Reducing inter-core cache contention with an adaptive bank mapping policy in DRAM cache

Conference Paper · September 2013

DOI: 10.1109/CODES-ISSS.2013.6658988

3 authors, including:

Fazal Hameed
Institute of Space Technology
35 PUBLICATIONS **465** CITATIONS

SEE PROFILE

Lars Bauer
Karlsruhe Institute of Technology
102 PUBLICATIONS **2,494** CITATIONS

SEE PROFILE

Reducing Inter-Core Cache Contention with an Adaptive Bank Mapping Policy in DRAM Cache

Fazal Hameed, Lars Bauer, and Jörg Henkel
Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Germany
{hameed, lars.bauer, henkel}@kit.edu

ABSTRACT

On-chip DRAM cache has the advantage of increased cache capacity that may alleviate the memory bandwidth problem. Recent research has demonstrated the benefits of high capacity on-chip DRAM cache that leads to reduced off-chip accesses. However, state-of-the-art has not taken into consideration the cache access patterns of concurrently running heterogeneous applications that can cause inter-core cache contention. We therefore propose an adaptive bank mapping policy in response to the diverse requirements of applications with different cache access behaviors that – as a result – reduces inter-core cache contention in DRAM-based cache architectures. On average, our adaptive bank mapping policy increases the harmonic mean instruction-per-cycle throughput by 19.3% (max. 71%) compared to state-of-the-art bank mapping policies.

1. INTRODUCTION

Future multi-core systems are expected to execute memory intensive applications with large working set sizes which exacerbate the "Memory Bandwidth" problem [13] because of the significantly increasing processor-memory speed gap. A recent trend towards mitigating the memory bandwidth problem is to use a large on-chip DRAM Last-Level-Cache (LLC). DRAM cache provides greater capacity benefits (~8× [3, 8]) compared to SRAM caches and allows to store large working sets, hereby reducing off-chip accesses. For example, the IBM POWER7 employs a 32 MB DRAM-LLC using embedded DRAM technology to reduce the off-chip accesses [15].

Typically, a cache is composed of multiple banks. *Bank Mapping* refers to the assignment of a memory block to a cache bank. Each memory block (also called cache line; 64 bytes in our case) is a contiguous series of bytes and describes the granularity at which the cache operates. Recent approaches [3, 9] and commercial multi-core processors [2, 14] use the least significant bits of the memory block address to define the cache bank that houses the tag/data of the block. However, this policy is not generally beneficial as it may cause inter-core cache contention, where one core could evict useful cache lines used by another core [4]. Several bank mapping policies [4] at larger mapping granularity (i.e. segment level) have been proposed for multi-bank SRAM caches. These policies map a memory segment (typical segment size is 4 KB; also called as page) to a unique cache bank so that consecutive memory blocks (within the same segment) belong to the same cache bank. These policies decide the assignment of a segment to a cache bank on a segment miss, i.e. when the segment is referenced for the first time.

Existing bank mapping policies [2, 3, 4, 9, 14] are application-unaware and do not consider inter-core cache contention. After discussing state-of-the-art in Section 2 and the technical background in Section 3, we present an example in Section 3.4 that shows how some applications may slowdown other applications using state-of-the-art bank mapping policies and how judicious bank mapping can be used to mitigate inter-core cache contention.

Multi-core systems execute multiple applications with diverse cache access patterns. An application is said to exhibit *thrashing* behavior if its memory accesses have low temporal locality and if it places a large amount of data in the cache with majority of these data not reused in the future. These thrashing applications may evict useful data belonging to non-thrashing applications, and as a result, cause inter-core cache contention and severe performance degradation of non-thrashing applications.

We make the following new contributions:

1. We propose the *adaptive bank mapping* (*ABM*) policy after analyzing that existing bank mapping policies may not work efficiently due to inter-core cache contention [2, 3, 9, 14] and under-utilization/over-utilization [4] of cache resources (details in Section 3.4). Our *ABM* policy tracks the run-time miss rate information of concurrently executing diverse applications and adapts the bank mapping policy on a per-core basis. It comes with a negligible hardware overhead compared to existing policies (details in Section 4.5).
2. Our *ABM* policy maps the majority of memory segments from thrashing applications onto separate cache banks, since these applications may evict useful cache lines belonging to other applications. This provides *performance isolation* between thrashing and non-thrashing applications as it leads to reduced inter-core cache contention. In contrast, state-of-the-art [2, 3, 9, 14] bank mapping policies suffer from inter-core cache contention when thrashing applications run concurrently with non-thrashing applications.
3. Our *ABM* policy provides cooperative cache sharing for non-thrashing applications (details in Section 4.3) in order to mitigate under-utilization/over-utilization of cache resources whereas state-of-the-art [4] bank mapping policies suffer from under-utilization/over-utilization of cache resources for non-thrashing applications (details in Section 3.4).

2. RELATED WORK

There is a considerable amount of related studies on reducing inter-core cache contention in SRAM [5, 7, 12, 17] and DRAM caches [6, 8] applied on a cache set level. The work presented in [5, 7] adapts the cache line insertion policy in SRAM cache by tracking the run-time miss rate information of the individual applications. However, they require non-trivial changes to the

978-1-4799-1417-3/13/$31.00

existing least recently used (LRU) policy. The work presented in [12, 17] uses way-partitioning that allocates more cache resources (i.e. ways) to the applications that can use them better. They require modifications to LRU policy and use a Utility Monitoring Circuit (UMON) that comes with a significant area overhead for a larger DRAM cache. For instance, it requires SRAM storage overhead of 296KB for a 4-core system with 64MB DRAM cache. The work in [6] combines shared SRAM and DRAM caches to form a hybrid last-level-cache that exploit the latency benefits of SRAM cache and capacity benefits of DRAM caches. In contrast to [6], this paper does not employ shared SRAM cache in the cache hierarchy and uses DRAM as the last-level cache. However, our adaptive bank mapping (*ABM*) policy can be built on top of hybrid last-level-cache hierarchy [6] to provide inter-core performance isolation and to mitigate thrashing.

The most prominent work that considers the inter-core cache contention in a shared DRAM cache is the **A**daptive **M**ulti-**Q**ueue policy (*AMQ-policy*) [8]. The *AMQ-policy* organizes each DRAM cache set as multiple queue structures that reduce contention between thrashing and non-thrashing applications. However, the *AMQ-policy* has the following drawbacks compared to our *ABM* policy. First, the *AMQ-policy* requires non-trivial changes to the LRU policy. In contrast, our *ABM* policy does not require any such modifications. Second, the *AMQ-policy* needs to store the tags in the SRAM array which incurs a significant area overhead for larger DRAM caches. In contrast, *ABM* stores the tags in the DRAM array. Finally, the data movement between different queue structures in the *AMQ-policy* introduces additional latency and hardware complexity compared to our work.

In summary, related studies [5, 7, 8, 12, 17] that consider inter-core cache contention in SRAM and DRAM caches require modification to the conventional LRU policy. This leads to an increased hardware overhead and complexity compared to the LRU policy. In contrast, our *ABM* policy mitigates inter-core cache contention via judicious bank mapping and does not require any modifications to the LRU policy. Related studies [5, 6, 7, 8, 12, 17] statically determine the bank mapping policy to determine the cache bank that houses the tag/data of a block. In contrast, we adapt the bank mapping policy at runtime considering the cache access behavior of concurrently executing applications. The policies at cache set level proposed in [5, 6, 7, 8, 12, 17] can be combined with our adaptive bank-level policy to further mitigate inter-core cache contention. That means it is possible to devise a hybrid approach by integrating set-level policies with the bank-level policy proposed in this paper (such a combination is beyond the scope of this work).

3. BACKGROUND

3.1 DRAM Cache Organization

A primary challenge in architecting a large DRAM cache is the design of the tag store which is required to identify a cache hit/miss. A 64 MB DRAM cache has a tag storage requirement of 6 MB when using 6 bytes per tag entry. The tags can be stored in a separate SRAM tag array which eliminates slow DRAM access if the SRAM tag array indicates a cache miss. This is called "Tags-In-SRAM" approach which incurs a high area overhead. To avoid large area overhead, state-of-the-art DRAM caches [6, 9, 10] store the tags in the DRAM cache as well (called "Tags-In-DRAM" approach). They co-locate the tags and data for an entire cache set in the same row as shown in Figure 1. The tags indicate the actual location of a data block stored in the row. A typical DRAM row size of 2 KB can store up to 32 64-byte blocks. The Tags-In-DRAM approach reserves 29 blocks for data (i.e. an associativity of 29 ways per set) and 3 blocks for tags (29×6 = 174 bytes). It mitigates the storage overhead limitations of the Tags-in-SRAM approach because the DRAM cache provides greater capacity benefits (~8× [3, 8]) compared to SRAM caches. However, it performs a slow DRAM access to identify a hit/miss before the request is sent to off-chip main memory (in case of a miss). To combat this problem, recent work [9] uses a low overhead SRAM based structure named as *MissMap* (1.7 MB compared to 6 MB Tags-in-SRAM) that accurately determines whether an access to a DRAM cache will be a hit or a miss (details in Section 3.2; see Figure 1 and Figure 2), while the tags are stored in the DRAM.

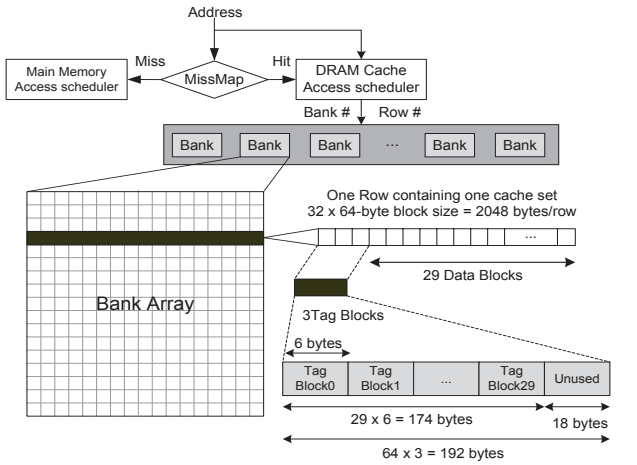


Figure 1: Tags-In-DRAM DRAM cache organization [6, 9]

3.2 *MissMap* Organization

The design of the Tags-in-DRAM *MissMap* is illustrated in Figure 2. If the *MissMap* identifies a hit, the request is sent to the DRAM cache scheduler (see Figure 1). A *MissMap* miss (i.e. data is not available in the DRAM cache) makes DRAM cache misses faster because the DRAM cache does not need to be accessed to determine a DRAM cache miss. Each *MissMap* entry represents a memory segment (we use a segment size of 4 KB) and tracks the presence of the cache blocks (we use a cache block size of 64 bytes) of that segment. Therefore, each *MissMap* entry contains a tag (*Seg-Tag*; see "*MissMap* tag-array" in Figure 2) corresponding to the address of the tracked memory segment and a bit vector (*Seg-BV*; see "*MissMap* Data array") with one bit per cache block that stores the hit/miss information of the block. If a corresponding *Seg-BV* entry is 1, the block within the segment is present in the DRAM cache, otherwise it is absent.

On a *MissMap* access, the set index field (see Figure 2) of the requested physical address is used to index a *MissMap* set in the *MissMap* tag-array. All tag entries within that *MissMap* set (an associativity of 4 is shown in Figure 2) are then compared to the *Seg-Tag* field of the physical address to identify a segment hit/miss. A segment miss implies that the requested block is absent in the DRAM cache. Following a segment hit, the vector index field of the requested physical address is used to index the *Seg-BV* entry of the hit-segment to identify a block hit/miss.

When a block b_i is placed into the DRAM cache (i.e. the tag/data of block b_i is stored in the DRAM cache), then the *MissMap* segment entry S to which b_i belongs needs to be accessed. If no segment entry for S exists in the *MissMap*, a new entry is allocated for it (see below). Then, the *Seg-BV* entry i (i.e.

the i[th] bit of *Seg-BV*) of segment S is set. When block b_i is evicted from the DRAM cache, the *Seg-BV* entry *i* of segment S is cleared. The *MissMap* is managed as a set-associative cache. Whenever a block is placed in the DRAM cache and the *MissMap* does not contain a corresponding entry (i.e. segment miss), then a new *MissMap* entry must be allocated and a victim segment is chosen using the least recently used (LRU) policy. If some *Seg-BV* entries of the victim segment *S* were set (i.e. some of its cache blocks are present in the DRAM cache), then all corresponding cache blocks must be evicted from the DRAM cache. This guarantees that the *MissMap* always accurately determines whether an access to a DRAM cache will be a hit or a miss.

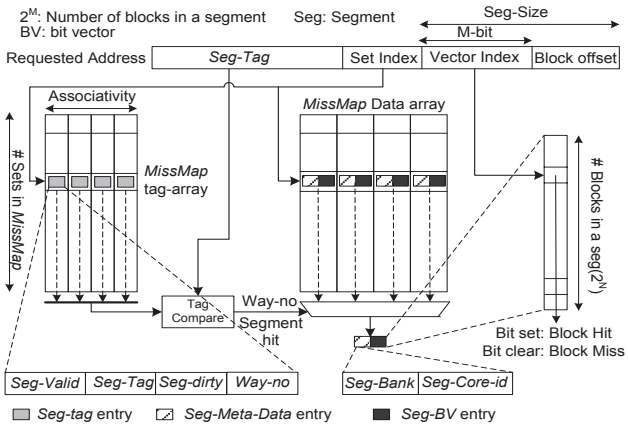


Figure 2: MissMap [9] for LLC-DRAM hit-miss detection. For our proposed extension we add the Seg-Meta-Data field to the MissMap to support our adaptive bank mapping policy

For the rest of the paper, the term "bank mapping policy" refers to the assignment of a memory segment to a DRAM cache bank after a *MissMap* segment miss, i.e. when an application accesses a new segment that is currently absent in the *MissMap*.

3.3 *State-of-the-art Bank Mapping Policies*

When an application running on $core_i$ accesses a new segment *S* that is currently absent in the *MissMap*, then a new *MissMap* entry *E* is allocated for segment *S* and *S* is assigned to a DRAM cache bank D_{bank}. We assume *N* cores and *K* DRAM cache banks where $N \leq K$ and *K* is a multiple of two. Figure 3 illustrates an LLC-DRAM cache with N=4 and K=16, where B_i correspond to a single DRAM cache bank.

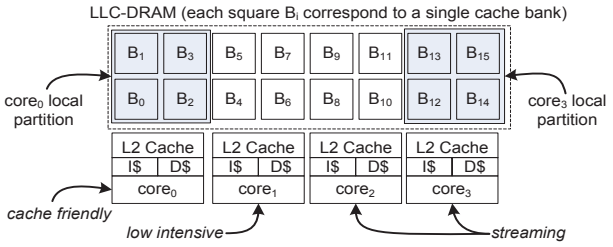


Figure 3: LLC-DRAM cache with N=4 cores and K=16 Banks for PS-policy [4] executing a cache friendly, a low intensive, and two streaming applications

We now describe the different policies from state-of-the-art as follows.

SLIP [2, 3, 9, 14]**:** uses the least significant $\log_2(K)$ bits of the memory block address to decide D_{bank} as illustrated in Figure 4.

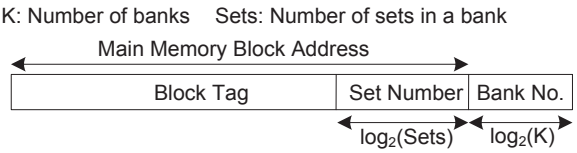


Figure 4: Shared Line Interleaving policy (SLIP) [2, 3, 4, 9]

PS-policy [4]**:** *S* is assigned to the local partition of the requesting core where each local partition contains *K*/*N* cache banks. For the example shown in Figure 3, D_{bank} is chosen among B_0, B_1, B_2, and B_3 for a new segment *S* requested from $core_0$.

RRS-policy [4]**:** D_{bank} is chosen for *S* among B_0, B_1, …, B_{K-1} in a round robin way.

3.4 *Motivational Example*

The multi-programmed applications from SPEC2006 [1] vary widely in terms of their cache requirements. Figure 5 illustrates this observation showing LLC Misses Per Thousand Instructions (MPKI) for different LLC sizes and different SPEC2006 applications with 64 MB LLC-DRAM cache. Recent work [7, 16] has classified applications into the following categories:

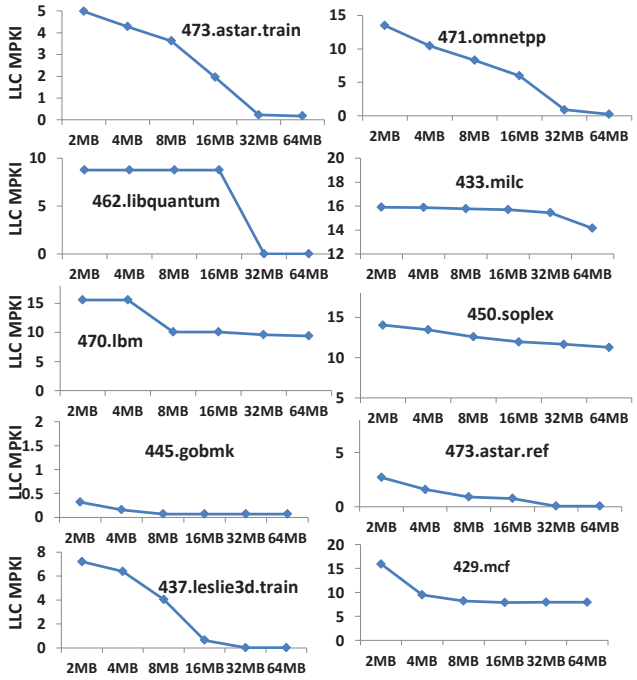


Figure 5: LLC misses per thousand instructions (LLC MPKI) for different SPEC2006 applications [1] and LLC sizes

1) *cache-friendly* applications are very sensitive to the amount of cache resources allocated to them. Increasing the cache resources of these applications (e.g. *473.astar.train*, *471.omnetpp*, and *437.leslie3d.train*) provides significant reduction in MPKI.

2) *low-intensive* applications (e.g. *445.gobmk*, *473.astar.ref*) have a small working set size compared to the available cache size and have a low cache miss rate.

3) *cache-fitting* applications require a huge working set size to perform well. For example, *462.libquantum* requires a working set size of 29 MB. When running alone or running concurrently with *low-intensive* applications, these applications have a very low miss-rate. When running concurrently with *cache friendly* applica-

tions, they can cause thrashing when the cache resources given to them is less than their working set sizes.

4) *streaming* applications have a high cache miss rate and a high cache access rate. These applications (e.g. *450.soplex* and *433.milc*) get negligible benefit from increasing the cache resources.

Let us consider a scenario where $core_0$ is running a *cache-friendly* application, $core_1$ is running a *low-intensive* application, and $core_2$ and $core_3$ are running *streaming* applications as depicted in Figure 3. In the *PS-policy*, some of the local partitions may be under-utilized, whereas others may be severely over-utilized. For example, the local partition associated with $core_0$ (running a *cache-friendly* application) will be over-utilized and the local partition associated with $core_1$ (running a *low-intensive* application) will be under-utilized. The *SLIP* and *RRS-policy* improve the utilization of partitions as the new segment *S* can be assigned to any partition. However, they may incur inter-core cache contention when *cache-friendly* and *streaming* applications run concurrently. For instance, the *streaming* applications running on $core_2$ and $core_3$ may evict useful data belonging to the *cache-friendly* application running on $core_0$. Our Adaptive Bank Mapping (*ABM*) policy (details in Section 4) reduce inter-core cache contention by mapping the majority of memory segments from *streaming* applications to a dedicated single cache bank. Doing so reduces inter-core cache contention between *cache-friendly* applications and *streaming* applications. Furthermore, our *ABM* policy provides cooperative cache sharing for non-thrashing applications (details in Section 4.3) in order to mitigate under-utilization and over-utilization of cache resources.

4. ADAPTIVE BANK MAPPING POLICY

Our adaptive bank mapping (ABM) policy targets a cache hierarchy with on-chip DRAM Last-Level-Cache (LLC) and a *MissMap* [9] as shown in Figure 6. The *MissMap* needs to be modified as it uses the traditional *SLIP* [2, 3, 9, 14] policy whereas we use our adaptive bank mapping policy (details in Section 4.3) to define the cache bank that houses the tag/data of the relevant block. *SLIP* does not require storing the bank field in the *MissMap* as the bank number is statically determined by the least significant bits of the memory block address. Our *ABM* policy needs to store the cache bank number (called *Seg-Bank*) in the *MissMap* because they are assigned adaptively. Additional per-segment fields (called *Seg-Meta-Data*, see Figure 2) are required in the *MissMap* to guide our *ABM* policy. On a *MissMap* segment hit in our *ABM* policy, the *Seg-Bank* field determines the cache bank that houses the corresponding segment. This modification to the *MissMap* provides the basic support to implement our *ABM* policy. The *MissMap* is accessed (explained in Section 3.2) after an L2 cache miss. A *MissMap* hit indicates that the line is present in the DRAM cache. In that case, the line is brought from the DRAM bank (specified by *Seg-Bank* field of corresponding *MissMap* entry) and forwarded to the requesting core. After a *MissMap* miss, the line is brought from the memory and filled into a DRAM bank (specified by *Seg-Bank* field of corresponding *MissMap* entry). Additionally, the line is forwarded to the requesting core and filled in the core private caches.

Our *ABM* policy consists of three major components (see Figure 6):

1. **Application Profiling Unit (APU):** In order to provide sufficient information for our adaptive bank mapping policy, the APU profiles the application behavior (thrashing or non-thrashing) by tracking run-time miss rate information of all concurrently executing applications (described Section 4.1).
2. **Policy Selection Unit (PSU):** reads the runtime statistics provided by APU to determine the suitable bank mapping policy for each application (Section 4.2).
3. **Bank assignment:** allocation of segments to a suitable bank as determined by the PSU (Section 4.3).

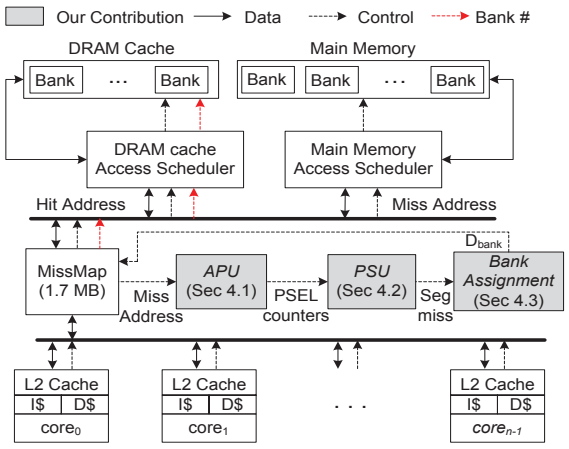
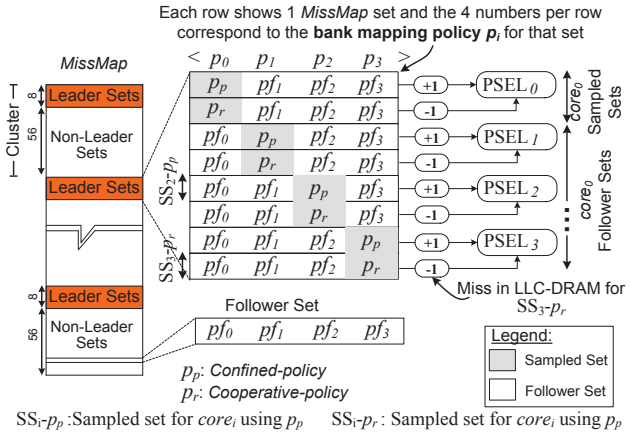


Figure 6: Organization of our Adaptive Bank Mapping policy

4.1 Application Profiling Unit (*APU*)

Figure 7 shows the details of the **A**pplication **P**rofiling **U**nit (*APU*) that is based on set dueling. Set dueling is a well established mechanism [7, 8] to adaptively choose between two competing policies P0 and P1. In set dueling, a few sampled sets of the cache (we apply set dueling on *MissMap*) are dedicated to always use P0 and other few sampled sets to always use P1. A saturating *k*-bit policy selection counter (counting from 0 to 2^k-1 and initialized with 2^{k-1}) estimates which of the two policies (P0 or P1) leads to a smaller number of misses.



SS$_i$-p_p :Sampled set for $core_i$ using p_p SS$_i$-p_r : Sampled set for $core_i$ using p_p

Figure 7: Application Profiling Unit (APU) as example for a 4-core system

We employ the set dueling mechanism to adaptively choose between two different bank mapping policies p_p and p_r for each core (details in Section 4.3). In our *ABM* policy each set in the *MissMap* uses a policy vector $<p_0, \ldots, p_{n-1}>$, where p_i denotes the bank mapping policy for the new requested segments from $core_i$. To simplify explanation, Figure 7 shows an example for a quad-core system where the sets of the *MissMap* are clustered into groups of 64 sets with 8 "leader sets" and 56 "non-leader sets".

The 8 leader sets are composed of two "sampled sets" per core (grey boxes in Figure 7). A leader set that is a sampled set for $core_i$ is a "follower set" for $\underline{core_{j,j\neq i}}$ (white boxes in Figure 7). The non-sampled sets are follower sets for all cores. The two sampled sets of $core_i$ namely SS_i-p_p and SS_i-p_r use fixed bank mapping policies p_p and p_r. For example, the first set of each cluster always uses the p_p policy for $core_0$ ($p_{0=}p_p$ for this set). The 12-bit saturating policy selection counter $PSEL_i$ for $core_i$ estimates which of the bank mapping policies (p_p or p_r) leads to the smaller number of *MissMap* misses (a miss in the *MissMap* implies an *LLC-DRAM miss*). A *MissMap* miss incurring in the sampled set dedicated for p_p of $core_i$ increments $PSEL_i$ while a *MissMap* miss incurring in the sampled set dedicated for p_r of $core_i$ decrements $PSEL_i$. The $PSEL_i$ counters remain unchanged for misses in the non-leader sets. $PSEL_i$ estimates which of the two policies (p_p or p_r) leads to the smaller number of misses. If the MSB of $PSEL_i$ is '0', then policy p_p is used for all "follower sets", otherwise p_r is used.

4.2 Policy Selection Unit (*PSU*)

The **P**olicy **S**election **U**nit (*PSU*) reads the $PSEL_i$ counters for each competing application to determine the bank mapping policy pf_i (pf_i stands for the policy that is used by the follower sets of $core_i$) at the end of every execution interval (in our case 100 million cycles, see below). If the MSB of $PSEL_i$ is 0, then pf_i is p_p, otherwise p_r. During an interval *j*, the *PSU* determines pf_i which was computed at the end of the previous interval *j-1*. During an interval, the $PSEL_i$ counters are updated for the *MissMap* leader sets on receiving *a MissMap* miss to track application phase changes. It is possible that pf_i rapidly switches between p_p and p_r while not actually converging to a good policy. To mitigate this, we invoke the *PSU* only once every 100 million cycles (interval length) to slow down the policy change rate.

4.3 *ABM* policy

Figure 8 shows the details of our *ABM* policy for a set of *N* cores which organizes *K* DRAM cache banks into two regions namely "private region" and "shared region" as shown in Figure 9. The "private region" contains *N* banks where each $core_i$ has its own dedicated cache bank B_i to which only new segments from $core_i$ are assigned. The "shared region" contains *K-N* cache banks to which segments from any core can be assigned. The division of the cache banks into two regions provides the basic support to implement the two following bank mapping policies.

***Confined-policy*:** assigns a new segment *S* from $core_i$ to the core local bank B_i, i.e. $D_{bank} = B_i$.

***Cooperative-policy*:** For a new segment *S* from $core_i$, D_{bank} is chosen from the shared region in a round robin way.

These policies are used by the "sampled sets" of $core_i$ SS_i-p_p (p_p=*Confined-policy*) and SS_i-p_r (p_r=*Cooperative-policy*). Our *ABM* policy chooses the best per-core policy among p_p and p_r for the "follower sets" of $core_i$ guided by the *PSU* (explained in Section 4.2). It attempts to choose p_p for thrashing applications and p_r for non-thrashing applications based on the run-time miss rate information from the application profiling unit *APU*. This reduces inter-core cache contention between thrashing and non-thrashing applications by mapping segments from a thrashing application to the core local bank.

Each $core_i$ in the *ABM* policy maintains two counters. The first $local\text{-}counter_i$ tracks the number of *MissMap* segments assigned to the local bank B_i of $core_i$. The second $total\text{-}counter_i$ tracks the total number of segment entries residing in the *MissMap* for $core_i$.

When an application running on $core_i$ accesses a new segment *S* currently absent in the *MissMap*, then a new *MissMap* entry *E* is allocated for *S* (line 1 in Figure 8) and $total\text{-}counter_i$ is incremented (line 2). The *ABM* policy assigns a DRAM bank D_{bank} to *S* for $FSet_i$ ($FSet_i$ stands for "follower set" of $core_i$) as illustrated by lines 5-9. The bank mapping policy used by $FSet_i$ is guided by the behavior of the $core_i$ "sampled sets" SS_i-p_p and SS_i-p_r following fixed policies. SS_i-p_p assigns *S* to B_i (use a single cache bank) and SS_i-p_r assigns *S* from the "shared region" in round robin way (to provide cooperative cache sharing).

// New segment ***S*** requested by $core_i$ currently absent in *MissMap*
Input:
MSET: *MissMap Set* determined by set index field in Figure 2
SS_i-p_p: Sampled set for $core_i$ using fixed policy p_p (Figure 6)
SS_i-p_r: Sampled set for $core_i$ using fixed policy p_r (Figure 6)
***$FSet_i$*:** Follower set for $core_i$ (Figure 6)
***pf_i*:** Policy chosen by PSU (see Section 4.2) for the **FSets**
Output: *DRAM cache bank* $\boldsymbol{D_{bank}}$ assigned to ***S*** (Figure 6)
1. Allocate a *MissMap* entry ***E*** for ***S***
2. $total\text{-}counter_i$ ++
3. if (**MSET** ∈ **SS_i-p_p**) {$D_{bank} := B_i$} // *Confined-policy*
4. else if (**MSET** ∈ **SS_i-p_r**) {D_{bank} is chosen from shared region (see Figure 9) in round robin way} // *Cooperative-policy*
5. else { // pf_i is determined for $FSet_i$ by *PSU* in Section 4.2
6. if ($pf_i == p_p$) {$D_{bank} := B_i$ }
7. else if ($local\text{-}counter_i$ / $total\text{-}counter_i$ < 1/(*K* - *N*)) {$D_{bank} := B_i$}
8. else {D_{bank} is chosen from shared region (see Figure 9) in round robin way for $core_i$}
9. }
10. set *Seg-Valid*, update *Seg-Tag* based on address, *Seg-Dirty*, and way-number (Figure 2) field of *MissMap* entry ***E***
11. store D_{bank} in *Seg-Bank* field of *MissMap* entry ***E***
12. store *i* in *Seg-core-id* field of *MissMap* entry ***E***
13. if ($D_{bank} = B_i$) {$local\text{-}counter_i$ ++}

Figure 8: Our *ABM* policy

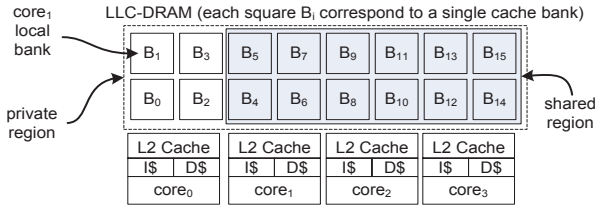


Figure 9: LLC-DRAM cache with N = 4 and K = 16 for the *ABM* policy

As discussed in Section 3.4, thrashing applications typically cause inter-core cache contention with concurrently running applications which causes increased off-chip accesses. Therefore, the *ABM* policy predicts per-core thrashing behavior (using the *PSU* in Section 4.2) by taking into consideration the run-time miss rate behavior of the "sampled sets" (using the application profiling unit *APU* in Section 4.1). If an application running on $core_i$ exhibits thrashing behavior, then for a follower set $FSet_i$ the *ABM* policy assigns segment *S* requested by $core_i$ to B_i (line 6). Confining the new requested segments from the thrashing application to a single cache bank provides **performance isolation** between thrashing and other applications.

If an application running on $core_i$ exhibits non-thrashing behavior, then for a follower set $FSet_i$ the *ABM* policy assigns *S* to B_i, if B_i is under-utilized (line 7), otherwise the *ABM* policy assigns *S* to the "shared region" in a round robin fashion (line 8).

For a non-thrashing application running on $core_i$, the *ABM* policy attempts to balance the segment distribution across B_i and the "shared region" by tracking *MissMap* segments assigned to B_i and "shared region" using $local\text{-}counter_i$ and $total\text{-}counter_i$. This provides efficient utilization of the "shared region", hereby providing **cooperative cache sharing** between non-thrashing applications. When D_{bank} is assigned to *S*, the *Seg-Valid*, *Seg-Tag*, *Seg-Dirty*, and way-number (see Figure 2) field of the allocated *MissMap* entry *E* are updated (line 10). D_{bank} is stored in the *Seg-Bank* field of *MissMap* entry *E* and the *Seg-core-id* field of *E* is set to *i* (lines 11-12). If the D_{bank} is same as B_i, then $local\text{-}counter_i$ is incremented (line 13). Figure 10 illustrates the hardware organization of set selection logic used to identify $SS_i\text{-}p_p$, $SS_i\text{-}p_r$, and $FSet_i$ for $core_i$ used by lines 3-9.

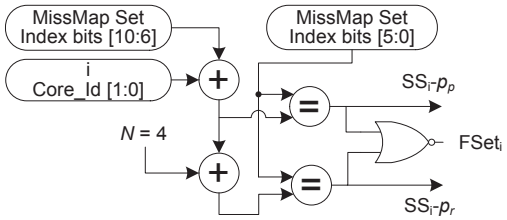


Figure 10: *MissMap* set selection logic for a quad-core system

4.4 Implementation of our *ABM* policy

Figure 11 shows the steps involved in LLC-DRAM lookup operation in our proposed *ABM* policy for a new request from $core_i$, which are explained as follows:

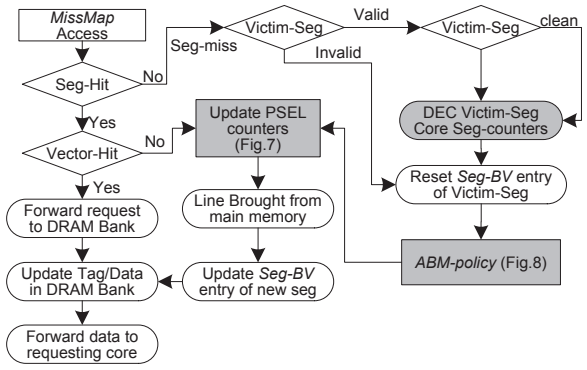


Figure 11: Steps involved in LLC-DRAM lookup operation with *MissMap*. Boxes in shades show our contribution.

***MissMap* Seg-hit/*Seg-BV* hit:** For a *MissMap* hit (i.e. hit in the segment tag and the corresponding bit vector entry as shown in Figure 2) after an L2 miss, the *MissMap* forwards the hit address and the cache bank (determined by the *Seg-Bank* field of the hit segment) to the DRAM access scheduler (Figure 6). When the data is returned from LLC-DRAM, it is forwarded to the requesting $core_i$.

***MissMap* Seg-hit/*Seg-BV* miss:** For a *MissMap* miss (i.e. hit in the segment tag and miss in the corresponding bit vector entry) after an L2 miss, the *MissMap* forwards the miss address and the *Seg-Bank* field to the main memory access scheduler (Figure 6). When the data is returned from memory, it is forwarded to the requesting $core_i$ and the corresponding *Seg-BV* entry is updated (see Figure 11). The $PSEL_i$ counter associated with $core_i$ is incremented for $SS_i\text{-}p_p$ and decremented for $SS_i\text{-}p_r$. The $PSEL_i$ counter remains unchanged for $FSet_i$.

***MissMap* Seg-miss:** For a *MissMap* segment miss (miss in the segment tag) after an L2 miss, a new *MissMap* entry *E* is allocated and a victim segment *V* is chosen using the least recently used policy. If some *Seg-BV* entries of *V* are set, all corresponding cache blocks of *V* resident in the DRAM cache must be evicted. If the Seg-core-id field of *V* is *i*, then $total\text{-}counter_i$ is decremented. If *i* is equal to Seg-Bank field of *V* then $local\text{-}counter_i$ is decremented, otherwise it remains unchanged. The $PSEL_i$ counter associated with $core_i$ is incremented for $SS_i\text{-}p_p$ and decremented for $SS_i\text{-}p_r$ (Figure 7). The $PSEL_i$ counter remains unchanged for $FSet_i$. The D_{bank} field of the new *MissMap* entry *E* is determined by our *ABM* policy illustrated by lines 3-9 in Figure 8.

4.5 Overhead

Table 1 summarizes the hardware overhead for *SLIP* [2, 3, 9, 14], *RRS-policy* [4], *PS-policy* [4], and our *ABM* policy. We employ an 8192-set 16-way *MissMap* structure for *SLIP*, *RRS-policy*, and *PS-policy*. The *ABM* policy needs to store the *Seg-Bank* and *Seg-core-id* field in each *MissMap* entry which requires additional storage. To stay within the same *MissMap* storage budget used by *SLIP*, we employ an 8192-set 15-way *MissMap* structure (i.e. reducing the *MissMap* associativity by one) for our *ABM* policy. On average, our *ABM* policy only suffers 0.37% reduction in performance (harmonic mean instruction per cycle) using a 15-way *MissMap* structure instead of using a 16-way *MissMap* structure. Note that in the result section, we always report the conservative performance results based on the 8192-set 15-way *MissMap* structure.

Table 1: Hardware overhead for a quad-core system with 16 DRAM banks (^{X} shows additional hardware overhead required for our *ABM* policy)

Size of original *MissMap* entry (1 valid bit + 1 dirty bit + 31-bit tag + 4-bit LRU + 4-bit way number + 64-bit *Seg-BV*)	105 bits
MissMap size in bits for *SLIP* [2, 3, 9, 14] [8192 set * 16 entry/set * 105 bits/entry]	1.68 MB
MissMap size in bits for *PS-policy* [4], and *RRS-policy* [4] [8192 set * 16 entry/set (105 bits + 4-bit *Seg-Bank*)/entry]	1.744 MB
MissMap size in bits for our *ABM* policy [8192 set * 15 entry/set * (105 bits + 4-bit *Seg-Bank*+ 2-bit *Seg-core-id*)/entry]	1.665 MB
^{X} 4 12-bit PSEL counters (one per core)	48 bits
^{X} 4 12-bit adders/subtractors (one per core)	4 adders
^{X} 4 1-bit storage to store pf_i (one per core)	4 bits
^{X} 8 17-bit counters for $local\text{-}counter_i$ and $total\text{-}counter_i$	136 bits
^{X} 2 6-bit comparators, 2 6-bit adders, and one NOR gate for the set selection logic	
^{X} 4 round robin bank selector, 8 comparators, 2 adder, 8 shifters, 12 *AND* gates, 12 inverters, 12 multiplexers, and 4 *OR* gate for bank selection logic	for 4 cores

Our *ABM* policy requires negligible additional hardware overhead compared to existing policies as illustrated in Table 1. For each core, the *ABM* policy requires a 12-bit counter for $PSEL_i$, a 12-bit adder/subtractor for incrementing/decrementing the $PSEL_i$ counter, two 17-bit counters (one for $local\text{-}counter_i$ and other for $total\text{-}counter_i$) and 1-bit required to store the value of pf_i. Implementing the *ABM* policy requires the set selection logic (used to identify $SS_i\text{-}p_p$, $SS_i\text{-}p_r$, and $FSet_i$ for $core_i$) and the bank selection logic used by lines 3-9. The set selection logic (Figure 10) requires two 6-bit adders, two 6-bit comparators, and one NOR gate. For each core, the bank selection logic requires the round robin bank selector, two comparators, one adder, two shifters, three *AND* gates, three inverters, three multiplexers, and one *OR*

gate. The set selection and the bank selection logic do not affect the *MissMap* access latency as they are not on the critical path.

5. EXPERIMENTAL SETUP AND RESULTS

We use the SimpleScalar (zesto) simulator [11] to simulate a quad-core system for various multi-programmed workloads from SPEC2006 [1] as shown in Table 3. The core, cache, *MissMap*, and main memory parameters are listed in Table 2. Off-chip memory timing parameters (in nanoseconds) are based on Samsung K4B510446E-ZCH0 (t_{RAS}=45, t_{RCD}=11.25, t_{RP}=11.25, t_{CAS}=11.25, t_{RC}=11.25). DRAM cache timing parameters (in nanoseconds) are based on [9] (t_{RAS}=20, t_{RCD}=6, t_{RP}=6, t_{CAS}=6, t_{RC}=6).

Table 2: Core, Cache, and Main memory parameters

ROB size	128	L1 Cache	32 KB
RS size	32	L2 Cache	256 KB
LDQ size	32	*MissMap* latency	7 cycles
STQ size	24	DRAM cache size	64 MB
Decode width	4	DRAM bus width	128 bits
Commit width	4	DRAM-cache banks	16
Core Frequency	3.2 GHz	Main memory bus width	64 bits

Table 3: Application mixes

Name	Benchmarks
Mix_01	*433.milc, 437.leslie3d.ref, 471.omnetpp, 473.astar.ref*
Mix_02	*401.bzip2, 437.leslie3d.train, 450.soplex, 473.astar.train*
Mix_03	*473.astar.train, 429.mcf, 437.leslie3d.ref, 462.libquantum*
Mix_04	*437.leslie3d.ref, 437.leslie3d.train, 473.astar.ref, 433.milc*
Mix_05	*462.libquantum, 433.milc, 471.omnetpp, 437.leslie3d.train*
Mix_06	*401.bzip2, 462.libquantum, 433.milc, 433.milc*
Mix_07	*470.lbm, 433.milc, 462.libquantum, 401.bzip2*
Mix_08	*429.mcf, 450.soplex, 437.leslie3d.train, 462.libquantum*
Mix_09	*462.libquantum, 471.omnetpp, 473.astar.train, 437.leslie3d.ref*
Mix_10	*471.omnetpp, 473.astar.train, 450.soplex, 462.libquantum*
Mix_11	*473.astar.train, 470.lbm, 471.omnetpp, 437.leslie3d.ref*
Mix_12	*455.gobmk, 471.omnetpp, 429.mcf, 470.lbm*

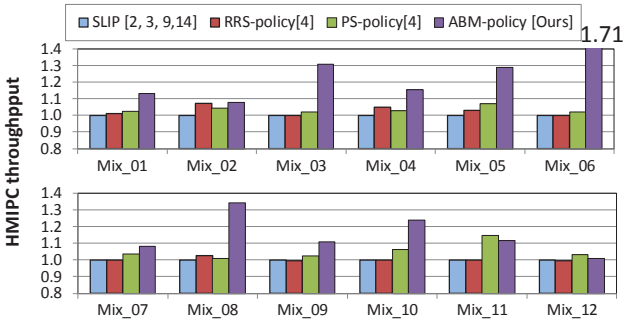


Figure 12: Normalized HMIPC throughput relative to *SLIP* [2, 3, 9, 14]

For evaluation, we have compared our *ABM* policy with state-of-the-art bank mapping policies, namely *SLIP* [2, 3, 9, 14], *RRS-policy* [4], and *PS-policy* [4], which are explained in detail in Section 3.3. Figure 12 shows the HMIPC (harmonic mean instruction per cycle) results for all configurations with the speedup normalized to *SLIP*. On average, our *ABM* policy increases HMIPC by 19.3%, 17.2%, and 14.2% compared to *SLIP*, *RRS-policy*, and *PS-policy*, respectively.

The performance improvement of our *ABM* policy over existing static bank mapping policies is mainly due to reduced off-chip accesses. Figure 13 shows the LLC-DRAM miss rate for all configurations with the miss rate normalized to *SLIP*. On average, our *ABM* policy reduces the LLC-DRAM miss rate by 20.1%, 19.5% and 16.4% compared to *SLIP*, *RRS-policy*, and *PS-policy* respectively.

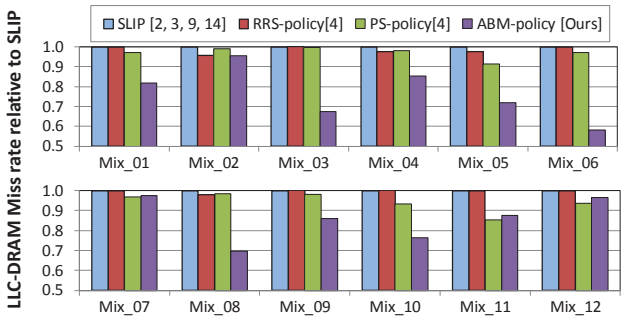


Figure 13: Normalized LLC-DRAM miss rate relative to *SLIP* [2, 3, 9, 14]

Result Analysis: Our *ABM* policy adaptively chooses the bank mapping policy between p_p and p_r depending on the run-time miss rate behavior of concurrently running applications by taking into account the global miss-rate instead of just the local miss-rate. Figure 14 illustrates this by showing the percentage of time each core spent in the p_p policy (confined to a single bank) and p_r policy (cooperative cache sharing) for the *ABM* policy. For example, in Mix_01, *433.milc* (exhibits thrashing behavior) running on $core_0$ (the leftmost of the four bars in group) spent 98.5% of its time in p_p policy and 1.5% of its time in p_r policy for the "follower sets" of $core_0$. We found that the thrashing behavior of some applications depends upon the mix of applications (e.g. *462.libquantum* and *437.leslie3d.ref*). That is, for one application mix a particular application exhibits thrashing behavior (*462.libquantum* in Mix_05, Mix_09 and Mix_10) while for another mix it exhibits non-thrashing behavior (*462.libquantum* in Mix03, Mix_06, Mix_07, and Mix_08). For example in Mix_05, 462.*libquantum* running on $core_0$ exhibits thrashing behavior (it spent 81.7% of its execution time in p_p policy and 19.3% time in p_r policy). However, it exhibits non-thrashing behavior running on $core_1$ in Mix_06 (it spent 12.6% of its execution time in p_p policy and 87.4% time in p_r policy).

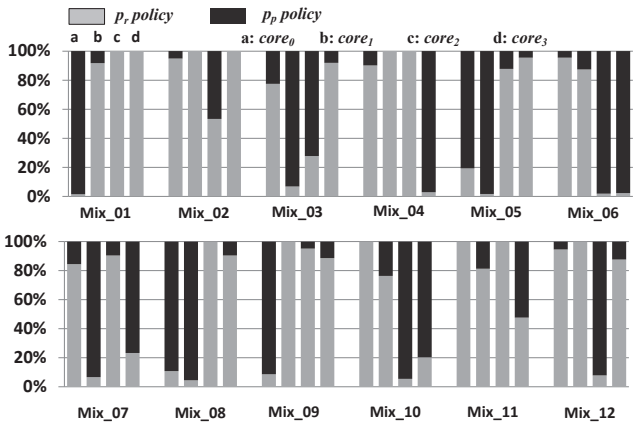


Figure 14: Percentage of time each application running on $core_i$ spent in p_p and p_r for the $FSet_i$ of $core_i$ in the *ABM* policy

Our *ABM* policy reduces inter-core cache contention between applications by isolating thrashing applications to separate cache

banks via the p_p policy. The *ABM* policy exhibits its benefits in situations where thrashing applications run concurrently with cache friendly applications (e.g. Mix_10). In such a case, the thrashing applications (e.g. *462.libquantum* and *450.soplex* in Mix_10) are confined to the local cache banks, thus provide **performance isolation** between thrashing and cache friendly applications. The cache friendly applications (e.g. *471.omnetpp* and *473.astar.train* in Mix_10) achieve dramatic speedup through efficient utilization of the "shared region" (see Figure 9) using **cooperative cache sharing**.

The main drawback of the *SLIP* [2, 3, 9] and *RRS* policies [4] is that they suffer from inter-core cache contention for a mix of thrashing and cache friendly applications (e.g. Mix_01, Mix_03, Mix_05, Mix_06, Mix_08 and Mix_10 etc.). The *PS-policy* [4] provides inter-application performance isolation but suffers from over-utilization of cache resources for *cache fitting* applications (e.g. *462.libquantum*). For example, the local partition (size is 16 MB for a baseline 64 MB DRAM cache in a quad-core system) of the core running *462.libquantum* (see Figure 5) is over-utilized because it requires a huge working set size of 29 MB to perform well. Furthermore, the *PS-policy* suffers from under-utilization of cache resources for *low-intensive* applications (e.g. *445.gobmk*, *473.astar.ref*). For example, the local partition of the core running *445.gobmk* is under-utilized because its working set size is very small compared to the local partition size of 16MB. Our *ABM-policy* reduces under-utilization of cache resources when *low-intensive* applications (e.g. *445.gobmk*, *473.astar.ref*) run concurrently with cache friendly applications via cooperative sharing.

6. Conclusions

Cache management has become more challenging in multi-core systems because of increased inter-core cache contention and limited memory bandwidth. This paper presents the Adaptive Bank Mapping (*ABM*) policy that mitigates inter-core cache contention (via performance isolation) between thrashing and non-thrashing applications and inter-core cooperative cache sharing between cache friendly applications.

We compared our proposed *ABM* policy with state-of-the-art bank mapping policies. On average, our *ABM* policy increases the performance (harmonic mean instruction throughput) by 19.3%, 17.2% and 14.2% compared to *SLIP* [2, 3, 9, 14], *RRS-policy* [4], and *PS-policy* [4], respectively, while requiring negligible additional hardware overhead compared to these policies.

7. Acknowledgement

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).

REFERENCES

[1] Standard Performance Evaluation Corporation. http://-www.spec.org.

[2] R. X. Arroyo, R. J. Harrington, S. P. Hartman, and T. Nguyen. IBM POWER7 Systems. *IBM Journal of Research and Development*, 55(3):2:1 – 2:13, 2011.

[3] B. Black, M. Annavaram, N. Brekelbaum, J. DeVale, and L. Jiang. Die-Stacking (3D) Microarchitecture. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–479, December 2006.

[4] S. Cho and L. Jin. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 455–468, December 2006.

[5] F. Hameed, L. Bauer, and J. Henkel. Dynamic Cache Management in Multi-Core Architectures through Run-time Adaptation. In *Proceedings of the 14th conference on Design, Automation and Test in Europe (DATE)*, pages 485–490, March 2012.

[6] F. Hameed, L. Bauer, and J. Henkel. Adaptive Cache Management for a Combined SRAM and DRAM Cache Hierarchy for Multi-Cores. In *Proceedings of the 15th conference on Design, Automation and Test in Europe (DATE)*, pages 77–82, March 2013.

[7] A. Jaleel, W. Hasenplaugh, M.K. Qureshi, J. Sebot, S. Steely Jr., and J. Emer. Adaptive Insertion Policies for Managing Shared Caches. In *Int'l Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 208–219, 2008.

[8] Gabriel H. Loh. Extending the Effectiveness of 3D–stacked Dram Caches with an Adaptive Multi-Queue Policy. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 174–183, 2009.

[9] Gabriel H. Loh and Mark D. Hill. Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 454–464, 2011.

[10] Gabriel H. Loh and Mark D. Hill. Supporting Very Large DRAM Caches with Compound Access Scheduling and MissMaps. In *IEEE Micro Magazine, Special Issue on Top Picks in Computer Architecture Conferences*, 2012.

[11] Gabriel H. Loh, S. Subramaniam, and Y. Xie. Zesto: A Cycle-Level Simulator for Highly Detailed Microarchitecture Exploration. In *Int'l Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[12] M. K. Qureshi and Y. N. Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-performance, Runtime mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 423–432, 2006.

[13] B.M. Rogers, A. Krishna., G.B. Bell, K. Vu, X. Jiang, and Y. Solihin. Scaling the Bandwidth Wall: Challenges in and Avenues for CMP Scaling. *SIGARCH Computer Architecture News*, 37(3):371–382, 2009.

[14] M. Shah, J. Barreh, J. Brooks, R. Golla, G. Grohoski, R. Hetherington, P. Jordan, M. Luttrel, O. Chistopher, B. Saha, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A Highly-Threaded, PowerEfficient, SPARC SOC. In *IEEE Asian Solid State Circuit Conference*, pages 22–25, 2007.

[15] D. Wendel, R. Kalla, R. Cargoni, J. Clables, J. Friedrich, R. Frech, J. Kahle, B. Sinharoy, W. Starke, S. Taylor, S. Weitzel, S.G. Chu, S. Islam, and V. Zyuban. The Implementation of Power7TM: A Highly Parallel and Scalable Multi-core High-end Server Processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 102–103, February 2010.

[16] Y. Xie and G. H. Loh. Dynamic Classification of Program Memory Behaviors in CMPs. In *2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, June 2008.

[17] Y. Xie and G. H. Loh. PIPP: Promotion/Insertion Pseudo-Partitioning of Multi-Core Shared Caches. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA)*, pages 174–183, 2009.