCache interconnect. Note that these overheads are suffered only when the corresponding cache slices are merged. A reconfiguration decision is performed by comparing two counters and making two additions, which can be performed in 3 cycles. This decision can be delivered to the switches in the MorphCache interconnect in 5 cycles. Considering the 300 million cycles reconfiguration interval of MorphCache, this overhead of reconfiguration is negligible. Our goal is to perform reconfigurations at a granularity which is similar to context-switch/thread scheduling interval in order to reduce any impact it may have on data accumulated in the ACF. Note that, our simulation framework faithfully captures any additional overheads this may incur.

4.1 Workloads

To quantitatively evaluate the effectiveness of our MorphCache approach on CMPs, we used both multithreaded and multiprogrammed workloads consisting of applications. We used all applications from the SPEC CPU 2006 benchmark suite which includes both integer and floating point benchmarks for constructing our multiprogrammed workloads. All the SPEC benchmarks use the reference input sets. Table 4 shows these benchmarks and their the average ACF sizes in the L2 and L3 cache. A value of 1.0 represents 100% cache slice utilization (average). The table also shows the temporal standard deviation, σ_t (standard deviation in the set of ACFs collected for each benchmark across different epochs). Temporal standard deviation is a measure

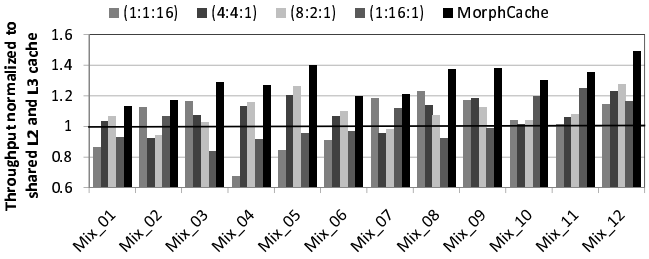


Figure 13. Throughput of MorphCache as compared to various static configurations for multiprogrammed (SPEC CPU 2006) applications.

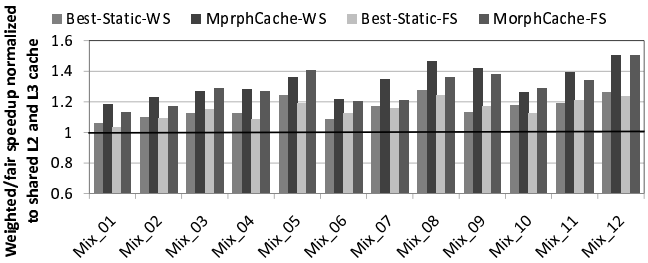


Figure 14. Weighted speedup (WS) and Fair speedup (FS) of MorphCache as compared to the best performing static configuration for each mix.

of how far the different ACF sizes over different epochs are from the average value over all epochs. We expect that when a benchmark with a high ACF is combined with one with a low ACF at a particular level, it is going to encourage a shared topology at that level. Therefore, we divided the benchmarks into four classes based on whether their L2 and L3 ACFs were low or high. Each application's class is indicated in parenthesis in Table 4, following its name. We constructed 12 workload mixes with varying representations of different types of benchmarks. Each multiprogrammed workload comprises 16 single-threaded benchmarks. The details of these workloads are in Table 5.

We also used all the applications from the PARSEC benchmark suite with simlarge inputs for evaluation with multithreaded workloads. Each multithreaded workload consists of a single application that is executed with 16 threads. Table 4 also shows the average ACFs in the L2 and L3 cache for all the PARSEC benchmarks, averaged over all threads of the benchmarks. Apart from showing the temporal standard deviation for each benchmark (in this case, the average of temporal deviations across the 16 threads of each application), the table also shows the spatial standard deviation for each benchmark, which is defined as the standard deviation among the ACFs of different threads in the same epoch. Spatial standard deviation is a measure of how far the different ACFs of different threads are from the average value over all threads of the application.

5 Experimental Results

We compare the performance of MorphCache to the baseline shared L2 and L3 cache configuration (i.e., (16:1:1)) and four other static configurations: (1:1:16), (4:4:1), (8:2:1), and (1:16:1). Note that (1:1:16) corresponds to an per core private L2 and L3 cache topology, while (1:16:1) represents a topology with per core L2 with a shared L3, as explained in Section 1.2.

5.1 Multiprogrammed Workloads

Throughput Results: Figure 13 shows the throughputs of MorphCache in comparison with other static topologies normalized to the baseline shared L2 and L3 cache topology. Mixes 1-3, 6-7, and 10 include more applications that have a large ACF in both the L2 and L3 caches. Consequently, they derive smaller benefits as compared to others that have a fair mix of workloads with high and low ACFs coupled with significant temporal variation in the ACFs. On average, MorphCache improves the throughput of all the mixes by 29.9% over the baseline, by 29.3% over (1:1:16) topology, by 19.9% over (4:4:1) topology, by 18.8% over (8:2:1) topology, and 27.9% over (1:16:1) topology.

Speedup Results: The throughput metric may be unfairly maximized by maximizing the IPC of a small subset of applications. The weighted speedup (WS) metric, in contrast, gives equal weight to the relative performance of each application. The fair speedup (FS) metric has been shown to balance both fairness and performance speedups [25]. Therefore, we compare (in Figure 14) the performance of MorphCache against the *best static topology* on both these metrics. On average, MorphCache performs 32.8% better on the WS metric as compared to the baseline case and 12.3% better than the best static topology of (2:2:4) (the static topology that achieved the best WS). It also performs 29.7% better on the FS metric with respect to the baseline configuration and 10.8% better than the best static topology of (4:4:1) (the static topology that achieved the best FS).

Comparison with an Ideal Offline Scheme: Given the adaptive behavior of MorphCache, it is important to know the effectiveness of its ability to adapt to the best topology at runtime. In order to quantify this effectiveness, we compare MorphCache to an ideal scheme that executes the workloads in all the static configurations discussed above and, at the beginning of each epoch, chooses the one that will give the best performance for that epoch. Note that, it is not practical to implement the ideal offline scheme in practice, given the extensive offline analysis required by such a scheme. It is also impossible to have this information for a new/arbitrary workload. Figure 15 shows the comparison of throughput normalized to our baseline case in MorphCache and this ideal offline scheme. We see that, with the help of a simple ACF estimation hardware, MorphCache is able to obtain almost 97% of the performance benefits of an ideal offline scheme. We also observed that in many cases, the improvements obtained by MorphCache were due to *asymmetrical topologies* adopted (which were better than any symmetrical static topology chosen by the ideal offline scheme). For example, for Mix 10, no static symmetrical topology was able

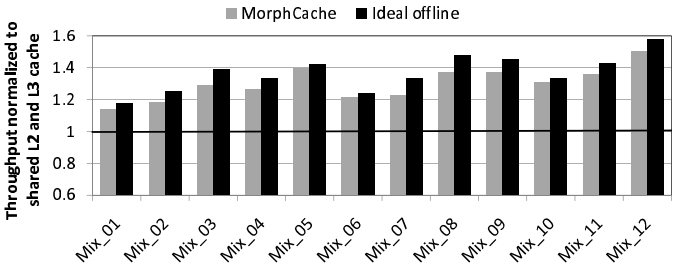


Figure 15. Throughput compared to an ideal offline scheme.

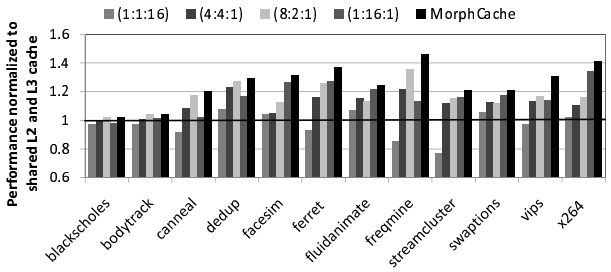


Figure 16. Comparison with various static configurations for multithreaded (PARSEC) applications.

to achieve similar performance to the asymmetrical topology selected in step 4 (after T3) for cores 0-3. The topology used by MorphCache facilitates capacity sharing in L2 between cores 2 and 3, and in L3 among cores 0-3. In the following interval, cores 0 and 1 require L2 capacity sharing. This is clearly not possible in any static symmetric topology.

5.2 Multithreaded Workloads

In case of multithreaded workloads, we measure the performance as the *inverse of execution time*. We compared the performance of all the multithreaded PARSEC applications in MorphCache to the baseline case of (16:1:1) and the other static topologies. The results are shown in Figure 16. We can see that facesim, ferret, freqmine, and x264 derive higher performance benefits than most others. This can be attributed to the high spatial variance among the ACFs of their threads. As seen from Table 4, facesim and ferret have a high spatial standard deviation in L2 while, freqmine and x264 have a high spatial standard deviation in L3. A high spatial standard deviation implies a higher requirement for capacity sharing among the threads. On average, MorphCache improves the performance of our multithreaded applications by 25.6% over (16:1:1), by 30.4% over (1:1:16) topology, by 12.3% over (4:4:1) topology, by 7.5% over (8:2:1) topology, and 8.5% over (1:16:1) topology.

5.3 Throttling MSAT - Controlling QoS

The merge aggressive policy (of merging cache slices whenever we have an option) we have adopted thus far leads to overall improvements in performance. However, this may be achieved at the cost of hurting some applications when providing significant improvements for the other applications. This leads to a lack of *quality of service* (QoS). With a small modification to MorphCache, we can assure that the performance improvement is obtained without degrading any individual application's performance below the performance obtained by each application getting its fair share of cache space (similar to private configuration). We leverage the fact that MorphCache bases its reconfiguration decisions on the miss/split aggressiveness threshold (MSAT) at each reconfiguration step. Note that, from the perspective of providing QoS, it is important to control when a merge occurs as merges are majorly responsible for degradation of QoS. We can track the number of misses incurred by an application before and after each merging reconfiguration step. Each time we observe an increase in the number of misses, we throttle up the MSAT (increase the higher bound and decrease the lower bound of MSAT). By throttling up the MSAT, we are moving towards a private configuration that provides the necessary QoS. But, whenever we do not see any change or observe an improvement after a merge step, we throttle down the MSAT. This helps in improving overall system performance. The storage overhead required to maintain the miss information per cache slice is two registers of 4 bytes each. This accounts for a total of 8 bytes per cache slice overhead for a QoS aware MorphCache scheme.

5.4 Summary of Sensitivity Experiments

We performed several sensitivity experiments to study the sensitivity of MorphCache to various configuration parameters including cache sizes (at different levels), set associativities, and the number of cores. We found that increasing the L2 cache size from 256 KB/slice to 512 KB/slice leads to an increase in the performance improvements obtained by MorphCache by 2.1%, on average. This can be attributed to intelligent management of the additional capacity by MorphCache. Increasing the L3 cache size had a slightly smaller impact leading to an improvement in MorphCache performance by 1.8%, on average. Doubling the associativities of the cache (at the cost of increased access latencies) did not bring in additional benefits for MorphCache. We also experimented with 8 core configurations and found the throughput improvements by MorphCache with multiple 8-application mixes to be comparable. The actual benefits were 0.7% lesser than that observed in the 16 core case, on average. This can be attributed to the reduced flexibility for MorphCache to reconfigure the topology in an 8 core case.

5.5 Extensions and Scalability

So far we were restricting the reconfigurations to cache sharing only among 2, 4, 8, or 16 cores and only among neighboring cores. We can potentially relax these assumptions. In some cases, depending on the active utilizations of the caches, it may be beneficial to allow groups of n slices where n is not a power of 2 or groups of slices that comprise of non-neighboring slices to be configured in shared mode. In its current form, the interconnect assumed in this work does not support such groups to operate independently. However, this configuration can still be supported by creat-

ing physical groups that are supersets of the required logical groups. In this case, these logical groups use the same physical interconnect and identify the messages intended for them using unique logical group IDs. The communication latency incurred in this case depends on the greatest distance between any two slices connected to the same physical fabric. For instance, a sharing between L2 cache slice of core 1 and core 7 can be realized by creating a physical connection that spans all 8 L2 slices and using separate logical IDs for the two groups. Similarly, a sharing among L3 cache slices 1, 2, and 3 can be realized by a physical connection spanning 4 L3 slices and using separate logical IDs for the two groups. We also performed experiments to evaluate the benefits of such policies. Allowing arbitrary number of neighboring cores to be shared, on average, improves the throughput of all the multiprogrammed mixes by 3.6% over the throughput achieved by restricted sharing (default) scheme. However, relaxing the constraint of sharing among neighbors only, and allowing non-neighboring cores to share cache slices leads to a 7.1% degradation on the throughput metric. This is basically because of the large access latencies to the distant cache slices becoming dominant. This means that, even for a chip with 16 cores, accessing non-neighboring cores may be harmful to performance. Therefore, instead of allowing such an arbitrary sharing, we believe it will be more beneficial to employ appropriate application/thread scheduling strategies that would take advantage of the MorphCache architecture, which is part of our future work. Fortunately, this observation also provides us with a good way to scale MorphCache to future manycore CMPs. Although the segmented-bus based design of the interconnect may not be efficiently scalable beyond 16 cores, higher core counts in a CMP can effectively exploit the advantages of MorphCache by using a tile-based architecture, where each tile of at most 16 cores would use a cache hierarchy managed as a MorphCache, while the tiles themselves would be connected using a more scalable interconnection network. Threads that share code or data would be scheduled on the cores within a tile. Consequently, we can leverage the advantages of both the performance of a bus based network within a tile as well as the scalability of a more general interconnection network across tiles.

6 Related Work

Shared last level cache management. Liu et al. [15] evaluated the performance of managing the allocation of cache resources on a bus-based CMP and proposed a profile-driven approach to determine which cache banks to share between processors and which to reserve as private at the last level of shared cache. In contrast, MorphCache dynamically determines the cache requirements at multiple levels of cache for the workloads and reconfigures the cache topology suitably.

Rafique et al. [21] proposed an OS scheme that implements a variety of policies including those for fair cache sharing and achieving differentiated instruction throughput. The overheads of implementing cache management from the OS are typically higher and a OS managed reconfigurable cache is also hampered by the coarse granularity at which it can be reconfigured. Chang and Sohi [5] have proposed cooperative cache partitioning (CCP), wherein they use multiple time sharing partitions to resolve cache contention. Obtaining information about cache requirement curves (as required by CCP) may require a high training time or may not be practical to profile. In contrast, MorphCache uses simple ACF estimation hardware to obtain all the necessary information for a reconfigurable cache topology. Qureshi and Patt [20] proposed a utility based cache partitioning scheme where the share received by an application was proportional to the utility rather than its demand. Qureshi et al. [19] also proposed dynamic insertion policies (DIP) that is a flexible replacement policy that dynamically chooses between multiple insertion policies to incur fewer misses. Further, Jaleel et al. [10] extended the work on DIP and proposed thread aware dynamic insertion policies (TADIP) that takes into account the memory requirements of each application based on the feedback from set-dueling monitors. In contrast, MorphCache uses the LRU replacement policy for all applications as it adapts the cache topology for different applications and insulates any cache-thrashing applications as it learns the ACFs with the help of ACFVs. Xie and Loh proposed promotion/insertion pseudo-partitioning (PIPP) [28] to manage shared caches. PIPP combines dynamic insertion and promotion policies to provide the benefits of cache partitioning, adaptive insertion, and capacity stealing all with a single mechanism. However, simply extending PIPP to multiple levels of the cache can result in sub-optimal results as it is "topology-unaware". We also found that while PIPP performs on par with MorphCache with a 2 core CMP, it does not scale well to 16 cores or further.

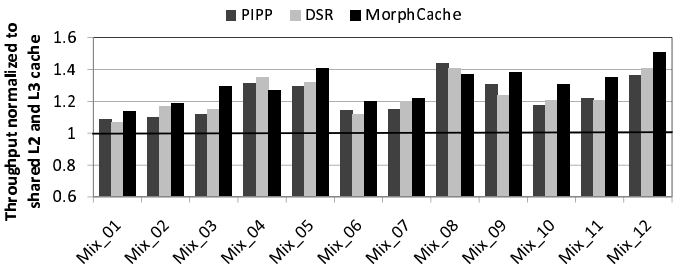


Figure 17. Performance of MorphCache as compared to PIPP [28] and DSR [18] schemes extended to multiple level cache hierarchy.

Figure 17 presents a comparison of PIPP extended to both L2 and L3 caches against the proposed MorphCache architecture. We observe that MorphCache outperforms the extended version of PIPP on all but two mixes. There is little variation in the ACFs of the applications in these two

mixes, MIX_04 and Mix_08 leading, to slightly smaller benefits from the MorphCache architecture. However, on average, MorphCache performs 6.6% better than PIPP on the throughput metric.

Private last level cache management. Many previously proposed schemes [4, 6, 3] for managing the last level cache in CMPs as private caches suggest the use of replication in order to take advantage of the trade-off between latency and on-chip capacity. All these schemes tune the degree of replication in different ways to strike the right trade-off. MorphCache, on the other hand solves an orthogonal problem of capacity sharing at different levels of caches in CMPs. The most closely related proposal to MorphCache in the literature is dynamic spill receive (DSR) [18] architecture proposed for capacity sharing in private cache organization for last level cache of CMP. DSR uses set dueling monitors to learn if each cache has to act as a spiller or a receiver. While DSR performs well for a single level of cache hierarchy, it is topology agnostic and does not extend well to multiple levels. We compared the performance of MorphCache architecture against a multilevel private cache management scheme which is an extension of DSR to multiple levels of private caches. The results are shown in Figure 17. We notice that MorphCache outperforms DSR on all but the same two mixes where it does not perform well due lesser variations among ACFs of the applications involved. On an average, MorphCache outperforms DSR by 5.7%.

7 Concluding Remarks

In this paper, we present a novel dynamically reconfigurable cache hierarchy for CMPs called MorphCache. Using active cache footprint of per core cache slices and the overlap between footprints to estimate the data sharing, MorphCache deciphers the benefits of dynamically sharing multiple cache slices among cores. It alters the cache topology dynamically by merging or splitting cache slices and modifying the accessibility of different cache slice groups to different cores in a CMP. We evaluated MorphCache on a 16 core CMP using a full system simulator and found that MorphCache significantly improves both average throughput and harmonic mean of speedups of diverse multithreaded and multiprogrammed workloads. Although we believe that the segmented-bus architecture would lead to reduced power consumption in MorphCache, we would like to quantify this improvement in the future.

References

[1] M. Azimi et al. Integration challenges and tradeoffs for tera-scale architectures. *Intel Technology Journal*, 11(03), 2007.

[2] R. Balasubramonian et al. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, 2000.

[3] B. M. Beckmann et al. ASR: Adaptive selective replication for CMP caches. In *MICRO*, 2006.

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

[5] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS*, pages 242–252, 2007.

[6] Z. Chishti et al. Optimizing replication, communication, and capacity allocation in CMPs. In *ISCA*, 2005.

[7] P. P. Gelsinger. Intel architecture press briefing. `http://download.intel.com/pressroom/archive/reference/Gelsinger_briefing_0308.pdf`, 2008.

[8] J. Guo et al. Energy/area/delay tradeoffs in the physical design of on-chip segmented bus architecture. *IEEE Trans. Very Large Scale Integr. Syst.*, 15(8), 2007.

[9] L. R. Hsu et al. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT*, 2006.

[10] A. Jaleel et al. Adaptive insertion policies for managing shared caches. In *PACT*, 2008.

[11] R. Kalla et al. IBM POWER5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2), 2004.

[12] S. Kim et al. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, pages 111–122, 2004.

[13] R. Kumar et al. Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling. *SIGARCH Comput. Archit. News*, 33(2), 2005.

[14] L. Li et al. CCC: Crossbar connected caches for reducing energy consumption of on-chip multiprocessors. *Euromicro Symposium on Digital Systems Design*, 0:41, 2003.

[15] C. Liu et al. Organizing the last line of defense before hitting the memory wall for CMPs. In *HPCA*, page 176, 2004.

[16] P. S. Magnusson et al. SIMICS: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[17] M. M. K. Martin et al. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. *SIGARCH Comput. Archit. News*, 33(4), 2005.

[18] M. K. Qureshi. Adaptive spill-receive for robust high-performance caching in CMPs. In *HPCA*, 2009.

[19] M. K. Qureshi et al. Adaptive insertion policies for high performance caching. In *ISCA*, 2007.

[20] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[21] N. Rafique et al. Architectural support for operating system-driven CMP cache management. In *PACT*, 2006.

[22] M. V. Ramakrishna et al. Efficient hardware hashing functions for high performance computers. *IEEE Trans. Comput.*, 46(12), 1997.

[23] P. Ranganathan et al. Reconfigurable caches and their application to media processing. In *ISCA*, 2000.

[24] J. T. Robinson. Generalized Tree-LRU Replacement. Technical Report RC23332, IBM Research Division, September 2004.

[25] J. E. Smith. Characterizing computer performance with a single number. *Commun. ACM*, 31(10):1202–1206, 1988.

[26] S. Srikantaiah et al. Adaptive set pinning: managing shared caches in chip multiprocessors. In *ASPLOS*, 2008.

[27] G. E. Suh et al. Dynamic partitioning of shared cache memory. *J. Supercomput.*, 28(1):7–26, 2004.

[28] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of multi-core shared caches. *SIGARCH Comput. Archit. News*, 37(3), 2009.

[29] S.-H. Yang et al. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA*, 2002.