Exploiting Reuse Locality on Inclusive Shared Last-Level Caches

JORGE ALBERICIO, PABLO IBÁÑEZ, and VÍCTOR VIÑALS, University of Zaragoza
JOSE MARÍA LLABERÍA, UPC Barcelona Tech

Optimization of the replacement policy used for Shared Last-Level Cache (SLLC) management in a Chip-MultiProcessor (CMP) is critical for avoiding off-chip accesses. Temporal locality, while being exploited by first levels of private cache memories, is only slightly exhibited by the stream of references arriving at the SLLC. Thus, traditional replacement algorithms based on recency are bad choices for governing SLLC replacement. Recent proposals involve SLLC replacement policies that attempt to exploit reuse either by segmenting the replacement list or improving the rereference interval prediction.

On the other hand, inclusive SLLCs are commonplace in the CMP market, but the interaction between replacement policy and the enforcement of inclusion has barely been discussed. After analyzing that interaction, this article introduces two simple replacement policies exploiting reuse locality and targeting inclusive SLLCs: *Least Recently Reused* (LRR) and *Not Recently Reused* (NRR). NRR has the same implementation cost as NRU, and LRR only adds one bit per line to the LRU cost.

After considering reuse locality and its interaction with the invalidations induced by inclusion, the proposals are evaluated by simulating multiprogrammed workloads in an 8-core system with two private cache levels and an SLLC. LRR outperforms LRU by 4.5% (performing better in 97 out of 100 mixes) and NRR outperforms NRU by 4.2% (performing better in 99 out of 100 mixes). We also show that our mechanisms outperform rereference interval prediction, a recently proposed SLLC replacement policy and that similar conclusions can be drawn by varying the associativity or the SLLC size.

Categories and Subject Descriptors: B.3.0 [Memory structures]: General

General Terms: Design, Performance

Additional Key Words and Phrases: Replacement policy, shared resources management

ACM Reference Format:

Albericio, J., Ibáñez, P., Viñals, V., and Llabería, J. M. 2013. Exploiting reuse locality on inclusive shared last-level caches. *ACM Trans. Architect. Code Optim.* 9, 4, Article 38 (January 2013), 19 pages.
DOI = 10.1145/2400682.2400697 <http://doi.acm.org/10.1145/2400682.2400697>

1. INTRODUCTION

In order to reduce the average latency of memory accesses, a hierarchy of cache levels is essential. In a multicore chip, the memory hierarchy usually contains one or two levels of private cache and a shared last-level cache (SLLC). A key task of the cache hierarchy is to exploit the locality usually found during program execution. Specifically, under a demand-fetch policy, the exploitation of locality is directly related to the replacement algorithm at every level of the hierarchy.

This work was supported in part by grants TIN2007-60625 and TIN2010-21291-C02-01 (Spanish Government and European ERDF), gaZ: T48 research group (Aragón Government and European ESF), Consolider CSD2007-00050 (Spanish Government), and HiPEAC-3, FP7/ICT 287759.

Authors' addresses: J. Albericio (corresponding author), P. Ibáñez, V. Viñals, Dpto. De Ing. De Sistemas e Informática, University of Zaragoza and Instituto de Investigación en Ingeniería de Aragón (I3A); email: jorge.albericio@gmail.com; J. M. Llabería, DAC, UPC Barcelona Tech.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2013 ACM 1544-3566/2013/01-ART38 \$15.00

DOI 10.1145/2400682.2400697 <http://doi.acm.org/10.1145/2400682.2400697>

Traditionally, all levels of the memory hierarchy employ algorithms that consider temporal locality in order to select the cache line to replace. In particular, least recently used (LRU) is a widespread replacement algorithm. It predicts that a recently accessed line (either hit or miss) will be used again soon. LRU gives good results on first-level caches because the complete stream of references from the processor is observed but, as many previous authors have shown, it has poor performance as a replacement policy for SLLCs [Jaleel et al., 2008; Lin and Reinhardt, 2002; Qureshi et al., 2007; Subramanian et al., 2006].

Private caches exploit short-distance reuses. Frequently, they even satisfy all the accesses to a given line and, in this case, the SLLC only receives the initial miss request. From the SLLC standpoint these are single-use lines. Therefore, using a recency-based replacement policy such as LRU is not efficient in the SLLC: retaining single-use lines is useless but, nonetheless, LRU will insert those lines in the most recently used (MRU) stack position, thus maximizing their stay. Moreover, in the case of a multicore chip running a multiprogrammed workload the replacement inefficiency may be amplified by interference between programs. A program with a harmful memory access pattern (i.e., a burst of single-use lines) may prevent other programs exploiting reuse opportunities and there may be large accumulated losses in performance.

Although the reference stream observed by the SLLC may exhibit little temporal locality in the conventional sense, it does exhibit *reuse locality*. The concept underlying this type of locality can be described as follows: lines accessed at least twice tend to be reused many times in the near future and, moreover, recently reused lines are more useful than those reused earlier. That is, in reuse locality future references are only expected after the first hit to a line. In contrast, with temporal locality there is an expectation of future references straight after the first reference to a line, a miss. However, only a few lines in the SLLC have reuse locality. Indeed, most lines in the SLLC are *dead*, and they will not receive any further references during their lifetime [Kaxiras et al., 2001; Lai et al., 2001; Qureshi et al., 2007; Xie and Loh, 2009].

Reuse locality has been identified and exploited in cache memories for disks. A representative proposal modified the LRU algorithm in order to protect reused pages against access patterns that result in poor performance such as thrashing or scanning [Karedla et al., 1994]. On the other hand, recent research in SLLC replacement policies relies on predicting the rereference interval [Gao and Wilkerson, 2010; Jaleel et al., 2010b; Khan et al., 2012; Wu et al., 2011]. According to the predicted rereference interval, the utility assigned to each line in these schemes can take one of several values. In contrast, as the reuse locality is a binary property, the derived replacement policies will only require two utility¹ values: to keep or not to keep.

In addition, most proposals consider *noninclusive* SLLCs [Gao and Wilkerson 2010; Jaleel et al. 2008; Khan et al. 2012; Qureshi et al. 2007; Wu et al. 2011], meaning that the lines present in the private caches may or may not reside in the SLLC [Baer and Wang, 1988]. Several commercial processors have instead an *inclusive* SLLC that always keeps a superset of the contents of private caches [Intel, 2011]. This choice greatly simplifies the cache coherence protocol, and is usually implemented by invalidation. When an SLLC line is evicted, inclusion is enforced by sending invalidation messages to all the copies present in the private caches, if any [Baer and Wang, 1988; Chen et al., 2006]. However, another replacement inefficiency arises when the replacement of an inclusive SLLC is managed by an LRU-based policy: a heavily referenced line with a short reuse distance may remain in private caches for a long time. During this time this *hot line*, despite being actively accessed by the core, may move down in the LRU stack of the SLLC, to the point of being evicted. This will force invalidation of the line

¹Utility in this context is associated with the probability of a line being used in the future.

in the private cache, though the processor will request the line again producing a new SLLC miss [Jaleel et al., 2010a].

In this article, we show that recency-based replacement algorithms such as LRU and NRU can be adapted with minor modifications to take advantage of reuse locality rather than temporal locality. Our work introduces two replacement policies for inclusive SLLCs: *least recently reused* (LRR) and *not recently reused* (NRR). They try to retain in the SLLC the lines present in the private caches and the reused lines. Both policies are built upon two simple assumptions about line behavior. First, lines present in the private caches are being used by the running programs. Therefore, these lines will be the last to be evicted. Second, a small subset of lines have reuse locality. Therefore, these lines are valuable and, when it is necessary to select a victim among them, the reuse order will provide a basis for the selection.

With the LRR and NRR policies, lines are replaced as follows: first, lines neither present in the private caches nor showing reuse (*nonreused* lines) are evicted at random; if there are none of these, a line not present in the private caches but reused (*reused* lines) is evicted; and, finally, if there are none of these, a victim line is selected from the private caches (*being-used* lines), this last case occurring relatively rarely.

Under the LRR policy, the lines are ordered depending on their last hit. That is, a least recently reused stack of lines is maintained in each SLLC set. Thus, lines belonging to the reused group are totally ordered (following the LRR order), while there is no relative order among the elements of the nonreused group.

The LRR policy has the drawback of the implementation cost increasing with the square of the set associativity. Also based on recency, the *not recently used* (NRU) algorithm is an inexpensive alternative to LRU ordering [Sun Microsystems, 2007]. Indeed, it is used in the SLLC of commercial processors, such as the Intel Itanium or Sun SPARC T2, and by using only one bit per line, the NRU cost increases linearly with the set associativity. The NRR policy we propose adapts the NRU algorithm for tracking reuse in SLLC sets. Under this NRR policy, every line is provided with an *NRR* bit. In contrast with NRU, the reuse bit will be unset only on hits, not on line refilling. Accordingly, all the not recently reused lines are victim candidates (NRR bit set) and the remaining lines are not.

The proposals are evaluated in an eight-core system with two private cache levels and an inclusive SLLC. By running a rich set of multiprogrammed workloads, we show that LRR outperforms LRU by 4.5%, and NRR outperforms NRU by 4.2% with exactly the same cost. We also show that our mechanisms outperform Rereference interval prediction (RRIP) [Jaleel et al., 2010b], a recently proposed SLLC replacement policy. Similar conclusions can be drawn for a range of associativity values and SLLC sizes.

The article is structured as follows. Section 2 presents experimental evidence of reuse locality and the usefulness of not evicting the SLLC lines present in private caches. Section 3 explains the LRR and NRR replacement policies, giving details of the implementation and associated costs. Section 4 describes the experimental methodology, baseline system and performance indices, and Section 5 reports and discusses the results of all the experiments. Section 6 deals with related work, and finally, Section 7 concludes the article.

2. MOTIVATION

In this section, we analyze the behavior of an example program mix (mix #91) running in the hierarchy of a multicore chip made up of an SLLC and private caches. We highlight three effects, namely: (i) by sharing the SLLC space the working set of an application spreads towards distances greater than the cache associativity; (ii) the principle of inclusion may force hot lines in the private levels to be invalidated, but

Table I. Number of SLLC Accesses and Misses per Kilo-Instruction of Each Application in the #91 Mix, APKI and MPKI, Respectively

	<i>milc</i>	<i>wrf</i>	<i>dealII</i>	<i>hammer</i>	<i>dealII</i>	<i>omnetpp</i>	<i>libquantum</i>	<i>gobmk</i>
APKI - one thread	27.41	0.98	0.15	2.09	0.14	4.26	30.77	0.36
MPKI - one thread	27.37	0.03	0.01	0.02	0.01	1.90	30.77	0.08
APKI - eight threads	27.64	1.16	0.33	2.15	0.31	4.31	30.77	0.65
MPKI - eight threads	27.63	0.53	0.27	1.14	0.27	3.38	30.77	0.46

private caches will request the line again straight away; and (iii) most SLLC hits come from sustained reuse among a small subset of lines.

The selected mix is composed of eight applications of the SPEC CPU 2006 benchmark suite and runs in an eight-core CMP system with an inclusive SLLC; see the simulation details in Section 4. The first two rows of the Table I show the number of SLLC accesses and misses per thousand instructions, when the programs are run alone. We can observe very different behaviors. For instance, *dealII* seems to fit well in the private caches and barely accesses the SLLC (0.15 APKI) and *hammer* almost always hits in the SLLC (2.09 APKI and only 0.02 MPKI), while *milc* and *libquantum* access the SLLC many times (27.41 and 30.77 APKI, respectively) and almost always miss.

Figure 1 plots the number of *wrf* hits in a set-associative SLLC as a function of the LRU stack distance under three different boundary conditions (gray or black bars). The horizontal axis represents 64 LRU stack distances in the SLLC; the first 16 distances belong to real cache storage while the next 48 distances are tracked using shadow tags.

The working set may spread beyond the available storage. In a CMP, different applications share the SLLC and compete for placing their working set into the cache. The lines of an application are displaced in the LRU stack by the lines inserted into the same set by other applications. Figure 1(a) shows the LRU stack when *wrf* runs alone in the CMP (it has the whole SLLC to itself). It can be observed that all the hits arise at a distance of 1 to 8 (0.95 HPKI overall), and there are no additional hits that a larger cache could capture. Now let us consider Figure 1(b), which plots the hit distances of *wrf* when it runs along with the other seven applications shown in Table I. We can see that the bars are smaller and spread over much longer LRU distances than before (from distance 1 to 16, 0.63 HPKI overall). This is because other applications such as *libquantum* or *milc* load a large number of lines which in turn displaces the *wrf* lines (see LLC MPKIs in Table I). Consequently, the MPKI of *wrf* in a 16-way associative SLLC increases from 0.03 when running alone to 0.53 when running together with other applications.

Cache inclusion plus high SLLC miss ratios mean hot line thrashing. Hot lines, those lines with a sustained core reuse, remain silently in private caches for a long time and, therefore, they may become stale in the LRU stack of the SLLC. Before replacing a hot line in the SLLC, the copies present in the private caches are invalidated, but as they are being used by the core, misses will occur and the private caches will request these lines again straight after the invalidation. In Figure 1(b), we can see a peak at a distance of 17 followed by significant number of references at (17–20), meaning that the core is requesting recently invalidated lines. The solid line crossing the aforementioned peak is the distribution of LRU stack distances for a noninclusive SLLC (from distance 1 to 16, 0.71 HPKI overall). The noninclusive SLLC performs better because, even though it may be evicting the same hot lines as the inclusive counterpart, they are not invalidated in the private levels.

The reused lines fit within the associativity. Figure 1(c) shows the distribution of the LRU stack distances for a noninclusive SLLC calculated in the following way. LRU stack entries are tagged as *reused lines* when the first hit occurs, and all the remaining

Fig. 1. Distribution of hits along the LRU stack (HPKI = hits per thousand instructions) in three experiments; the vertical line signals the associativity of the used SLLC cache: (a) *wrf* running alone in the CMP; (b) *wrf* running with 7 other applications in an inclusive (bars) and in a noninclusive (solid line) SLLC; and (c) *wrf* running with 7 other applications in a noninclusive SLLC (reuse stack).

lines (*nonreused*) are ignored when calculating the stack distance. By doing this, we can see how the distance distribution of the reused lines concentrates at the top positions, without exceeding a distance of 10. This indicates that if the SLLC replacement policy were focused on keeping the cache lines that can be expected to be reused, SLLC performance would be significantly improved.

3. REUSED-BASED REPLACEMENT

The goal of reuse-based replacement is to identify reuse locality and exploit it as effectively as possible. In order to achieve this objective, we are inspired by replacement algorithms that try to exploit temporal locality. These algorithms are based just on line use, either a hit or a miss. In contrast, since we aim to exploit reuse locality, our algorithms will consider reuse to establish the replacement order.

Whenever a line is in the private caches, we assume its temporal locality is being well exploited and so it should be among the last ones to be evicted from the SLLC. Among the lines not present in the private caches, replacement should be carried out following the reuse order (total or partial). We consider an SLLC line to be reused if there has been at least one hit on it since it was loaded into the SLLC. The most recently reused lines are the last candidates to be replaced. Conversely, the lines that have not yet been

Fig. 2. The reused-based stack.

reused will be the first eviction candidates, but as there is no basis for ordering them, victims are selected randomly.

To record whether a line has been reused or not, we add a bit to each SLLC line, the *reused* bit. The reused bit is unset when a line is loaded into the SLLC (on a miss), and it is set when the line is reused (on a hit). We use another bit to record whether an SLLC line is present in any private cache, the *being-used* bit. When the line is present in a private cache the being-used bit is set and it is unset when the line only resides in the SLLC. Provided that the coherence protocol we use has precise knowledge of the line presence in the private caches², the being-used bit is not needed.

Next, we modify two classic recency-based replacement policies, LRU and NRU, to take better advantage of reuse locality. We call our proposals LRR and NRR.

3.1. LRR: Least Recently Reused

Each cache set is organized as a single logical stack with being-used lines on top, reused lines in the middle, and nonreused lines at the bottom (Figure 2). The replacement policy searches for a victim starting from the bottom and finishing at the top of the stack. Neither nonreused nor being-used lines are ordered. Thus, random replacement is employed when a line from these groups has to be evicted. Reused lines, on the other hand, are selected following the reuse order. This behavior is shown in Algorithm 1.

The LRR policy does not establish size bounds for any of the three groups. Accordingly, once a line has been classified as reused, it will not be evicted while there are any nonreused lines. This implies that the group of reused lines may grow to occupy all the available space, while the nonreused group may decrease to the point of disappearing. However, the number of nonreused lines increases each time a line without any hits in the SLLC (reuse bit unset) is evicted from the private caches, at which point the line joins the nonreused group. In Section 5, we will show how the number of lines evolves in each group over the course of the execution of a given mix of programs, as an example.

3.2. NRR: Not Recently Reused

LRR and LRU share the same scalability problem: their implementation cost grows quadratically with cache associativity. An alternative to LRU, also based on recency, is the *not recently used* (NRU) replacement algorithm [Sun Microsystems, 2007]. The implementation cost of the NRU approach grows linearly with cache associativity (1 bit per line). We propose a reuse-based algorithm inspired by NRU that we call *not recently reused* (NRR). NRR has the same implementation cost as NRU.

As with LRR, NRR organizes each cache set as a single logical stack with being-used lines on top, reused lines in the middle, and nonreused lines at the bottom of the stack (Figure 2). However, NRR does not consider lines to have any order within each group.

NRR uses a bit per line, the *NRR* bit, in order to distinguish the recently reused lines from the not recently reused ones. When a line is loaded into the SLLC because

²Nonsilent eviction of clean blocks is assumed in the private caches.

ALGORITHM 1: Least-Recently Reused (LRR) algorithm

Cache **hit** on line h :

- i) Move ' h ' to the *most-recently reused* position
- ii) Set the *being-used* bit of h to '1'
- iii) Set the *reused* bit of h to '1'

Cache **miss** on line h :

- i) Randomly select a line with *being-used* and *reused* bits to '0'
- ii) If found go to vi)
- iii) Select the LRR line with *being-used* bit to '0'
- iv) If found go to vi)
- v) Select the LRR line
- vi) Replace the selected line
- vii) Insert the line h
- viii) Set the *being-used* bit of h to '1'
- ix) Set the *reused* bit of h to '0'

of a miss, its NRR bit is set (Algorithm 2). When there is a hit (a reuse), its NRR bit is unset. If the NRR bits of all the lines in the not-being-used group become unset, no useful information about the reuse order remains. Thus, emulating the NRU behavior (but based on reuse), NRR sets the NRR bit of every line in the not-being-used group, except for the line receiving the last hit.

While a line is in some local cache, its NRR bit is not changed. The algorithm replaces any line with the NRR bit set from the not-being-used group.

In summary, the differences between NRR and NRU are: (i) NRR gives an additional level of protection to blocks present in the local caches, and (ii) NRR unsets the bit associated with a line only on hits, while NRU unsets the bit on hits and on misses. Therefore, the hardware cost of NRU and NRR is the same if we consider that the presence information in the local caches is provided by the coherence protocol.

ALGORITHM 2: Not-Recently Reused (NRR) algorithm

Cache **hit** on line h :

- i) Set the *NRR* bit of h to '0'
- ii) Set the *being-used* bit of h to '1'
- iii) If the *NRR* bit of all the remaining *not being-use* lines are '0', set them to '1'

Cache **miss** on line h :

- i) Search for the first not *being-used* line with the *NRR* bit to '1'
- ii) If found go to iv)
- iii) Search for the first line with *NRR* bit to '1'
- iv) Replace the selected line
- v) Insert the line h
- vi) Set *NRR* bit of line h to '1'
- vii) Set *being-used* bit of line h to '1'

3.3. Hardware Complexity

Our algorithms have the same cost that two well-known and already implemented proposals: LRU and NRU.

The LRR algorithm, as it is required by LRU, has to keep an order between all the lines present in a set. This objective can be achieved by using different approaches. Being n the associativity of the cache, two common implementations are: (i) a matrix

of n by n bits per cache set, (ii) a counter of $\log^2 n$ bits per line. The division between reused and no-reused elements that LRR considers is achieved without adding any hardware storage. All the no-reused elements can be represented by one unique state. E.g. n zeroes on alternative (i) or a counter equal to zero on alternative (ii).

The NRR algorithm requires one bit per cache line to track reuse (the NRR bit), and another to record if the line is present in the private caches. However, note that this last bit of presence is already implemented in the directory of an inclusive SLLC.³ Therefore, NRR requires one bit per cache line, as NRU.

The logic to find the replaced element in both algorithms operates with the result obtained by NORing the reuse bit vector and the presence bit vector. When all lines in a set are present in private caches, the logic operates directly with the reuse bit vector. Thus, NRR and LRR have similar replacement logics to LRU and NRU, respectively.

4. METHODOLOGY

4.1. Experimental Setup

We have used *Simics*, a full-system execution-driven simulator, to evaluate our proposal [Magnusson et al., 2002]. We simulate in-order processors and employ the *Ruby* plugin from the *Multifacet GEMS* toolset, to model the memory hierarchy with a high degree of detail [Martin et al., 2005]: coherence protocol, on-chip network, communication buffering, contention, etc., and we added a detailed DDR3 DRAM model.

We used multiprogrammed *SPEC CPU 2006* workloads running on the Solaris 10 Operating System. In order to locate the end of the initialization phase, we used hardware counters on a real machine and ran all the SPARC binaries with the reference inputs until completion.

For an eight-core system, we produced a set of 100 random mixes of eight programs each, taken from among all the 29 *SPEC CPU 2006* programs (no effort was made to distinguish between integer and floating point values). At each checkpoint, it is guaranteed that no application is in its initialization phase. The cycle-accurate simulation starts at those checkpoints, 500 million cycles are run to warm up the memory hierarchy, and then we collect statistics for the next billion cycles. Table III shows the average of misses per kilo-instruction (MPKI) in all the instances of an application at the three levels of the cache hierarchy when the eight applications of a given mix are run together.

4.2. Baseline System

The baseline system has eight in-order cores. Each core has two levels of private caches and all the cores share the LLC. This shared LLC has four banks interleaved at cache line granularity (64 B). A MOSI protocol maintains the memory system coherent, allowing thread migration between cores. A crossbar communicates the local caches and the shared LLC banks. There is a single DDR3 memory channel and the DRAM memory bus runs at a quarter of the core frequency. Table II gives additional implementation details.

5. RESULTS

In this section, we evaluate the LRR and NRR replacement policies. Section 5.1 analyzes the performance achieved by the proposed policies and compares them with LRU, NRU, and RRIP [Jaleel et al., 2010b]. Sections 5.2 and 5.3 give insight into the behavior of LRR and NRR algorithms, and the RRIP comparison, respectively. Finally,

³Evictions of clean lines on the private caches are nonsilent.

Table II. Baseline System Configuration

Private L1 I/D	32 KB, 4-way LRU replacement, 64 B line size, 1-cycle access latency
Private unified L2	256 KB, 8-way LRU replacement, 64 B line size, 7-cycle access latency
Shared L3	8 MB inclusive (4 banks of 2 MB each), 64 B interleaving, 64 B line size. Each bank: 16-way, LRU replacement, 10-cycle access latency. 16 demand MSHR
DRAM	1 rank, 16 banks, 4 KB page size, Double Data Rate (DDR3 1333Mhz). 92-cycle raw access latency
DRAM bus	667Mhz, 8 B wide bus, 4 DRAM cycles/line, 16 processor cycles/line

Table III. Average MPKI at Each Cache Level

Application	L1	L2	LLC	Application	L1	L2	LLC
perlbench	3.7	0.8	0.6	povray	11.0	0.3	0.3
bzip2	8.2	4.3	2.1	calculix	13.8	3.7	1.5
gcc	21.8	7.1	6.2	hmmer	2.9	2.2	1.7
bwaves	20.3	19.6	19.6	sjeng	4.2	0.5	0.5
gamess	75.3	46.2	28.6	GemsFDTD	25.8	25.7	21.6
mcf	22.9	22.2	18.1	libquantum	36.6	36.6	36.6
milc	21.6	21.6	21.5	h264ref	3.5	0.7	0.6
zeusmp	12.3	6.4	6.3	tonto	4.88	0.86	0.52
gromacs	8.71	5.91	5.91	lbm	68.1	39.2	39.2
cactusADM	13.9	1.4	0.7	omnetpp	7.3	4.4	1.2
leslie3d	29.5	18.1	17.7	astar	6.9	0.9	0.7
namd	1.4	0.2	0.1	wrf	4.1	1.6	0.5
gobmk	9.5	0.5	0.4	sphinx3	13.8	8.0	6.3
dealII	2.3	0.3	0.3	xalancbmk	8.2	7.0	6.4
soplex	6.7	5.8	4.8				

Sections 5.5 and 5.4 describe the sensitivity of LRR and NRR performance to associativity and cache size, respectively.

5.1. Results for the Baseline System

Besides LRR and NRR, we also consider the replacement algorithms usually implemented in real processors (LRU and NRU), and a state-of-the-art alternative (RRIP [Jaleel et al., 2010b]). We select RRIP as a reference because to the best of our knowledge it is the best proposal for inclusive SLLC caches and one of the few that suggest a reuse-based algorithm for this kind of SLLCs. Specifically, we implement thread-aware dynamic rereference interval prediction (TA-DRRIP), the extension to shared caches proposed by the authors [Jaleel et al., 2010b]. We also compare our proposals with an enhanced model of TA-DRRIP that protects SLLC lines while they are in the private caches, which we call TA-DRRIP+.

Figure 3(a) shows NRU, TA-DRRIP, TA-DRRIP+, NRR, and LRR performance relative to LRU for the program mix used as an example in Section 2. The rightmost group of bars is the geometric mean of individual speedups. LRR improves LRU performance for all the applications in the mix, while NRR has the same effect with respect to NRU. TA-DRRIP is the only algorithm that causes losses in some applications (milc), while TA-DRRIP+ is better than TA-DRRIP in six of the eight applications and eliminates

Fig. 3. Performance improvement and miss reduction in the multiprogrammed workload #91 used as example in Section 2.

losses. On average in this mix, our proposals perform better than the other algorithms tested.

Figure 3(b) shows, for each application in the example mix, the percentage reduction in the number of misses achieved for each mechanism relative to LRU replacement. As can be observed in the figure, LRR achieves the greatest reduction in all applications. NRR is the second best mechanism in six of the eight applications, and DRRIP in the other two (hmmer and omnetpp). Furthermore, DRRIP is the most irregular mechanism. First, it gets a much smaller reduction in the two instances of the application dealII. Also, DRRIP does not achieve any reduction in milc and libquantum. These two applications have high miss ratios, pointing to a possible thrashing behavior. DRRIP correctly recognizes the lack of reuse and acts easing the eviction of their cache lines. However, our mechanisms work with a finer grain, identifying the few lines showing reuse, and giving them higher priority than the rest. As a result, NRR and LRR manage to reduce misses even in these applications (LRR eliminates 1.8% and 2.7% of misses, while NRR eliminates 0.5% and 1.6% in libquantum and milc, respectively).

Figure 4(a) plots all the 100 mixes on the horizontal axis. The different mixes are sorted by the LRR speedup over LRU. In the same way, Figure 4(b) plots NRR performance relative to NRU. LRR outperforms LRU in 97 mixes out of 100, while NRR is better than NRU in all but one of the mixes.

Fig. 4. Relative performance for all the mixes evaluated.

Fig. 5. Summary of speedups.

Figure 5 shows the mean performance of NRU, TA-DRIP, TA-DRIP+, NRR, and LRR relative to LRU for the one hundred mixes described in Section 4. On average, LRR improves LRU performance by 4.5%, while NRR outperforms NRU by 4.2%. On

Fig. 6. Stack profile for *wrf* with LRR as replacement policy.

Fig. 7. Temporal evolution of the group sizes in a cache set when LRR is used.

the other hand, TA-DRRIP and TA-DRRIP+ increase LRU performance by 3% and 3.3%, respectively.

5.2. LRR/NRR Behavior

LRR and NRR are intended to cause lines that will not be reused to be removed from the cache. It is interesting to explore the degree to which this expectation is met, and also to understand how the dynamic evolution among line groups explains the observed behavior. Therefore, we consider next the resulting stack profile under LRR, and the temporal evolution of the classification of lines in a sample set of the SLLC.

Figure 6 shows again the stack profile of Section 2 for mix #91 under LRR. As can be seen, hits are concentrated towards the top of the stack, almost always at a distance of less than 16. Thus, by applying LRR, the SLLC is effectively keeping the cache lines likely to be reused, and this is why the SLLC performance improves. The NRR policy, not illustrated in the figure, produces a similar behavior.

Figures 7 and 8 plot the evolution of the number of lines classified as being-used, nonreused, and reused (from bottom to top) in a sample SLLC set, over a short period of execution of mix #91, with LRR and NRR as replacement policies, respectively.

Under LRR, the boundary between reused and nonreused lines moves down each time a line is reused for the first time. As we pointed out in Section 3.1, this single-direction movement is counteracted every time a hit occurs on a reused block (the block

Fig. 8. Temporal evolution of the group sizes in a cache set when NRR is used.

Fig. 9. Distribution of speedups relative to LRU for all applications.

moves to the being-used group), and each time a new block is loaded into the cache set (miss) and all the nonreused lines in that cache set are also being-used.

Under NRR, when all the not-being-used lines become reused, the replacement algorithm itself converts all of them but one to nonreused lines. That is to say, when all the not-being-used lines have the reused bit set, then NRR unsets it for all of them except the line receiving the last hit. This behavior can be clearly seen at times 201 and 601 in Figure 8.

5.3. Comparison with RRIP

To obtain insight into how the replacement algorithms affect individual applications, Figure 9 shows the distribution of speedups by application. The number of mixes in which each application appears is shown along the top of the graph. For each replacement policy (TA-DRIP, LRR, and NRR) five speedups are plotted, namely the minimum, the first quartile, the median, the third quartile, and the maximum.

For the mix #91, we saw (in Figure 3) how TA-DRIP improved the performance of several applications, but also reduced it in one case (milc). In Figure 9, we note that this behavior is quite common. In 24 out of the 29 applications, TA-DRIP performs worse than LRU in some multiprogrammed mixes, whereas LRR and NRR reduce this number to 12. Therefore, it can be concluded that reuse-based replacement is more fair than TA-DRIP.

Fig. 10. Performance comparison for five mixes of a SPEC2K6 subset in a four-core system.

The imbalance introduced by TA-DRRIP may be due to the control mechanism deciding which replacement algorithm is used for each application. Specifically, TA-DRRIP uses set dueling [Qureshi et al., 2007] to identify the best suited replacement policy for each application, dynamically choosing between scan-resistant SRRIP and thrash-resistant BRRIP [Jaleel et al., 2010b]. That is, if an application greatly reduces its miss rate with a given configuration, even at the cost of increasing the misses of other applications, the configuration that benefits itself will prevail.

On the other hand, the average speedup we obtain with DRRIP seems to be lower than that reported by the authors. We believe that the explanation may lie in the different methodological approaches used. They model a four-core system with a 4MB SLLC, executing a varied workload, among which there is only a subset of five SPEC 2006 applications. Therefore, in the next experiment we simulate that system and run the five mixes of four applications resulting from combining the applications of the same SPEC 2006 workload (*cactusADM*, *sphinx3*, *hmmcr*, *mcf*, and *bzip2*).

Figure 10 shows the average speedup of NRU, TA-DRRIP, NRR, and LRR over LRU for the five aforementioned mixes. Notably, in comparison to Figure 5, TA-DRRIP increases its speedup the most (from 1.02 to 1.04).

Hardware complexity. The NRR algorithm requires one bit per cache line (as shown in Section 3.3), while DRRIP requires N bits per cache line, N being the number of bits required to classify lines in segments. The aging logic for NRR is simpler than that of DRRIP. DRRIP aging requires incrementing the counters of all the lines in a set, whereas NRR aging only requires resetting the reuse bits to the lines not present in the private caches (using the presence bit vector). Moreover, TA-DRRIP requires a per-thread policy selection counter, and the logic for choosing a set dueling monitor. This logic decides whether a miss occurs or not in the (sampled) sets belonging to the monitor of the corresponding thread.

5.4. Sensitivity to the SLLC Associativity

In this section, we explore the sensitivity of reuse-based replacement to cache associativity, testing the values 8, 16, and 32. The cache size is kept constant at 8MB.

Fig. 11. Sensitivity to the cache associativity and cache size.

Figure 11(a) shows three groups of bars for the three cache associativities. Each group has five bars which represent NRU, TA-DRRIP, TA-DRRIP+, NRR, and LRR mean performance relative to LRU. The speedup with respect to LRU decreases with increasing associativity. LRR shows the best performance in all the associativities, while the low-cost proposal, NRR, is the second best option for associativities 16 and 32.

5.5. Sensitivity to the Cache Size

In this section, we consider the sensitivity of reuse-based replacement to cache size, testing the values 4, 8, and 16MB. The cache associativity is kept constant at 16.

Figure 11(b) shows three groups of bars for the three cache sizes. Each group has five bars which represent the mean performance of NRU, TA-DRRIP, TA-DRRIP+, NRR, and LRR relative to LRU. The speedup with respect to LRU decreases for both 4 and 16MB cache sizes. Moreover, for a 4MB cache, all replacement algorithms lead to poorer performance than LRU except LRR. LRR gives the best performance in all the cache sizes, while the low-cost proposal, NRR, is better than TA-DRRIP for 4MB and 8MB caches.

6. RELATED WORK

Numerous innovations have been proposed for improving the performance of replacement policies. We only summarize those closely related to LRR and NRR.

Reuse Locality. Reuse locality was first observed and exploited in cache memories for disks. Segmented LRU [Karedla et al., 1994] tries to protect useful lines against harmful behaviors (i.e., a burst of single-use accesses) by dividing the classical LRU stack into two different logical lists, the *referenced* and the *nonreferenced* list. The boundary between the lists is fixed and victims are selected in order to preserve that limit.

Recent proposals have applied this idea to the replacement policy of noninclusive SLLCs. Both dynamic segmentation [Khan et al., 2012] and dueling segmented LRU [Gao and Wilkerson, 2010] consider these two logical LRU divisions and try to dynamically find an optimal configuration by using set dueling. The former uses a set dueling predictor and a decision tree to dynamically move the border between the reused and nonreused segments. Another level of set dueling chooses between dynamic segmentation and plain LRU. Whenever the size of the reused segment becomes the smallest (one line), bypass is switched on. In spite of the cache being bypassed, one of every thirty-two lines is stored in cache in order to prevent the working set from becoming stale.

Dueling segmented LRU adds random promotion and aging to the basic segmentation. Random promotion acts by randomly tagging some nonreused lines as being reused, while aging acts in the opposite way. The mechanism uses set dueling in order to dynamically choose between segmented and plain LRU. In addition, they suggest using adaptive bypass. Some shadow tags are required to evaluate the bypass benefit and switch it on or off accordingly.

In contrast, LRR and NRR are designed for inclusive caches (which are widespread in commercial processors), and are much simpler than previous proposals. They are implemented by adding slight modifications to the LRU and NRU replacement algorithms present in commercial processors. Similar to the last two proposals (dueling segmented LRU and dynamic segmentation), LRR and NRR both exploit the idea of reuse by classifying lines as reused and nonreused. In addition, since the SLLC is unaware of line usage in the private caches, our algorithms classify these lines in a third category (being-used), giving them the highest stay priority. Set sampling is not required, and only one replacement algorithm is required rather than two. LRR does not need an explicit aging mechanism, and NRR uses exactly the same one as NRU. No random promotion is required and bypassing is not implemented.

Recently, *MRU-tour*-based algorithms also propose using reuse to divide the elements of a set into different groups and randomly evict elements with a number of MRU-tours lower than a given value [Valero et al., 2012].

Insertion Policy. Several studies propose to change the insertion point in the recency stack of lines that reach the cache. Their goal is to avoid that thrash or scan workloads evict the useful cache lines.

The dynamic insertion policy (DIP) involves a hybrid cache replacement [Qureshi et al., 2007]. Under this scheme, LRU and Bimodal Insertion Policy (BIP) are dynamically selected by using set dueling. LRU always inserts the incoming lines in the MRU position of the recency stack. BIP inserts the majority of incoming lines in the LRU position and with a low probability in the MRU position.

Promotion/Insertion PseudoPartitioning (PIPP) allows each thread to insert its lines at a different point in the recency stack [Xie and Loh, 2009]. It provides the benefits of cache partitioning, adaptive insertion, and capacity stealing among threads. However, it requires a utility monitor with shadow tags to detect the thread's behavior.

Although industry has widely adopted inclusive SLLC schemes in commercial processors, specific content management for this kind of cache organization has not received much attention from academia. An exception is ReReference Interval Prediction (RRIP), that has been proposed for an inclusive hierarchy [Jaleel et al., 2010b]. RRIP involves a modified LRU that considers a chain of segments, where all the cache lines in a segment are supposed to have the same rereference interval [Jaleel et al., 2010b]. An N -bit counter is added to each line to classify it in 2^N segments. In static-RRIP, new lines are inserted into the segment corresponding to *intermediate* rereference interval (2^{N-2} segment) making the algorithm *scan*-resistant. *Thrash*-resistant bimodal-RRIP

inserts lines with *long* rereference interval but a small fraction of randomly chosen lines that are introduced with an *intermediate* rereference interval. Finally, this proposal includes a dynamic version (DRRIP) that uses *set dueling* to select between SRRIP and BRRIP. For an SLLC in a CMP, thread-aware-DRRIP requires a set dueling monitor for each thread.

We have compared TA-DRRIP with our proposals in terms of performance and hardware cost in Section 5.

Frequency-Based Replacement. Frequency-based replacement algorithms classify according to the number of times cache lines have been accessed, giving more priority to the more accessed ones. The *least frequently used* (LFU) approach relies on the access frequency of cache lines to attempt to avoid harmful patterns that may evict useful lines [Lee et al., 2001]. Although frequency-based replacement improves the performance of applications with frequent *scans*, it is not good if the workload exhibits temporal locality. In a way, our SLLC replacement proposals are a very simple form of a frequency-based replacement since lines get classified in only two counts: either those accessed one time or more than one time during their stay in the SLLC. However, LRR and NRR also work correctly if the workload exhibits temporal locality because they use recency within each segment.

Dead-Block Prediction. Dead-block prediction tries to identify dead lines by means of hardware predictors [Khan et al., 2010; Lai et al., 2001; Liu et al., 2008]. For instance, prediction relies on recording the instruction sequences performing the last touch of a given line, or counting the number of accesses performed on a line before it gets replaced. When a line experiences again some of those past behaviors, it is tagged as dead and becomes a replacement candidate. Dead-block prediction requires expensive prediction hardware. The underlying behavior of lines assumed in dead-block prediction and reuse-based replacement is the same: a big fraction of cache lines is dead at any moment. However, the opportunity that shows up is exploited in different ways. Our mechanisms classify a priori all lines entering the SLLC as dead (since we realize most of them are indeed touched once). A line becomes alive once it is referenced a second time during its stay in the SLLC.

Replacement on Inclusive Hierarchies. In inclusive hierarchies, the core caches absorb most of the temporal locality and the hot lines may lose positions in the LRU stack of the SLLC, up to the point of being evicted. A recent paper shows ways to solve this problem by identifying lines in the core caches and preventing their replacement in the SLLC [Jaleel et al., 2010a]. Three ways are proposed: sending hints to the SLLC about the core accesses (*TLH*), identifying temporal locality by early invalidation of lines in the core caches (*ECI*), or querying the core caches about the presence of the victim lines (*QBS*). We also address this problem by using the information present in the coherence directory, assuming nonsilent eviction of clean blocks in the private caches. However, our main contribution is the design of two replacement algorithms as exploit reuse locality with the same hardware cost as those used in commercial processors.

7. CONCLUSIONS

Private cache levels filter short-distance reuses, and thus the SLLC of a CMP observes a stream of references that may have very little temporal locality. On the other hand, these references may have *reuse locality*. The concept of reuse locality can be described as follows: lines accessed at least twice tend to be reused many times in the near future and, moreover, recently reused lines are more useful than those reused earlier.

Further, a heavily referenced line with a short reuse distance may remain in private caches for a long time, steadily losing position in the LRU stack and eventually being

evicted. Thus, if an SLLC follows an inclusive scheme such hot lines will be invalidated and fetched again and again. Consequently, traditional replacement algorithms based on recency such as LRU and NRU are poor choices for inclusive SLLCs.

In this article, we have shown how to adapt these algorithms to take advantage of reuse locality rather than temporal locality. We have proposed two simple replacement policies for inclusive SLLCs that exploit reuse locality: *least recently reused* (LRR) and *not recently reused* (NRR). Both policies are intended to retain in the SLLC the lines present in the private caches as well as the reused lines.

In contrast with previous studies that select the application subset sensitive to the replacement algorithm, our proposals have been evaluated by running a rich set of multiprogrammed workloads created from all SPEC CPU 2006 applications.

For an eight-core system with two private cache levels and an inclusive SLLC, we have found that LRR outperforms LRU by 4.5% (97 out of 100 mixes) and NRR outperforms NRU by 4.2% (99 out of 100 mixes). A detailed comparison with RRIP [Jaleel et al., 2010b], a recent SLLC replacement proposal, indicates that LRR and NRR give 1.5% and 0.89% better performance, respectively. Additionally, we have shown that our algorithms are more fair than TA-DRIP and that similar conclusions can be drawn considering a range of different associativity values and SLLC sizes.

Unlike previous proposals, which require prediction mechanisms or use several algorithms and dynamically select the best one through techniques such as set dueling, LRR and NRR have costs similar to replacement algorithms implemented in commercial processors. NRR has the same implementation cost as NRU, and LRR only adds one bit per line to the LRU cost.

REFERENCES

- BAER, J. AND WANG, W.-H. (1988). On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Computer Architecture Symposium*. 73–80.
- CHEN, X., YANH, Y., GOPALARKISHNAN, G., AND CHOU, C. T. 2006. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Proceedings of the International Conference on Formal Methods in Computer Aided Design (FMCAD'06)*. 81–88.
- GAO, H. AND WILKERSON, C. 2010. A dueling segmented lru replacement algorithm with adaptive bypassing. In *Proceedings of the 1st JILP Workshop on Computer Architecture Competitions*.
- INTEL. 2011. Intel core i7 processor. <http://www.intel.com/products/processor/corei7/specifications.htm>
- JALEEL, A., BORCH, E., BHANDARU, M., STEELY JR., S., AND EMER, J. 2010a. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. IEEE Computer Society, 151–162.
- JALEEL, A., HASENPLAUGH, W., QURESHI, M., SEBOT, J., STEELY, S., AND EMER, J. 2008. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM Press, New York, 208–219.
- JALEEL, A., THEOBALD, K., STEELY, S., AND EMER, J. 2010b. High performance cache replacement using reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM Press, New York, 60–71.
- KAREDLA, R., LOVE, J., AND WHERRY, B. 1994. Caching strategies to improve disk system performance. *Comput.* 27, 3, 38–46.
- KAXIRAS, S., HU, Z., AND MARTONOSI, M. 2001. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th Annual International Computer Architecture Symposium*. 240–251.
- KAHN, S., WANG, Z., AND JIMENEZ, D. A. 2012. Decoupled dynamic cache segmentation. In *Proceedings of the IEEE 18th International Symposium on High Performance Computer Architecture (HPCA'12)*.
- KAHN, S. M., TIAN, Y., AND JIMENEZ, D. A. 2010. Sampling dead block prediction for last-level caches. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'10)*. 175–186.
- LAI, A.-C., FIDE, C., AND FALSAFI, B. 2001. Dead-Block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Computer Architecture Symposium*. 144–154.

- LEE, D., CHOI, J., KIM, J.-H., NOH, S., MIN, S. L., CHO, Y., AND KIM, C. S. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Trans. Comput.* 50, 12, 1352–1361.
- LIN, W. AND REINHARDT, S. K. 2002. Predicting last-touch references under optimal replacement. Tech. rep. CSE-TR-447-02, University of Michigan.
- LIU, H., FERDMAN, M., HUH, J., AND BURGER, D. 2008. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proceedings of the 41st IEEE/ACM International Symposium on Microarchitecture (MICRO'08)*. 222–233.
- MAGNUSSON, P. S., CHRISTENSSON, M., ESKILSON, J., FORSGREN, D., HALLBERG, G., HOGBERG, J., LARSSON, F., MOESTEDT, A., AND WERNER, B. 2002. Simics: A full system simulation platform. *Comput.* 35, 2, 50–58.
- MARTIN, M., SORIN, D., BECKMANN, B., MARTY, M., XU, M., ALAMELDEEN, A., MOORE, K., HILL, M., AND WOOD, D. 2005. Multifacets general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News* 33.
- QURESHI, M., JALEEL, A., PATT, Y., STEELY, S., AND EMER, J. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA'07)*. ACM Press, New York, 381–391.
- SUBRAMANIAN, R., SMARAGDAKIS, Y., AND LOH, G. H. 2006. Adaptive caches: Effective shaping of cache behavior to workloads. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 385–396.
- SUN MICROSYSTEMS. 2007. UltraSPARC T2 supplement to the UltraSPARC architecture 2007. Draft D1.4.3, 19 Sep 2007.
- VALERO, A., SAHUQUILLO, J., PETIT, S., LOPEZ, P., AND DUATO, J. 2012. Combining recency of information with selective random and a victim cache in last-level caches. *ACM Trans. Archit. Code Optim.* 9, 3, 16:1–16:20.
- WU, C.-J., JALEEL, A., HASENPLAUGH, W., MARTONOSI, M., STEELY, S. C., AND EMER, J. 2011. Ship: Signature-Based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'11)*. ACM Press, New York, 430–441.
- XIE, Y. AND LOH, G. 2009. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM Press, New York, 174–183.

Received June 2012; revised September 2012; accepted November 2012