ReD: A Policy Based on Reuse Detection for Demanding Block Selection in Last-Level Caches

Javier Díaz[†], Pablo Ibáñez[†], Teresa Monreal*, Víctor Viñals[†], José M. Llabería*

Aragón Institute of Engineering Research (I3A), University of Zaragoza, and Hipeac [†]

Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya and Hipeac*

***Abstract*—*In this paper, we propose a new block selection policy for Last-Level Caches (LLCs) that decides, based on Reuse Detection, whether a block coming from main memory is inserted, or not, in the LLC. The proposed policy, called ReD, is demanding in the sense that blocks bypass the LLC unless their expected reuse behavior matches specific requirements, related either to their recent reuse history or to the behavior of associated instructions. Generally, blocks are only stored in the LLC the second time they are requested in a limited time window. Secondarily, some blocks enter the LLC on the first request if their associated requesting instruction has shown to request highly-reused blocks in the past. ReD includes two table structures that allow tracking, measuring and correlating reuse for specific block addresses and requesting program counters within a constrained storage budget. It can be implemented on top of any other base replacement algorithm. Other parts of the base replacement policy, such as promotion or victim selection, can remain unchanged, enabling our policy to work along with many state-of-the-art replacement algorithms.*

I. Introduction

The insertion policy of a replacement algorithm determines the position of incoming blocks in the replacement list. For example, the Least Recently Used (LRU) replacement algorithm gives incoming blocks the highest priority to stay, inserting them into the MRU position. Several studies show the inefficiency of this policy for Last-Level Caches (LLCs) and propose inserting new blocks either with an intermediate priority [4], or with the lowest priority within their cache set [6, 7]. Alternatively, some incoming blocks could be selected to not be stored (be bypassed) in the LLC, if their computed priority is lower than the minimum in the set [3, 5]. Throughout this paper, we use the expression “block selection policy” to refer to the part of the insertion policy that decides whether a new block has to be stored in the LLC.

The starting point for our proposed policy is the observation that most cache blocks are not requested again from the LLC after they are stored. Our policy also relies on the reuse locality property of the LLC access stream, which states that 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 [1]. Therefore, in our policy, blocks requested for the first time are by default not stored in the LLC. They are only stored when a following request is detected, that is, when they are reused. As our policy is based on reuse detection, we have called it the Reuse Detector (ReD).

With this default policy, blocks with reuse are going to experience two LLC misses. To avoid the second miss, we propose to exploit the correlation between the reuse pattern of the blocks and the instructions that request them for the first time. A similar correlation has been pointed out and exploited in a previous study [8]. In our policy, we focus on the instructions that request blocks with a high reuse probability. LLC misses coming from such instructions will always trigger LLC block storage, avoiding the second miss.

ReD can supersede the block selection policy previously used by any other replacement algorithm, leaving the rest of the components unchanged. In the policy we have submitted for the 2nd Cache Replacement Championship (CRC-2), SRRIP is used as the base replacement algorithm [4].

The structure of this paper is as follows. Section II describes how ReD works. Section III gives implementation details and storage costs. Section IV shows results, and Section V summarizes our contributions.

II. ReD Block Selection Policy

The goal of our ReD policy is to store in the LLC only blocks that have demonstrated reuse. To achieve this, we primarily track block addresses for requests that miss in the LLC. Secondarily, we track the program counters (PCs) that request those blocks. In order to avoid interference among cores, and assure a fair distribution of resources, both tracking mechanisms are private for each core.

Our primary reuse detector is the Address Reuse Table (ART). It stores addresses of requests that have recently missed in the LLC, to check whether they are requested again. A request that misses both in the LLC and the ART is marked as a candidate for bypass in the LLC, and its address is stored in the ART. We call this an initial request. A request that misses in the LLC but hits in the ART is stored in the LLC, because the ART hit is a true indication of reuse. We call this a first-reuse request. Subsequent requests are expected to hit in the LLC.

Using only the previous mechanism, a block with reuse would experience two LLC misses, because both the initial and the first-reuse request would miss in the LLC. To avoid the second miss, we include a secondary mechanism that aims to predict the reuse pattern of blocks at their initial request. For this mechanism, we assume that the reuse of a block is correlated with the instruction that performs the initial request, the trigger instruction. We record the reuse behavior of blocks

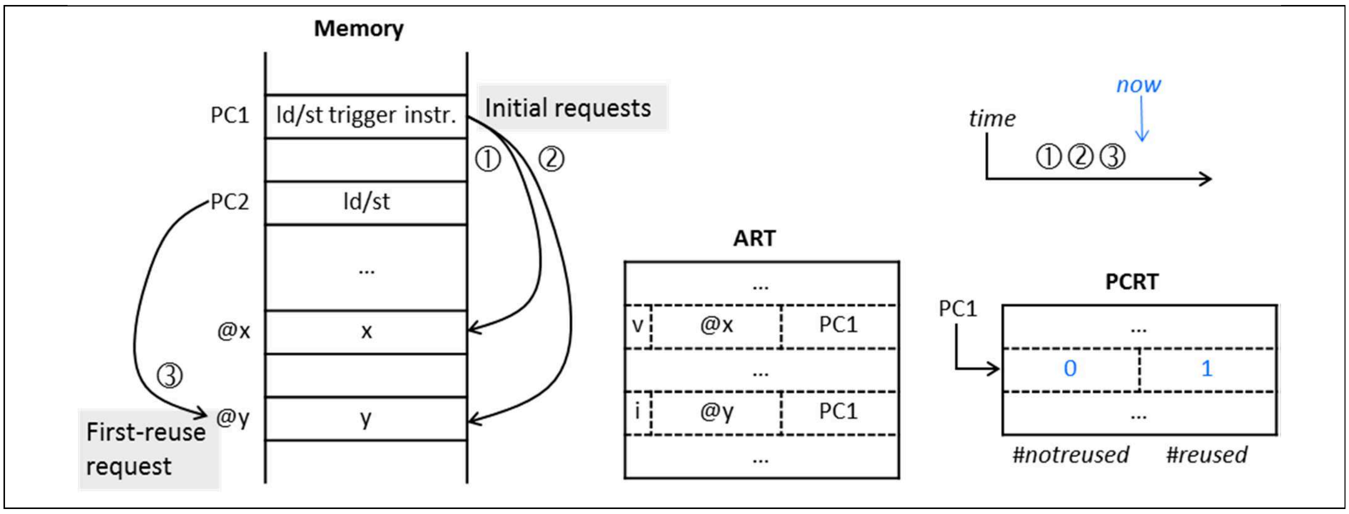

Fig. 1. State of ReD internal tables after two initial requests ①②, and a first-reuse request ③. It is assumed that the ART set shown uses PC sampling.

brought in by initial requests in a table indexed by the PC of the trigger instruction, to be able to compute the reuse probability of a new block from the values recorded by its trigger instruction. We call this table the Program Counter - Reuse Table (PCRT). The PCRT stores two counters per entry, namely *#reused* and *#notreused*. To manage these counters, some sampled sets of the ART are expanded to keep the PC of the trigger instruction together with its block address.

When a first-reuse request hits in the ART, the retrieved trigger PC is used to increment the corresponding *#reused* counter in the PCRT, and the ART entry is invalidated. When a valid (and therefore not reused) block is evicted from the ART, the associated trigger PC is used to increment the corresponding *#notreused* counter. The reuse probability of the trigger instruction can be calculated as the quotient *#reused*/(*#reused* + *#notreused*). More details are given in Section III-B.

Requests that have been marked as candidates for bypass by the ART are checked against the PCRT. If the trigger instruction is found to have a high probability of reuse, the bypass mark is ignored and the block is inserted into the LLC, to avoid the miss in the expected first-reuse L2 request.

To help visualize the whole mechanism, Figure 1 shows a schematic view of the ART and the PCRT, and their state after three requests.

III. Implementation Details

A. Address Reuse Table (ART)

The ART is organized as a set-associative buffer with 16 ways and 512 sets. We use a FIFO replacement policy that requires 4 bits per set. In order to reduce the hardware cost, the ART uses partial address tags (PAt) and is organized in sectors. An entry or sector tracks four consecutive blocks, and hence, four valid bits per entry are required to distinguish between them; see Figure 2(a). The partial tag size is 11 bits, a value that shows a good tradeoff between size and performance in our experiments.

We use a sample of 1/4 of the ART sets to gather information for the PCRT. In each entry of those ART sets, we include the PCs of the trigger instructions of the four blocks in the sector. We only store the 8 bits required to index the PCRT; see Figure 2(b).

B. Program Counter - Reuse Table (PCRT)

The PCRT is tagless and has 256 entries, a value that shows a good tradeoff between size and performance in our experiments. A tagless design with this relatively low number of entries is sufficient for ReD because it is used only as a secondary mechanism. For example, if two aliased PCs show markedly different behaviors, one with high reuse and the other with low reuse, and the PCRT categorizes their reuse probability as low, not all initial requests would be sent to the LLC, but the ART would still act correctly on first-reuse requests.

The PCRT is indexed with 8 bits of the trigger PC, bits 2-9. Each PCRT entry has two 10-bit counters (*#reused* and *#notreused*); see Figure 2(c). When a counter reaches its maximum, both counters of the entry are divided by two.

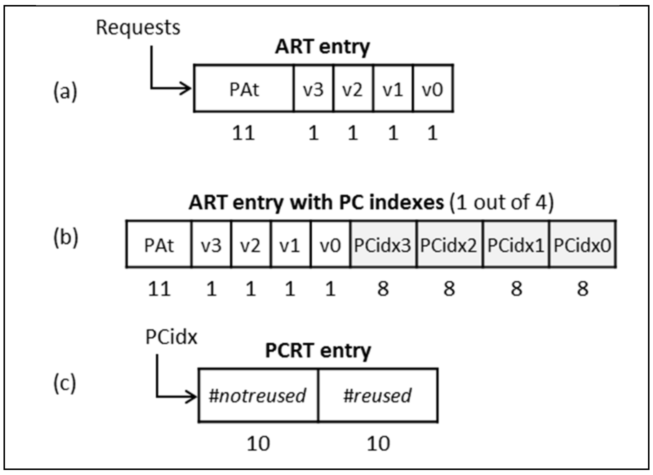

Fig. 2. Entry of the Address Reuse Table without (a) and with (b) PC sampling, respectively. Entry of the Program Counter - Reuse Table (c).

The minimum reuse probability that forces all initial requests to be sent to the LLC is set to 1/4. This value has been set experimentally, and corresponds to a *#notreused/#reused* ratio of 3.

C. *Increasing the effectiveness of the ART*

The PCRT also allows the identification of initial requests that it is not worth keeping in the ART. We use the information stored in the PCRT to reduce the insertion rate of addresses in the ART in the following two specific cases:

- Addresses coming from a trigger instruction with a very low reuse probability (less than 1/64).
- Addresses coming from a trigger instruction with a high reuse probability (more than 1/4). Since ReD already stores all blocks requested by this category of instructions in the LLC, it is not worth keeping all their initial requests in the ART.

Reducing the insertion rate of these addresses makes it possible to keep other (more useful) ones for longer, increasing the effectiveness of the ART.

The reduced insertion rate is set to 1 in 8 times. It is not advisable to reduce it to 0 because ReD needs to insert some addresses and their associated PCs in the ART for tracking changes over time in the behavior of trigger instructions. It is also important to address thrashing in the ART: the reduced insertion rate enables the ART to store at least a portion of the thrashing working set.

D. *Other details*

The base replacement policy used for the CRC-2 submission and considered in the results section of this paper is 2-bit SRRIP. On insertion, it is applied only if ReD decides not to bypass a block.

Prefetch requests are handled like demand requests. Write-back requests are ignored by ReD and SRRIP. If they miss, they are always allocated in the LLC, but with minimum priority. The simulation infrastructure does not allow bypassing them.

E. *Storage costs*

Table I summarizes the storage costs per core of ReD (ART and PCRT), plus the costs of SRRIP.

IV. RESULTS

We have simulated our policy using the ChampSim simulator of the CRC-2. We have considered the four configurations defined: single core without prefetching (c1), single core with data prefetching (c2), four-core without prefetching (c3) and four-core with data prefetching (c4).

For single-core configurations, we have used 45 traces from different parts of the execution of the 29 applications of the SPEC CPU 2006 benchmark suite. For multi-core configurations, we have created 80 mixes using these 45 traces.

TABLE I. ReD HARDWARE COST, PER CORE

ART	Parameters	512 sets, 16 ways, 4 blocks/sector
	# bits / entry	11 tag, 4 valid
	# bits / set	4 (FIFO replacement)
	Cost	512 * (16 * 15 + 4) = 124928 bits = 15616 bytes
ART sampled sets	Parameters	128 sets, 16 ways, 4 blocks/sector
	# bits / entry	4 * 8 bits PC
	Cost	128 * 16 * 32 = 65536 bits = 8192 bytes
PCRT	Parameters	256 entries
	# bits / entry	2 * 10
	Cost	256 * 20 = 5120 bits = 640 bytes
SRRIP	Parameters	2048 sets, 16 ways
	# bits / entry	2
	Cost	2048 * 16 * 2 = 65536 bits = 8192 bytes
Total cost: 15616 + 8192 + 640 + 8192 = **32640 bytes (31.875 KB)**		

Figure 3 shows results achieved with our proposed policy, and additionally for SRRIP as a reference. For the single-core configurations (c1 and c2), we plot speedup over LRU, while for multi-core configurations, we plot average speedup, over all instances of the trace in all mixes, relative to the performance with LRU. We only show results for traces that achieve more than a 2% speedup when increasing LLC capacity from 2M to 8M with the LRU replacement algorithm, in a single-core configuration. The geometric mean of speedups over all selected traces is 4.4% using configuration c1, 2.4% using c2, 5.6% using c3 and 3.6% using c4.

Over the non-plotted traces, 26 in total, the geometric mean of speedups is 0.1% in c1, 0.2% in c2, 1.5% in c3 and 1.4% in c4. The average bypass rate of the plotted executions, using configuration c1 is 32.8%, with a maximum of 82.1% in 429.mcf.

V. CONTRIBUTIONS

The main contributions of our work are:

- We focus only on the block selection policy (that is, the decision of whether or not to bypass the cache), as we believe it is the key component of the LLC replacement policy. ReD can be combined with any other LLC replacement policy, either by adding it as a block selection policy or by substituting for the one used in the base policy.
- We design a block selection policy that combines, in a synergistic way, two different approaches to computing the reuse likelihood of a block that misses the LLC: *a*) the detection of a recent-past use of the block as an indicator of future reuse, and *b*) the past reuse behavior of blocks requested by the instruction that requests the block.
- We design a separate block reuse detector that remembers addresses that have recently missed in the LLC. In other policies that have been proposed, the LLC cache is used to perform a similar task [2].

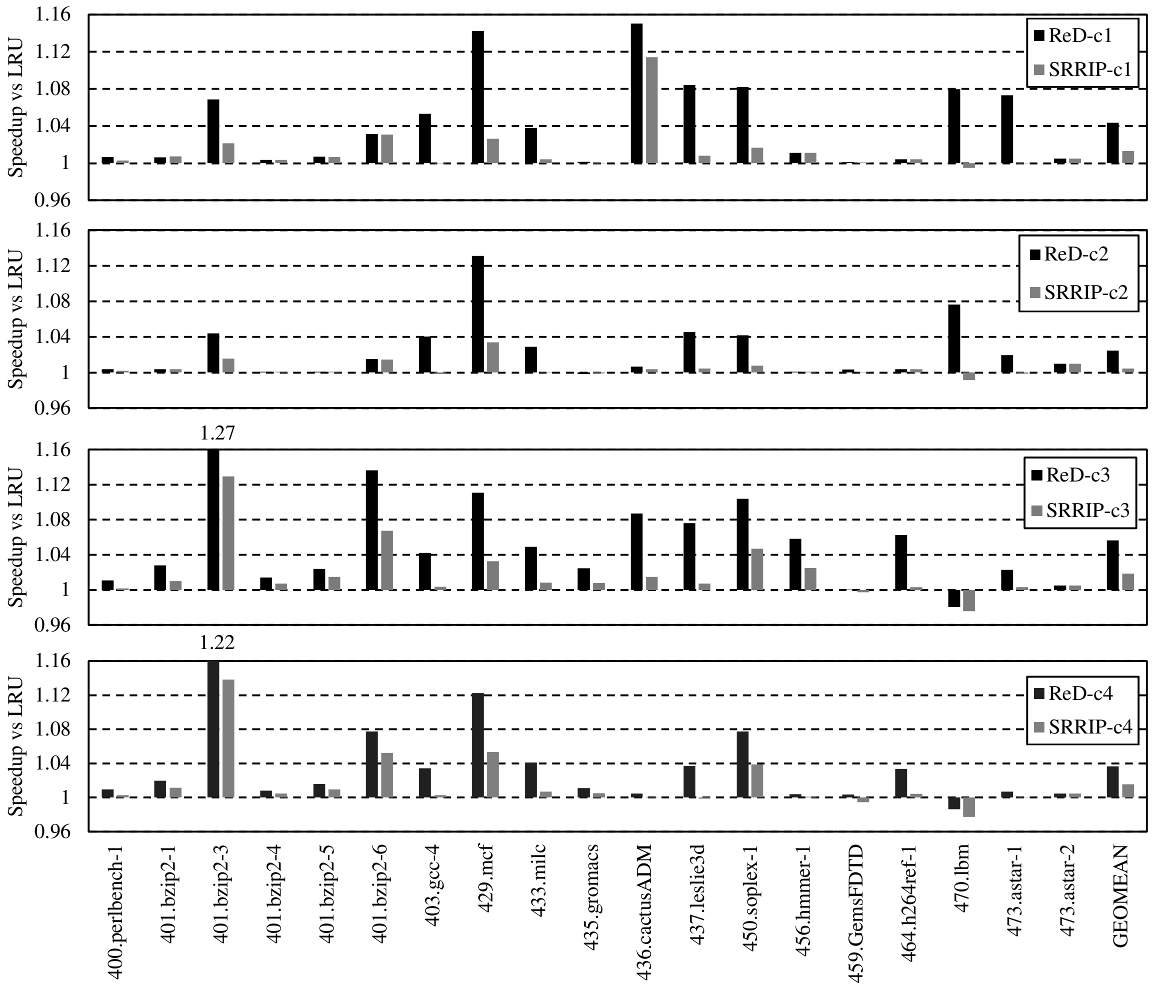

Fig. 3. Performance results: Speedup vs LRU for ReD and SRRIP. Results for all Spec2006 benchmarks that show more than a 2% improvement in IPC between a 2MB and a 8MB LRU-managed LLC. From top to bottom: c1) single core without prefetching, c2) single core with data prefetching, c3) four-core without prefetching, and c4) four-core with data prefetching.

- We also include a PC-indexed store that tracks the reuse of blocks requested by each instruction, and is able to predict reuse behavior in some cases. A similar table has been used in a previous study [8], but we use it in a different way. First, we train it with the reuse observed in the address detector instead of the LLC, and second, in ReD, it is a secondary mechanism that only acts in specific cases: to avoid the miss of the first-reuse request and to reduce the number of insertions into the address detector.
- Both mechanisms are implemented in private per-core tables, to ensure a fair distribution of resources and to avoid potential thrashing caused by a single thread.

ACKNOWLEDGMENTS

This work was supported in part by grants TIN2013-46957-C2-1-P, TIN2016-76635-C2-1-R (AEI/FEDER, UE), TIN2015-65316-P, Consolider NoE TIN2014-52608-REDC (Spanish Gov.), and gaZ: T48 research group (Aragón Gov. and European ESF).

REFERENCES

[1] J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería. 2013. Exploiting reuse locality on inclusive shared last-level caches. ACM Trans. Archit. Code Optim., 9(4):38. 1-19.

[2] J. Albericio, P. Ibáñez, V. Viñals and J.M. Llabería. 2013. The reuse cache: downsizing the shared last-level cache. In Proc. of the 46th Int. Symp. on Microarchitecture. 310-321.

[3] Gao and C. Wilkerson. 2010. A dueling segmented LRU replacement algorithm with adaptive bypassing. In Proc. of the 1st JILP Workshop on Computer Architecture Competitions.

[4] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In Proc. of 37th Int. Symp. on Computer architecture. 60-71.

[5] S. M. Khan, Y. Tian, and D. A. Jimenez. 2010. Sampling Dead Block Prediction for Last-Level Caches. In Proc. of the 43rd Int. Symp. on Microarchitecture. 175-186.

[6] P. Michaud. 2010. The 3P and 4P cache replacement policies. In Proc. of the 1st JILP Workshop on Computer Architecture Competitions.

[7] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely Jr., and J. Emer. 2007. Adaptive Insertion Policies for High-Performance Caching. In Proc. of the 34th Int. Symp. on Computer Architecture. 381–391.

[8] C. Wu, A. Jaleel, W.Hasenplaugh, M. Martonosi, S. C. Steely, Jr., J. Emer. 2011. SHiP: signature-based hit predictor for high performance caching. In Proc. of 44th Int. Symp. on Microarchitecture. 430-441.