Journal of Systems Architecture 55 (2009) 446–456

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Reusability-aware cache memory sharing for chip multiprocessors with private L2 caches

Hyunhee Kim [a], Sungjun Youn [b], Jihong Kim [a,*]

[a] School of Computer Science and Engineering, Seoul National University, San 56-1, Shinlim-9dong, Gwanak-gu, Seoul, 151-742, Republic of Korea
[b] LG Electronics Corporation, Seoul 152-702, Republic of Korea

ARTICLE INFO

Article history:
Received 3 December 2008
Received in revised form 22 August 2009
Accepted 15 September 2009
Available online 18 September 2009

Keywords:
CMPs
Private L2 Cache
Reusability
Cache management

ABSTRACT

In this paper, we propose a novel on-chip L2 cache organization for chip multiprocessors (CMPs) with private L2 caches. The proposed approach, called reusability-aware cache sharing (RACS), combines the advantages of both a private L2 cache and a shared L2 cache. Since a private L2 cache organization has a short access latency, the RACS scheme employs a private L2 cache organization. However, when a cache block in a private L2 cache is selected for eviction, RACS first evaluates its reusability. If the block is likely to be reused in the near future, it may be saved to a peer L2 cache which has space available. In this way, the RACS scheme effectively simulates the larger capacity of a shared L2 cache. Simulation results show that RACS reduced the number of off-chip memory accesses by 24% compared to a pure private L2 cache organization on average for the SPLASH 2 multi-threaded benchmarks, and by 16% for multi-programmed benchmarks.

1. Introduction

Chip multiprocessors (CMPs) are rapidly emerging as an alternative architecture for high-end embedded systems, providing high performance and low power consumption. Efficient management of the on-chip memory hierarchy is important for these CMPs to achieve their maximum performance potential, since on-chip cache memory space is limited and off-chip memory accesses take much longer than on-chip memory accesses. Therefore, reducing the number of off-chip memory accesses by carefully managing the on-chip cache space can improve the overall system performance [1,2].

Most CMPs have several levels of on-chip cache memory which can be accessed more quickly than off-chip memory. Typically, L1 caches are designed in a small size because the access latency of the L1 cache affects system performance directly. On the other hand, a second level of large cache can be arranged in two ways, either all the processors share a single L2 cache or each processor has its own private L2 cache.

A shared L2 cache has the advantage of using cache space more flexibly because data blocks do not have to be replicated and cache is relatively large, which can reduce the number of off-chip memory accesses. However, increasing the size of the cache leads to larger access latency and energy consumption. This arrangement also causes more bus contentions than a private L2 cache because of a shared bus. On the other hand, data in a private L2 cache can be accessed more quickly because it is closer to the processor and relatively small. Private caches, however, use space inefficiently because data must often be replicated in several private L2 caches. In addition, private L2 caches of equal size are a poor match for applications with different-sized working sets. When a working set is smaller than the private cache size, the cache space is not fully utilized; the working set is larger than the cache size, only part of it can be accommodated.

In this paper, we propose an on-chip L2 cache organization which combines the advantages of both a private L2 cache and a shared L2 cache for CMPs. The proposed L2 cache organization is based on a private L2 cache with low-latency as shown in Fig. 1 but writes blocks from a local private L2 cache to a peer L2 cache which is a private L2 cache of the other processor when they are evicted to aggregate the private L2 caches. Writing evicted blocks to a peer L2 cache has already been proposed in CMP_CC [6]. However, this scheme does not consider the reusability of evicted blocks and writes the evicted block to the peer L2 cache with a given probability which is not change dynamically depending on the workloads behavior. Our contribution is to consider the reusability of the evicted blocks in deciding whether they will be saved in the peer L2 caches and dynamically adjust the number of writes to the peer L2 cache depending on the concurrently running workloads.

In the proposed scheme, when a cache block in the private L2 cache is selected for eviction, the reusability of that block is evaluated. If the block is predicted to be reused, it is saved in the private

* Corresponding author. Tel.: +82 2 880 8792; fax: +82 2 871 4912.
E-mail addresses: hh0726@davinci.snu.ac.kr (H. Kim), spica81@davinci.snu.ac.kr (S. Youn), jihong@davinci.snu.ac.kr (J. Kim).

1383-7621/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2009.09.003

L2 cache of other processors, replacing blocks which are not reusable. Since it is faster to access the L2 cache of a nearby processor than the off-chip memory, saving blocks that are likely to be reused in a peer L2 cache will improve performance. In addition, the proposed scheme dynamically adjusts the number of evicted blocks written to the peer L2 caches considering the characteristics of the concurrently running workloads, such as a size of working set and data reuse ratio. When the large number of evicted blocks is written to the peer L2 caches and degrades the performance by polluting them, the proposed scheme decides the reusability of a block more conservatively. On the other hand, if there is an available space in the peer L2 caches and it could receive the evicted blocks without the performance loss, the proposed scheme decides the reusability of a block more aggressively and selects a destination cache where the evicted blocks are written. In effect, this arrangement simulates a shared L2 cache organization more efficiently. By considering the reusability of cache blocks and dynamically adjusting the number of evicted blocks written to the peer L2 cache, the proposed technique can always achieve better performance improvement by up to 4.8% and 17.4% over the best and the worst CMP_CC probability setting, respectively.

The rest of the paper is organized as follows. Several previous research works on a cache management technique for CMPs are introduced in Section 2. In Section 3 we explain the motivation of our idea. In Section 4 we explain the proposed reusability-aware cache sharing technique. Our experimental results are presented in Section 5. Finally, Section 6 summarizes our paper and suggests future work.

2. Related work

There have been many research works which proposed different on-chip cache organizations in order to use on-chip cache memory space more efficiently in CMPs. Existing techniques such as CMP-SNUCA [3], victim replication [4], CMP-NuRAPID [5] and CMP_CC [6] all aimed to combine the low latency of the private L2 cache with the low miss-rate of the shared L2 cache. The CMP-SNUCA [3] applies a non-uniform cache structure [9] to the CMPs architecture and migrates data blocks close to the requesting processor, so as to reduce the wire-delay. The victim replication [4] scheme attempts to keep copies of data evicted from the L1 caches within a local slice of the L2 cache, so as to reduce the wire-delay in the shared L2 cache. The CMP-NuRAPID [5] scheme copies data close to the requesting processor to allow fast access for read-only sharing, but not for read-write sharing, in order to avoid coherence misses. It also includes a method of stealing capacity from a neighbor's cache when a processor's private cache space is not large enough.

CMP_CC [6] redistributes private L2 cache space by randomly writing evicted blocks from a local L2 cache to a peer L2 cache with a given probability from 0% to 100%. It also offers a single control point, called cooperation throttling, which uses the probability when deciding whether the evicted block is written to the peer cache or not. However, it does not propose a mechanism to dynamically adjust the probability even though the performance improvement changes depending on workload behavior. A technique proposed in [10] similarly redistributes evicted blocks (as done in CMP_CC) but its redistribution decisions require stronger conditions for evicted blocks. It only writes evicted blocks from the local L2 cache to the peer L2 cache only when there is invalidated or shared line. Unlike our proposed approach, these schemes do not consider the reusability of an on-chip cache block.

The earlier version of the RACS technique proposed in [7,8] considers the reusability of an on-chip cache block only when deciding whether the block should be kept on an on-chip cache or not. However, the technique described in this paper has significantly improved the decision procedures for determining the reusability of an on-chip cache block and the destination cache where the evicted block should be written. In the extended technique, the criterion of deciding the reusability of a block is dynamically adapted by dynamically adapting the threshold value, unlike in the earlier version of RACS where the constant threshold value was used for the reusability check. Adapting the threshold value dynamically depending on the changing execution environment allows more cache blocks to be reused. We also improved the procedure for deciding the destination cache by considering the working set size and data locality of the destination cache in the OS level, unlike the earlier version of RACS where the selection of the destination cache was decided by purely in the hardware level. Taking into accounts of OS level information (e.g. data locality of a cache), the extended technique outperforms the earlier version by choosing the destination cache which has the least performance degradation from accepting the evicted block.

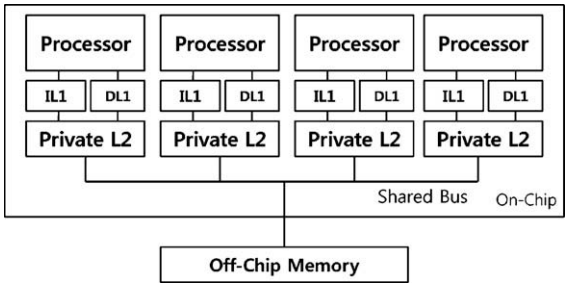

Fig. 1. A CMP architecture with a private L2 cache and shared bus.

3. Motivation

CMP_CC [6] transfers evicted blocks to peer L2 caches with a given probability which is decided before an execution and does not change through the execution. However, this technique does not identify whether the evicted block is likely to be reused or not in the near future. Fig. 2 shows how many of the blocks written to peer L2 caches by the CMP_CC with a given 100% probability, which writes all of the evicted blocks to the randomly selected peer L2 cache, are reused and how many are not, for the SPLASH 2 benchmarks [13]. As can be seen, for most benchmarks, the majority of evicted blocks are not reused. When the large number of the blocks is not reused even though they are written to other cache, the performance of a system can be damaged because they may pollute a peer L2 cache space unnecessarily and generate additional transactions which cause conflicts on an on-chip shared

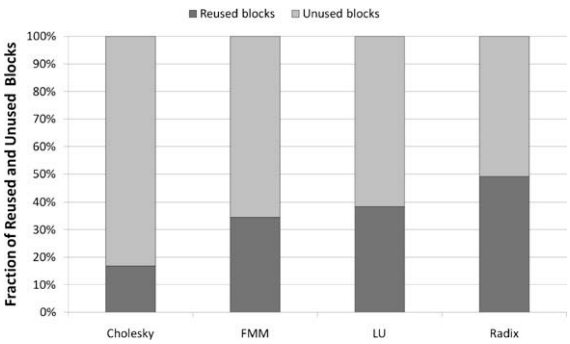

Fig. 2. The number of unused blocks and reused blocks under CMP_CC with 100% probability.

bus. This means that an adaptive writing is necessary depending on the reusability of evicted L2 blocks.

As well as the reusability of blocks, selecting the cache where the evicted L2 block is written, a destination cache, is important because accepting a peer cache's block might pollute its own cache space and degrade its performance. However, if the space of the peer L2 cache is available, more evicted blocks with reusability could be transferred. In this paper, an available space in an L2 cache refers to the space where evicted blocks from peer L2 caches could be written without performance decrease. In this paper, we identify the available space in the peer L2 cache and use it as the destination cache.

Fig. 3 presents how instructions per cycle (IPC) for the SPEC2000 benchmarks improves as the size of the L2 cache is increased by increasing the number of ways from 1-way to 8-way. It shows which programs can have the available space for the peer caches. As can be seen, some programs, such as *mcf* and *gap*, are insensitive to the size of the L2 cache, which means that the corresponding cache can provide their space to other programs without affecting their own performance. If we exploit this available space effectively, we can keep more reusable blocks on the chip without any significant reduction in overall performance.

Therefore, this paper proposes reusability-aware cache sharing (RACS) Technique, which predicts the reusability of a block and selects a peer L2 cache to which an evicted block is transferred. In the next section, the proposed technique, RACS, is described in detail.

4. Reusability-aware cache sharing technique

In this section, we explain the proposed RACS technique in more details. RACS consists of two techniques, one for predicting the reusability of an on-chip cache block and one for selecting the cache with an available space.

4.1. Predicting block reusability

4.1.1. Access time interval and frequency (ATIF) pattern

Typical replacement algorithms such as lest recently used (LRU) and most recently used (MRU) exploit a recency to keep blocks in the cache which are likely to be reused in the near future. On the other hand, LRFU [11], a cache or a buffer replacement policy, exploits the frequency of accesses or its recency to predict its reusability. However, in the proposed scheme, it cannot exploit the recency of data because it should consider the reusability of a block after it is evicted from a local private L2 cache. Instead, it considers the frequency of accesses to a block and a time interval between consecutive accesses.

It is observed that if a block is accessed frequently with a short time interval but rarely accessed with a long time interval, it usually is not reused after the eviction because it has only temporal locality. On the other hand, in most cases, if a block is rarely accessed with a short time interval but frequently accessed with a long time interval, it is likely to be reused after the eviction. Based on this observation, RACS classifies blocks by access time interval and frequency (ATIF) pattern and then decides the reusability of the blocks according to the reuse ratio of each pattern by monitoring how many blocks of each pattern written to the peer L2 cache are actually reused and how many are unused. A level of reusability is thus associated with each pattern.

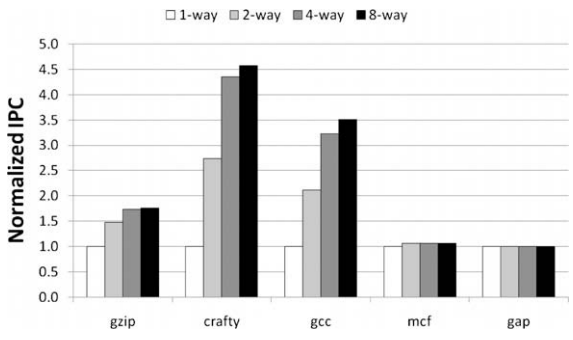

Fig. 3. IPC variation with a cache size.

The ATIF pattern associated with a block is determined by the number of accesses that occur after short or long time intervals while the block is in a private L2 cache. We distinguish quite simply between a short and a long time interval: the interval is long if there is an intervening access to any block to the same set. Otherwise, the interval is considered to be short. We use a 4-bit counter to record the number of accesses to each block with a short time interval and a 2-bit counter to record the number of accesses with a long time interval. We use a wider counter for the short time interval counters because there are usually more accesses with a short time interval than the accesses with a long time interval because of the temporal locality. The ATIF pattern of a block is then determined from these two counts when the block is evicted from the private L2 cache. The blocks are classified into 16 ATIF patterns using the upper 2 bits of the 4-bit counter and the whole width of the 2-bit counter. In order to record the reuse ratio of blocks in each ATIF pattern, we add 16 counters to each private L2 cache.

If a block written to the peer L2 cache is subsequently reused, then the corresponding ATIF counter in the L2 cache from which the block originated is incremented by one. However, if the block is evicted from the peer L2 cache without reuse, the ATIF counter is decremented by one. When the ATIF pattern counter becomes zero, we predict that the blocks which belong to the ATIF pattern have very low reusability. Thus, the pattern counters provide ongoing estimates of reusability for each pattern. The initial value for each ATIF counter is half of its maximum value to allow evicted blocks to be written to the peer L2 cache during the start time of the execution. During the start time, all of the evicted blocks are written to the peer L2 cache because the value of the ATIF counter is not zero. However, if many of written blocks are evicted without a reuse during the execution, the value of the counters becomes zero, which prohibit evicted blocks from being written to peer L2 caches. On the other hand, if many of the written blocks are evicted after a reuse, the value of the counters remains non-zero, which leads evicted blocks to be written to peer L2 cache continuously. While the increments of a short or long interval counter should be performed every time a hit occurs in the cache line, it could be performed in parallel with the cache access because the hit could be known earlier after completing only tag lookups. Once the tag lookups is completed, the counter of the corresponding hit block can be incremented without increasing the L2 cache hit time.

Fig. 4 shows what proportion of the blocks written to the peer L2 caches by CMP_CC with the probability 100% is reused for each ATIF pattern. *X*-axis represents the 16 ATIF patterns. The first number of each pattern represents the upper 2-bit value of the short time interval counter and the second number represents the 2-bit value of the long time interval counter. In most cases, ATIF patterns that correspond to a lot of accesses with a long time interval, such as (13), (22), (23), and (33), have the relatively larger number of reused blocks. Also, when the first number of the ATIF pattern, corresponding to the number of accesses after short time intervals, is zero, like (00), (01), (02), and (03), the blocks do not have a temporal locality but many of them are reused after the eviction. Consequently, ATIF patterns used in RACS can identify blocks with high reusability.

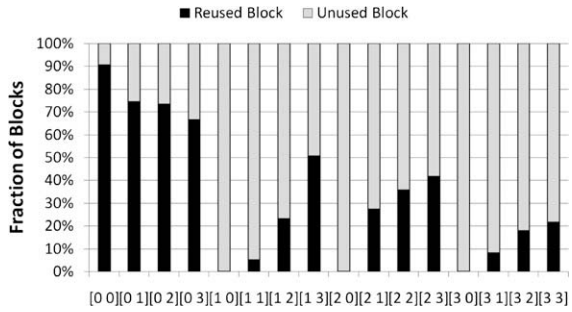

Fig. 4. Distributions of reused blocks and unused blocks under different 16 ATIF patterns.

4.1.2. Dynamic prediction of block reusability

The accuracy of our reusability prediction depends on the size of the pattern counters. A longer counter predicts that more blocks are reusable, because a small counter becomes zero so quickly that reusability of the block may be unnecessarily underestimated. Fig. 5 shows the normalized IPC for a private L2 cache scheme, with different numbers of bits in the ATIF counters. Each of four processors is running a different program; *parser*, *gcc*, *gzip*, and *twolf*. As shown in Fig. 5, the 8-bit counter gives the best performance. On the other hand, when using the 10-bit counter, IPC decreases because too many unused blocks are allowed to escape eviction. This indicates that dynamic technique for predicting block reusability is likely to be preferable.

For example, when a peer L2 cache has no available space to save evicted blocks, a more conservative prediction can improve performance. In this case, we can avoid polluting the other cache and generating bus traffic by retaining only the blocks with high reusability. Conversely, if there is available space in the peer cache, classifying more blocks as reusable can improve the performance. We, therefore, use a reusability threshold at each cache to adjust a local prediction of the reusability of the blocks dynamically.

We use ATIF counters which, unregulated, will predict an excessive number of blocks to be reusable. This is fine if cache space in peer L2 caches is available, but the reusability threshold is used to produce a more conservative estimate of the reusability when other caches have no space to keep evicted blocks. To determine the reusability of a block, we compare the corresponding ATIF counter with the local reusability threshold. If the value of the ATIF pattern counter is larger than the reusability, threshold the block is classified as reusable. Therefore, decreasing the reusability threshold means that more blocks are classified as reusable. Initially, the reusability threshold is set to half the maximum value of the counter. If there is available space in the peer caches, then the reusability threshold is decreased, so as to predict the reusability of the blocks more aggressively. However, if no space is available, its value is increased.

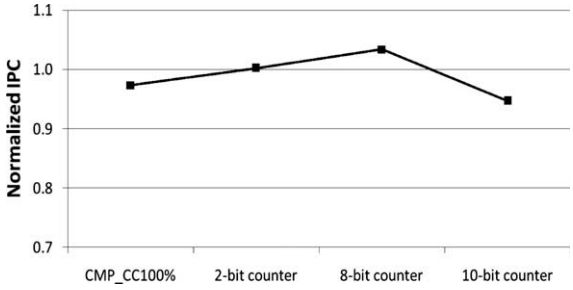

Fig. 5. Normalized IPC with different numbers of bits in the ATIF counters.

4.2. Cache selection technique

When an evicted block is to be written to a peer L2 cache, we have to select a destination peer L2 cache carefully. For example, as described in Section 3, if we write the evicted L2 block to the cache running *gcc*, its performance might be seriously affected because it deteriorates as the size of the cache decreases. Instead, it is better to send the evicted block to an another peer cache which has space available. In this case, the cache running *mcf* or *gap* is the best destination because it is characterized by low reusability and not significantly affected by a reduction in the size of the cache. Even though predicting reusability dynamically can prevent the evicted blocks with low reusability from being written to peer L2 caches, we need to get more on-chip space to keep more blocks evicted from other caches if there is the peer L2 cache whose performance is insensitive to the size of the cache. This allows to exploit the on-chip cache space more aggressively and improve the overall performance.

We propose a simple heuristic. If the performance of a program will not be affected by accepting the evicted block from a peer L2 cache, the more evicted blocks are allowed to be written to this cache. We call this sort of cache a destination cache. There are two cases in which evicted blocks will be saved in a destination cache: either the program running on the cache has a low level of data reuse, therefore reducing the cache space will not affect its performance; or the program has a very small working set, therefore it never fills the cache. we find caches which meet one or both of these condition at run-time and write evicted blocks to them. A cache being used by a program with a low level of data reuse is considered first because its performance is less dependent on available cache space than a program with a small working set.

Table 1 shows proposed cache selection heuristic, which chooses destination cache, *dst_cache*, at which there is a low level of data reuse, and adapts the reusability threshold, *reusability_threshold*, dynamically. This heuristic is executed through an

Table 1
Heuristic to select the cache.

```
/* Initialize*/
selection_threshold = MAX_COUNTER * NUM_OF_COUNTERS/2;
reusability_threshold [] for all the program = MAX_COUNTER/2;
/* for each period, select the destination cache */
for each period
    for each i in the dst_cadidate{} for the previous period
        if (IPC_history(prev)[i] − IPC_cur[i] < 0) selection_threshold *= 2;
        else selection_threshold/= 2;
    end for
    for each program i running concurrently
        pattern_sum[i] = sum of the 16 pattern counters;
        if (pattern_sum[i] ⩽ selection_threshold)
            dst_candidate{} += i;
        else (pattern_sum[i] > selection_threshold)
            src_candidate{} += i;
        end if
    end for
    while (src_candidate{} and dst_candidate{} are not empty)
        src_cache = the cache with the maximal pattern_sum
        in src_candidate{};
        src_candidate{} -= src_cache;
        /* to predict the reusability aggressively */
        reusability_threshold [src_cache]/= 2;
        dst_cache = the cache with minimal pattern_sum
        in dst_candidate{};
        dst_candidate{} -= dst_cache;
        /* to predict the reusability conservatively*/
        reusability_threshold [dst_cache] *= 2;
        shared_cache[src_cache] = dst_cache;
    end while
end for
```

OS level daemon periodically. At every time period, the OS daemon reads the information needed for the decision and notifies all the caches of its decision, namely *dst_cache* and *reusability_threshold*.

In our cache selection heuristic, a shared cache register, *shared_cache[i]*, stores the destination cache number at each cache. If a cache does not have a destination cache, the register holds its own cache number. As well as the reusability threshold, we use a selection threshold, *selection_threshold*, to identify programs whose IPC does not depend on the space of the cache. At each cache, a variable *pattern_sum*, which represents the data reuse characteristics of the program, is used to decide the destination cache. If the sum of the 16 ATIF pattern counters, *pattern_sum*, at a cache is equal or less than the *selection_threshold*, we consider it as a candidate for the destination cache, *dst_candidate{}*. Therefore, if we increase *selection_threshold*, more caches are considered as the candidate destination cache for the next period. However, if we select a program that is sensitive to the amount of space remaining in its cache as a destination cache due to a high *selection_threshold*, the performance of that program might be degraded. We therefore adapt *selection_threshold* dynamically using feedback from the performance during the previous period.

Therefore, at the end of each period, we check the IPC of each program. If the IPC of the cache selected as the destination cache for the current period is less than it was in the previous period, it indicates that too many blocks are being written to that cache. We then halve the *selection_threshold* by two times to select candidates for the destination cache more conservatively. When we decide whether the IPCs has improved or not, we do not use the IPC for the previous period, because it might be affected by program phase change. Instead we use $IPC_{history}$, which is averaged over the recent history of program execution as follows.

$$IPC_{history(new)} = \frac{IPC_{history(prev)} \times 3 + IPC_{cur}}{4} \quad (1)$$

IPC_{cur} is the IPC for the current period. If IPC_{cur} is smaller than $IPC_{history}$, it implies that the wrong destination cache is being used, and we decrease the *selection_threshold*. Conversely, a value of IPC_{cur} that is the same as or larger $IPC_{history}$, suggests that the program is maintaining its performance even though the cache has received blocks evicted from peer L2 caches. The *selection_threshold* is initialized to half of *MAX_COUNTER*NUM_OF_COUNTERS* to start from the middle of the sum of the 16 ATIF pattern counters.

As explained in Section 4.1.2, the reusability threshold is also used for dynamic reusability prediction. Each of the *i* caches has a local reusability threshold, *reusability_threshold [i]*. By adapting this threshold, we can predict reusability of a block dynamically to reflect the availability of space in peer L2 cache. After the *dst_candidate{}* and *src_candidate{}* lists have been created, the source and destination caches, *src_cache* and *dst_cache*, are determined. The source cache is the cache with the largest sum of pattern values for all the caches in the *src_candidate{}* because this cache can be assumed to have the largest number of reusable blocks. Destination cache has the smallest sum of pattern values for all caches in *dst_candidate{}* because this cache can be assumed to have the fewest reusable blocks, so that its performance is likely to be least affected by receiving evicted blocks. At the end of period, the *shared_cache[i]* register of the source cache is set to the index of the destination cache according to the result of cache selection heuristic.

If the cache selection heuristic cannot find the destination cache with a low level of data reuse, we select the destination cache by considering the size of the working set at each cache. The size of the working set is inferred from the memory demand, because a processor can be expected to require more memory when replacement occurs in its private L2 cache. We use a replacement time interval $Repl_{history}$ as a measure of a processor's memory demand at each private L2 cache, and this value is updated every time replacement occurs, as follows:

$$Repl_{history(new)} = \frac{Repl_{history(prev)} \times 3 + Repl_{interval}}{4} \quad (2)$$

$Repl_{history(prev)}$ is the previous prediction of memory demand and $Repl_{history(new)}$ is the latest value. $Repl_{interval}$ is the time interval between the last two consecutive replacements. This average over time is used because it is not affected directly by a change of program phase. If an L2 cache has a small $Repl_{history}$ its processor requires more memory. This value is only used to compare the memory demand between L2 and does not quantify the memory demand. To calculate $Repl_{interval}$, we use a 8-bit counter to measure the time since the last replacement. This counter is incremented by one every 32 processor cycles, and reset to 0 when replacement occurs.

4.3. Procedure of the RACS technique

Fig. 6 summarizes the RACS steps in processing an evicted L2 cache block α.

1. A block that is evicted from a private L2 cache is not written to any peer L2 cache (a) if the state of that block is shared, because it means that the same block is present in another L2 cache; (b) if the block was transferred from an other L2 cache but was not reused while residing in the peer cache, because such blocks have already had a chance to be reused; (c) if the reusability of the block is low, which is determined by comparing corresponding ATIF pattern counter to the reusability threshold of its own cache.
2. If a block is still eligible for transfer, we look for a block with low reusability at the bottom of the LRU stack in one of the peer L2 caches. If such a block exists, we write the candidate block to the peer L2 cache.
3. Otherwise, we check the destination cache register first. If a destination cache has been determined by the cache selection heuristic, the evicted block is written to it.
4. If cache does not have a destination cache, RACS decides whether the victim block will remain on-chip from the memory demand and reuse ratio. Writing to a peer L2 cache does not cause a subsequent write to the other peer L2 cache to avoid a ripple effect.

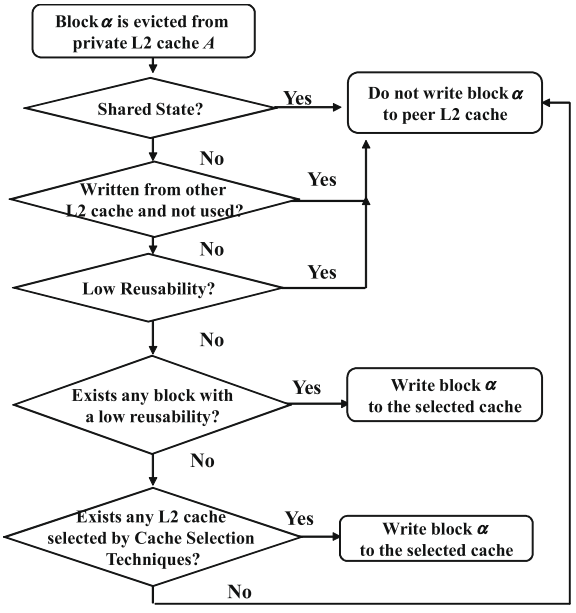

Fig. 6. Overall procedure of the RACS technique.

The process of deciding where to write victim blocks requires communication, and we assume that the necessary peer-to-peer communication lines are present. If a block has a high reusability, its cache sends the set number of the block and the value of $Repl_{history}$ to all the peer L2 caches. These caches send two bits of information on reply: one bit indicates whether the cache has a block with low reusability at the bottom of its LRU stack; and the other bit indicates whether the value of $Repl_{history}$ for that cache is larger than the broadcast value of $Repl_{history}$. The block is written to the peer L2 cache which sends an appropriate response. This process could be implemented in the L2 controller. Although there are several steps to decide if the evicted block is written to the peer cache, the cost of each step is small because most of the steps only require bit check and the first three steps could be processed in parallel. Furthermore, it should be noted that the decision whether to write a block to another cache is not on the critical path because it can be made after the block is evicted from its original cache and placed in the write queue.

4.4. Overhead

The RACS scheme has hardware overhead compared to a pure private L2 cache organization because it requires additional counters for the two prediction schemes and peer-to-peer communication lines between the L2 caches. Predicting reusability involves a 4-bit counter and a 2-bit counter at each block, to record the number of accesses with long and short time intervals, respectively. An additional 2 bits are required for each set to distinguish between long and short time interval accesses. These bits record the most recently accessed block of each set. In addition, each block needs 2 bits to indicate which processor writes it and a further bit indicates whether the block has been reused or not. For each private L2 cache, we also need 16 10-bit pattern counters, a 10-bit reusability threshold and a 2-bit destination cache register. To predict the memory demand, a 8-bit counter is used to record the time from the last replacement and another 8-bit counter records the replacement history. This comes to a total of 9 bits per block, 2 bits per set, and 188 bits per cache. For 256 KB private L2 cache with 128 B block size, there are 4096 blocks and 512 sets in each cache. The total memory overhead becomes 5 KB for each cache, which is less than 2% of each private L2 cache, therefore is negligible.

Furthermore, it is only necessary for OS to execute the cache selection heuristic every 2,000,000 cycles. This period long enough to mitigate the overhead incurred by the cache selection heuristic while achieving the performance improvement from the heuristic. When the migration occurs to balance the loads, the value of the 16 ATIF counters should be stored in the process data structure of OS and restored in the ATIF counters of the new node. OS should keep other information, for example, a *selection threshold* and an array of *the reusability threshold*, used in the cache selection heuristic, which may be used after migration to identify the reusability of each pattern and decide the destination cache. The overhead to keep the required data when the migration occurs is not large.

5. Performance evaluation

5.1. Simulation environment

We modify CATS multiprocessor simulator [12] to evaluate the proposed technique and support a cache-to-cache transfer based on a MESI protocol [14] for cache coherency. In this MESI protocol, a read miss occurred in a local L2 cache causes a read transaction on a bus, which broadcasts a block address and a read transaction signal to other L2 caches and memory. Since multiple processors may have a copy of the memory block in their cache, only one to supply the data on the bus needs to be selected. If no other cache has the requested data, the memory supplies the data. We implemented this cache-to-cache transfer in our simulator considering the overheads. On the other hand, it is also possible for both local L2 accesses and remote L2 queries from different processors to happen at the same time, causing contentions for cache tag accesses. In order to solve this potential performance hazard, we employ a dual-tag system for L2 cache which was also used in the previous work such as CMP_CC. Therefore, the local access can be served without interference with remote L2 queries.

Table 2 shows simulation parameters for the processor, cache, and memory configuration. We evaluated our scheme by varying the cache size from 128 KB to 512 KB to explore sensitivity to cache size. We implemented and evaluated the following schemes: private L2 caches, CMP_CC with 30%, 70% and 100% probability, RACS. Table 3 shows the benchmarks used for the evaluation. These are four multi-threaded benchmarks selected from SPLASH 2 and four combinations of multi-programmed benchmarks each consisting of four programs of SPEC2000.

We selected four programs with the different locality and the number of cache accesses from SPLASH 2 programs. FMM requires a larger number of L1 cache accesses compared to RADIX while FMM has a higher temporal locality. On the other hand, LU and Cholesky are similar benchmarks but LU has a higher level of temporal locality. Even though these multi-threaded benchmarks from SPLASH 2 show the different characteristics, the working sets of the programs mostly fit in the cache and the behavior of each thread in these benchmarks is not much different. Therefore, the selected programs are adequate to demonstrate the performance improvement of the proposed scheme because the performance can be improved mainly from identifying the reusability of the block, which can reduce the bus traffic and pollution of the other cache.

For the multi-programmed workloads, we select the programs from SPEC benchmarks which have different characteristics of how the performance varies depending on the cache size, which could be classified into two categories. First group includes mcf and gap whose performance does not have benefit as the cache size is increased. mcf has a large number of cold misses and mcf has a working set larger than the cache size. Second group includes other programs whose performance is significantly affected by the cache

Table 2
Simulation parameters.

Parameter	Value
Number of processors	4
Processor model	in-order
Processor issue width	4
Number of ALUs	2
Branch predictor	2-level, hybrid, 8 K entries
L1 D-Cache	16 KB, 1-way, 32 B block, 1 cycle
L1 I-Cache	16 KB, 1-way, 32 B block, 1 cycle
L2 Private cache	256 KB, 4-way, 128 B block, 6 cycle
Interconnect	Shared bus, 4 bytes bus width, pipelined
Off-chip memory	500 cycle access latency

Table 3
Multi-threaded and multi-programmed benchmarks.

Multi-threaded	*Cholesky, FMM, LU, Radix*	
Multi-programmed	MultiProg1	*gap, parser, gcc, crafty*
	MultiProg2	*parser, gcc, gzip, twolf*
	MultiProg3	*bzip, parser, gcc, mcf*
	MultiProg4	*parser, twolf, mcf, crafty*

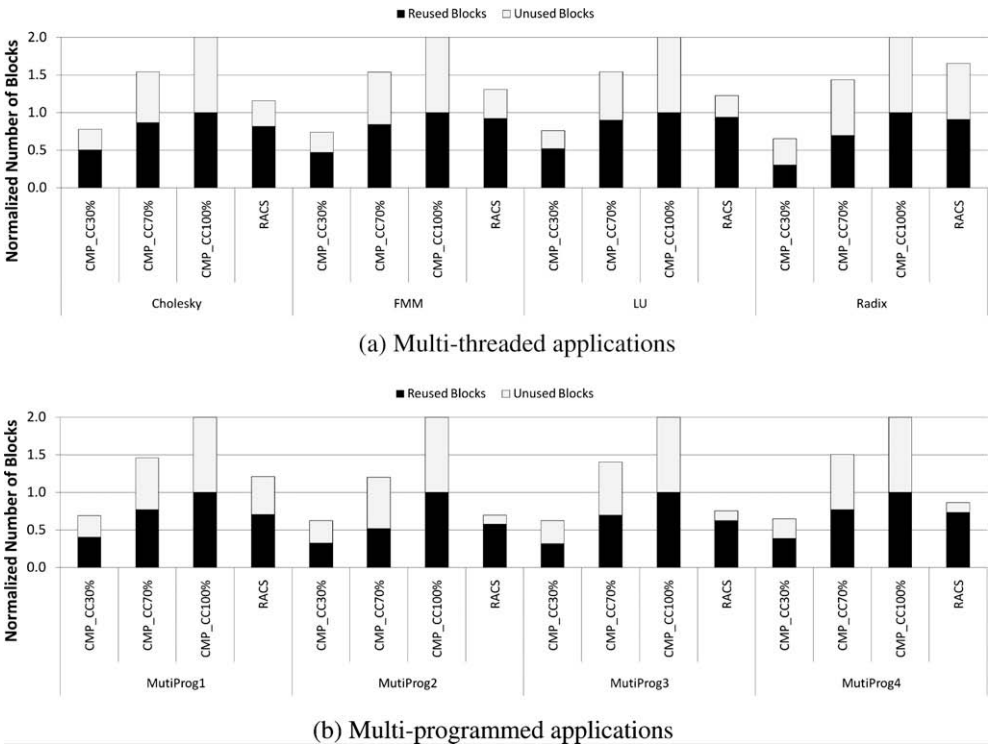

Fig. 7. Normalized number of unused and reused blocks.

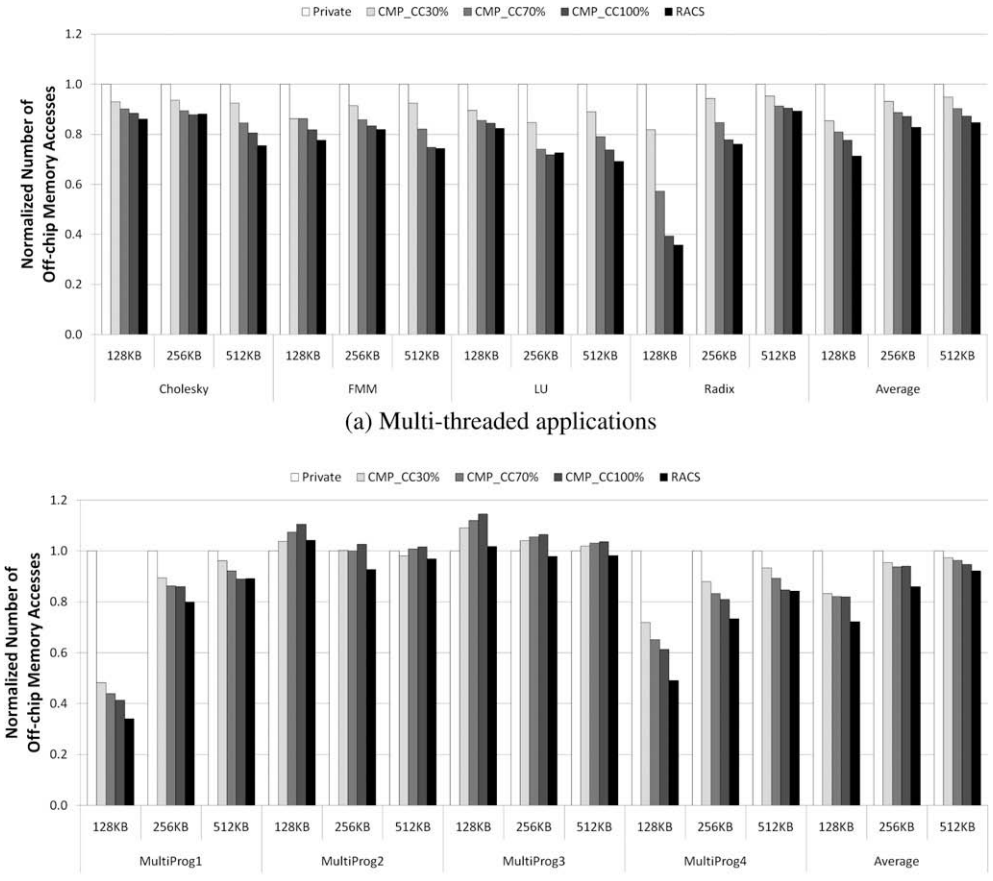

Fig. 8. Normalized number of off-chip memory accesses.

size. After classifying the programs, we made the 4 mixes of these programs by randomly choosing the programs from two groups. As a result, MultiProg1, MultiProg3, and MultiProg4 have one program from the first group and three programs from the second group. In these combinations, we could evaluate how the proposed technique exploits the available peer L2 cache space aggressively. For example, we could evaluate from these combinations how the programs from the second group exploit the available cache space of the program from first group. On the other hand, MultiProg2 has the four programs from the second group. In this combination of the program, we could evaluate how the proposed technique prevents the evicted block from being written to the peer cache when there are excessive evicted blocks which could pollute the peer L2 cache while there is no available cache space in the peer L2 cache.

5.2. Experimental results

Fig. 7 shows the number of the unused and reused blocks written to the peer L2 caches, for each scheme. Using the CMP_CC scheme, the total number of blocks written increases with the probability, for both multi-threaded and multi-programmed benchmarks. For the multi-threaded benchmarks, they show that the RACS scheme writes almost the same number of reused blocks as CMP_CC with the probability 100% but the number of unused blocks is reduced by 65% on average compared to CMP_CC with the probability 100%. For the four multi-programmed benchmarks from SPEC2000, there are many more unused blocks than there are in the multi-threaded benchmarks because more blocks are evicted due to the larger working sets. Again, with CMP_CC, the number of unused blocks increases as the probability increases, and our scheme reduces the number of unused blocks by up to 88% for *MultiProg2*. This benchmark includes *gcc*, *gzip* and *twolf*, which have large working sets, and the performance of *gcc* and *gzip* is easily degraded when their caches are taken up with blocks from other caches. As no cache space remains available, the reusability threshold of each cache is adjusted to predict the reusability of the evicted blocks more conservatively. As a result, dynamic reusability prediction decreases the number of unused blocks as well as the number of reused block. Only blocks with high reusability are kept in the on-chip cache space, to avoid polluting other caches and generating bus traffic.

The workload of each thread in multi-threaded benchmarks has similar behavior and relatively smaller working set size than the multi-programmed benchmarks so that they may not benefit from the dynamic prediction of block reusability technique and the destination cache selection technique although the proposed technique has similar performance over CMP_CC with 100% probability. The performance improvements of the multi-threaded applications arise mainly from identifying the reusability of a block. On the other hand, for the multi-programmed workloads, each program in each combination has different behavior which could benefit from the dynamic prediction of block reusability technique and the destination cache selection technique.

Fig. 8 shows the normalized totals of off-chip memory accesses for each scheme, while varying the cache size. The results for the multi-threaded programs shows that the number of off-chip accesses decreases with the CMP_CC scheme as the probability

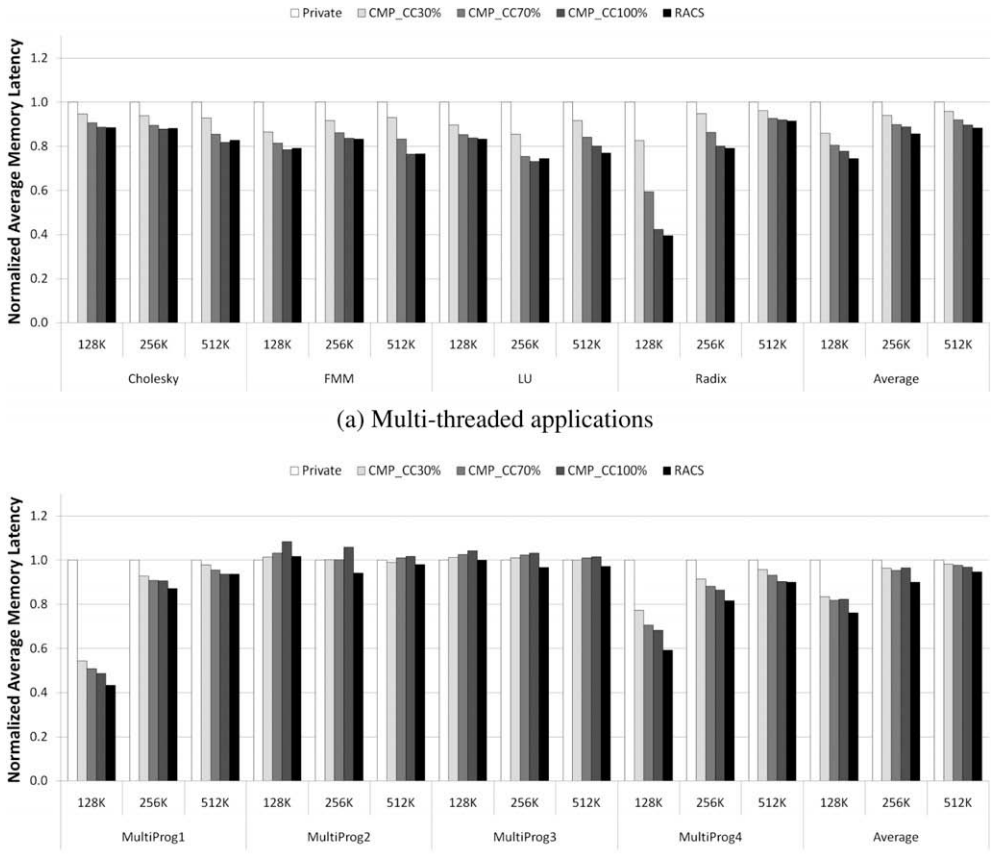

Fig. 9. Normalized average memory access latency.

increases for most programs. However, for the results for the multi-programmed benchmarks are different depending on the characteristics of the programs concurrently running and the sizes of their working sets. For *MultiProg2* and *MultiProg3*, there are more off-chip accesses with CMP_CC, especially when the probability is 100%, than with the private cache scheme. Since the programs in these benchmarks have large working sets, there are much more blocks which are evicted from local L2 caches. These large number of evicted blocks may pollute the peer caches and cause bus traffic. Nevertheless, our RACS scheme produces a fewer off-chip accesses than the private cache scheme by adapting reusability threshold and selection threshold.

In *MultiProg1* and *MultiProg4*, the RACS scheme achieves much more significant reductions in the number of off-chip accesses. *MultiProg1* and *MultiProg4* contain programs such as *mcf* and *gap* whose performance is not affected by the blocks evicted from other cache, or whose working set size is small. The RACS scheme can use their cache space that becomes available effectively, and the number of off-chip memory accesses is reduced by 24%, compared to the private cache and by 2.4%, compared to the CMP_CC scheme, averaged over the multi-threaded benchmarks. For the multi-programmed benchmarks, the number of off-chip memory accesses is reduced by average of 16% and 7.7%, respectively, compared to the private and CMP_CC schemes.

Fig. 9 shows how the normalized average memory access latency of each scheme varies with the cache size. In CMP_CC, the average memory access latency decrease as the probability increases in most cases, since the number of off-chip accesses is reduced. For the multi-threaded benchmarks, RACS reduces the average memory access latency by 21.3% and 5.5%, respectively, compared to the private and CMP_CC with the 100% probability schemes on average, respectively. For the multi-programmed benchmarks, RACS reduces the latency by 13% and 5.7% compared to the private and CMP_CC with the 100% probability schemes.

Fig. 10 shows how the normalized average IPC of each scheme varies with cache size. These results are almost same as those for average memory access latency because that is strongly linked to IPC. However, the improvement in IPCs is less because IPC is affected by other factors. For the multi-threaded benchmarks, the average improvement in IPC is 5% and 1%, compared to the private and CMP_CC with the probability 100% schemes. For the multi-programmed benchmarks, the corresponding improvements are 4% and 3%.

Although the improvements of 1% for the multi-threaded benchmarks and 3% for the multi-programmed benchmarks is small, the contribution of the proposed technique is that it could dynamically control the number of evicted blocks written to the peer L2 cache depending on the available space in the peer L2 cache while CMP_CC is a static approach based on a preset probability of evicted blocks being written to the peer L2 cache. Therefore, when CMP_CC is used, this preset probability is quite different depending on the characteristics of concurrently running programs. Since CMP_CC presets this probability, it can be difficult to adapt to the changing program execution behaviors. Our proposed scheme, on the other hand, is a dynamic scheme, outperforming the CMP_CC scheme even when the preset probability is

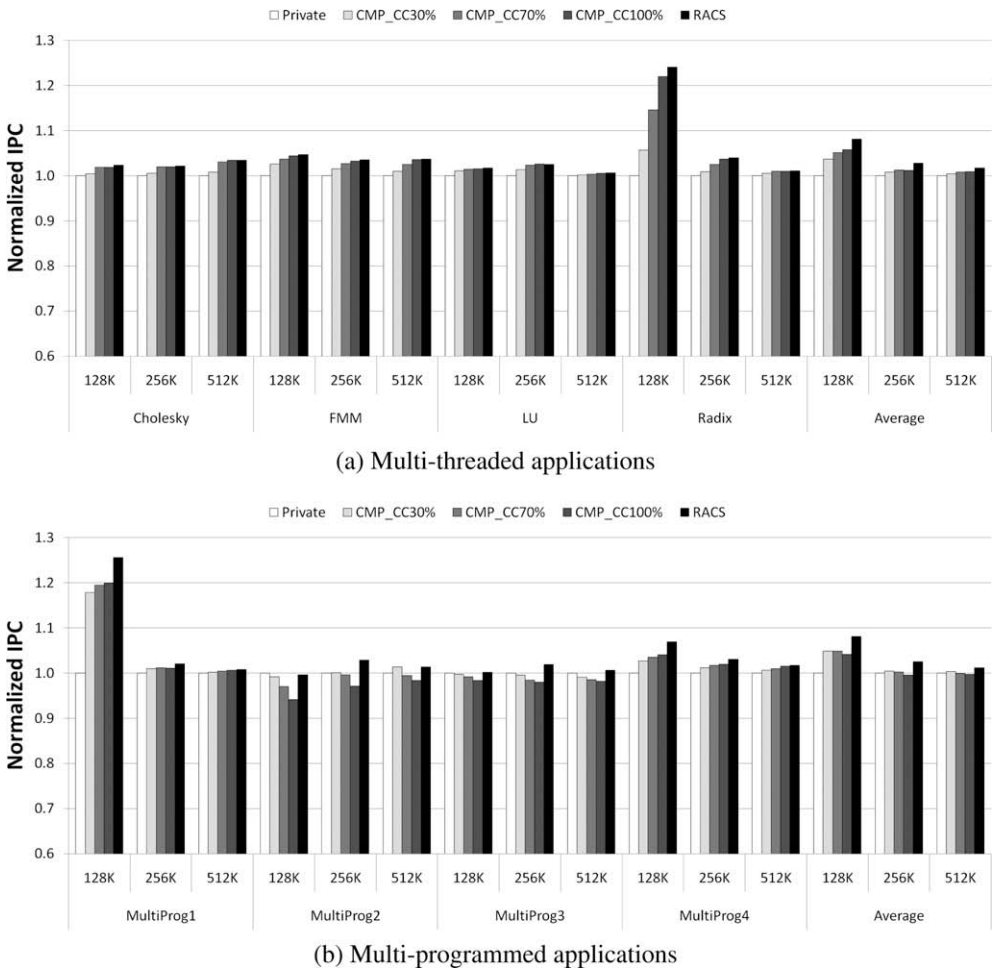

Fig. 10. Normalized performance.

set using an oracle (that is, assuming that we know the program behavior a priori).

For example, in our experiments, for MultiProg1 using a 128 KB L2 cache, RACS achieves up to 4.8% over the best CMP_CC probability setting of 100%. On the other hand, over the CMP_CC with 100%, the proposed scheme improves IPC by up to 6% for MultiProg2 using 256 KB L2 cache. For this program combination, CMP_CC with 30% probability shows the best performance because the performance decreases as the probability increases, which is caused by the pollution of other cache and bus traffics. Even in this case, RACS can achieve about 3% performance improvement than the best CMP_CC probability setting of 30%. Consequently, the proposed technique can improve the performance more compared to the worst CMP_CC probability setting. For example, for RADIX, the performance improvement is 17.4% over the CMP_CC with 30% probability because the proposed technique can avoid the pollution of the other cache by identifying the reusability of the block. It only writes the evicted block with the reusability and replaces the block without the reusability in the peer cache. RACS also improves the performance by 6.7% for MultiProg1 using a 128 KB L2 cache and by 5.9% and 6% for MultiProg2 using a 128 KB and 256 KB L2 cache, respectively. These experiment results indicate that the proposed scheme can identify the reusability of cache blocks and control dynamically the number of evicted blocks written to other L2 cache compared to the previous technique with static probability by showing that it always achieves better performance improvement by up to 4.8% and 17.4% over the best and the worst CMP_CC probability setting, respectively.

6. Conclusions

We have proposed an on-chip L2 cache organization for CMPs called RACS which combines private and shared L2 caches to reduce access latency. A significant feature of the proposed scheme is that it takes into account the reusability of evicted blocks, so as to use the on-chip memory space efficiently. When a data block in a private L2 cache is selected for eviction, RACS evaluates its reusability. If the cache block is likely to be reused in the near future, we save it in one of the other L2 caches. Our second important contribution is to consider the properties of the program running at each processor so as to use the on-chip cache space more efficiently. When a cache space is available, we predict the reusability of evicted blocks more aggressively to give them more chance of being kept on the chip, while a reusability threshold avoids impacting the performance of the other programs, which could dynamically adjust the number of evicted blocks written to the peer L2 cache depending on the workloads behavior. We use a simple but effective heuristic to select an appropriate destination cache. We evaluated the performance improvements using multiprogrammed workloads with a different working set size and locality.

We evaluated the RACS scheme using a modified CMP simulator and compared its performance with the private cache scheme, and the CMP_CC scheme with variable probability. RACS scheme reduces the number of unused blocks written to a peer L2 cache by up to 65% over CMP_CC with the probability 100% for multi-threaded SPLASH 2 benchmarks and by up to 88% over the CMP_CC with the probability 100% for SPEC2000 multi-programmed benchmarks. Compared to the private cache and CMP_CC with the probability 100% schemes, it also reduces the average memory access latency by 13% and 5.7% and improves the average IPC by 4% and 3%, respectively, for the multi-programmed benchmarks. Furthermore, the proposed technique improves the performance by up to 4.8% and 17.4% over the best and the worst CMP_CC probability setting, which means that it can identify the reusability of cache blocks and adjust dynamically the number of evicted blocks written to other L2 cache compared to CMP_CC which uses the static probability.

The RACS scheme could be improved in several ways. First, the prediction heuristic might be made more effective: our results show that around 43% of blocks are not reused even though the prediction heuristic classifies them as highly reusable. Second, the RACS scheme has a significant hardware overhead with a large number of processors because it requires peer-to-peer communication lines between the private L2 caches. It is a challenge to make the RACS scheme more scalable so that it is suitable for large-scale CMP processors.

Acknowledgements

This work was supported by the Korea Science and Engineering Foundation (KOSEF) grant funded by the Korea government (No. R0A-2007-000-20116-0) and World Class University (WCU) program through KOSEF funded by the Ministry of Education, Science and Technology (No. R33-2008-000-10095-0). This work was also supported by the Korea Research Foundation Grant (KRF-2008-013-D00097) and the Brain Korea 21 Project in 2009. The ICT at Seoul National University and IDEC provided research facilities for this study.

References

[1] J. Huh, D. Burger, S.W. Keckler, Exploring the design space of future CMPs, in: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, September 2001, pp. 199–210.
[2] B.A. Nayfeh, L. Hammond, K. Olukotun, Evaluation of design alternatives for a multiprocessor microprocessor, in: Proceedings of the International Symposium on Computer Architecture, May 1996, pp. 67–77.
[3] B.M. Beckmann, D.A. Wood, Managing wire delay in large chip-multiprocessor cache, in: Proceedings of the International Symposium on Microarchitecture, December 2004, pp. 319–330.
[4] M. Zhang, K. Asanovic, Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors, in: Proceedings of the International Symposium on Computer Architecture, June 2005, pp. 336–345.
[5] Z. Chishti, M.D. Powell, T.N. Vijaykumar, Optimizing replication, communication and capacity allocation in CMPs, in: Proceedings of the International Symposium on Computer Architecture, June 2005, pp. 357–368.
[6] J. Chang, G.S. Sohi, Cooperative caching for chip multiprocessors, in: Proceedings of the International Symposium on Computer Architecture, June 2006, pp. 357–368.
[7] S. Youn, H. Kim, J. Kim, A reusability-aware cache memory sharing technique for high-performance low-power CMPs with private L2 Caches, in: Proceedings of International Symposium on Low Power Electronics and Design, August 2007, pp. 56–61.
[8] S. Youn, H. Kim, J. Kim, A reusability-aware cache memory sharing technique for high-performance low-power CMPs with private L2 caches, in: Proceedings of the Workshop on Chip Multiprocessor Memory Systems and Interconnects, February 2007, pp. 27–32.
[9] C. Kim, D. Burger, S.W. Keckler, An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches, in: Proceedings of the Architectural Support for Programming Languages and Operating Systems, October 2002, pp. 211–222.
[10] E. Speight, H. Shafi, L. Zhang, R. Rajamony, Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors, in: Proceedings of the International Symposium on Computer Architecture, June 2005, pp. 346–356.
[11] D. Lee, J. Choi, J. Kim, S.H. Noh, S. Min, Y. Cho, C. Kim, LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies, IEEE Transactions on Computers 50 (12) (2001) 1352–1361.
[12] D. Kim, S. Ha, R. Gupta, CATS: cycle accurate transaction-driven simulation with multiple processor simulators, in: Proceedings of the Design, Automation, and Test in Europe, April 2007, pp. 749–754.
[13] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, A. Gupta, The SPLASH-2 programs: characterization and methodological considerations, in: Proceedings of the International Symposium on Computer Architecture, June 1995, pp. 24–36.
[14] D.E. Culler, J.P. Singh, A. Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufman, 1998, ISBN 1-55860-343-3.

Hyunhee Kim received the B.E. degree in Computer Science and Engineering from the Chunang University, Seoul, Korea, in 2004, and the M.E. degree in Computer Science and Engineering from Seoul National University, Korea, in 2006. She is currently working toward the Ph.D. degree at Seoul National University. Her research interests include Chip Multiprocessor Architecture and On-chip Memory Management.

Sungjun Youn received the B.E. degree and the M.E. degree in Computer Science and Engineering from Seoul National University, Seoul, Korea, in 2005 and 2007, respectively. He is with LG Electronics Corporation. His interests include Low-power Chip Multiprocessor Architecture and On-chip Memory Management.

Jihong Kim received the B.S. degree in Computer Science and Statistics from Seoul National University, Seoul, Korea, in 1986, and the M.S. and Ph.D. degrees in Computer Science and Engineering from the University of Washington, Seattle, WA, in 1988 and 1995, respectively. Before joining SNU in 1997, he was a Member of Technical Staff in the DSPS R&D Center of Texas Instruments in Dallas, Texas. He is currently a Professor in the School of Computer Science and Engineering, Seoul National University. His research interests include Embedded Software, Low-power Systems, Computer Architecture, and Multimedia and Real-time Systems.