View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by UPCommons. Portal del coneixement obert de la UPC

Adapting Cache Partitioning Algorithms to Pseudo-LRU Replacement Policies

Kamil Kędzierski, Miquel Moreto
Technical University of Catalonia (UPC)
and Barcelona Supercomputing Center (BSC)
Barcelona, Spain
{kkedzier, mmoreto}@ac.upc.edu

Francisco J. Cazorla
Spanish National Research
Council (IIIA-CSIC) and BSC
Barcelona, Spain
francisco.cazorla@bsc.es

Mateo Valero
UPC
and BSC
Barcelona, Spain
mateo@ac.upc.edu

***Abstract*— Recent studies have shown that cache partitioning is an efficient technique to improve throughput, fairness and Quality of Service (QoS) in CMP processors. The cache partitioning algorithms proposed so far assume Least Recently Used (LRU) as the underlying replacement policy. However, it has been shown that the true LRU imposes extraordinary complexity and area overheads when implemented on high associativity caches, such as last level caches. As a consequence, current processors available on the market use pseudo-LRU replacement policies, which provide similar behavior as LRU, while reducing the hardware complexity. Thus, the presented so far LRU-based cache partitioning solutions cannot be applied to real CMP architectures.**

This paper proposes a complete partitioning system for caches using the pseudo-LRU replacement policy. In particular, the paper focuses on the pseudo-LRU implementations proposed by Sun Microsystems and IBM, called Not Recently Used (NRU) and Binary Tree (BT), respectively. We propose a high accuracy profiling logic and a cache partitioning hardware for both schemes. We evaluate our proposals' hardware costs in terms of area and power, and compare them against the LRU partitioning algorithm.

Overall, this paper presents two hardware techniques to adapt the existing cache partitioning algorithms to real replacement policies. The results show that our solutions impose negligible performance degradation with respect to the LRU.

***Keywords*-CMP; Shared last level cache; Pseudo-LRU;**

I. Introduction

The hardware cost of exploiting the remaining instruction-level parallelism (ILP) in the applications has motivated the use of thread-level parallelism (TLP) as an effective strategy to improve processor performance. One of the most common TLP paradigms is chip multiprocessing. In Chip Multiprocessor (CMP) architectures the last level cache (LLC) is commonly shared among cores. For example, both cores share the L3 cache in the IBM POWER6 [8], similarly the cores share the L2 in the Sun UltraSPARC T2 [28] and the L3 in the Intel i7 architecture [1].

The LLC, the L2 cache in our baseline processor setup, has been identified as one of the major sources of contention between threads in CMP architectures. If the allocation of the LLC is not controlled properly, some threads can severely affect the performance of the other running threads, degrading the final throughput and/or Quality of Service (QoS) [15], [17]. This has motivated researchers to propose several Cache Partitioning Algorithms (CPAs) in order to control the interaction between threads in the LLC [4], [5], [11], [14], [22]. Since CPAs deliver a flexible and easy-to-manage infrastructure to control threads' behavior in shared caches, they have become the central element of current QoS frameworks for CMPs [7], [10], [17], [18].

In this paper, we focus on *dynamic* CPAs that work at *way* granularity. Dynamic CPAs divide the execution of the workload into time intervals and at each interval boundary, the CPA tries to optimize a given target metric by assigning a new *cache partition* that specifies the number of ways to assign to each running thread. In order to implement a dynamic CPA, a *profiling* and a *partitioning* logics are required.

Profiling logic: The profiling logic gathers the number of cache misses that each thread would have if it had run in isolation, as we vary the number of assigned ways. In particular, for the Least Recently Used (LRU) replacement scheme, the profiling logic observes positions in the LRU stack [22] for each cache access. When using CPAs each thread owns a separate copy of the tag directory, namely *Auxiliary Tag Directory* (ATD). While all the threads access the L2 cache, each ATD is only accessed by its *owner* thread.

According to the stack property [13] of the LRU, by observing the stack position in which each cache access hits, we are able to predict the miss rate of the thread when it is assigned any number of ways [22]. Therefore, by collecting (from the ATD) the stack positions of all the accesses in the past interval, which are stored in the Stack Distance Histogram (SDH) registers, we can derive the L2 miss rate as a function of the assigned ways to that thread. Based on this information, the CPA decides how many ways are assigned to each thread, according to a given target metric, such as minimum

978-1-4244-6443-2/10/$26.00 ©2010 IEEE

number of misses, maximum IPC throughput, fairness or QoS.

Partitioning logic: With CPAs, a thread is allowed to hit in any cache way. However, this logic enforces that a thread only replaces lines in its *assigned* ways. To do so, the logic introduces changes into the cache replacement scheme. It selects as a victim line the LRU line in the assigned ways only, instead of the entire cache set.

The ATDs in the profiling logic and the partitioning logic are the most hardware costly components in terms of area and complexity of a CPA. For example, the area cost of the ATD for a 64-bit 8-core architecture with a shared 2MB, 16-way L2 cache requires 53,248 bytes, which is similar to the size of L1 caches in current designs. Due to its high area cost, the ATD has received researchers' attention, leading to several proposals that require only dozen of bytes [20]–[22]. As a consequence, the ATD has been removed as a limiter to implement CPAs in real processors.

The partitioning logic is the second limiting factor to implement CPAs in real processors. One of the common characteristics of the already proposed partitioning logics is that they are based on, and only work with, the stack property of the true LRU replacement policy. However, it has been shown that for highly associative caches, such as the L2 caches, implementing true LRU is complex and incurres a high hardware cost. The complexity comes from both the high number of storage bits to support LRU replacement scheme ($A \times log_2(A)$ LRU bits, where A is the cache associativity) and the high number of operations to be conducted on each access (in the worst case the position of each line in the LRU stack needs to be updated). As a result, current CMP processors implement simpler pseudo-LRU policies in the last level caches, which hinders the use of so-far proposed CPAs in real processors. In our view, a key aspect of any CPA to be considered by the industry is that it has to work with the replacement policies already implemented in real processors.

In this work, we show the required changes in the hardware components of a CPA that allow its implementation in a cache using a pseudo-LRU replacement policy. We focus on two implementations of the pseudo-LRU based replacement schemes: the Not Recently Used (NRU) implemented in the the UltraSPARC T2 [28] and the Binary Tree (BT) proposed by IBM [4]. Our hardware proposals are based on the fact that neither NRU nor BT have the stack property of true LRU, requiring a new profiling logic. Hence, in this paper we propose two new profiling logics: one that works with the NRU and another with the BT. Overall, this paper presents two hardware techniques to adapt the existing cache partitioning algorithms to real replacement policies.

We show that our CPAs for NRU and BT experience a small performance degradation with respect to the performance of the CPA on top of LRU. When running 49 two-, four- and eight-thread workloads from the SPEC CPU 2000 benchmark suite [2] in a CMP architecture with a 16-way 2MB L2 cache, the results show that our CPA for NRU suffers only a 0.3%, 3.6% and 7.3% performance degradation, respectively. In the case of our CPA for BT, the degradation is 1.4%, 3.4% and 9.7%, respectively. We also conduct a detailed study of the complexity of the LRU, NRU and BT policies. We depict the area overhead of each replacement logic. We further analyze the number of bits that need to be updated on particular events related to the cache access.

The rest of this paper is organized as follows. Section II introduces the components of a dynamic cache partitioning system, whereas Section III analyzes the required architecture changes when migrating from LRU to NRU and BT replacement schemes. The methodology description in Section IV is followed by our simulation results in Section V. Section VI discusses the related work. We conclude in Section VII.

II. Background

Figure 1 shows the architectural changes required to support dynamic CPA in a CMP with a shared L2 cache. In our baseline CMP processor setup, each core has a private L1 instruction and data caches, while the unified L2 cache is shared between the cores. The L2 cache partitioning includes *profiling* and *partitioning* logics, together with the corresponding modification of the replacement logic.

A. Profiling Logic

The profiling logic gathers the number of cache misses each thread would have if it had run in isolation, as we vary the number of ways it is assigned. The logic consists of an Auxiliary Tag Directory (ATD) and a Stack Distance Histogram (SDH) per thread.

The ATD is a separate copy of the tag directory, used to profile threads' accesses to the cache. Both the L2 cache and the ATD of each thread have the same associativity (A) and are accessed in parallel on every cache access. Since each thread accesses its own ATD, we can observe how the thread behaves in the ATD as if it runs alone with an A-associativity cache. A miss in the ATD indicates that a given thread would miss in a cache even if it is allowed to use the entire A-associativity L2 cache.

Each ATD is associated with one SDH. On every cache access, the ATD reports the LRU stack position, in which the access hits, to the SDH [5], [11], [13], [22],

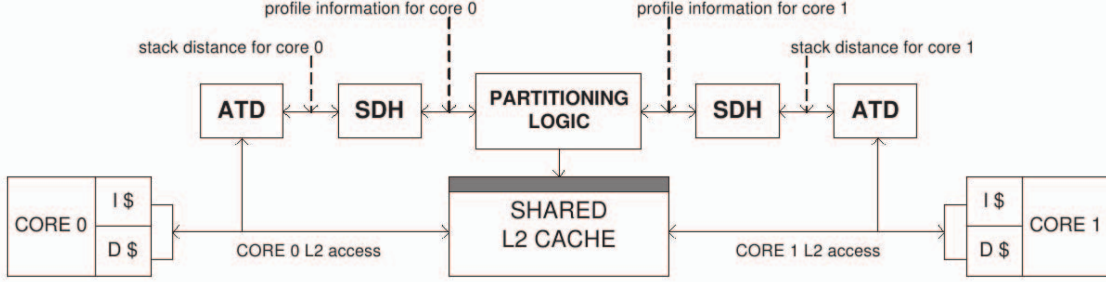


Figure 1. Baseline architecture supporting cache partitioning algorithms. *I$* stands for L1 Instruction Cache, *D$* stands for L1 Data Cache.

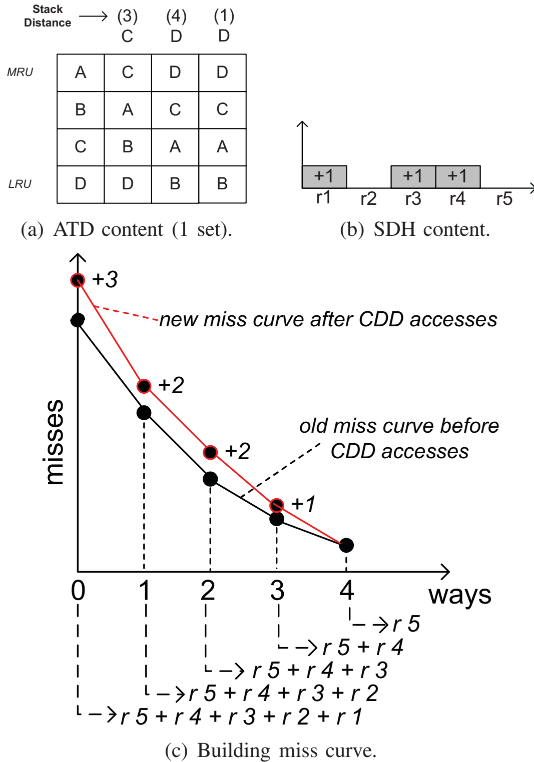


Figure 2. 4-way ATD and SDH state after CDD accesses.

[27]. In the case of a miss, it reports the position A+1, where A is the cache associativity. Hence, the SDH consists of A+1 registers, each containing the number of accesses in the past interval to the corresponding LRU stack positions. The values stored by the registers allow us to derive the miss curve of the thread as a function of the ways assigned to a thread.

Let's assume a 4-way associative cache with one thread executing. Figure 2(a) illustrates the content of the LRU stack for a sample set in the ATD. Further assume that initially the set stores lines {A, B, C, D}, in which A is the Most Recently Used (MRU) line and D is the LRU line, as Figure 2(a) depicts. After {C, D} accesses, the line D is promoted to the MRU position, whereas line B is degraded to the LRU position. For the next access, the stack position of the line D equals 1. Figure 2(b) shows the corresponding SDH structure. For a 4-way ATD, the SDH is built of 5 registers: *r1*, *r2*, *r3* and *r4* store number of accesses for stack distances equal 1, 2, 3 and 4, respectively. Register *r5* stores the number of accesses that miss in the ATD. Since the stack distance equals 1 for the second access of the line D, we increase register r1.

We can derive the number of misses for a given thread by reading the values stored in the corresponding SDH registers. For example, if a thread owns 2 ways, it will suffer $r3+r4+r5$ misses, as Figure 2(c) shows. Thus, by profiling a thread with the ATD and SDH, we are able to predict the number of misses it would have when assigned any number of ways.

Periodically, at every interval boundary, the SDH register values are scaled down to prevent their saturation. In our approach, we divide all register contents by 2. This operation requires only right bit shift in each counter and ensures a fair ratio between the stack positions corresponding to the past and future intervals.

B. Partitioning Logic

We consider CPAs that work at a way granularity and are dynamic, meaning that on every time interval boundary, the CPA selects a new partition. The partitioning logic has two main roles: 1) to determine which ways are given to each thread to optimize a given target metric 2) and to enforce that threads only evict data from their assigned ways.

Partition selection: In our setup we use the MinMisses [22] algorithm with an interval of 1 million cycles. The *MinMisses* policy assigns ways to the running threads so that it minimizes the overall number of misses, giving at least one way per thread. This mechanism increases the *overall* performance. Further goals can be reached, when the policy is modified to favor fairness or QoS [14].

Enforcement logic: So far, the proposed CPAs are based on the true LRU replacement. However, this scheme has been shown to have a high implementation

cost in highly associative caches. To specify a position in the LRU stack, each line needs to be augmented with $log_2(A)$ LRU bits, where A is the cache associativity. For example, in a 4-way associativity L2 cache the MRU position may be represented with bits *00*, and the LRU position with *11*. When looking for a victim, the logic searches for the *11* value in all the lines, sets to *00* the bits of the incoming line and increases the LRU bits of the remaining lines. On a hit, each line that is between the MRU line and the hit line increments its LRU bits, and the hit line is promoted to the MRU position. Not only does it increase design complexity, but it also leads to a high area overheads of the replacement logic.

So far, two enforcement mechanisms have been proposed to work on top of LRU.

1) *Per-set counters* [22]: In a CMP with N cores, $log_2(N)$ bits are added to each line to specify the core that wrote the data in that line. We refer to those bits as *owner core* bits. Moreover, each set has N counters, each of $log_2(A)$ bits, specifying the number of lines in the set that belong to a given core. Whenever a thread replaces a line of a different thread, the corresponding counter of the first thread increases, while the counter of the second thread (whose data is evicted) decreases.

The changes introduced in the replacement logic are the following: on a cache miss, the augmented LRU policy compares the number of lines belonging to the missing thread with the number of ways allocated to that thread. If the thread owns less ways than assigned, the replacement engine selects the LRU line among the lines that do not belong to the thread. Otherwise, it selects the LRU line among the owned ways. This significantly increases replacement logic complexity. This partitioning scheme requires additional $A \times log_2(N) + N \times log_2(A)$ bits per each set[1].

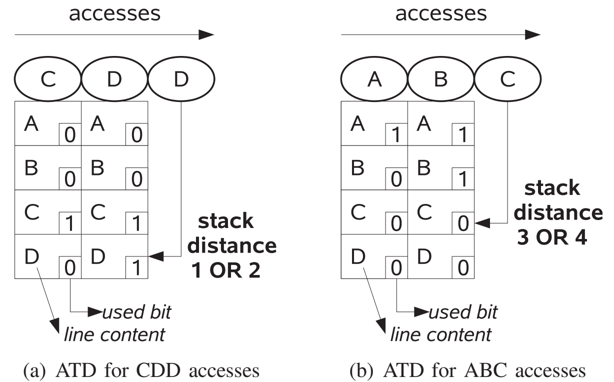
2) *Global replacement masks* [5]: In this scheme, there is a global replacement mask for each core, which specifies the ways that a given core is allowed to search for a victim line. Each mask consists of A bits, each bit specifying whether a given core may access a way in the case of a miss. On a miss, a thread evicts data from its assigned lines.

III. Cache Partitioning Algorithms with Pseudo-LRU Replacement

Two major obstacles prevent the implementation of CPAs in real architectures: the ATD and the replacement logic complexity.

The ATD has been recently removed as a limiting factor, since several solutions have been proposed to reduce its size [21], [22]. In this paper we use approach proposed in [22], where the authors decrease the size of

[1] $A \times log_2(N)$ for the owner core bits and $N \times log_2(A)$ counters' bits



(a) ATD for CDD accesses (b) ATD for ABC accesses

Figure 3. Used bits in a 4-way ATD using NRU for three consecutive accesses. The arrows point to the line of the last access with the estimated stack distance next to it.

the ATD without significantly affecting the final performance. They apply set sampling, so that the number of sets represented in the ATDs is smaller than in the L2 cache. In this scenario, an access to the L2 cache does not necessarily cause an access to the ATD structure, depending whether a given set has been sampled in the ATD. In our environment we sample 1 every 32 sets, so in total the ATD size per core is 3.25KB (for 64-bit architecture with 47 tag bits and 2MB, 16-way L2 cache). Other solutions [20] can further decrease the size of the entire monitoring logic, to only tens of bytes.

The second limiting factor is assuming that LRU is the underlying replacement policy. However, the industry has identified the implementation cost of the true LRU replacement scheme as excessive for high associativity caches. The complexity comes both from the high number of storage bits to support that replacement scheme ($A \times log_2(A)$ LRU bits) and the high number of operations to be conducted on each access (in the worst case the position of each line in the LRU stack needs to be updated). As an alternative solution, high associative caches use pseudo-LRU schemes with similar performance and significantly reduced implementation costs. There are two major types of pseudo-LRU based replacement policies, NRU used in the the UltraSPARC T2 [28] and the BT proposed by IBM [4].

To the best of our knowledge, so far a complete cache partitioning solution for pseudo-LRU schemes does not exist. In our view, the industry can benefit from dynamic cache partitioning only if it is adapted to the current replacement policies. Below we discuss both pseudo-LRU replacement schemes introduced in previous sections, as well as proposals of new profiling logics for dynamic cache partitioning designs that work on top of both pseudo-LRU implementations.

A. NRU-based Cache Partitioning Algorithm

The NRU replacement applies a used bit scheme [28], where every line is augmented with a used bit. Whenever a line is accessed, either on a hit or miss, its used bit is set to 1. If on an access, all the other used bits of the lines in a set are 1, they are reset to 0 except the bit of the line that is accessed. In addition, the L2 cache is extended with a *replacement pointer*, one for all running threads. The pointer is used only on a miss. When looking for a victim line, it shows the first way to be considered for a replacement. A line can be replaced if its used bit is reset to 0. If it is not the case, we search for the next position, until we find a line with a reset used bit. Finally, the replacement pointer is rotated forward one way. The usage of the replacement pointer, one for all the sets, guarantees a random-like replacement [28].

Profiling logic under NRU. Figure 3 depicts a snapshot of a sample set in a 4-way ATD employing the NRU replacement. Let's assume that the given set stores lines {A, B, C, D}. There are two situations when computing the stack distance of an access.

- *Access to a line whose used bit is 1*: Figure 3(a) depicts the used bits of the lines when accessing with the CDD pattern. After {C, D} accesses, the used bits of the lines C and D are set to 1. On the next access to D, we find a used bit already set to 1. We observe that D is the MRU line (accessed just before). This corresponds to a stack distance equal to 1 in LRU replacement. With our new profiling method, when accessing a cache set, we compute the total number of used bits set to 1, denoted U. If a line that is accessed has its used bit set to 1, we estimate its stack distance to be within 1 and U. For the case depicted in Figure 3(a), $U = 2$ and the stack distance may be either 1 or 2. With our profiling method we increase both SDH registers *r1* and *r2*, assuming the stack distance to be 2. Therefore, we tend to overestimate the stack distances. We further discuss this point in the next subsection, where we propose a correction to the SDH overestimation problem.

- *Access to a line whose used bit is 0*: Let's now focus on a different access pattern {A, B, C}, depicted in Figure 3(b). The first two accesses establish the corresponding used bits to 1, and the last access finds its used bit reset to 0. We observe that A and B lines are the most recently used, which implies that the line C has a stack distance equal to at least 3. If a line has its used bit reset to 0, we estimate the stack distance to be within $U+1$ and A. In Figure 3(b) the stack distance may be either 3 or 4. With our methodology, we assume the stack distance to be 4. In this particular case we do not update SDH registers, given that increasing all of them does not change the shape of the miss curve they store. This simplifies the profiling logic design and has a neglible performance cost.

By gathering the stack distances for the NRU scheme, we construct an *estimated* SDH (eSDH). In Section V we prove that the eSDH achieves negligible performance degradation when compared to the SDH, while it enables CPAs to be implemented in real market processors that use the NRU replacement policy.

Increasing the accuracy of the eSDH: As stated above, if the accessed line has its used bit set to 1, the stack distance is within 1 and U. For the eSDH update, however, so far we assume it to be U, which tends to overestimate the profile information. To mitigate this problem, we propose *scaled* eSDH: we assume the stack distance to be $S \times U$, where S is a *scaling factor*. For example, if $S = 0.5$ and there are $U = 8$ lines in a given set with used bits set to 1 (including the line that is accessed), we assume the stack distance to be $S \times U = 4$. If $S \times U$ value does not result in an integer number, we select the closest upper integer. For example, if $U = 7$, we compute $S \times U = 3.5 \simeq 4$.

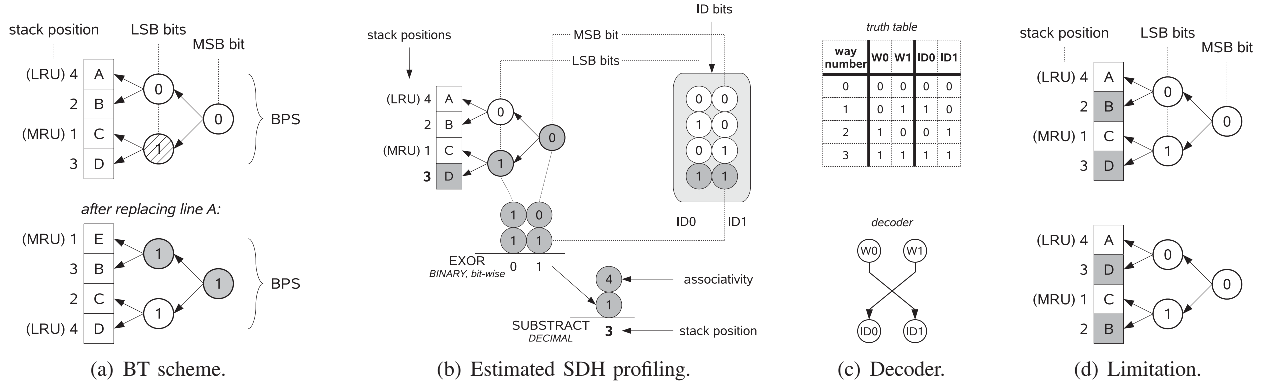
In this paper we analyze three scaling factors: 1.0, 0.75 and 0.5. The former value, 1.0, corresponds to the default case where scaling does not modify eSDH distribution.

Enforcement logic: In the case of a cache miss, the access of each core to the L2 cache is ANDed with the corresponding global replacement masks, to select the ways in which we search for a victim line. If the replacement pointer selects a way that is not within the set of accessible ways for a given core, we rotate it forward one way. This operation is repeated until we find a candidate for the replacement. In the case of a hit, we allow the core to access any line in a set. On an access, if all the used bits of the owned ways are set to 1, we reset all used bits except the one that belongs to the line currently accessed.

B. BT-based Cache Partitioning Algorithm

The Binary Tree (BT) replacement applies a tree structure comprised of $A-1$ bits, as Figure 4(a) shows[2]. Each node of the tree contains a bit that specifies whether upper sub-tree or lower sub-tree contains the most recently used (MRU) line: the value 0 means that the MRU line is in the lower sub-tree, while the value 1 means that the MRU line is in the upper sub-tree. On a miss, when looking for the line to evict (i.e., the pseudo-LRU line), the replacement logic reads the value of the most significant bit (MSB) in the tree structure. In this case the bit value 0 says that the pseudo-LRU position is in the upper sub-tree, whereas value 1 indicates it is in the lower sub-tree. For example, in Figure 4(a), the

[2]In this section we focus on 4-way implementation only. However, similar discussion can be conducted for any cache associativity.



(a) BT scheme. (b) Estimated SDH profiling. (c) Decoder. (d) Limitation.

Figure 4. BT scheme ilustration (a) and profiling logic for the BT replacement policy (b). BPS stands for *bits per set*. On (c) we show decoder for ID bits extraction from the way number. On (d) we show two stacks with the same BT bits.

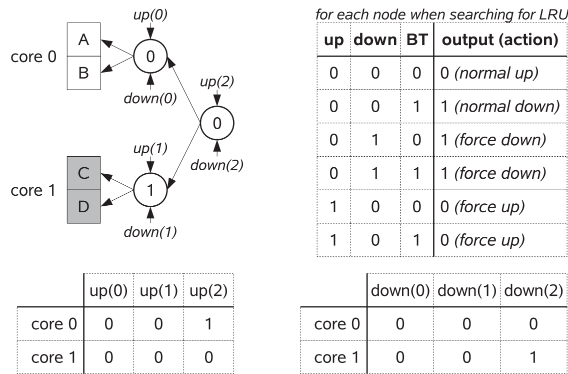


Figure 5. Partitioning logic for the BT replacement policy.

MSB bit specifies that line A or B holds the pseudo-LRU position. Next, the replacement logic reads the value of the corresponding less significant bit (LSB), which in our example points to A as the pseudo-LRU line. The scheme requires to traverse the entire path, from MSB to LSB bits to find the evicted line. There are $log_2(A)$ bits that need to be read to find this line. We observe that all the bits, except the MSB one, specify that a more recently used line is in the upper or the lower sub-tree. However, it does not imply that the line holds the absolute MRU position. For example, the value of the bit in the node connecting lines A and B (hatched node in Figure 4(a)) says that line B is more recently used than line A, though line B is not the MRU line.

In the case of miss, the replacement logic searches for the LRU line, it replaces the line and advances it to the MRU position. For example, in the bottom part of Figure 4(a), line A is replaced with line E and advanced to the MRU position. Shaded nodes represent the bits that are updated to promote the line to the MRU position - we set both bits to 1, as line E is in the most upper sub-tree position. In the case of a hit, the replacement logics accesses the line and similarly promotes the line to the MRU position.

Profiling logic under BT. Figure 4(b) proposes a novel, scalable profiling technique for the BT replacement. For each position in the stack we determine what would be the BT bits values if a given line held the LRU position. We call these bits the *identifier bits* (ID). For example, if line D stays at the LRU position, it is determined with 11 BT bits ($MSB = 1$ and $LSB = 1$). Therefore, for the 4th way storing line, D the identifier bits are 11 ($ID0 = 1$ and $ID1 = 1$), as Figure 4(b) shows.

Next, we perform a bit-wise XOR operation on the ID (11 in the example) and the actual BT bits (10), as Figure 4(b) shows ($11 \oplus 10$ operation). There are $log_2(A)$ bits to be XOR-ed. Finally, we substract the fixed associativity number ($A = 4$ in Figure 4(b)) with the results of the XOR operation. Substraction is a $log_2(A)$-bit operation, and gives the estimated position in the stack (3 in the example). Therefore, we build an *estimated* SDH (eSDH) for each thread.

Figure 4(c) shows a simple decoder extracting ID bits from the current way number for a 4-way L2 cache. We represent a way number with the $W0$ and $W1$ bits. For example, in a 4-way L2 cache the decoder finds the ID bits using formula $ID0 = W1$ and $ID1 = W0$. Therefore, for the 2nd way ($W0 = 1$ and $W1 = 0$) the decoder finds the ID bits $ID0 = 0$ and $ID1 = 1$. There is one decoder for the entire L2 cache. This solution imposes negligible hardware costs even for high associativity caches.

Limitations of the profiling logic in BT. The BT replacement logic does not store sufficient information to determine the actual order of the lines in the pseudo-LRU stack - BT bits only specify in which sub-tree (upper or lower) is the pseudo-LRU/MRU position. For example, Figure 4(d) depicts two different stack contents with identical BT bits. The stacks differ in the position of the B and D lines (marked with grey). Since our profiling logic uses BT bits and cache associativity as the inputs to estimate the stack position, and for both

Table I

COMPLEXITY OF THE LRU, NRU AND BT REPLACEMENT SCHEMES. THE CALCULATIONS IN BRACKETS CORRESPOND TO A 16-WAY 2MB L2 CACHE WITH 128B LINES, ACCESSED BY 2 CORES, 64-BIT ARCHITECTURE (WITH 47 TAG BITS) AND DO NOT INCLUDE THE COST OF THE PROFILING LOGIC. BPS, A, AND N STAND FOR BITS PER SET, CACHE ASSOCIATIVITY AND NUMBER OF CORES, RESPECTIVELY.

(a) Number of the bits that serve in the replacement logic for the LRU, NRU and BT schemes.

	LRU	**NRU**	**BT**
No partitioning	$A \times log_2(A)$ BPS (8 KB)	A BPS + replacement pointer ($\simeq$ 2 KB)	$A-1$ BPS (1.875 KB)
Global replacement masks	$A \times log_2(A)$ BPS + $A \times N$ owner masks bits ($\simeq$ 8 KB)	A BPS + $log_2(A)$ replacement pointer bits + $A \times N$ owner mask bits ($\simeq$ 2 KB)	$A-1$ BPS + $log_2(A)$ $\overrightarrow{up}$ bits per core + $log_2(A)$ $\overrightarrow{down}$ bits per core ($\simeq$ 1.875 KB)

(b) Number of bits that need to be read/updated.

Replacement/partitioning logic			
Event	**LRU**	**NRU**	**BT**
TAG comparison	$A \times TAG\ bits$ (752 bits)	$A \times TAG\ bits$ (752 bits)	$A \times TAG\ bits$ (752 bits)
Update position without partitioning (worst case)	$A \times log_2(A)$ (64 bits)	$A-1$ NRU bits (15 bits) + $log_2(A)$ replacement pointer bits (4 bits)	$log_2(A)$ BT bits (4 bits)
Update position for partitioning (worst case)	Find owned lines:		
	$N \times A$ (32 bits)	$N \times A$ (32 bits)	already solved by $\overrightarrow{up}$ and $\overrightarrow{down}$
	Find LRU in owned lines:		
	$A-1 \times log_2(A)$ (52 bits)	$A-1$ NRU bits (15 bits) + $log_2(A)$ replacement pointer bits (4 bits)	$log_2(A)$ BT bits (4 bits) + $log_2(A)$ $\overrightarrow{up}$ bits (4 bits) + $log_2(A)$ $\overrightarrow{down}$ bits (4 bits)
Get data (hit)	*line size* (1024 bits)	*line size* (1024 bits)	*line size* (1024 bits)

Profiling logic			
Event	**LRU**	**NRU**	**BT**
Read/estimate the stack distance	read $log_2(A)$ LRU bits (4 bits)	count number of used bits (16 bits)	$XOR\ 2 \times log_2(A)+$ $SUB\ 2 \times log_2(A)$ (16 bits)

cases the inputs remain unchanged, we *estimate* instead of determine the real stack position.

To sum up, the profiling proposal consist of a low-overhead decoder and two operations, to estimate the stack position of each line in a set. In Section V-B we evaluate the proposal including both leakage and dynamic power of the additional structures.

Enforcement logic under BT. Figure 5 shows the partitioning scheme that we apply in our setup, similar to [4]. Let's assume that the partitioning logic assigned lines A and B to core 0, and lines C and D to core 1. We extend the BT replacement with two global vectors (of bits) for the entire L2 cache per each core, $\overrightarrow{up}$ and $\overrightarrow{down}$. The size of the vectors equals the number of BT bits, $log_2(A)$. The vectors can force the replacement scheme for a given core to search the LRU position in the upper or lower sub-tree. In the case $\overrightarrow{up}$ signal equals 1, the replacement logic overwrites current BT bit with 0, forcing a given thread to search LRU line in the upper sub-tree. Similarly, if $\overrightarrow{down}$ signal equals 1, the replacement logic overwrites BT bit with 1, forcing a given thread to search LRU line in the lower sub-tree. If both signals equal 0, the value stored in BT bit determines the sub-tree to be searched. Figure 5 shows the truth table for the $\overrightarrow{up}$, $\overrightarrow{down}$ signals and BT bit. The partitioning logic ensures that both $\overrightarrow{up}$ and $\overrightarrow{down}$ signals cannot be equal to 1 at the same time.

C. Complexity Evaluation

In this section we analyze the overheads of adapting the NRU and BT pseudo-LRU replacement schemes to support dynamic CPAs.

LRU. This scheme employs $A \times log_2(A)$ bits per set to store the LRU bits, which translates into 8KB cost for a 16-way 2MB L2 cache with lines of 128 bytes. In the worst case for a hit in the LRU position, the logic updates the positions of all the lines in the set, as Table I(b) shows. This corresponds to advancing the line from the LRU to the MRU position and moving all the other lines one step towards the LRU position, which affects $A \times log_2(A)$ bits. If we use global replacement masks to partition the cache, we need $N \times A$ additional bits to specify the lines available for the replacement. To find the stack position in all the partitioning scenarios, the profiling logic reads $log_2(A)$ replacement bits associated with a given line.

NRU. The NRU scheme significantly reduces the area overhead of the replacement-supporting bits associated with each line, as depicted in Table I(a). If we do not apply any cache partitioning, the area cost for the same cache configuration as above is 2KB plus 4 bits for

Table II
BASELINE PROCESSOR CONFIGURATION (LEFT) AND WORKLOAD SUMMARY (RIGHT).

Processor setup
CORE: 8 wide, out-of-order, 98 entry reserv. station; Branch predictor: select best from bimodal & gshare; BTB: 1KB, 4-way; min penalty - 3 cycles
L1 CACHES: Icache: 64KB, 2-way, 128B line, LRU, 11 cycles miss penalty; Dcache: 32KB, 2-way, 128B line, LRU, 11 cycles miss penalty
L2 CACHE: Unified: 2MB, 16-way, 128B line size, 250 cycles miss penalty, *MinMisses* policy

Workload	Benchmarks
2T_01	apsi, bzip2
2T_02	mcf, parser
2T_03	twolf, vortex
2T_04	vpr, art
2T_05	apsi, crafty
2T_06	bzip2, eon
2T_07	mcf, gcc
2T_08	parser, gzip
2T_09	applu, gap
2T_10	lucas, sixtrack
2T_11	facerec, wupwise
2T_12	galgel, facerec
2T_13	applu, apsi
2T_14	gap, bzip2
2T_15	lucas, mcf
2T_16	sixtrack, parser
2T_17	applu, crafty
2T_18	gap, eon
2T_19	lucas, gcc
2T_20	sixtrack, gzip
2T_21	crafty, eon
2T_22	gcc, gzip
2T_23	mesa, perlbmk
2T_24	equake, mgrid

Workload	Benchmarks
4T_01	apsi, bzip2, mcf, parser
4T_02	parser, twolf, vortex, vpr
4T_03	apsi, crafty, bzip2, eon
4T_04	mcf, gcc, parser, gzip
4T_05	applu, gap, lucas, sixtrack
4T_06	lucas, galgel, facerec, wupwise
4T_07	applu, apsi, gap, bzip2
4T_08	lucas, mcf, sixtrack, parser
4T_09	vpr, wupwise, gzip, crafty
4T_10	fma3d, swim, mcf, applu
4T_11	applu, crafty, gap, eon
4T_12	lucas, gcc, sixtrack, gzip
4T_13	crafty, eon, gcc, gzip
4T_14	mesa, perl, equake, mgrid
8T_01	apsi, bzip2, mcf, parser, twolf, swim, vpr, art
8T_02	apsi, crafty, bzip2, eon, mcf, gcc, parser, gzip
8T_03	twolf, mesa, vortex, perl, vpr, equake, art, mgrid
8T_04	applu, gap, lucas, sixtrack, facerec, wupwise, galgel, facerec
8T_05	applu, apsi, gap, bzip2, lucas, mcf, sixtrack, parser
8T_06	lucas, mcf, sixtrack, parser, facerec, twolf, wupwise, art
8T_07	galgel, vpr, twolf, apsi, art, swim, parser, wupwise
8T_08	gzip, crafty, fma3d, mcf, applu, gap, mesa, perlbmk
8T_09	applu, crafty, gap, eon, lucas, gcc, sixtrack, gzip
8T_10	wupwise, mesa, facerec, perl, galgel, equake, facerec, mgrid
8T_11	crafty, eon, gcc, gzip, mesa, perl, equake, mgrid

the shared replacement pointer. In the worst case only $A-1$ bits have to be updated (all used bits were 1 and are reset, except the replacement pointer position). Similarly, when cache partitioning is applied, additionally $N \times A$ bits for the masks specify the line available for the replacement. The profiling logic requires reading A used bits on each access to the ATD.

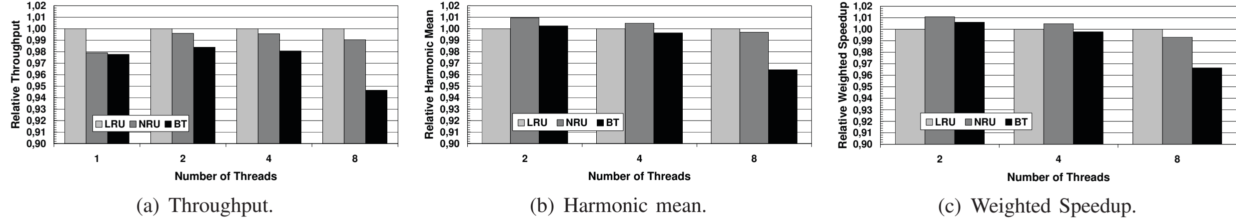
BT. For the scenarios without cache partitioning, BT scheme requires $A-1$ replacement bits per each set, to maintain the BT structure. This translates into 1.875KB in our L2 cache baseline setup. On every cache access $log_2(A)$ bits have to be updated. The replacement bits area slightly increases (by 8 bits) when applying cache partitioning using global replacement masks. There is no need for the owner mask bits, since $\overrightarrow{up}$ and $\overrightarrow{down}$ vectors already specify the set of available lines for a given core, as Figure 5 shows. On a cache access $log_2(A)$ bits of the $\overrightarrow{up}$ and $\overrightarrow{down}$ vectors need to be read additionally. The profiling logic requires two simple operations on each access to the ATD: bit-wise XOR and substract, as Section III-B shows.

Since the global replacement masks, or $\overrightarrow{up}$ and $\overrightarrow{down}$ vectors, already specify the set of ways that can be searched for a victim, we do not need the information on how many lines each core has put in a given set. Thus, we do not require each line to be marked with the core that has put the line in the cache, as it is in the case of the per-set counters [22]. Table I summarizes the complexity analysis of the LRU, NRU and BT replacement schemes.

IV. METHODOLOGY

We use an enhanced version of a detailed cycle-accurate IBM's Turandot simulator [9], [16], the *Parallel Turandot CMP* (PTCMP) [6]. Table II shows the baseline processor configuration. We model 2-, 4- and 8-core CMP processors with 1 thread executing in each core. Both instruction and data first level caches are private to each core, while the L2 cache is shared between all the cores. The processor configuration remains constant for all the experiments. We use three performance metrics: IPC throughput defined as the sum of the threads IPCs, $\sum_{i=1}^{N} IPC_i$; the weighted speedup [25], defined as the sum of relative IPCs, $\sum_{i=1}^{N} IPC_i^{CMP}/IPC_i^{isolation}$; and the harmonic mean of relative IPCs [12], defined as $N/(\sum_{i=1}^{N} IPC_i^{isolation}/IPC_i^{CMP})$. We also evaluate power and relative energy (*CPI x Power*) of the entire processor and memory. We model leakage and dynamic power of all the processor's components. We also take into account the power overhead of accessing off-chip memory. We assume that the energy cost of a memory access is 150 times higher than an access to L2 [3].

We use the SPEC CPU 2000 suite [2] to evaluate our proposal. We combine benchmarks into 24 two-thread workloads, 14 four-thread workloads and 11 eight-thread workloads, in which the benchmarks have been selected randomly. We generate traces using SimPoint methodology [19]. We stop the simulation when each of the threads commits 100 million instructions. Table II depicts the summary of the evaluated workloads.



(a) Throughput. (b) Harmonic mean. (c) Weighted Speedup.

Figure 6. Performance of LRU, NRU and BT. Analysis for 1, 2, 4 and 8 core CMPs using a 16-way 2MB L2 cache with 128 bytes lines.

V. Results and Analysis

A. Pseudo-LRU Schemes on Non-Partitioned Caches

Figure 6 compares the performance results of the NRU and BT schemes of a non-partitioned L2 cache with respect to the LRU replacement policy. In general, pseudo-LRU schemes obtain lower performance than LRU. The behavior of the NRU is similar to the performance of a random replacement policy. If there are several lines in the set that have their used bits reset to 0, the candidate for the replacement is determined by the current position of the replacement pointer. Since there is only one replacement pointer for the entire cache and it is used by all the sets, as introduced in Section III-A, we can assume that the candidate is selected randomly. As a result, this random-like policy achieves lower performance than the LRU scheme. The maximum throughput degradation does not exceed 2.1% across 1, 2, 4 and 8-core architectures with respect to the LRU scheme. The BT replacement tends to spread the lines of each thread across entire set, since each node selects alternally upper and lower sub-tree as the best candidate for the replacement. In this scenarion we observe higher performance degradation: 2.2%, 1.6%, 1.9% and 5.3% throughput reduction for 1-, 2-, 4- and 8-core CMP, respectively. We obtain similar results with harmonic mean and weighted speedup metrics.

B. Pseudo-LRU and Cache Partitioning Algorithms

Figure 7 analyzes the performance of the LRU, NRU and BT schemes when applying dynamic CPAs with our hardware proposals. We evaluate 2, 4 and 8-core CMP architectures, when all the cores share a 16-way 2MB L2 cache. We characterize a given configuration with three parameters. First, we compare configurations with the owner counters per set per each running thread (denoted C), and the global replacement masks (denoted M) in the L2 cache. Second, we compare the LRU (L), the NRU (N) and the binary tree (BT) replacement scheme in the profiling logic and the L2 cache. Third, for the NRU scheme, we evaluate three eSDH scaling factors: 1.0, 0.75, 0.5.

The acronym describing a given architecture consists of these three parameters. For example, we describe a configuration using 1) global replacement masks in the L2 cache, 2) the NRU scheme in the profiling logic and L2 cache, and 3) 0.75 as eSDH scaling factor, by *M-0.75N*. All the results in Figure 7 are relative to the baseline architecture with the dynamic CPA, using the owner counters per set, and with the LRU replacement in both the L2 cache and the profiling logic (configuration *C-L*).

Owner counters vs. global replacement masks. We compare two architectures using LRU replacement: one using per-set counters (*C-L*) and the second using global replacement masks (*M-L*). Figure 7 shows a negligible throughput, fairness and weighted speedup variation, less than 0.5% for any core count. We conclude that the global replacement masks do not impose considerable performance costs, while reducing the cost and the complexity of the replacement logic, as introduced in Section II-B. For this reason we use global masks for all the pseudo-LRU mechanisms evaluated in this paper.

Profiling accuracy with the NRU scheme. We evaluate the new profiling method proposed in Section III-A for the NRU replacement. We apply the NRU scheme to both the L2 cache and ATDs. We evaluate the following values for the eSDH scaling factor: 1.0 (denoted as M-1.0N), 0.75 (denoted as M-0.75N) and 0.5 (denoted as M-0.5N). The eSDH with the 1.0 scaling factor tends to overestimate the number of misses when profiling threads, which causes some performance degradation. The value 0.5, underestimates threads' miss rate, incurring performance penalties. The value 0.75 for the scaling factor presents the best results. With M-0.75N the throughput degrades by 0.3%, 3.6% and 7.3% with respect to the baseline C-L configuration for 2-, 4- and 8-core architectures, respectively. There are two reasons for this performance degradation. First, NRU replacement itself achieves lower performance than LRU scheme, as Figure 6 shows. Second, instead of true stack distances, eSDH gathers estimated positions in the stack. Due to estimation error, our proposal adds additional performance cost.

Profiling accuracy with the BT scheme. Next, we compare the configuration with BT replacement, M-BT, with the baseline C-L architecture, where the latter uses LRU scheme. If the replacement logic uses BT algorithm, it suffers the highest performance degradation

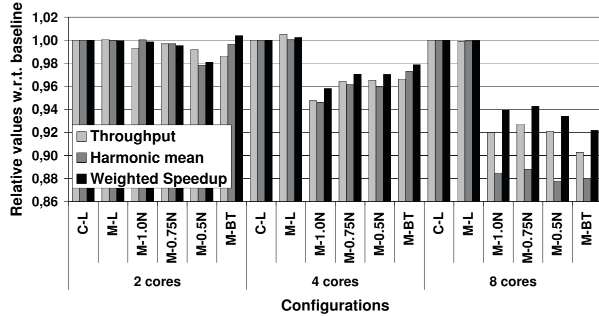
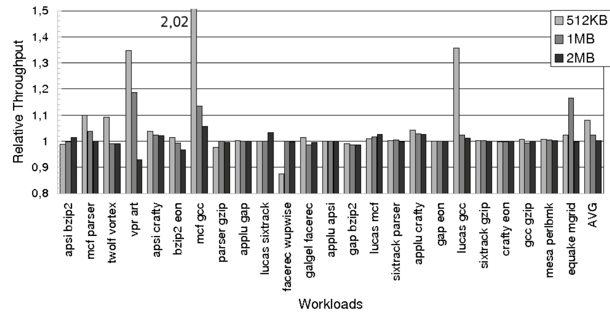


Figure 7. Performance results for the dynamic cache partitioning algorithms in the 2-, 4- and 8-core CMP. All the results are relative to the baseline, *C-L* configuration. Analysis done for a 16-way 2MB L2 cache with 128 bytes lines.

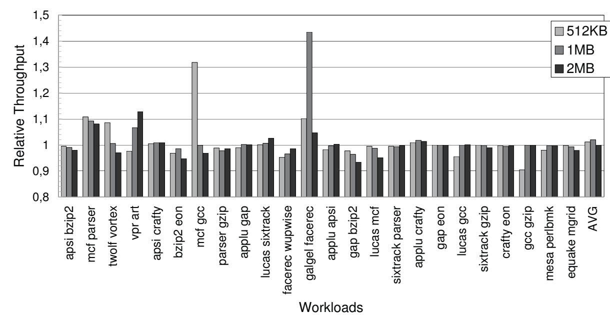
with respect to LRU case, due to reasons discussed in Section V-A. However, we observe that adding cache partitioning on top of the BT scheme does not impose significant performance costs when comparing to similar partitioned LRU-based cache. When the L2 cache is not partitioned, BT introduces a 1.6%, 1.9% and 5.3% lower throughput for 2-, 4- and 8-core CMPs with respect to LRU, as Figure 6(a) shows. When the L2 is partitioned, BT replacement introduces a 1.4%, 3.4% and 9.7% throughput degradation for 2-, 4- and 8-core architectures with respect to LRU. The BT achieves lower throughput than the NRU in partitioned caches due to the replacement policy itself and lower BT performance with respect to NRU, as Figure 6 depicts.

Effect of partitioning the L2 cache. Figure 8 depicts throughput results of the dynamic CPA for LRU, NRU and BT schemes with respect to the non-partitioned L2 cache of the same replacement algorithms. In each case we vary the cache size from 512KB to 2MB. We observe limited performance improvements for big caches, as running threads fit into the L2 cache and do not trash each other data to a high extent. Since the threads' interference increases for smaller caches, dynamic CPAs can recognize the best partitioning scenarios and adapt to current benchmarks' phases. For example, for 512KB L2 cache size MinMisses improves throughput by 8% (for LRU) and 8.1% (BT), for 1MB by 2.4% (LRU) and 4.7% (BT), and for 2MB by 0.2% (LRU) and 0.5% (BT). Due to the SDH estimation limited accuracy, we do not observe average improvements higher than 2% for the NRU policy accross all evaluated cache sizes. However, estimation error does not limit improvements for BT scheme, since BT achieves much lower performance for non-partitioned L2 cache (see Figure 6), and thus creates much relaxed baseline in our studies. We observe similar trends for 4- and 8-core CMPs. We leave further estimation accuracy research for our future work.

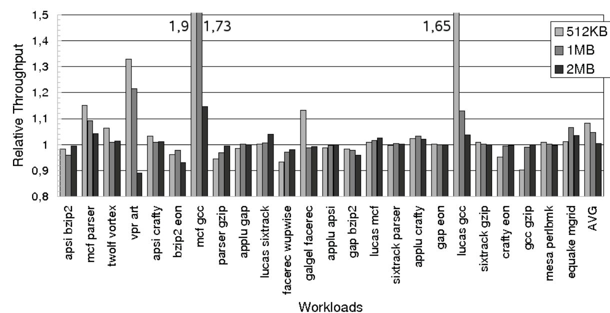
In all the evaluated configurations we assume the same L2 cache access latency, namely 11 clock cycles



(a) *M-L* architecture vs. non-partitioned LRU-based L2 cache.



(b) *M-0.75N* architecture vs. non-partitioned NRU-based L2 cache.



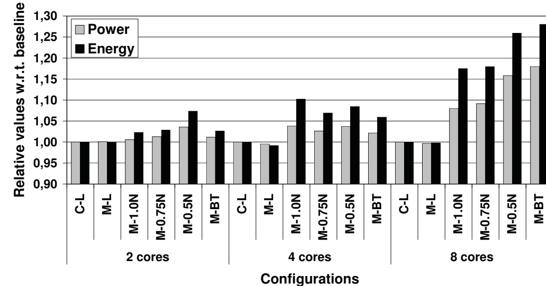
(c) *M-BT* architecture vs. non-partitioned BT-based L2 cache.

Figure 8. Throughput for the LRU, NRU and BT schemes when applying dynamic cache partitioning in a 2-core CMP. The results are relative to the cases without cache partitioning. Analysis done for L2 cache size varied from 512KB to 2MB, with 16 ways and 128 bytes line size in all the cases.

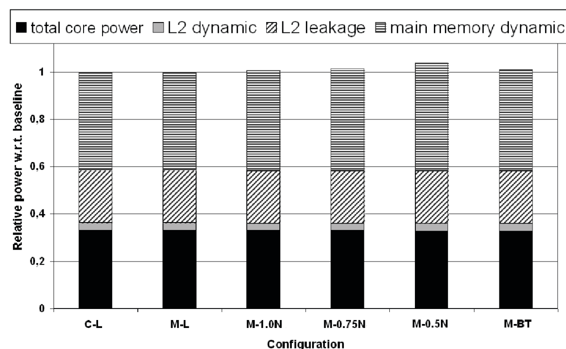
as Table II depicts. However, in Section III-C we show, that both pseudo-LRU schemes require less complex replacement logic, which translates into less delays on each access to the L2. For example, for our baseline setup with non-partitioned L2 cache, LRU updates 64 replacement bits (for a hit in the LRU position), NRU updates 23 bits (all the used bits reset except the replacement pointer position) and BT updates 4 bits (for all the accesses). Hence, an efficient cache design may decrease the access latency for the pseudo-LRU schemes. However, in this paper we focus on the worst-case scenario, in which the pseudo-LRU logic latency does not decrease with the design simplification.

C. Power and Energy Consumption

In this section we evaluate both power and energy consumption for LRU, NRU and BT replacement schemes. The total power include the cores' and L2 cache's dynamic and static power, together with the dynamic main memory power, as Figure 9(a) shows. We include all the leakage cost according to equations in Table 1(a), and all the dynamic power costs according to equations in Table 1(b). We observe that power and energy consumption numbers have the same tendency than the performance numbers. Since the only difference between the evaluated architectures refers to the L2 cache replacement and partitioning logic, different performance translates into different miss rates. High miss rates cause low performance and a high number of energy-consuming off-chip accesses to the main memory. This can be observed when evaluating the power of each processor components as a fraction of the whole power consumption in Figure 9(b). The power of the cores and the L2 cache remains unchanged, whereas main memory dynamic power increases for the configurations with lower performance, due to off-chip accesses. We conclude that the power consumed by the proposed new profiling logic is a negligible component of the whole power - it always remains below 0.3% of the total power. Therefore, to reduce power consumption one needs to improve the performance and reduce off-chip accesses. We believe that efficient cache designs, transforming lower complexity of the pseudo-LRU caches into lower latency caches, can increase the power efficiency of the NRU and BT replacement schemes.



(a) Total power and energy consumption.



(b) Power of each component for a 2-core CMP.

Figure 9. Power and energy consumption for the evaluated configurations. All the results are relative to the *C-L* configuration for 2, 4, and 8 cores, respectively. For all the cases we use a 16-way 2MB L2 cache with 128 bytes lines.

VI. Related Work

The true LRU replacement policy has the *stack property* [13]. This allows to build the Stack Distance Histograms (SDH), obtained during the execution by running the thread alone in the system [5] or by adding some hardware profile counters [22], [26]. Qureshi et al. [22] presented a low-overhead circuit to measure SDHs using an Auxiliary Tag Directory (ATD).

Previous work proposed to partition shared caches, assigning more cache space to the applications that improve a given metric. In these approaches, static and dynamic Cache Partitioning Algorithms (CPA) monitor the L2 cache accesses and decide a partition, in order to maximize throughput [5], [15], [22], [24], [26] or fairness [11], [14]. Zhou et al. used SDHs to improve the management of the main memory and reduce the number of page faults [29].

Other authors propose to use CPAs to ensure Quality of Service (QoS) in CMP architectures. Rafique et al. [23] suggest to manage shared caches with a hardware cache quota enforcement mechanism. They also explore an interface between the architecture and the OS, to let the latter decide the quotas. Nesbit et al. [17] introduce Virtual Private Caches (VPC), which consist of an *arbiter* that controls cache bandwidth, and a *capacity manager* that controls cache storage. However, the authors do not discuss how to decide on resource assignments. A similar framework is presented by Iyer et al. [10], where resource management policies are guided by thread priorities. Guo et al. [7] present an extension of this work with an admission mechanism to accept jobs. Similarly, Moreto et al. [14] attain QoS objectives converting IPC into resource assignments with a specialized hardware. They can also optimize other IPC-related metrics, such as throughput, fairness, and weighted speed up, which gives an enhanced flexibility missing in previous proposals.

However, these proposals are based on the true LRU replacement policy. In this paper we have shown how to adapt dynamic CPAs to NRU and BT schemes already used in the current processors.

VII. Conclusions

Dynamic CPAs have shown to be an effective technique to improve performance in the CMP architectures. However, the solutions proposed so far target LRU replacement scheme. Unfortunately, the LRU imposes

high complexity and implementation costs for high associativity caches, which motivates processor vendors to use the pseudo-LRU policy. Hence, the so-far used CPAs have to be adapted to the replacement schemes available in the current processors.

In this paper, we propose a complete partitioning design that targets two pseudo-LRU replacement policies. In particular, we focus on the Not Recently Used (NRU) replacement, implemented in the L2 cache in the market UltraSPARC T2 processor and the Binary Tree (BT) proposed by IBM. Our proposal covers novel, high accuracy profiling logic. The results show a negligible performance degradation. Namely, our design for NRU loses as much as 0.3%, 3.6% and 7.3% throughput for 2, 4 and 8-core CMP architectures, respectively. For BT the proposal degrades throughput by 1.4%, 3.4% and 9.7%, respectively.

We conclude that the proposals depicted in this paper allow current pseudo-LRU schemes in high associativity shared caches to be easily extended with dynamic cache partitioning algorithms with a small performance degradation.

Acknowledgements

This work was supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625 and grants AP-2005-3776 and AP-2005-3318, by the HiPEAC Network of Excellence (IST-004408) and a Collaboration Agreement between IBM and BSC with funds from IBM Research and IBM Deep Computing organizations. The authors are grateful to the reviewers for their valuable comments.

References

[1] http://www.intel.com/design/corei7/documentation.htm.

[2] http://www.specbench.org/.

[3] S. Borkar and et al. Hundreds of cores: Scaling to terascale architecture. In *Intel Developer Forum*, 2006.

[4] T. Chen, P. Liu, and K. C. Stelzer. *Implementation of a pseudo-LRU algorithm in a partitioned cache*. U. S. Patent Office, June 2006. Patent number 7,069,390.

[5] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *DAC*, 2000.

[6] J. Donald and M. Martonosi. Power efficiency for variation-tolerant multicore processors. In *ISLPED*, 2006.

[7] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO*, 2007.

[8] H.Q. Le, W.J. Starke, J.S. Fields, F.P. O'Connell, D.Q. Nguyen, B.J. Ronchetti, W.M. Sauer, E.M. Schwarz, and M.T. Vaden. Ibm power6 microarchitecture. *IBM J. Res. & Dev.*, 51(6), 2007.

[9] Z. Hu, D. Brooks, V. Zyuban, , and P. Bose. Microarchitecture-level power-performance simulators: Modeling, validation and impact on design. tutorial. In *MICRO*, 2003.

[10] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, 2007.

[11] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, 2004.

[12] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *ISPASS*, 2001.

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2), 1970.

[14] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. FlexDCP: a QoS framework for CMP architectures. *ACM SIGOPS Operating System Review, Special Issue on the Interaction among the OS, Compilers, and Multicore Processors*, April 2009.

[15] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Mlp-aware dynamic cache partitioning. *HiPEAC Conference*, 2008.

[16] M. Moudgill, J.-D. Wellman, and J. H. Moreno. Environment for powerpc microarchitecture exploration. In *IEEE Micro*, volume 19, 1999.

[17] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. *ISCA*, 2007.

[18] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. Multicore Resource Management. *IEEE Micro*, 38(3), June 2008.

[19] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood, and B. Calder. Using simpoint for accurate and effcient simulation. In *ACM SIGMETRICS Performance Evaluation Review*, 2003.

[20] M. Qureshi and et al. Set-dueling controlled adaptive insertion for high-performance caching. In *IEEE MICRO*, 2008.

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

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

[23] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT*, 2006.

[24] A. Settle, D. Connors, E. Gibert, and A. Gonzalez. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing*, 1(3-4), 2005.

[25] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic job scheduling with priorities for a simultaneous multithreaded processor. In *ASPLOS*, 2000.

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

[27] G. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1), 2004.

[28] Sun Microsystems, Inc. UltraSPARC T2 supplement to the UltraSPARC architecture 2007, Draft D1.4.3. 2007.

[29] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, 2004.