ReD: A Reuse Detector for Content Selection in Exclusive Shared Last-Level Caches

JAVIER DÍAZ, Aragón Institute of Engineering Research (I3A), University of Zaragoza, and Hipeac
TERESA MONREAL, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya and Hipeac
PABLO IBÁÑEZ, Aragón Institute of Engineering Research (I3A), University of Zaragoza, and Hipeac
JOSÉ M. LLABERÍA, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya and Hipeac
VÍCTOR VIÑALS, Aragón Institute of Engineering Research (I3A), University of Zaragoza, and Hipeac

Manuscript communications:
imarin@unizar.es
+34 976 76 21 07

ABSTRACT

The reference stream reaching a chip multiprocessor Shared Last-Level Cache (SLLC) shows poor temporal locality, making conventional cache management policies inefficient. Few proposals address this problem for exclusive caches. In this paper, we propose the Reuse Detector (ReD), a new content selection mechanism for exclusive hierarchies that leverages reuse locality at the SLLC, a property that states that blocks referenced more than once are more likely to be accessed in the near future. Being placed between each L2 private cache and the SLLC, ReD prevents the insertion of blocks without reuse into the SLLC. It is designed to overcome problems affecting similar recent mechanisms (low accuracy, reduced visibility window and detector thrashing). ReD improves performance over other state-of-the-art proposals (CHAR, Reuse Cache and EAF cache). Compared with the baseline system with no content selection, it reduces the SLLC miss rate (MPI) by 10.1% and increases harmonic IPC by 9.5%.

Key Words:

Chip multiprocessor, shared last-level cache, exclusion, reuse, content selection, bypass

1. INTRODUCTION

Nowadays, chip multiprocessor (CMP) systems dominate the market in high-performance servers, desktop or embedded systems, and mobile devices. Their most common design includes a multilevel memory hierarchy, ending with a shared last-level cache (SLLC). This cache is critical in terms of cost and performance. In cost, because it occupies nearly 50% of the chip area. In performance, because it is the last resource before accessing the DRAM memory, located outside the chip, which is much slower.

Several studies show that conventional SLLC designs are inefficient because they waste a large portion of the cache. This is because they hold many dead blocks, i.e., blocks that are never re-accessed before their eviction. Frequently, those blocks are already dead when they enter the SLLC [Khan et al. 2010; Qureshi et al. 2007; Gaur et al. 2011]. This occurs in multilevel hierarchies because private caches, often encompassing two levels (L1 and L2), exploit most of the temporal locality, which is effectively filtered out before reaching the SLLC [Jaleel et al. 2010a; Jaleel et al. 2010b]. To address this drawback and increase the SLLC hit rate, several proposals suggest new SLLC insertion and replacement policies. Most of the work refers to inclusive or non-inclusive caches, and only a small group [Gaur et al. 2011; Chaudhuri et al. 2012] focuses on exclusive SLLCs [Jouppi and Wilton 1994].

Exclusive or partially exclusive SLLCs are already being used in some chip multiprocessors [Conway et al. 2010]. The aggregate on-chip capacity of private caches increases with the number of cores, thus making exclusive hierarchies more appealing than inclusive ones. Over the next few years, we can expect many-core designs with more cores within the chip, and SLLCs not much larger than the current ones [Lotfi-Kamran et al. 2012]. Therefore, using an inclusive cache will be even more inefficient and, unless there are drastic changes in the basic design of the memory hierarchy, the usefulness of exclusive SLLCs will grow in the future [Jaleel et al. 2015].

This work focuses on enhancing the efficiency and performance of an exclusive SLLC in a chip multiprocessor. Exclusive caches offer the opportunity to implement a content selection (bypass) mechanism with low complexity, in contrast to inclusive hierarchies. Our proposal is also an efficient solution to reduce traffic from private caches to the SLLC, one of the drawbacks of exclusive designs.

Specifically, we propose a new content selection mechanism to take advantage of the reuse locality existing in the stream of requests to the SLLC. A block is said to have reuse locality if it has been referenced at least twice. A block with reuse locality is more likely to be accessed in the near future [Albericio et al. 2013a].

After an in-depth analysis of previous proposals that exploit reuse locality, we have identified three aspects where there is still room for improvement:

- Most of them predict reuse by linking it with a cache block feature, such as the instruction that brought the block into the SLLC or the memory area the block belongs to. The accuracy of these predictors is usually low.
- Most proposals detect reuse by keeping track of past accesses in a store embedded into the SLLC. In such proposals, the size of the SLLC restricts the number of detected blocks. They effectively lengthen the life of the blocks flagged as reused in the SLLC, but they are not able to detect further blocks.
- As far as we know, global thrashing may appear in all of them, since the reuse detection mechanism is shared among all the threads running on the CMP. A thread bringing too many blocks in the on-chip hierarchy can prematurely replace existing data from other applications, worsening their reuse detection.

The aim of our proposal is to fill up these gaps. To achieve this, we monitor the traffic between L2 and the SLLC, by means of a specialized store that remembers addresses of cache blocks recently evicted from each L2 cache. That address store, called hereafter the Reuse Detector or ReD, detects which blocks of those evicted from L2 do not have reuse, and avoids inserting them into the SLLC. Dirty blocks are sent directly to main memory (bypass). ReD is a separate private store near each L2 cache, sized and organized regardless of the SLLC configuration.

We evaluate ReD using a set of multiprogrammed workloads running on a chip multiprocessor with eight cores, a three-level cache hierarchy, and TC-AGE as SLLC replacement policy [Gaur et al. 2011]. Results show that the Reuse Detector enhances performance, above other recent

proposals such as CHAR [Chaudhuri et al. 2012], Reuse Cache [Albericio et al. 2013b], and EAF cache [Seshadri et al. 2012].

The work is structured as follows. Section 2 explains the motivation for this work. Section 3 describes in detail the proposed Reuse Detector. Section 4 gives insight into the ReD operation. Section 5 details the methodology used, including the experimental environment and the configuration of the simulated systems. Section 6 presents results and explores the trade-offs in the design of ReD. Section 7 presents and analyzes additional results, and compares them against other relevant proposals. Section 8 reviews the state of the art in the matter. Finally, in Section 9 we summarize our conclusions.

2. MOTIVATION

2.1 Problem analysis

Several studies have shown that, in a memory hierarchy, most of the blocks have already received all accesses when they are evicted from the caches close to the processor. Caches that are further away from the processor are used inefficiently because the stream of references that reaches them has very little temporal locality. Instead, these references show *reuse locality*. The reuse locality property has been empirically proved in several works [Gaur et al. 2011; Chaudhuri et al. 2012; Albericio et al. 2013a; Albericio et al. 2013b]. It can be stated as follows: lines accessed at least twice tend to be reused many times in the near future.

An experiment is conducted to quantify the number of blocks with reuse and the amount of reuse. Figure 1 plots a classification of the blocks evicted by an exclusive SLLC depending on the number of SLLC accesses that each block registered during their stay in the on-chip caches. Each block is classified according to whether it has received a single access (U), two accesses (R, reuse), or more than two accesses (M, multiple reuse). The average number of reuses for each M block is shown on top of the bars. We use TC-AGE as replacement policy. The figure shows the distribution for eight applications running together in an example mix.

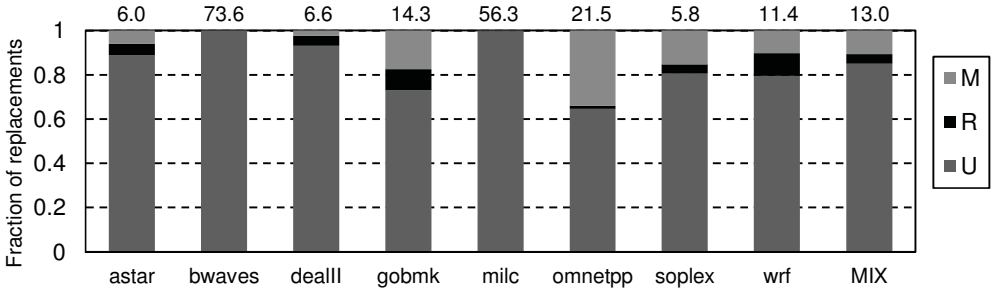

Figure 1: Fraction of blocks evicted from the SLLC cache, in an example mix, according to the number of accesses received before its eviction: (U) one access, (R) two accesses, and (M) three or more accesses. Figures on top of each bar show the average number of reuses for an M block.

On average, 85% of the blocks do not receive any hit in the SLLC (U). These blocks could bypass the SLLC without loss of performance. Blocks with only one reuse (R) are 4% of the total, and those with more reuses (M) are 11%. For each block classified as M, there are 13.0 reuses on average. A content selection policy that only stored blocks with reuse (at least two accesses, R + M) would keep the small set of blocks that produces most hits. Furthermore, this policy would prevent the storage in the SLLC of the large set of U blocks, reducing the likelihood of M blocks being replaced. Our proposal is a mechanism that detects the second use of a block to classify it as reused, and only stores these reused blocks in the SLLC.

2.2 Reuse detector design

We have analyzed previous mechanisms designed to classify blocks as reused for an SLLC and have identified three aspects where they could be improved:

Prediction accuracy. Most works predict reuse by linking it with some cache block feature, such as the instruction that brought the block into the SLLC or the memory area the block belongs

to [Qureshi et al. 2007, Jaleel et al. 2010b, Wu et al. 2011, Gaur et al. 2011, Chaudhuri et al. 2012, Li et al. 2012]. The accuracy of these predictors is limited. As an example, a mechanism that associates SLLC reuse with PC will only be accurate if most blocks brought by each PC present the same behavior, whether having reuse in the SLLC or not. However, accuracy will drop if some blocks brought by a PC have reuse in SLLC, but others have not. Our proposal relies on detection instead of prediction of the reuse locality. Our mechanism keeps the addresses of all blocks evicted from the private caches during a certain time window (reuse detection window). The eviction from private caches of a block whose address is already stored is a true indicator of reuse locality, and thus the block is tagged as such. In Section 7 we provide results to test the accuracy of our proposal against CHAR, a state of the art predictor [Chaudhuri et al. 2012].

Thread-global reuse detection. All the previous proposals have in common an important constraint: their reuse detector is shared among all threads running on the CMP. A thread delivering lots of misses will cause premature evictions of addresses previously inserted into the detector by other threads, restricting the detector's ability to discover more reuse for those other threads. In other words, a thread missing a lot in its private caches shrinks the reuse detection window of the remainder applications. In fact, these mechanisms reproduce in the reuse detector the thrashing problem they try to avoid in the SLLC. To overcome this, we propose implementing reuse detectors that are private to each core. Each detector is placed next to the private L2 cache, and remembers the addresses of all blocks evicted only by its associated L2 cache. In Section 7 we provide results to compare the amount of reuse detected by our proposal versus a global detector.

Size of the reuse detector. A larger detector size allows it to remember more blocks for longer, thereby increasing the opportunity to classify more blocks as reused. Most previously proposed techniques track reuse patterns using the SLLC [Gaur et al. 2011; Chaudhuri et al. 2012; Albericio et al. 2013a; Gao and Wilkerson 2010; Khan et al. 2012; Wu et al. 2011; Li et al. 2012]. In these proposals, the SLLC size defines the size of the reuse detector and, consequently, this detector is not able to discover more reuse than an LRU-managed SLLC of the same size would do [Mattson et al. 1970]. In other words, blocks categorized as reused do not increase in number. In fact, increasing the lifespan of reused blocks indirectly shortens the life for those blocks that have not yet shown reuse, due to the capacity limit. This leads to a reduction in the detection window size relative to a cache with an LRU replacement policy. Our proposal aims to increase the number of blocks detected as reused. This requires a larger detection window. To achieve this, we include an additional store that is able to remember more block addresses than the SLLC can keep. In Section 7 we provide results that show the amount of reuse detected by our proposal with detection windows of different sizes.

Our ReD proposal is an efficient content selection policy that detects with high accuracy when a block has been reused, and only stores these reused blocks in the SLLC. This reduction in the traffic of unused blocks enables a SLLC keeping more blocks with reuse and for longer time. Compared with previous proposals that also exploit reuse locality, ReD is more accurate detecting reused blocks, permits a greater visibility window, and does not suffers from global thrashing. In fact, among all those proposals, ReD is the only that manages to have on average more alive than dead blocks in the SLLC. The use of a private and separate store makes the SLLC replacement policy not adversely affect the detector efficiency, so ReD can be implemented in an SLLC managed with any replacement policy. ReD is specifically designed for exclusive SLLCs and chip multiprocessor systems, turning out to be a bypass mechanism that has low complexity, is simple and easy to implement.

3. DESIGN AND IMPLEMENTATION OF THE REUSE DETECTOR

3.1 Baseline design

The baseline design is a three-level cache hierarchy consisting of an SLLC whose contents are managed in exclusion with respect to the contents of two-level private caches, which are inclusive.

Coherence is kept by means of a directory that holds, for each block in the hierarchy, both its status and precise location, which can be one or several private caches or the SLLC.

Blocks coming from main memory are sent directly to the requesting L2 cache. Eventually, when a block is evicted from L2 it is sent to SLLC. From here on, either the block is requested again from any L2 cache, then being sent and invalidated in SLLC, or it is replaced by another block that needs room for insertion. If a block placed in an L2 cache is requested by another L2 cache, the directory detects this situation and the block is retrieved from the former to be delivered to the latter. Shared blocks are inserted in the SLLC only when the last copy is evicted from the L2 caches.

3.2 The Reuse Detector

We propose placing our Reuse Detector next to every L2 cache, in the path from each L2 cache to SLLC. ReD receives the addresses of every block evicted from the corresponding L2 cache, see Figure 2. Being located outside the critical path from SLLC to L2 caches, ReD does not affect the SLLC read latency. Instead, it slightly increases the time that a block evicted from the L2 cache takes to be sent to SLLC or main memory.

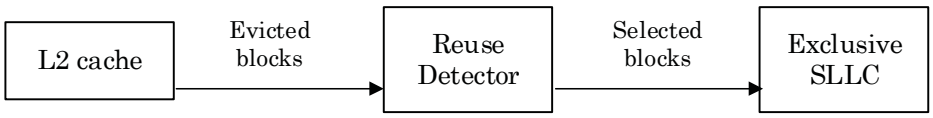

Figure 2: Placement of the Reuse Detector. Every private L2 cache has one ReD.

When a block is evicted from the L2 cache, ReD decides between sending the block to the SLLC and bypassing it. The decision is driven by the block reuse history: if a block has a single use, it is bypassed. If it has one or more reuses, it is stored in the SLLC.

A block is classified as reused if it satisfies one of these conditions:

- The Reuse Detector remembers the address of the block. ReD is a buffer storing block addresses coming from L2 evictions. It detects whether it is the first time the block is evicted from L2 or it has already experienced a previous eviction. A first eviction means no reuse detected, while following evictions mean reuse.
- The block was provided to the private cache by the SLLC. We add one bit to each L2 cache block, the Reuse bit, which remembers whether the block came from SLLC or main memory.

ReD is structured as a set associative buffer, and its capacity, associativity, and replacement policy are design parameters. We define ReD capacity as the number of tracked evicted blocks times block size. For instance, a ReD able to track 1024 blocks of 64B has a capacity of 64KB. The ReD capacity is a metric that allows us to measure its tracking potential relative to the SLLC size. Capacity is a key parameter, since an effective reuse detection requires storing a significant number of addresses between consecutive L2 evictions of a given block.

Neither the SLLC nor the directory require structural changes to be adapted to the new mechanism. In order to take into account a possible bypass action, the coherence protocol and control logic will need to be adapted. In addition, our mechanism requires to add the Reuse bit to each L2 cache block.

3.3 Reuse Detector operation

Figure 3 shows a diagram illustrating the operation of ReD.

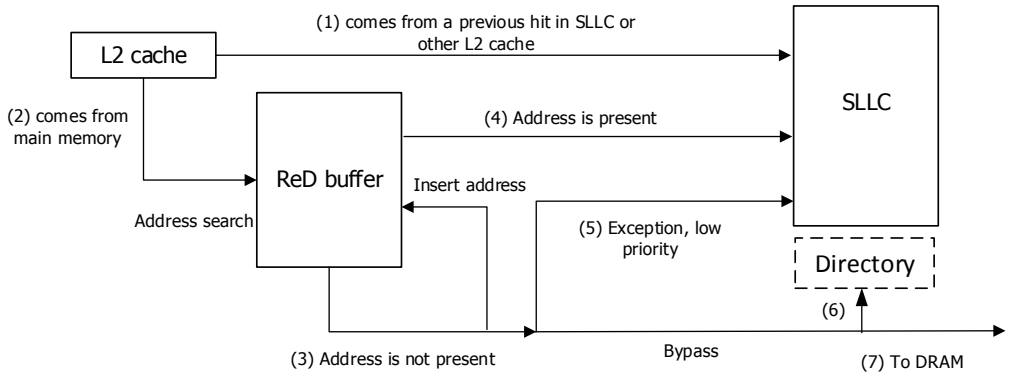

Figure 3: Reuse Detector operation.

On L2 evictions the Reuse bit is first checked. If the evicted block came from SLLC or another private cache, it is stored again in the SLLC, without looking up ReD (1). Otherwise, if the evicted block came from main memory, its address is looked up in ReD (2). A miss means no reuse, so the block is bypassed, but the address is added to ReD (3). A hit means reuse, so the block is sent to SLLC (4). Bypassed blocks send a control message to update the directory (6). Then, clean blocks are discarded and dirty blocks are written to main memory (7).

As an exception, a small fraction of bypassed blocks are sent to SLLC with "low insertion priority" (5). This means the SLLC will store them only if there are free ways in the corresponding set; moreover, those blocks will be inserted with the highest replacement priority. This exceptional filling policy comes from the observation that exclusive SLLCs experiencing at the same time many hits and bypasses may present many empty ways. Experimentation has shown us that diverting to SLCC one of every 32 bypassed blocks takes advantage of the free space and increases performance.

3.4 Implementation details

We implement the ReD buffer as a set associative cache, with entries containing tags for addresses, valid bits, and some bits for the replacement policy. We will use a 16-way ReD buffer; higher associativities lead to hardly any performance improvements. In order to reduce the ReD area, we propose storing sector tags and compressing them.

A sector is a set of consecutive blocks aligned to the sector size, a power of two. As our design requires per-block reuse tracking, every sector tag needs as many valid bits as the number of blocks a sector has. For example, a ReD sector size of four blocks, requires entries with four valid bits.

Compression aims to shorten the tag size while maintaining good ability to distinguish between sectors. To compress we propose the following bit folding: let t and c be the number of bits of the entire and compressed tags, respectively. The t bits are split into consecutive pieces of size c, filling with zeroes if the last piece does not consist of c bits. Then, the compressed tag results from an XOR operation to all pieces.

False positives may appear by using compression, as several sectors share the same compressed tag. Therefore, it might happen that blocks without reuse get inserted into the SLLC. Such wrong insertion is not a functional error, but can hurt performance. So, the right number of bits is a tradeoff between area and performance.

Our experiments point out that using a FIFO replacement policy to manage the ReD buffer works fine. FIFO replacement means that the age of an address relates to its insertion (first use) and not to its last access. This is in line with the buffer goal of finding out the first reuse of a block.

3.5 Hardware costs

In this section, we calculate the total number of bits required to implement the ReD buffer we attach to each L2 cache.

After trading off performance and cost (see Section 6), the chosen configuration has a capacity of 2MB, resulting from having 1K 16-way sets, a sector size of two blocks, and 10-bit tags. This balanced ReD requires 12 bits (10 tag bits and 2 valid bits) per entry, and four FIFO bits per set.

The number of entries for each ReD is 16K, which means 24.5 KB per core. The total size for the eight cores is 196 KB, a 2.3% of an 8 MB SLLC.

4. RED OPERATION INSIGHT

In this section we use an example workload to analyze in depth the ReD operation, and how it is able to reduce the SLLC miss rate.

We plot how ReD classifies the blocks it receives, into five classes: first use (U), first reuse (R), multiple reuse detected only by ReD (MD) [1], multiple reuse detected only because the block comes from SLLC or another private cache (MC), and multiple reuse detected by both mechanisms (MA). Figure 4 shows the distribution for the eight applications of an example mix.

A L2 eviction of a block classified as U causes an SLLC bypass, while an eviction of a block classified as any other class causes the insertion of the block into the SLLC. Evictions of blocks classified as U, R or MD denote that the block comes originally from main memory, while blocks classified as MC and MA denote a previous hit in the SLLC or in another private cache.

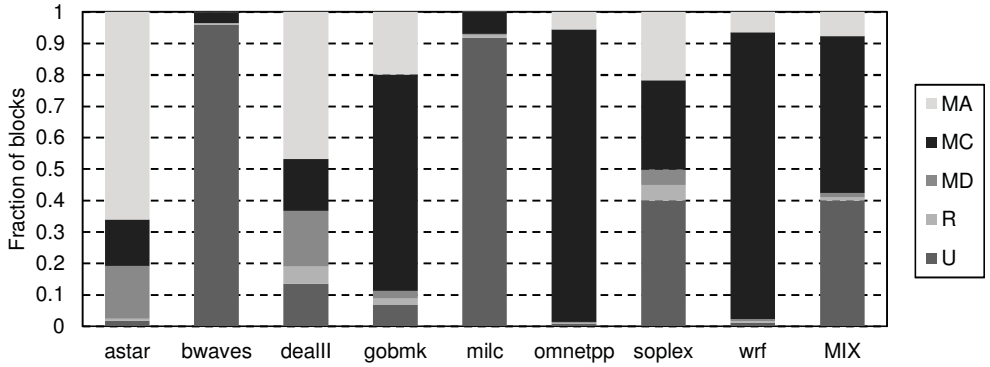

Figure 4: Fraction of blocks evicted from L2 caches, in an example mix, categorized from the ReD standpoint according to the type of reuse in: (U) first use, (R) first reuse, (MD) multiple reuse detected only by ReD, (MC) multiple reuse detected only because the block comes from SLLC or another private cache, and (MA) multiple reuse detected by any of them.

As shown, the amount of bypass varies from one program to another. In *bwaves* and *milc*, more than 91% of blocks evicted from L2 show a single use, and are bypassed. This is consistent with the measurements presented later in Table 1, which show that the SLLC miss rates for these programs are very high. At the opposite extreme, in *astar*, *omnetpp* and *wrf* less than 2% of the blocks evicted from the private caches do not show reuse, so there is almost no bypass for these applications. The rest of the programs present intermediate figures: *dealII*, *gobmk* and *soplex*, with bypass levels of 13%, 7% and 40%, respectively.

The number of blocks that are sent to the SLLC after detecting their first reuse (R class) varies between 0.2% and 5.5% for *omnetpp* and *dealII*, respectively, with an average of 1.0%. These few blocks showing a first reuse are accessed later multiple times (MD, MC, and MA classes). On average, for each block classified as R (first reuse), ReD detects 61 additional reuses.

Blocks classified as MD have been previously detected by ReD, classified as reused and stored in the SLLC, but they have been prematurely evicted from there. As ReD has a detection window that is larger than the SLLC, it can detect this situation and re-insert them into the SLLC. This occurs on average 13% of the times a multiple reuse is detected.

The bypass of blocks without reuse in *bwaves*, *milc*, *dealII* and *soplex* allows the SLLC to better preserve the useful blocks from these programs, because they will not be evicted as often. Moreover, other programs of the mix will also benefit from this.

Figure 5 shows the average fraction of the SLLC occupied by each program, for the baseline system and for ReD (left). It also shows the SLLC MPI reduction of each application with respect to the baseline (right). *Bwaves* and *milc* take much less SLLC space with ReD. However, the block selection done by ReD does not harm their performance, since both maintain a miss rate similar to that of the baseline (0.9% worse in *bwaves*). For the rest of the programs, having more space in the

[1] Although the proposed ReD hardware cannot distinguish between the R and MD classes, they were separated in the figure to illustrate that both reuse detection mechanisms are complementary.

SLLC and a better block selection mechanism allows them to keep more blocks with reuse and for a longer time, resulting in reductions in the SLLC MPI between 30.4% and 97.3% for *soplex* and *omnetpp*, respectively. The MPI reduction for the whole mix[2] is 24.0%, and the normalized hIPC is 1.28.

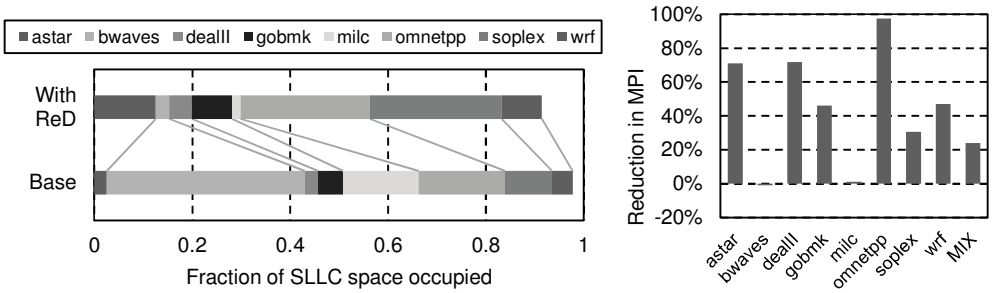

Figure 5: Left: fraction of the SLLC occupied by blocks of each program in the example mix. We take data every million cycles and show the average. Right: SLLC MPI reduction of ReD with respect to the base system.

5. EXPERIMENTAL SETUP

This section details the experimental framework and the configuration of the baseline system we use to evaluate the proposal.

5.1 The experimental framework

As a simulation engine we use the Simics full-system simulator [Magnusson et al. 2002], and the plugins Ruby and Opal from the GEMS Multifacet toolset [Martin et al. 2005] and DRAMSim2 from the University of Maryland College Park [Rosenfeld et al. 2011]. Ruby is used to accurately model the memory hierarchy of the CMP system: caches, directory, coherence protocol, on-chip network, buffering, and blocking of components. Opal (also known as TFSim) is used to model in detail a superscalar out-of-order processor. DRAMSim2 is used to model a cycle-accurate DDR3 memory system.

Our Simics platform simulates SPARC cores managed by Solaris 10, and runs a multiprogrammed workload made of applications from the SPEC CPU 2006 suite [Henning 2006]. For our system with 8 processors we have generated a set of 100 mixes, composed by random combinations of 8 benchmarks each, taken from among all the 29 included in the SPEC CPU 2006 benchmark suite. Each program appears between 18 and 41 times, this representing an average of 27.6 times with a standard deviation of 6.1.

In order to identify initialization phases, we run until completion all the SPARC binaries, with the reference inputs, on a real machine. During the execution we use hardware counters to detect the end of the initialization phase of each benchmark. For every mix, we ensure that no application was in its initialization phase by fast-forwarding the simulation until all the initialization phases are finished. Starting at this point, we first run 300 million cycles to warm up the memory system, and then collect data statistics for the next 700 million cycles.

The first three columns in Table 1 show the average number of misses per kilo-instruction (MPKI) in all three levels of the memory hierarchy. These figures are averages for each benchmark in all mixes in which appears, and when the eight benchmarks in each mix run together on the base system. The last column shows average multi-processor IPC.

Benchmark	L1	L2	LLC	IPC	Benchmark	L1	L2	LLC	IPC
astar	7.5	1.1	0.7	1.17	libquantum	45.8	33.2	32.2	0.28
bwaves	24.5	21.1	20.1	0.66	mcf	64.9	36.0	18.9	0.18
bzip2	8.4	3.9	0.9	1.30	milc	24.6	23.5	22.0	0.23
cactusADM	20.8	11.4	4.9	0.64	namd	1.7	0.2	0.2	3.16
calculix	8.5	4.3	1.5	1.61	omnetpp	12.6	9.2	2.2	0.63
dealII	1.6	0.5	0.3	2.69	perlbench	10.2	1.8	0.8	1.37

[2] If we order our 100 workloads, described in Section 5, from higher to lower normalized hIPC, this example mix is in position number 8.

gamess	6.7	1.0	0.6	2.60	povray	11.5	0.2	0.1	2.65
gcc	22	6.4	2.1	0.78	sjeng	6.9	0.8	0.5	1.28
gemsFDTD	42.7	29.7	22.8	0.45	soplex	8.9	7.1	3.1	0.63
gobmk	13.2	1.1	0.3	1.23	sphinx3	18.8	14.3	11.7	0.26
gromacs	11.7	3.0	1.2	1.60	tonto	6.7	1.3	0.5	2.18
hmmer	3.3	2.4	0.2	2.50	wrf	14.3	8.9	1.5	2.26
h264ref	4.2	1.4	0.7	1.36	xalancbmk	15.1	8.7	2.8	0.68
lbm	65.4	38.6	36.7	0.21	zeusmp	32.3	8.7	7.2	0.87
leslie3d	40.4	23.2	17.9	0.58					

Table 1: L1, L2 and LLC: Average MPKI at each cache level of the base system (exclusive SLLC with 8 MB and TC-AGE replacement policy). IPC: average multi-processor IPC.

5.2 Configuration of the baseline system

We model a base system of eight superscalar processors with speculative out-of-order execution. Each processor has a 4-wide pipeline of 18 stages and 10 functional units. Branch prediction uses a YAGS cache structure [Eden and Mudge 1998] with a direction pattern history table (PHT) of 4K entries. Table 2 summarizes all the parameters of the simulated processor.

Base architecture	SPARC v9
Cores	8, 4-way superscalar, 4 GHz
Pipeline	18 stages: 4 fetch, 4 decode, 4 dispatch/read, 1 (>4) execute, 3 memory, 2 commit
ROB size	128 entries
Register Files	int: 160 (logical) + 128 (rename) FP: 64 (logical) + 128 (rename)
Functional units	4 int, 4 FP, 2 load/store
Branch prediction	YAGS cache structure with a PHT of 4.096 entries

Table 2: Processor parameters

Each processor core has a two-level private cache hierarchy, being the exclusive third and last level cache shared among all the cores. The SLLC has a total size of 8MB, and is split into four banks that are cache line interleaved (64B).

A crossbar network connects the eight processors to the four SLLC banks. The DDR3 memory system is accessed through two memory channels running at a frequency of 667 MHz. Table 3 shows all the details of the cache hierarchy we simulate.

Private cache L1 I/D	32 KB, 4-way, LRU replacement, block size of 64 B, 3 cycles access latency
Private cache L2 unified	256 KB in inclusion with L1, 8-way, replacement LRU, block size of 64 B, 7 cycles access latency
Network	Crossbar, 80 bits bus width, 5 cycles latency
Shared cache L3 (SLLC)	8 MB exclusive (4 banks of 2 MB each), block interleaving, block size of 64 B. Each bank: 16-way, TC-AGE replacement with 2 bits, 10 cycles access latency, 32 demand MSHR
DRAM	Device Micron 32M 8B x8, 2 channels, 2 ranks per channel, 8 devices per rank, 8 GB total.
DRAM bus	2 channels at 667 MHz, Double Data Rate (DDR3 1333 MHz), 8 B bus width, 4 DRAM cycles/line, 24 processor cycles/line

Table 3: Memory hierarchy parameters

5.3 SLLC replacement policy

We use TC-AGE as the SLLC replacement policy for the baseline system because this policy has proved to be very efficient in exclusive SLLCs [Gaur et al. 2011]. It is equivalent to SRRIP for inclusive caches. It uses two bits to store the age of each cache line. The age is assigned when the block is inserted into the SLLC: if the block has previously received a hit in the SLLC, it is inserted with age 3, otherwise it is tagged with age 1. Each block in the private L2 cache stores one additional trip count bit to remember if it has had a hit in SLLC (the TC bit). This bit is also sent to the SLLC with the block when it is evicted from the L2 cache. TC-AGE selects a random victim

among those blocks in the younger group (age 0). If there is no block with age 0 in the cache set, the age of all blocks is decremented, and the victim selection restarts. In summary, TC-AGE assigns older age, and therefore less likelihood of replacement, to blocks that have been reused.

When ReD is implemented on an SLLC with TC-AGE replacement, only one bit is needed in the L2 cache to map the Reuse bit and the TC bit. So, in this case, our mechanism does not have any overhead in the L2 caches. From the TC-AGE perspective, this bit maintains the same meaning: it remembers whether the block came from SLLC or main memory. That is, it is reset when ReD discovers a first reuse (second time a block is evicted from L2); it is set in the subsequent L2 evictions (a block has already received at least three accesses). Therefore, TC-AGE is driven to give higher replacement priority in the SLLC to the blocks having one reuse, and less to those having multiple reuses.

Although in this work we use TC-AGE as SLLC replacement policy, it is possible to implement ReD with any other policy.

5.4 Performance metrics

Two performance metrics are mainly used: the harmonic mean of weighted IPCs [Luo et al. 2001; Eyerman et al. 2014] normalized to that of the base system (normalized harmonic IPC or normalized hIPC) and the reduction in misses per instruction against the base system (MPI reduction). Unless stated otherwise, figures show the average of the results obtained for each of the 100 workloads.

For each mix, the normalized harmonic IPC for a proposal "PROP" is calculated as

$$normalized\ hIPC^{PROP} = \frac{H_t\left(\frac{IPC_t^{PROP\ MP}}{IPC_t^{BASE\ SP}}\right)}{H_t\left(\frac{IPC_t^{BASE\ MP}}{IPC_t^{BASE\ SP}}\right)}$$

where $IPC_t^{\mathrm{PROP}\ MP}$ is the IPC obtained using *PROP* for processor *t* when run in the multiprogrammed experiment, $IPC_t^{\mathrm{BASE}\ MP}$ is the IPC obtained using the base system for processor *t* when run in the multiprogrammed experiment, $IPC_t^{\mathrm{BASE}\ SP}$ is the IPC obtained using the base system for processor *t* when run alone on the system, and function *H* is the harmonic mean, defined as

$$H_t(x) = \frac{t}{\sum_t \frac{1}{x_t}}$$

The harmonic IPC metric is used because it incorporates a notion of fairness, in addition to performance. This is because the harmonic mean tends to be lower when there is much variance among the different weighted IPCs of each processor.

The reduction in misses per instruction is calculated as

$$1 - \frac{\sum_t M_t^{PROP}}{\sum_t M_t^{BASE}} \cdot \frac{\sum_t I_t^{BASE}}{\sum_t I_t^{PROP}}$$

where M_T is the number of SLLC misses counted during simulation for processor *t,* and I_T is the number of instructions executed by processor *t*.

6. DESIGN SPACE EXPLORATION

The following three subsections evaluate the performance-cost trade-offs of ReD capacity, sector size and tag compression, searching for a balanced configuration.

6.1 ReD capacity

Here we study how the results vary depending on the ReD capacity (see Section 3.2). Figure 6 shows normalized hIPC and reduction in MPI, with respect to the baseline, as a function of the ReD

capacity per core. For this experiment, the ReD sector size is one block and it stores the entire tag. We show average values across the 100 mixes described in Section 5.

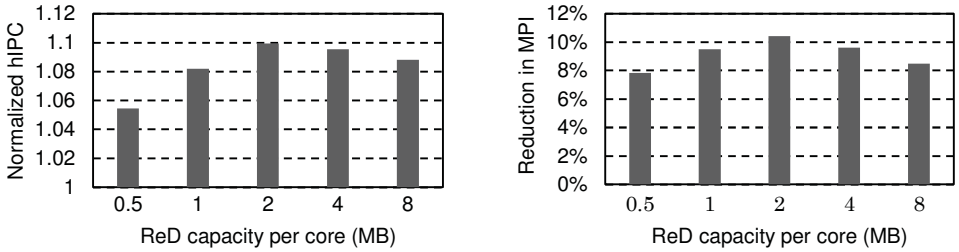

Figure 6: Normalized hIPC (left) and reduction of SLLC misses per instruction (right) with respect to the base system, as a function of the ReD capacity per core.

By increasing capacity, ReD can track blocks that have been evicted from the L2 cache longer ago, that is, it can detect more distant reuses. The optimal configuration is achieved with a capacity of 2 MB per core, which presents a hIPC increase of 9.9%, and reduces MPI by 10.4%. A ReD with capacity larger than 2 MB detects more blocks with reuse than those the SLLC can effectively store, leading to a performance decrease compared to the 2 MB ReD.

6.2 ReD sector size

Increasing sector size decreases the ReD hardware cost (see Section 3.4). We call ReD size the hardware cost of a given ReD configuration measured in bytes. Figure 7 shows ReD size as a function of capacity and sector size, when using tags with 10 bits.

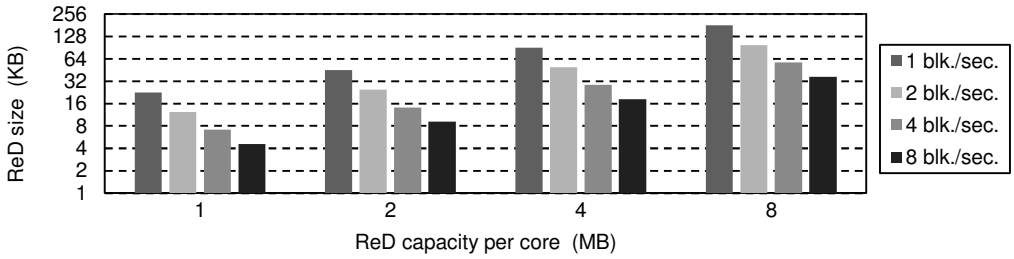

Figure 7: ReD size per core in KB, as a function of capacity and sector size. The sector size is the number of blocks associated with each ReD tag. We consider tags of 10 bits.

For a given ReD capacity, doubling the sector size allows to halve the number of ReD sets. Therefore, a ReD with bigger sector size requires less ReD size. This is because the storage saved by reducing the number of entries is greater than the storage needed to add valid bits for each block of a sector.

On the other hand, Figure 8 shows how performance varies when the ReD sector size increases.

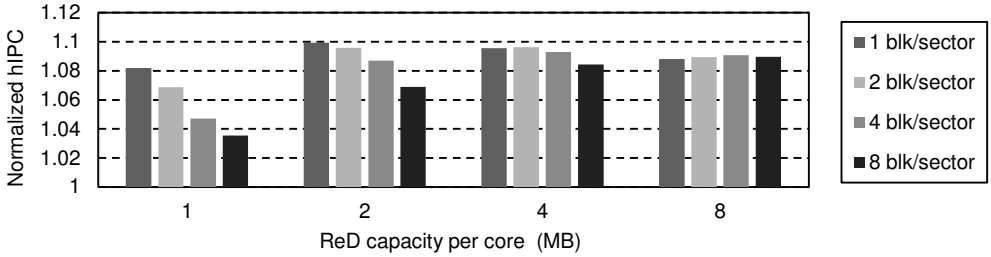

Figure 8: Normalized hIPC with respect to the base system, as a function of the ReD capacity per core, and for different sector sizes.

We have defined the reuse detection window as the set of block addresses that ReD remembers of an executing thread (ReD window for short). Note that it could be different from the ReD capacity

because some threads may not use the full capacity of the detector. If we increase the sector size while maintaining the ReD capacity, the ReD window decreases because sometimes the thread will not reference all the blocks of a sector. Therefore, ReD will detect less reuse, leading to a performance degradation for all the ReD capacities except for 8 MB, where ReD already detected more blocks with reuse than those the SLLC is able to store.

The best configuration in terms of performance, with 2MB of capacity and sectors with one block, requires a ReD size of 45 KB per core. However, other configurations have better performance/cost ratios: the one with 2 MB of capacity and sectors with 2 blocks shows 0.35% lower hIPC and a ReD size of 24.5 KB, 46% lower. The latter is the configuration selected for the remaining experiments.

6.3 ReD tag size

As explained in Section 3.4, ReD can store compressed tags to reduce the amount of storage required. Figure 9 shows on the left, depending on the tag size, the average error rate in reuse detection due to tag compression. These errors are false positives: a false reuse is detected because the compressed tag that is being searched matches with that of a different sector previously registered. Inserting not-reused blocks into the SLLC reduces the effectiveness of the mechanism. The error rate is less than 1% with a tag of only 12 bits.

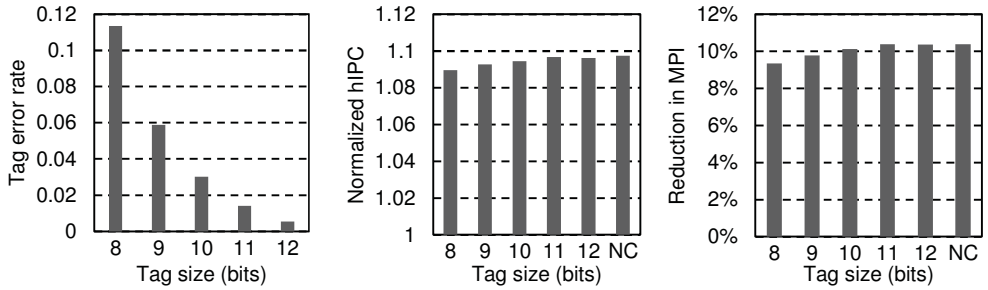

Figure 9: Left: average rate of detection errors in ReD due to tag compression. Center: normalized hIPC with respect to the base system. Right: SLLC MPI reduction with respect to the base system. The NC bar represents the tag with no compression.

Figure 9 shows also, in the center and on the right, normalized hIPC and MPI reduction obtained for our selected configuration (2MB of capacity, a sector of 2 blocks, and ReD size of 24.5 KB) as a function of tag size. The performance loss is almost negligible for a tag size of 10 bits: normalized hIPC decreases 0.29% while MPI increases 0.26% compared to the configuration with uncompressed tags. This justifies compression in order to reduce the amount of storage required.

Hereinafter, unless stated otherwise, we use this configuration in all the experiments: 2 MB ReD capacity, sector size of two blocks and 10 bit tags.

7. PERFORMANCE ANALYSIS OF RED

In this section we first compare ReD with state of the art proposals. Next, we analyze ReD performance using different SLLC sizes, and using an alternative replacement policy. In Section 7.4, we present data on the fraction of alive blocks in SLLC achieved by each proposal. In Section 7.5, we analyze the IPC results of our proposal broken down by application and mix. Finally, we present results on single-processor workloads.

7.1 Comparison with other proposals

In this section we compare the performance of our mechanism with three other recent proposals: cache hierarchy-aware replacement (CHAR) [Chaudhuri et al. 2012], Reuse Cache [Albericio et al. 2013b] and Evicted-Address Filter cache [Seshadri et al. 2012]. We also compare it with a base system with double the SLLC size, that is, 16 MB.

Comparison with CHAR: CHAR is a content selection proposal that bases the bypass decision on the access pattern that a block has at all levels of the memory hierarchy. CHAR was proposed both for inclusive and exclusive SLLCs. We use here the exclusive version.

Figure 10 plots normalized hIPC and MPI reduction against the baseline obtained by ReD and CHAR with an 8 MB SLLC. ReD outperforms CHAR both in MPI reduction (10.1% vs. 4.3%) and normalized hIPC (1.095 vs. 1.070).

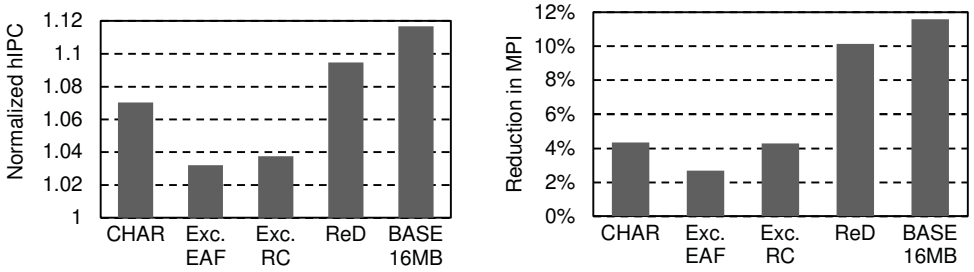

Figure 10: normalized hIPC (left) and MPI reduction (right) compared to the base system with 8 MB, for five systems: ReD (with the balanced configuration), CHAR, Evicted Address Filter, Reuse Cache (with RC-32/8 and NRR in tag array), and a base system with double the SLLC size (16 MB). All are implemented on an exclusive SLLC with TC-AGE in the data array.

CHAR uses a predictor to foresee which blocks will show or not reuse. In Section 7.7 we will show that the accuracy of predictor-based CHAR is lower than the accuracy of our detector-based ReD.

Comparison with Reuse Cache: The Reuse Cache is a content selection proposal for an SLLC whose tag and data arrays are decoupled, and that stores data only of those lines that have shown reuse. To be fair in the comparison, we have modelled a Reuse Cache in which the data array works in exclusion with the private L2 caches. Our exclusive Reuse Cache works as follows: each block in the L2 private caches includes a bit indicating whether it should be inserted into the SLLC when evicted from L2 (bypass / no bypass). On an SLLC miss (first block access), the block is sent from main memory to the L2 cache indicating “bypass”, and the tag is inserted into the SLLC tag array. This allows subsequent reuse to be detected. On a hit in the tag array of the SLLC that misses in the data array (second block access) the block is sent from main memory to L2 cache indicating “no bypass”. When the block is evicted again from L2, it is stored in the SLLC data array. In subsequent accesses, that hit both in tag and data arrays, the block is sent to the private L2 indicating “no bypass”, and is evicted from the SLLC data array. There are no changes in the SLLC tag and data arrays.

We use an exclusive Reuse Cache with a data array of 8MB and a tag array equivalent to 32MB. Among those with 8MB of data, this relationship between tags and data is the best we have found in our simulations. TC-AGE is used as replacement policy in the data array.

As shown in Figure 10, ReD outperforms the Exclusive Reuse Cache both in MPI reduction (10.1% vs. 4.5%) and normalized hIPC (1.095 vs. 1.038).

The Reuse Cache uses a global reuse detector, suffering as a result of interference between the distinct applications. We analyze this effect in Section 7.7. In addition, the Reuse Cache embeds the reuse detector into the SLLC tag array. This increases the detector complexity, since each entry must keep the complete tag along with coherency information, limiting the design opportunities.

Comparison with Evicted-Address Filter: The Evicted-Address Filter Cache is an SLLC that tracks the addresses of blocks that were recently evicted from the SLLC in a structure called the Evicted-Address Filter (EAF). Missed blocks whose addresses are present in the EAF are predicted to have high reuse, while the rest of the blocks are predicted to have low reuse. This prediction affects the insertion priority: high-reused blocks enter at the Most-Recently-Used (MRU) position and low-reused ones enter according to a bimodal policy (MRU with probability 1/64, otherwise LRU). The EAF is implemented using a Bloom filter, which is cleared periodically.

Even though EAF is a replacement policy and not a content selection policy, we have included it in our comparison because it also attaches a reuse detection mechanism. The information ReD

stores and acts upon is different, because it monitors traffic between L2 and the SLLC whereas EAF does it between SLLC and Main Memory. EAF cannot be used to implement a content selection policy, because it uses a Bloom filter to store the reuse information. The filter is periodically cleared, which produces a loss of information that leads to temporarily classify all blocks as not reused. This is beneficial when the detector is used to adjust the replacement policy, as it is in the original publication and in our setup. Conversely, it makes it unsuitable for use as a content selection mechanism, as it would lead to not inserting any new blocks into the SLLC after the reset, for any application, until the filter is adequately refilled.

To be fair in the comparison, we have modelled an EAF Cache in which the data array works in exclusion with the private L2 caches. The L2 caches are also extended to store the Reuse bit, which is sent to the SLLC on eviction. At the time the block enters into the SLLC, that is, when it is evicted from an L2 cache, the Reuse bit is checked first. If it is set, the block is inserted at the MRU position. If not, the EAF is checked, applying the described policy. We store the SLLC MRU information using 2 bits per block, in line with the 2 bits that we use for TC-AGE in our other models. Our experimental results show that for the exclusive version a larger Bloom filter is required. We obtain the best results with a filter 25% larger than in the original publication. This is consistent with the increase in distinct blocks in the whole cache subsystem due to the move from inclusion to exclusion, from 8 to 10 MB (8 cores with 256 KB of L2 cache each).

As shown in Figure 10, ReD outperforms the exclusive EAF Cache both in MPI reduction (10.1% vs 2.7%) and normalized hIPC (1.095 vs. 1.032).

Comparison with a double-sized base system: Figure 10 plots normalized hIPC and MPI reduction against the baseline obtained by ReD with an 8 MB SLLC, and by a base system with a 16 MB SLLC (BASE 16MB).

ReD with an 8 MB SLLC achieves 87% of the MPI reduction (10.1% vs 11.6%) and 81% of the increase in normalized hIPC (1.095 vs 1.117) of the double-sized base system, with only a 2.3% increase in SLLC space.

7.2 Additional cache sizes

In this section we show the performance of ReD using distinct SLLC cache sizes, and compare it with the selected three proposals.

Figure 11 plots normalized hIPC and MPI reduction against the base system obtained by CHAR, exclusive EAF, exclusive Reuse Cache and ReD for varying SLLC sizes. ReD outperforms all of the other proposals in both metrics, at all SLLC sizes considered.

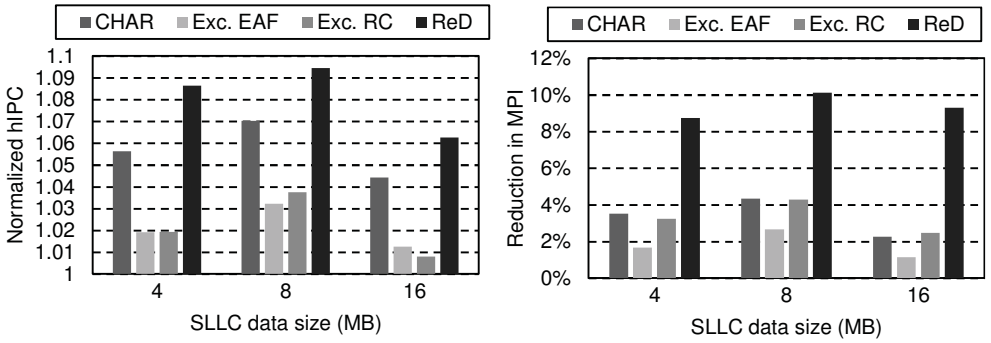

Figure 11: normalized harmonic IPC (left) and MPI reduction (right) compared to the base system, for different SLLC data sizes, for four systems: ReD (with a balanced configuration for each size), CHAR, Evicted Address Filter Cache and Reuse Cache (RC-16/4, RC-32/8 and RC-64/16). All are implemented on an exclusive SLLC with TC-AGE in the data array.

7.3 Alternative cache replacement policy: Least Recently Filled (LRF)

Content selection and replacement policies are usually aligned, as they share the same objectives. Therefore, we use TC-AGE as replacement policy: ReD selects reused blocks to be stored in the SLLC, and TC-AGE aims to retain the most reused blocks as long as possible. However, content selection and replacement policies are orthogonal. The former chooses which

blocks enter the SLLC and the latter which blocks are evicted to make room for others. Therefore, ReD can be implemented on an SLLC managed with any replacement policy.

As ReD has the detector store decoupled from the cache, the replacement policy is not able to adversely affect the detector efficiency. In other mechanisms where the SLLC itself is used as detector store, there is a clear dependence between detection and replacement. A poor replacement algorithm can adversely affect the detector.

Next, we analyze the impact of our proposal on an exclusive SLLC with a 4-bit Least-Recently-Filled (LRF) replacement policy, similar to LRU in inclusive caches. Figure 12 plots normalized hIPC and MPI reduction obtained by ReD when using LRF and TC-AGE, compared to a base system with the same replacement policy but without ReD. Adding ReD leads to better MPI reductions and higher normalized hIPC with LRF than with TC-AGE. This is not a surprising result, as the former is not as efficient as the latter and leaves more room for improvement.

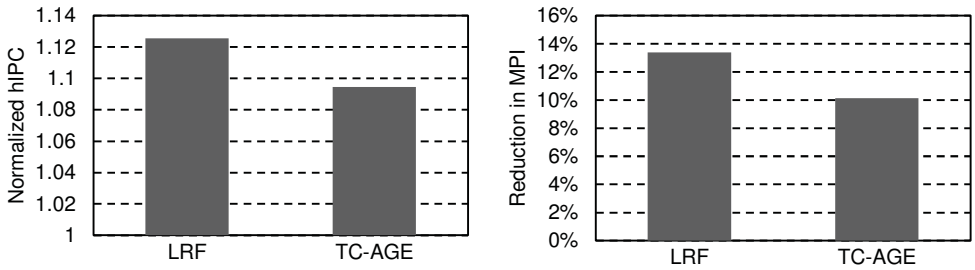

Figure 12: normalized hIPC (left) and MPI reduction (right) obtained when adding the ReD content selection mechanism to base systems with 4-bit LRF (Least-Recently-Filled) and 2-bit TC-AGE as replacement policies. Both are implemented on an exclusive SLLC with 8 MB in the data array.

7.4 Alive and dead blocks

In this section we present the average number of alive blocks that the SLLC stores at any given time. We define a block in the SLLC as alive at a given time if it receives a hit in the future before its eviction. Conversely, a block is defined as dead at a given time if it does not receive an additional hit before its eviction. Dead blocks waste storage.

Figure 13 plots these results for CHAR, exclusive EAF, exclusive Reuse Cache and ReD. Additionally, we include the baseline configuration (labelled TC-AGE), and NRF (Not Recently Filled) as the most basic replacement policy (NRF is analogous to NRU in inclusive caches [Sun 2007]). For each workload, we take measures every million cycles and calculate the average. We show the average over all our workloads.

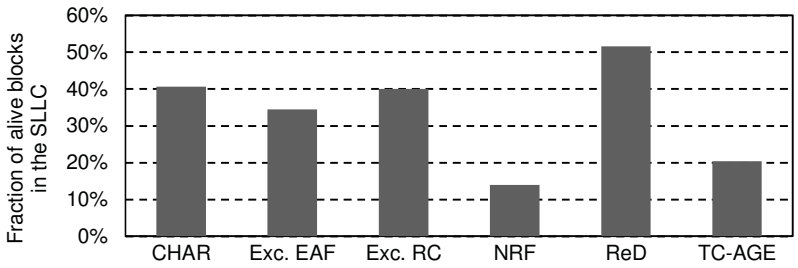

Figure 13: average fraction of alive blocks present at any given moment in the SLLC, for different cache management proposals, using an 8 MB SLLC.

As Figure 13 shows, when using the basic 1-bit NRF policy only 14.0% of the blocks are alive on average. Our baseline configuration (2-bit TC-AGE), increases that figure up to 20.4%. All other proposals, implemented on top of TC-AGE, improve the management of the SLLC, increasing the fraction of alive blocks. ReD achieves the best results with 51.5% of alive blocks, being the only proposal that manages to have on average more alive than dead blocks.

Comparing Figure 13 with previous Figure 10 (right), we realize that increasing fractions of alive blocks correlates to a higher reduction in misses per instruction, albeit it is not proportional. As ReD

prioritizes multiple-reused blocks in the SLLC (see Section 5.3), it takes more advantage of the alive fraction, thus leading to a higher miss rate reduction.

7.5 Per-application and per-mix performance

As explained previously, application performance depends both on the application itself and on the other applications running in the workload. Figure 14 shows box-and-whisker plots with the distribution of speed-ups (normalized IPC) by application, with respect to the baseline system, for all instances of the applications that are running in our 100 workloads. Five values are plotted, namely minimum, first quartile, median, third quartile, and maximum.

Out of all 29 applications, 5 show improved performance in all workloads they appear in (*astar*, *bzip2*, *hmmer*, *tonto* and *xalancbmk*), with medians as high as 1.41 for *xalancbmk*. Another 11 show improved performance starting with the first quartile (*bwaves*, *gamess*, *gobmk*, *gromacs*, *h264ref*, *mcf*, *omnetpp*, *sjeng*, *soplex*, *sphinx3*, *wrf*), although in some mixes they show reduction. In 8 of them (*cactusADM*, *dealII*, *gcc*, *libquantum*, *milc*, *namd*, *perlbench*, *povray*), the median shows improvement but the first quartile shows reduction. The 5 remaining applications (*GemsFDTD*, *calculix*, *lbm*, *leslie3d*, *and zeusmp*) show less performance in the median.

Performance results also vary by workload, depending on the applications it includes. Figure 15 plots the normalized harmonic IPC for all the workloads, relative to the baseline. Out of the 100 mixes, 94 show speed-up improvements of up to 1.70, the worst having a value of 0.98.

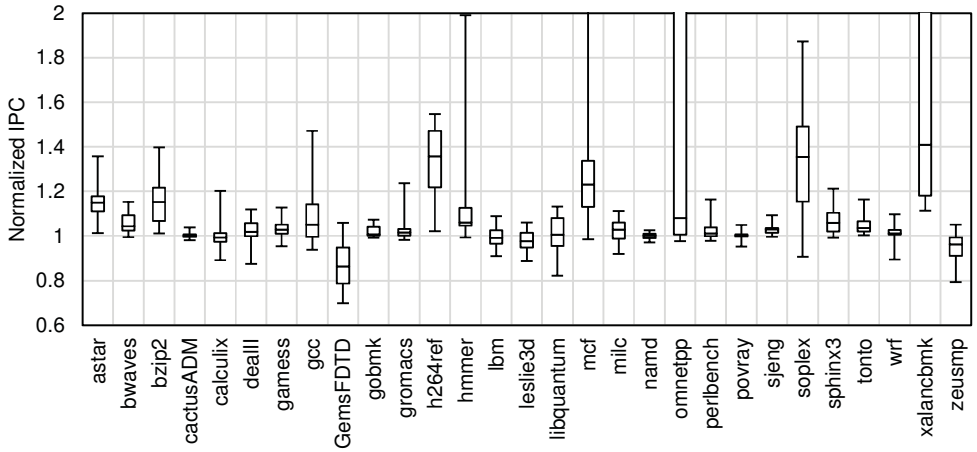

Figure 14: Distribution of normalized IPC, compared to the base system, for all applications in all workloads.

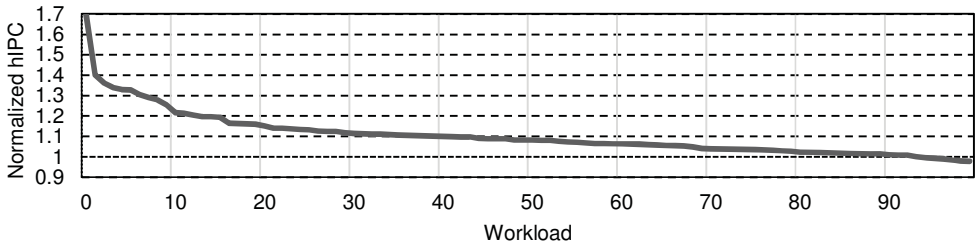

Figure 15: Normalized harmonic IPC, compared to the base system, for all 100 workloads.

7.6 Single-processor performance

In this section we show the performance of ReD for single-processor workloads. For these experiments we use a 1 MB LLC, the same per-processor amount as in our multiprocessor simulations. All other parameters are the same. We show results for all benchmarks that have MPKI > 2 at the LLC.

Figure 16 plots normalized IPC obtained by ReD compared to the base system. Although ReD is specifically designed for chip multiprocessor systems, it still provides performance enhancements

for 9 of these 14 sequential workloads, up to 12.8% speedup for *xalancbmk*. It decreases the performance of the other 5, up to 1.8% for *omnetppp*.

It is interesting to compare these results with those shown in Figure 14 for multi-processor workloads. O*mnetppp* shows there positive normalized IPC in most of the workloads it is present in (starting with the first quartile). As shown in Figure 5, the content selection made by ReD changes the distribution of space in the SLLC, and is often able to assign more space to the alive blocks of *omnetppp*, surpassing the 1 MB that we use in this section. With increasing space, the reused working set fits in the SLLC, and *omnetppp* turns to show IPC improvements in most multiprocessor benchmarks.

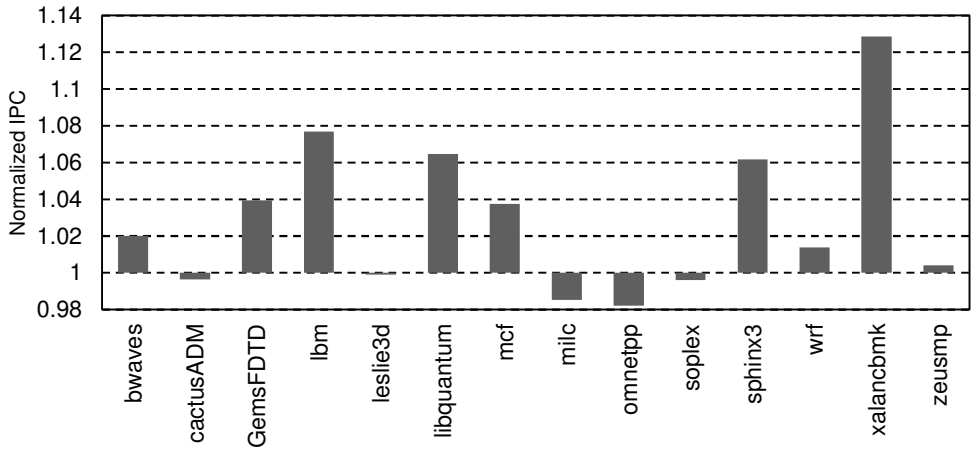

Figure 16: Normalized IPC obtained by ReD, compared to the base system, for single-processor workloads that have LLC MPKI >2, on an exclusive LLC with 1 MB in the data array

7.7 Detector efficiency

In this section we analyze ReD efficiency in terms of the number of blocks selected for SLLC insertion, and their usefulness. We show figures about the amount of blocks selected by the distinct detectors/predictors, and the subsequent reuse of these blocks.

Figure 17 plots the number of blocks in our example workload that, after coming from main memory to L2, are selected for SLLC insertion ("new blocks"). We show figures for ReD, CHAR, and a detector named "Shared ReD". This detector is similar to ReD, but it uses a single address buffer that is shared among all cores instead of multiple private ones. Figure 18 shows the accuracy of each content selection mechanism. This is defined as the percentage of new blocks that are accessed at least once after being sent to the SLLC. Figure 19 plots the MPI reduction of each mechanism versus our base system.

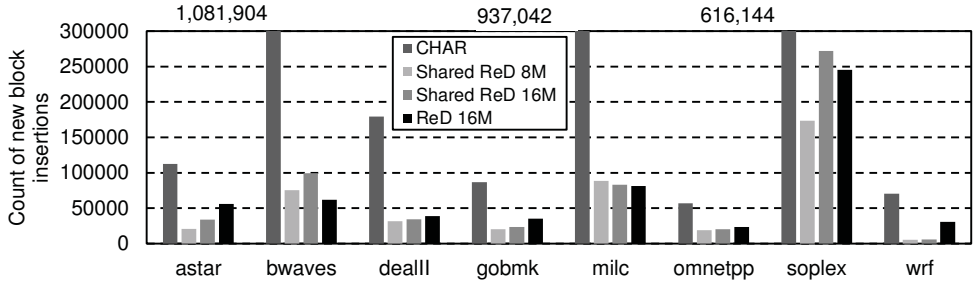

Figure 17: Number of blocks selected for SLLC insertion after coming from main memory, for all applications in the example workload. From left to right: CHAR, shared ReD (8 MB and 16 MB) and ReD (16 MB, 2MB per core)

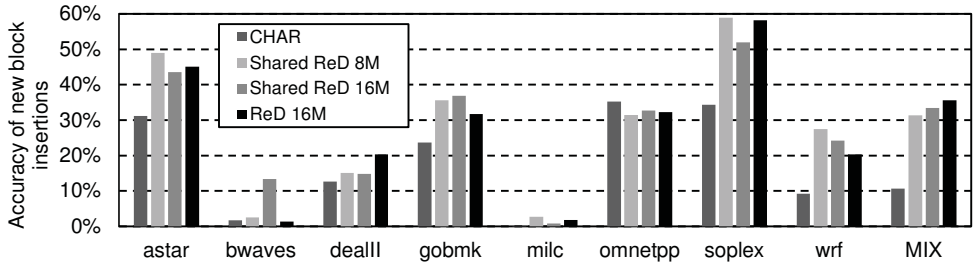

Figure 18: Accuracy of content selection mechanisms: percentage of new blocks used at least once after being sent to the SLLC, in the example workload. From left to right: CHAR, shared ReD (8MB and 16 MB) and ReD (16 MB, 2MB per core)

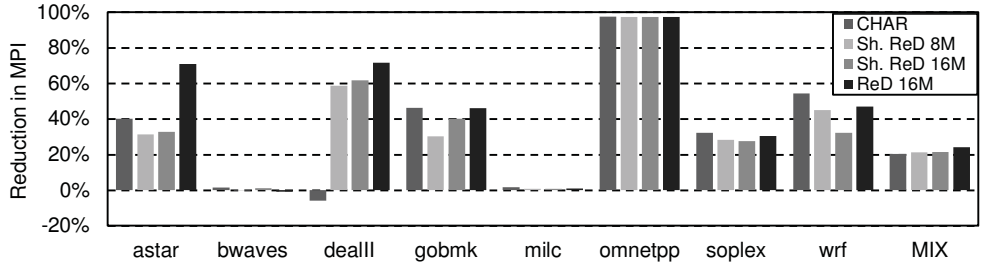

Figure 19: SLLC MPI reduction with respect to the base system. From left to right: CHAR, shared ReD (8MB and 16 MB) and ReD (16 MB, 2MB per core)

As shown in Figure 17 and Figure 18, predictor-based CHAR is less selective than our detector-based ReD. CHAR inserts many more blocks into the SLLC but its accuracy is low. For example, for *bwaves* CHAR inserts into the SLLC about 18 times the blocks inserted by ReD. However, both have similar SLLC miss ratio because only 2% of the blocks inserted by CHAR are used before being evicted. The behavior is similar for *milc* (12x and 0.1%). To store blocks for these two applications, CHAR evicts useful blocks from other cores. As a consequence, ReD is much better than CHAR at reducing MPI for two other applications of this mix (*astar* and *dealII*). For the whole mix the MPI reduction for CHAR is 20% vs 24% for ReD 16M (Figure 19). These differences, or even larger ones, appear consistently across our workloads, leading to the average 6% difference shown in Figure 10.

Shared ReD with a capacity of 16M overall inserts 32% more blocks than Shared ReD with 8 MB. Additionally, the accuracy increases by 2%. This does not directly translate into a much higher MPI reduction in this particular workload (only 0.3%), but on average (across our 100 workloads) it leads to a 1% reduction in MPI.

ReD 16M outperforms Shared-ReD 16M in 5 applications of the mix, and obtains similar performance in *bwaves*, *milc* and *omnetppp*. Figure 20 plots the average fraction of the reuse detector occupied by each program using these two mechanisms. Only two applications, *bwaves* and *milc*, occupy 76% of the shared detector, stealing capacity from the other six applications. The private detector in ReD *protects* all applications from this thrashing, which leads to a fair distribution of the detection window, and ultimately better performance of the workload.

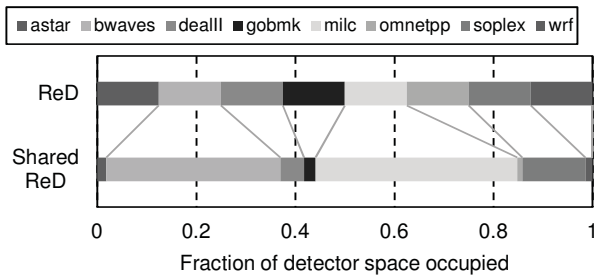

Figure 20: Fraction of overall detector space occupied by each application on the example mix, for ReD 16MB and Shared ReD 16MB.

8. RELATED WORK

Any cache content management mechanism is based on a model that forecasts whether a block is going to be used in the immediate future or not. We can classify state-of-the-art mechanisms for the SLLC into two groups: those that rely on the last touch to each block (PC based [Khan et al. 2010], PC-sequence based [Lai et al. 2001], counter based [Kharbutli and Solihin 2008], ...), and those that rely on the reuse locality [Qureshi et al. 2007; Gaur et al. 2011; Jaleel et al. 2010b; Chaudhuri et al. 2012; Albericio et al. 2013a; Albericio et al. 2013b; Gao and Wilkerson 2010; Khan et al. 2012; Wu et al. 2011; Li et al. 2012; Seshadri et al. 2012; Gupta et al. 2013].

In a similar way, Faldu and Grot [2016] classify management strategies into Dead Block Predictors (DBPs) and insertion policies. DBPs try to predict whether a block has reached the end of its useful lifetime on chip. Insertion policies try to predict when a block is dead on arrival (it will not see any reuse in the cache after its insertion). The paper concludes that DBPs are less accurate than insertion policies.

We focus on proposals relying on the reuse locality property of the SLLC blocks, which are "insertion policies" in Faldu and Grot [2016] taxonomy. We can classify them according to three characteristics:

- Replacement / content selection: In some proposals, the replacement algorithm gives higher priority to stay in the cache to those blocks showing reuse. On the other hand, some works leverage reuse locality to select for insertion only those blocks classified as reused, bypassing the rest.
- Detection / prediction: In order to classify blocks, some authors suggest reuse detection mechanisms, while others propose mechanisms to predict a reuse behavior before it appears.
- Address store: Both detection and prediction mechanisms need to remember past block addresses in order to identify the second access to a block. Detection mechanisms associate reuse to the block receiving a second access while prediction mechanisms associate reuse to a signature of the block receiving a second access. The structure that holds past accesses can be implemented in several ways, either embedded into the SLLC itself or as an independent structure.

Table 4 contains a sample of previous work classified according to this taxonomy.

		Addresses in the SLLC	In a different store
Replacement	Detection	Gao and Wilkerson 2010 Gaur et al. 2011 (TC-AGE) Khan et al. 2012 Albericio et al. 2013a	Seshadri et al. 2012 Gupta et al. 2013
	Prediction	Qureshi et al. 2007 Jaleel et al. 2010b Wu et al. 2011	
Content selection	Detection	Albericio et al. 2013b	**Our proposal (ReD)**
	Prediction	Gaur et al. 2011 (Bypass + TC-AGE) Chaudhuri et al. 2012 Li et al. 2012	

Table 4: Classification of previous work based on the reuse locality property, according to our taxonomy

8.1 Replacement policies

Both detection and prediction have been used to guide replacement algorithms, and most of them keep the reuse window in the SLLC itself.

Replacement mechanisms based on detection label a block as not reused when it comes from main memory (its first use in the reuse window that the SLLC is able to recall). Subsequent SLLC hits (second and later touches to the block) will flag the block as reused. Two proposals include an

added store to remember these blocks. Seshadri et al. [2012] use a Bloom Filter and Gupta et al. [2013] use a Bypass Buffer.

Replacement mechanisms based on prediction try to figure out whether a block will be reused before it really is, and flag the block as reused just after its first touch. Prediction policies categorize blocks according to certain features (signatures in Wu et al. [2011]), and study the reuse characteristics of any block in each category. Wu et al. [2011] analyze distinct types of signatures: memory region, program counter, or instruction sequence. As an example, the PC signature policy acts by classifying blocks according to the PC of the memory instruction responsible for bringing them in the chip. It identifies the reuse behavior of the blocks that each instruction loads (mainly categorizing them as reused or not reused), and assigns the same category to all blocks that the same instruction will bring in the future.

DIP and DRRIP mechanisms are also predictors [Qureshi et al. 2007; Jaleel et al. 2010b]. Using set-dueling techniques, these mechanisms analyze the reuse behavior of the entire application and apply it to all its blocks. All blocks are categorized into a single category, the one of their own application.

8.2 Content selection policies

Except the Reuse Cache proposed by Albericio et al. [2013b], all other content selection policies include some sort of prediction: Li et al. [2012] uses the PC signature policy, Gaur et al. [2011] the trip count and use count of blocks, and Chaudhuri et al. [2012] the behavior of blocks during their stay in private caches and their coherence status. For each class, an algorithm analyzes its SLLC behavior and extends it to all future blocks belonging to the same category.

Comparing our proposal with others using prediction, they tend to be more complex, and often require the transfer of data between cache levels or even to send the PC to the cache subsystem. Additionally, predictors show lower accuracy than detectors.

On the other hand, all previous content selection techniques track reuse (and reuse patterns) using the SLLC. Therefore, the SLLC size defines and limits the size of the reuse detection window.

Finally, all these proposals have in common an important constraint: their reuse detector is shared among all threads running on the CMP. A single thread can thrash the detector, shrinking the reuse detection window of the remainder applications. To overcome this, we propose implementing reuse detectors that are private to every core.

9. CONCLUSIONS

Previous publications reveal that the stream of references reaching the shared last level cache (SLLC) of a multiprocessor chip shows little temporal locality. However, it shows reuse locality, i.e., blocks referenced more than once are more likely to be referenced in the near future. This leads to an inefficient use of the cache if conventional management is performed. There are several proposals addressing this problem for inclusive caches, but few that focus on exclusive ones.

This paper proposes a novel content selection mechanism for exclusive SLLC that leverages the reuse locality embedded in the SLLC request stream. We propose adding a Reuse Detector (ReD), placed in between each L2 cache and the SLLC, to discover which of the L2 evicted blocks have not experienced reuse and avoid their insertion in the SLLC, bypassing them.

We analyze problems affecting similar recent mechanisms (low accuracy, reduced visibility window and thrashing in the detector) and design ReD to overcome them as much as possible.

We evaluate the proposal in a multicore chip with eight processors that executes a multiprogrammed workload. Properly designed, the Reuse Detector prevents the insertion of many useless blocks in the SLLC, and helps keeping the most reused.

Experimental results show that this allows for enhancing SLLC performance beyond other recent proposals. Specifically, ReD reduces the SLLC misses per instruction by 10.1% with respect to a base cache with TC-AGE replacement and no content selection, while CHAR and exclusive-cache versions of the Reuse Cache and the EAF cache reduce MPI by 4.3%, 4.5% and 2.7% respectively.

10. 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).

11. REFERENCES

[Albericio et al. 2013a] J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería. 2013. Exploiting reuse locality on inclusive shared last-level caches. ACM Trans. on Architecture and Code Optimization, vol. 9, nº 4, p. 38.

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

[Chaudhuri et al. 2012] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman. 2012. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In Proceedings of the 21st Int. conference on Parallel architectures and compilation techniques, 293-304.

[Conway et al. 2010] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. 2010. Cache hierarchy and memory subsystem of the AMD Opteron processor. IEEE micro, nº 30(2), 16-29.

[Eden and Mudge 1998] A. N. Eden, and T. Mudge. 1998. The YAGS branch prediction scheme. In Proceedings of the 31st Ann. ACM/IEEE Int. Symp. on Microarchitecture, 69-77.

[Eyerman and Eeckhout 2014] S. Eyerman and L. Eeckhout. 2014. Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance. *IEEE Computer Architecture Letters*, vol. 13, no. 2, 93-96.

[Faldu and Grot 2016] P. Faldu and B. Grot. 2016. LLC Dead Block Prediction Considered Not Useful. In 13th Workshop on Duplicating, Deconstructing and Debunking (WDDD).

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

[Gaur et al. 2011] J. Gaur, M. Chaudhuri, and S. Subramoney. 2011. Bypass and Insertion Algorithms for Exclusive Last-level Caches. In Proceedings of the 38th Int. Symp. on Computer Architecture, 81-92.

[Gupta et al. 2013] S. Gupta, H. Gao, and H. Zhou. 2013. Adaptive Cache Bypassing for Inclusive Last Level Caches. In Proceedings of the 27th Int. Symp. on Parallel & Distributed Processing, 1243-1253.

[Henning 2006] J. L. Henning. 2006. SPEC CPU2006 benchmark descriptions. ACM SIGARCH Computer Architecture News, vol. 34, nº 4, 1-17.

[Jaleel et al. 2010a] A. Jaleel, E. Borch, M. Bhandaru, S. C. Steely Jr., and J. Emer. 2010. Achieving Non-Inclusive Cache Performance with Inclusive Caches. Temporal Locality Aware (TLA) Cache Management Policies. In Proceedings of the 43rd Ann. Int. Symp. on Microarchitecture, 151-162.

[Jaleel et al. 2010b] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). In Proceedings of the 37th Int. Symp. on Computer Architecture, 60-71.

[Jaleel et al. 2015] A. Jaleel, J. Nuzman, A. Moga, S.C. Steely Jr., and J. Emer. 2015. High Performing Cache Hierarchies for Server Workloads. Relaxing Inclusion to Capture the Latency Benefits of Exclusive Caches. In Proceedings of the 21st Int. Symp. on High Performance Computer Architecture, 343-353.

[Jouppi and Wilton 1994] N. P. Jouppi, and S. J. E. Wilton. 1994. Tradeoffs in Two-Level On-Chip Caching. In Proceedings of the 21st Ann. Int. Symp. on Computer Architecture, 34-45.

[Khan et al. 2010] S. Khan, Y. Tian, and D. A. Jiménez. 2010. Sampling Dead Block Prediction for Last-Level Caches. In Proceedings of the 43rd Ann. Int. Symp. on Microarchitecture, 175-186.

[Khan et al. 2012] S. Khan, Z. Wang, and D. Jimenez. 2012. Decoupled dynamic cache segmentation. In Proceedings of the IEEE 18th Int. Symp. High Performance Computer Architecture HPCA, 1-12.

[Kharbutli and Solihin 2008] M. Kharbutli, and Y. Solihin. 2008. Counter-based cache replacement and bypassing algorithms. in the IEEE Transactions on Computers (Volume:57, Issue:4), 433-447.

[Lai et al. 2001] An-Chow Lai, C. Fide, and B. Falsafi. 2001. Dead-Block Prediction & Dead-Block Correlating Prefetchers. In Proceedings of the 28th Ann. Int. Symp. on Computer Architecture, 144-154.

[Li et al. 2012] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng. 2012. Optimal bypass monitor for high performance last-level caches. In Proceedings of the 21st Int. conference on parallel architectures and compilation techniques, 315-324.

[Lotfi-Kamran et al. 2012] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. 2012. Scale-Out Processors. ACM SIGARCH Computer Architecture News, vol. 40, nº 3, 500-511.

[Luo et al. 2001] K. Luo, J. Gummaraju, and M. Franklin. 2001. Balancing Throughput and Fairness in SMT Processors. In Proceedings of the IEEE Int. Symp. Performance Analysis of Systems and Software, ISPASS, 164-171.

[Magnusson et al. 2002] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. 2002. Simics: A full system simulation platform. Computer, nº 35(2), 50-58.

[Martin et al. 2005] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. 2005. Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. Computer Architecture News, nº 33(4), 92-99.

[Mattson et al. 1970] R. L. Mattson, J. Gecsei, D. R. Slutz and I. L. Traiger. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, vol. 9, no. 2, 78-117.

[Qureshi et al. 2007] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, and J. Emer. 2007. Adaptive insertion policies for high performance caching. In Proceedings of the 34th Ann. Int. Symp. on Computer Architecture, 381-391.

[Rosenfeld et al. 2011] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. In Computer Architecture Letters (Volume: 10, no. 1), 16–19.

[Seshadri et al. 2012] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. 2012. The evicted-address filter: a unified mechanism to address both cache pollution and thrashing. In Proceedings of the 21st Int. conference on Parallel architectures and compilation techniques, 355-366.

[Sun 2007] Sun Microsystems. UltraSPARC T2 supplement to the Ultra-SPARC architecture 2007. Draft D1.4.3.

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