The Reuse Cache: Downsizing the Shared Last-Level Cache

Jorge Albericio*
University of Toronto
jorge@eecg.toronto.edu

Pablo Ibáñez
University of Zaragoza
imarin@unizar.es

Víctor Viñals
University of Zaragoza
victor@unizar.es

José M. Llabería
UPC Barcelona Tech
llaberia@ac.upc.edu

ABSTRACT

Over recent years, a growing body of research has shown that a considerable portion of the shared last-level cache (SLLC) is dead, meaning that the corresponding cache lines are stored but they will not receive any further hits before being replaced. Conversely, most hits observed by the SLLC come from a small subset of already reused lines.

In this paper, we propose the reuse cache, a decoupled tag/data SLLC which is designed to only store the data of lines that have been reused. Thus, the size of the data array can be dramatically reduced. Specifically, we (i) introduce a selective data allocation policy to exploit reuse locality and maintain reused data in the SLLC, (ii) tune the data allocation with a suitable replacement policy and coherence protocol, and finally, (iii) explore different ways of organizing the data/tag arrays and study the performance sensitivity to the size of the resulting structures.

The role of a reuse cache to maintain performance with decreasing sizes is investigated in the experimental part of this work, by simulating multiprogrammed and multithreaded workloads in an eight-core chip multiprocessor. As an example, we show that a reuse cache with a tag array equivalent to a conventional 4 MB cache and only a 1 MB data array would perform as well as a conventional cache of 8 MB, requiring only 16.7% of the storage capacity.

Categories and Subject Descriptors

B.3.2 [**Hardware**]: Memory Structures—*Cache Memories*

General Terms

Design, Performance, Experimentation

*The work of the present paper was developed while the author was a PhD student at the University of Zaragoza.

MICRO '46, December 7-11, 2013, Davis, CA, USA.

http://dx.doi.org/10.1145/2540708.2540735

Keywords

Last-Level Cache Organization, Reuse

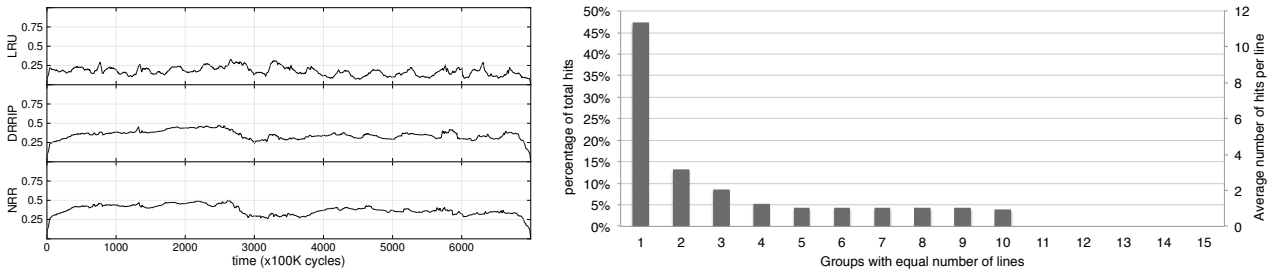
1. INTRODUCTION

Shared-memory chip multiprocessors (CMPs) reached high-performance and high-throughput computing markets in the last decade. Nowadays, they are widely accepted across all segments of the market, from cloud servers to desktop, embedded and mobile systems on a chip. Industry converts the continuously increasing number of available transistors into higher core counts and larger cache memories that expand linearly with the number of cores. This trend is, however, unlikely to continue in the future, obliging many-core systems to include lower cache-to-core ratios [21].

It is common that all the cores in a CMP share a last level of cache memory, referred to as the shared last-level cache (SLLC). This SLLC is required to perform two key tasks, manage coherence and store valuable data. Regarding the second task, the SLLC seeks to feed the misses of the private cache levels with low latency, exploiting to the maximum the scarce off-chip bandwidth with the external main memory.

Previous studies indicate that conventional SLLCs are effective but inefficient because their contents are mostly useless. In fact, a high proportion of the SLLC lines are dead, meaning they will not be requested again before being evicted, to the point that some lines are just used once, and are useless during their entire stay in the SLLC [16, 17, 26]. Efficient management of SLLC contents is difficult because temporal and spatial locality that govern private cache levels become dissolved in the stream of references observed by the SLLC [12, 26]. A great deal of research has addressed the problem from several angles, with the aim of improving the SLLC hit ratio. The suggestions proposed include decreasing the number of conflicts in the SLLC sets [27], improving replacement decisions [11], and prefetching useful data [4], among other techniques.

Despite these efforts, state of the art replacement policies only achieve average improvements within 5% of a commercial algorithm such as NRU, even using a set of benchmarks whose performance is sensitive to replacement [12, 1]. Moreover, the fraction of live lines in the SLLC does not increase much with any of the aforementioned techniques. Such facts encouraged us to explore an alternative approach: to develop a solution with which the SLLC area is drastically reduced without compromising performance. A solution of this sort would be very interesting, since the saved area could help to



(a) Changes in the fraction of live lines over time

(b) Distribution of hits among all lines loaded (or reloaded) into the LRU SLLC during their stay. Each group represents 0.5% of the loaded lines

Figure 1: Line usage patterns in a conventional 8 MB SLLC during a simulation period of 700 MCycles

cut manufacturing costs, reduce power consumption, and/or decrease the cache-to-core ratio, allowing the core count to be increased with the same die area.

A conventional SLLC has a non-selective allocation policy, meaning that any request coming from the lower levels always ends up storing the corresponding line in the SLLC (either right after the miss processing in an inclusive SLLC or in a deferred way in an exclusive one). Non-selective allocation is a good choice if temporal locality holds. We assume that recently referenced lines are likely to appear in the near future, so we want them to be kept in the cache. However, recent studies point out that the reference stream entered in the SLLC does not always exhibit temporal locality and, consequently, these authors propose varying the precise insertion point in the LRU stack [11, 12].

Other recent studies have shown that is not a good idea to base SLLC replacement on exploiting only temporal locality, given that local caches are already doing that job [10, 12, 5]. In fact, a recent study explicitly notes that the reference stream entered in the SLLC has reuse locality instead of temporal locality [1]. In short, the term reuse locality describes the property that the second reference to a line is a good indicator of forthcoming reuse, and recently reused lines are more valuable than other lines reused a long time ago.

In this paper, we introduce the reuse cache, a structure and a set of policies tuned to process request streams with reuse locality. A key design decision is to provide a selective allocation policy leveraging reuse locality. Specifically, only data that has already shown reuse will be kept in the SLLC data array, allowing drastic size reductions without performance loss.

A reuse cache decouples tag and data arrays, breaking the conventional 1:1 mapping. Besides its obvious functions, the tag array in a reuse cache supports reuse detection and inclusion maintenance. The data array may have far fewer entries than the tag array, because it only contains reused data.

We evaluate our proposal by simulating an eight-core CMP system which runs a rich set of multiprogrammed and multithreaded workloads. The reuse cache succeeds in identifying the small fraction of lines that receive most hits, and leveraging its decoupled tag/data design, the size of the data array can be dramatically shrunk without having a negative impact on system performance. Specifically, a reuse cache matches an 8 MB conventional cache in performance with tag and data arrays half and one-eighth the number of entries, respectively (a saving of 83.1% in storage capacity).

The paper is structured as follows. Section 2 provides experimental evidence of the small fraction of live lines in the SLLC and the concentration of hits in the reused lines. Section 3 explains the organization of the reuse cache, replacement algorithms and coherence protocol modifications, giving implementation details and costs. Section 4 presents the experimental methodology and the baseline system, while Section 5 presents the experiments and discusses the results, comparing them with two state of the art proposals, namely, dynamic re-reference interval prediction (DRRIP) [12] and non-inclusive cache, inclusive directory architecture (NCID) [35]. Section 6 summarizes related work and, finally, conclusions are drawn in Section 7.

2. MOTIVATION

In this section, we analyze the behavior of a representative multiprogrammed SPEC CPU workload[1] running in an eight-core chip with a memory hierarchy made up of an SLLC and private caches; see the simulation details in Section 4. We are going to highlight two effects, namely, *i*) most lines in the SLLC are dead, meaning they will not receive any further hits; and *ii*) most SLLC hits come from a small subset of lines (among all the lines the SLLC loads).

2.1 The fraction of live SLLC lines is small

Figure 1a shows the fraction of the SLLC lines that are live over the course of the execution of the example workload. We say that a line is *live* in a particular moment if it will receive some additional access during the rest of its stay in the cache. We simulated 700 M cycles and took a sample every 100 K cycles. In each sample, we compute the "instantaneous" fraction of live lines.

As we can see the fraction of live lines varies between 5.7 and 29.8% when LRU replacement is applied. On average, only 17.4% of the SLLC lines are live. Using Dynamic Re-Reference Interval Prediction (DRRIP)[12] and the Not Recently Reused (NRR)[1] algorithm, two state of the art replacement policies, this average increases to 34.8 and 37.9% respectively. The averages for the 100 workloads used in the experiments in Section 5 are 16.2, 35.9 and 40.0% for LRU, DRRIP and NRR respectively. In other words, 83.8% of the

[1]This example workload is composed of the following applications: *gcc*, *mcf*, *povray*, *leslie3d*, *h264ref*, *lbm*, *namd*, and *gcc*

lines stored in the SLLC do not provide any benefit, and replacement algorithms proposed in the literature are only able to reduce this percentage to about 64.9%. Hence, it can be reasonably argued that it should be possible to save the extra space and energy associated with those dead lines without compromising performance.

2.2 Hits come from a small subset of lines

Figure 1b shows the contribution to the total number of hits of a given group of lines during their stay in the SLLC over the course of the execution of the example workload. In order to obtain this distribution, we ran the simulation and, just after each line is evicted from the SLLC, inserted in a sorted list the number of hits the line received while in the cache (0 hits, 1 hit, 2 hits, etc.). If a line was loaded multiple times, i.e., it had multiple generations [14], its hit count appeared multiple times in the sorted list.

Once the simulation ends, we broke the sorted list into 200 groups of equal size. Thus, each group represents 0.5% of the loaded lines. The first bar to the left in Figure 1b represents the percentage of hits received by the group of line generations at the top of the list (47% of hits in 0.5% of lines). Alternatively, we can read the average number of hits per line generation from this group (11.5) from the vertical axis on the right.

As we can see, only 5% of all the loaded lines are useful, receiving one or more hits. Beyond that, the remaining 95% loaded lines are useless, because they will not receive any hits during their lifetime. Moreover, hits are concentrated in a very small portion of the useful lines. Specifically, 0.5% of all the loaded lines account for 47% of the SLLC hits.

In summary, the reference stream forwarded to the SLLC certainly exhibits reuse locality: very few lines are useful (receiving some hits), and once a line has received a hit, it has a high probability of receiving additional hits. Consequently, we propose to store in the SLLC only the lines showing reuse. Since these lines are a small portion of the total lines and receive most of the hits, we should be able to greatly reduce the cache size without impairing performance.

3. THE REUSE CACHE DESIGN

Starting from a conventional cache, we set out to design a reuse cache by reducing the data array and storing only the lines showing reuse. Regarding the tag array, it should reflect at least the lines retained in the tiny data array. In addition, the tag array should also record the lines present in the private levels (directory inclusion), so that coherence management is simplified [2, 10, 35]. Finally, in order to detect reuse, the tag array should maintain some usage history of recently used lines, lines that may not be in the data array or in the private levels.

Having more entries in the tag array than in the data array, a natural solution is to decouple them. The coherence protocol of the reuse cache has to be modified in order to reflect new coherence states (a line tag is in the tag array, but the corresponding data line is not in the data array).

On a miss in the tag array, the line is read from main memory and loaded into the corresponding private cache. Only the tag is loaded into the SLLC, with no associated data. On a hit in the tag array with no associated data, a reuse is detected. Thus, the line is read again from main memory and loaded in the private cache and SLLC data array at the same time. When a line is evicted from the data array, its tag remains in the tag array. A further access to that line hitting in the tag array will be taken as a reuse hint and then the line will be loaded in the data array. In order to take advantage of reuse locality, replacement in the data array is based on recency. On the other hand, tag replacement is designed to protect both private cache lines and recently reused lines.

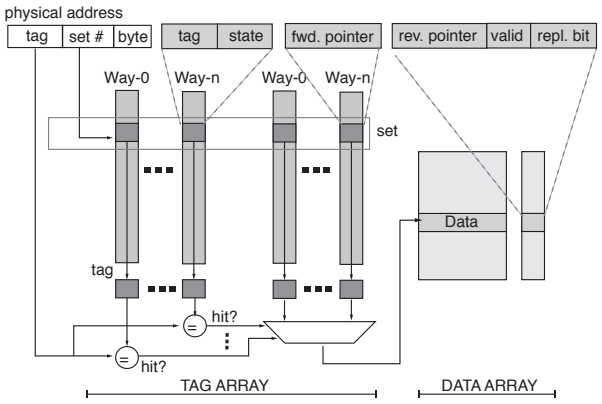


Figure 2: Reuse cache overview

In the following sub-sections, we discuss the reuse cache organization and replacement policies, present an example coherence protocol, and discuss the hardware costs of our proposal.

3.1 Organization

The reuse cache breaks the implicit one-to-one mapping between tag and data found in conventional caches.

Other authors have proposed decoupling tag and data arrays with them having the same [6] or different [30, 28, 27, 35] number of entries. Decoupling with the same number of entries allows the two arrays to be shaped differently, for instance enabling the concept of distance associativity [6]. In any case, all proposals rely on relating the entries of the two arrays by means of pointers. Some proposals need only *forward pointers* from the tag array to the corresponding data lines, if any [28, 6, 35], and others need only a *reverse pointer*, which links each data line to the corresponding tag [30], while in some cases both kinds of pointers are required [27].

An alternative organization, avoiding the need for pointers, uses the same number of sets in tag and data arrays and associates data to only a few tags for each set. The association between tags and data is fixed (for instance, only the tag in way 0 has associated data). This organization involves moving tags between the ways with and without data [35, 20].

In the reuse cache, a tag may have an associated data line or not. A particular coherence state identifies every possible situation, and a *forward pointer* and a *reverse pointer* relate the entries of the two arrays to one another. Figure 2 shows an overview of the reuse cache organization.

As the forward pointer indicates the exact position of a line in the data array, no additional lookup in the data array is required. Thus, the data array can be as associative as desired. The data array associativity is only related to replacement in the data array. By increasing associativity, the replacement algorithm has more options to choose a vic-

tim. The data array associativity also has a small impact on the hardware cost. By increasing associativity, the size of the pointers stored in tag and data arrays also increases. Section 3.3 details the data array organization, while Sections 3.5 and 3.6 analyze implementation issues, and Section 5.1 analyzes the influence of the data array associativity on the reuse cache size and performance, concluding that the impact of data array associativity is very limited both on cost and performance.

3.2 Tag Replacement Policy

A key benefit of decoupling is to specialize replacement, that is, to order and evict tags and data separately on the basis of their different roles. Any replacement policy may work in the reuse cache tag array if it fosters the presence of reused lines and takes into account inclusion and the trade-offs it brings [10]. In this paper, we adopt a *not recently reused (NRR)* [1] replacement policy, which is based on the *not recently used (NRU)* algorithm [24]. Both have the same implementation cost, one bit per line. In NRR, the *Non-Recently Reused (NRR)* bit, distinguishes recently reused lines from not recently reused ones. When a line is loaded into the SLLC due to a miss, its NRR bit is set (it has not been recently reused); when there is a hit (a reuse), the NRR bit is unset. NRR uses the full-map directory bits to distinguish between lines present or not in the private caches. Victim lines are randomly selected among lines having the NRR-bit set and not included in the private caches.

3.3 Data array: organization and replacement policy

The data array associativity is only related to the replacement in the data array. An associative search in the data array is never necessary because the forward pointer in the tag array indicates the set and way in the data array.

We assume a number of sets in the tag array greater than or equal to that in the data array, using in both arrays the least significant bits of the line address as set index. Therefore, a forward pointer only has to indicate the way of the data array where the line is, while a reverse pointer has to show the way of the tag array as well as the bits of the tag array index not included in the data array index (a number of bits equal to $\log_2$ the number of tag array sets - $\log_2$ the number of data array sets). For instance, a data array with only one set (fully associative) requires $\log_2$(the number of data array entries) bits for each forward pointer, and reverse pointers require $\log_2$(the number of tag array entries) bits.

Only reused lines are allocated in the data array. Thus, in order to exploit reuse locality, replacement should rely on recency. Given our low-cost design goal, we use NRU as the data array replacement algorithm. However, NRU performance decreases for high associativities. Thus, for the fully associative case, we have tested a suitable alternative, the low-cost Clock algorithm introduced in [7]. The implementation cost of both NRU and Clock is one bit per line.

When evicting a data line, the corresponding forward pointer in the tag array has to be invalidated. The corresponding tag array entry is located by following the reverse pointer of the just invalidated line (Figure 2).

3.4 TO-MSI: an example coherence protocol

Conventional coherence protocols assume that each line present in a cache has an entry in both the tag and data arrays. However, a reuse cache needs a coherence protocol able to deal with lines that have entries in the tag array but not in the data array.

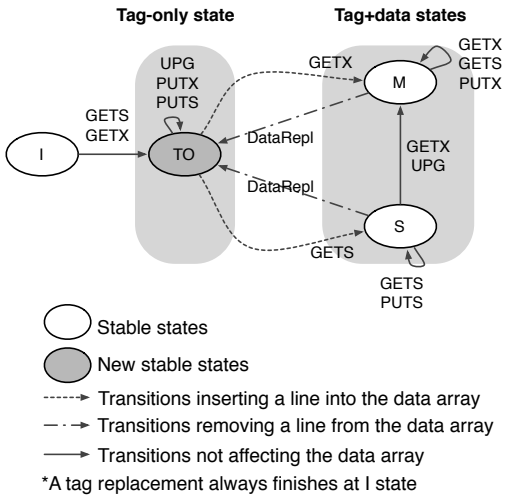


Figure 3: Functional description of the TO-MSI example coherence protocol

Figure 3 outlines an example coherence protocol based on the MSI protocol[2][8], which is able to work with decoupled tag and data arrays. Table 1 explains the states and events of the protocol. In this description, neither replacement nor external requests are represented. In every state except *I*, private caches may or may not have copies of the line. This information is stored in a *full-map* directory by using a presence bit vector.

Two different groups of states can be considered: *tag+data* states that contain lines in the data array; and *tag-only*, a single state in this simplified version of the protocol, that contains lines that are not present in the data array.

Transitions between the two groups always imply getting lines in or out of the data array.

1) *From tag-only to tag+data.* When the first SLLC hit (reuse) is observed the state changes from tag-only to a state of the tag+data group. These transitions are represented by *dash-dotted* arrows in the figure and are caused by GETS and GETX events when the state of the line is TO.

2) *From tag+data to tag-only.* When a line is evicted from the data array the state changes from the tag+data group to tag-only. Replacing a line in the data array requires the protocol to record that the tag no longer has associated data. The dashed arrows labeled with the *DataRepl* event, coming out of M and S, represent these state transitions.

3.5 Hardware Cost

In this section, we calculate the reduction in the total number of bits required by a reuse cache with respect to a conventional cache, by taking into account both the tag/data array reduction and the increase due to the forward and reverse decoupling pointers. As an example, for an eight-core

[2]For the sake of clarity, a simple protocol is shown here. In our evaluation, we consider a MSI-MOSI protocol with seven stable states. This protocol allows interconnection between several CMPs. The reuse cache needs three additional stable states to track the tag-only situations.

Name	Cache	Memory	Data
I	Invalid or not present	-	No
S	Unmodified	up-to-date	Yes
M	Modified	stale	Yes
TO	Only tag, no data	up-to-date or stale	No

(a) States of TO-MSI protocol

Event name	Description
GETS	Data read or fetch request
GETX	Write request
UPG	Upgrade request
PUTS	Eviction notification (clean)
PUTX	Eviction notification (dirty)
DataRepl	Eviction in the Data array

(b) Events of TO-MSI protocol

Table 1: States and events of the TO-MSI example coherence protocol

Component	Conv. 8MB	RC-4/1 Full	RC-4/1 16-way
Tag	21	22	22
Coherence	4	5	5
Full-map vector	8	8	8
Replacement	1	1	1
Fwd. pointer	-	14	4
Tot. tag entry (bits)	**34**	**50**	**40**
Data	512	512	512
Valid	-	1	1
Replacement	-	1	1
Reverse pointer	-	16	6
Tot. data entry (bits)	**512**	**530**	**520**
Tag array (K entries)	128	64	64
Data array (K entries)	128	16	16
Total size (Kbits)	**69888**	**11680**	**10880**
Reduction		**83.3%**	**84.4%**

Table 2: Hardware cost

system, we detail a 8 MB conventional cache and a reuse cache with a 1:8 scaling in the data array and a 1:2 scaling in the tag array. We consider 16-way and fully associative data array designs for the reuse cache.

The conventional cache is 16-way, and has 64-byte lines. Further, the conventional cache requires 34 bits per line in the tag array: 21-bit tags (assuming 40 bits of physical address space in a 64-bit architecture), 12-bit coherent state (4-bit state and 8-bit presence vector) and 1 bit for replacement (NRU algorithm[3]). The data array requires 512 bits per line. Overall, the conventional cache needs 69888 Kb (see Table 2).

The reuse cache has a 1 MB data array and a tag array with the same number of entries as a 4 MB conventional cache (RC-4/1 in the Table 2 headings). A tag array entry requires the same fields as a conventional cache plus a forward pointer per line and one additional bit for the coherence state[4]. The forward pointer requires 14 bits for the fully associative (16 K-line) data array but only 4 bits for the 16-way data array. Each data array entry requires 512 bits of data, a reverse pointer, one bit for the replacement policy (Clock/NRU), and one valid bit per entry. The reverse pointer requires 16 bits for the fully associative data array (4 and 12 bits to store way and set, respectively) but only 6 bits for the 16-way data array (4 and 2 bits to store way and set index, respectively).

Overall, the reuse cache (4 MB tag array / 1 MB data array) with fully associative data array needs 11680 Kb while the reuse cache with 16-way data array needs 10880 Kb. Thus, the set-associative organization of the data array requires 6.8% fewer bits than the fully associative design. Regarding the 8 MB conventional cache, the example reuse cache with fully associative data array would require only a 16.7% of its storage capacity (15.6%, considering the set-associative data array).

[3]Although LRU has been used as the replacement policy of the conventional cache in Section 5, NRU has been considered here to not bias the comparison.

[4]We consider the coherence protocol that supports our proposal roughly doubles the original in number of states, and thus we add on one additional bit.

Org.	Tag acc.	Data acc.	Total acc.
RC-8/8	+36%	same	+10%
RC-8/4	+36%	-16%	-3%

Table 3: Relative variations of the reuse cache access latency with respect to an 8MB conventional cache (organized in 4 banks of 2 MB).

3.6 Latency

This section describes a comparison of latencies between conventional and reuse caches. CACTI v6.5 was employed to model the cache access latency [25]. Serial access to SRAM tag and data arrays was assumed. Such arrays use a 32-nm technology node.

Table 3 shows relative latency variations with respect to an 8 MB conventional cache for two configurations: *i*) a reuse cache with the same tag and data array entries as the 8 MB conventional cache (RC-8/8); and *ii*) a reuse cache with the same number of tags but half the number of data array entries that there are in the 8 MB conventional cache (RC-8/4). With respect to a conventional cache with the same number of sets, the tag array access time of a reuse cache increases by 36% due to the additional bits of the forward pointers. Access latency to the data array decreases by 16% when its size is reduced from 8 to 4 MB. Overall, the RC-8/4 access latency is 3% lower than in the 8 MB conventional cache. It is important to note that the data array access latency of the 8 MB conventional cache is roughly three times larger than its tag array access latency.

In Section 5, when comparing reuse and conventional caches, the data array, or both the tag and the data arrays of the reuse cache are always smaller than those of conventional caches. Thus, we consider that the access time of the evaluated reuse cache configurations does not increase with respect to the conventional cache with which it is compared. Further, we assume the same latency in all reuse cache configurations, although the access time decreases significantly as the sizes of the tag and data arrays decrease.

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

Table 4: Baseline system configuration

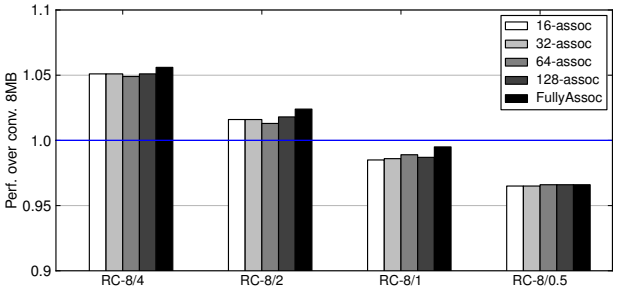


Figure 4: Average speedup relative to the baseline for reuse caches of various data array sizes and associativities. Tag array size and associativity are 8 MBeq and 16, respectively.

4. METHODOLOGY

4.1 Experimental setup

Simics, a full-system execution-driven simulator [22] was used as a simulation engine. The *Ruby* plugin from *Multifacet's multiprocessor simulator* toolset was employed to model the memory hierarchy with a high degree of detail: coherence protocol, on-chip network, communication buffering, contention, etc. [23]. Moreover, we added a detailed DDR3 DRAM model.

Multiprogrammed *SPEC CPU 2006* workloads were run on a Solaris 10 Operating System. In order to locate the end of the initialization phase, we used hardware counters on a real machine and ran all the SPARC binaries with the reference inputs until completion. For an eight-core system, we produced a set of 100 workloads, random combinations of 8 programs each, taken from all the 29 *SPEC CPU 2006* programs (no effort being made to distinguish between integer and floating point applications). The applications appear from 16 to 35 times, the average number of occurrences being 27.6 with an standard deviation of 4.5.

At each checkpoint, it was ensured that no application was in its initialization phase. This is achieved by running all the programs for as many instructions as the longest initialization phase among all the programs included in the multiprogrammed workload. The cycle-accurate simulation started at those checkpoints, 300 million cycles were run to warm up the memory hierarchy, and statistics were then collected for the next 700 million cycles. Table 5 shows the average of misses per kilo-instruction (MPKI) in all the instances of an application at the three levels of the cache hierarchy when the eight applications of a workload were run together.

4.2 Baseline system

The baseline system has eight in-order cores. Each core has two levels of private caches and all the cores share the last-level inclusive cache. This SLLC has four banks interleaved at cache line granularity (64 B). A MSI-MOSI protocol maintains the memory system coherent. A crossbar communicates the local caches and the SLLC banks. There is a single DDR3 memory channel and the DRAM memory bus runs at a quarter of the core frequency. Table 4 gives additional implementation details.

5. RESULTS

We first compare the performance of the reuse cache varying the data array size and associativity. We then study the optimal size ratio between tag and data arrays. In Section 5.3 and Section 5.4 we give insight into the reuse cache behavior by analyzing the percentage of lines not entered in the data array and the number of live lines when reducing the reuse cache size. Subsequently, in Section 5.5, we compare the reuse cache with DRRIP [12], a state of the art replacement algorithm, and NCID [35], a recent proposal that uses a decoupled tag-data cache. Finally, in Section 5.7, we analyze the behavior of the reuse cache when running parallel applications.

Throughout this section, results are expressed as speedups of the different reuse cache configurations relative to a baseline SLLC. The baseline SLLC considered is an 8 MB, 16-way conventional cache with LRU replacement.

When describing the reuse cache tag array, we use MBeq as the tag array equivalent to that of a 1 MB conventional cache. We always maintain a tag array associativity of 16 and a line size of 64 bytes. For instance, a 4 MBeq tag array has 64 K tags (4 MB / 64) organized in 4 K sets (64 K tags / 16). We use RC-x/y to refer to a reuse cache with a tag array equal to that of a x MB conventional cache (x MBeq) and a data array of y MB. As an example, RC-4/1, the reuse cache outlined in Table2, has a tag array equivalent to a 4 MB conventional cache with 1 MB data array.

5.1 Data array size and associativity

Figure 4 shows the performance of a reuse cache with 8 MBeq tag array, varying the data array size from 4 MB (RC-8/4) to 512 KB (RC-8/.5) and setting the data array associativity to 16, 32, 64, 128 or fully associative. Each bar represents average performance relative to baseline for the 100 workloads described in Section 4.

In general, performance varies very slightly and unevenly for associativities between 16 and 128. The reuse cache with a fully associative data array achieves better results for all sizes. However, the differences are not significant. For instance, the difference between 16-way and fully associative varies from -0.1% for RC-16/8 to +1% for RC-4/1. We can conclude that the fully associative and set-associative designs are very similar both in cost and performance. It is important to remember that the fully associative organization is easy to implement because it never needs associative lookups. Further, the clock replacement algorithm is really simple, being even cheaper than NRU in a set-associative organization with a high associativity. Unless stated otherwise, the remaining experiments are carried out with fully associative data arrays.

Regarding the size of the data array, a reuse cache with

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

Table 5: Average MPKI at each cache level of the baseline system (8 MB LRU).

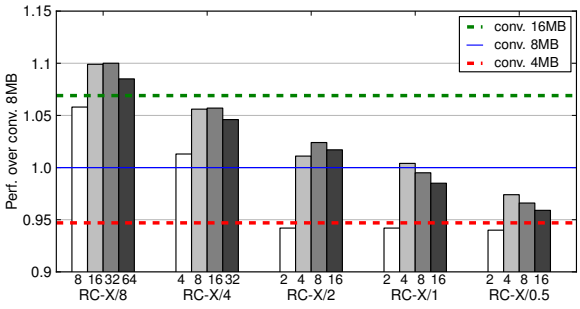


Figure 5: Average speedup relative to the baseline for reuse caches varying the tag and data array sizes. Tag array associativity is 16.

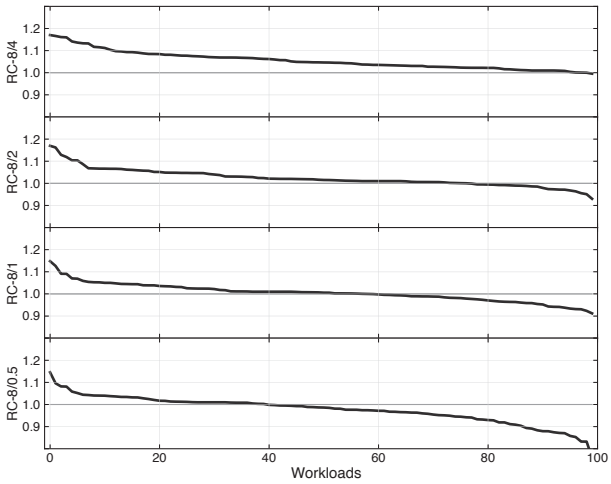


Figure 6: Speedup of the selected reuse cache configurations for all the 100 multiprogrammed workloads, relative to the 8 MB baseline. The data array ranges from 4 MB (RC-8/4) to 512 KB (RC-4/0.5).

one quarter the capacity of the baseline cache (RC-8/2) shows on average even better performance than the baseline cache (+ 2.4%). A further reduction in the data array, RC-8/1, marks a turning point with the reuse cache performing slightly less well than the baseline cache (-0.5%).

5.2 Tag array size

In this section, we investigate which tag array size achieves the best performance for each size of the data array. Figure 5 shows the relative performance of a reuse cache relative to the baseline 8 MB cache. For each size of the data array (X axis), we consider several different sizes of the tag array. In each configuration, the tag array must have more entries than the maximum of the data array and the sum of entries in the private caches (8x256 KB). In order to extend the comparison to a 16 MB conventional cache, we will also include a reuse cache with a 8 MB data array.

For a given data array size, increasing the size of the tag array beyond a certain limit is not worthwhile, because it only leads to identifying a larger reuse working set, the size of which is beyond the capacity of the data array. The optimum data-tag ratio is always 4 except for a 512 KB data array, where a ratio of 4 requires a 2 MBeq tag array, which is the minimum for tracking the aggregated 2 MB of private caches. Further, the small performance advantage of RC-32/8 over RC-16/8 would not justify selecting the 32 MBeq tag array. The same holds true comparing RC-16/4 and RC-8/4. Hence, for the remaining sections, the reference sizes of the reuse cache for each data array size will be: RC-8/4, RC-8/2, RC-4/1 and RC-4/0.5. We can identify RC-4/1 as the smallest reuse cache that performs better than a conventional 8 MB cache; indeed, RC-4/1 requires half the tags, one-eighth the data, and hence only 16.7% of the storage of the conventional 8 MB cache.

Figure 5 also shows the relative performance of 16-way LRU in 4 and 16 MB caches relative to the 8 MB baseline (red and black lines). We could replace both conventional caches with the smaller reuse cache giving some performance advantages. Specifically, a reuse cache with the same tag array but half the data array (RC-16/8) outperforms a 16 MB conventional cache. Likewise, for a 4 MB conventional cache it suffices a reuse cache with the same tag array but one-eighth the data array (RC-4/0.5).

Figure 6 plots the reuse cache speed-ups for every workload, varying the reuse cache size. We only show the previously selected configurations having the best performance/size tradeoff. In each plot, the different workloads are ordered along the horizontal axis according to their speed-up.

As can be seen, the speed-ups range increases as the size of the reuse cache decreases. RC-8/4 outperforms the baseline for almost all the workloads (99 out of 100). RC-4/1 seems to be a good design point, as it is better than the baseline for 64 out of 100 workloads, reaching a speedup of 1.14 and a slowdown of 0.82, only four and two workloads suffering losses and improving their performance by more than 10%, respectively.

5.3 Data lines entered in the data array

Table 6 shows, for the one hundred workloads, the mean

RC-x/y	8/4	8/2	4/1	4/0.5	Conv.
Avg. (%)	93	93	95	95	0
Min. (%)	81	81	89	89	0

Table 6: Mean and minimum percentage of lines not entered in the data array with respect to tags entered in the tag array for different reuse cache and conventional configurations.

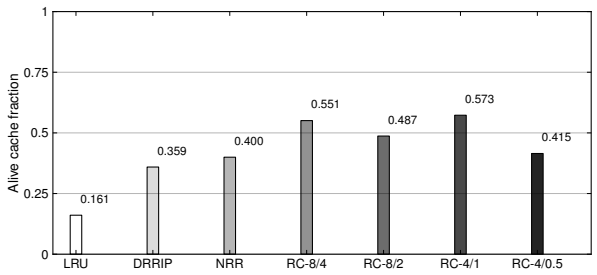


Figure 7: Average fraction of live lines for 8 MB LRU, DRRIP, and NRR conventional caches, and for the selected reuse cache configurations

and the minimum percentage of lines not entered in the data array with respect to tags entered in the tag array. Clearly, this percentage is zero in a conventional cache because tags and data are always allocated together. In a reuse cache, however, the selective allocation policy discards most lines. As we can see, the reuse cache is very selective allocating data lines, and selectivity increases as the tag array decreases. For instance, the RC-8/4 and the RC-4/1 configurations discard on average 93 and 95.4%, respectively. Even for the most demanding workloads more than 80% of the data lines are discarded. Hence, we can conclude that the reuse cache is very effective at protecting useful data lines against pollution, because the discarded data lines are not able to evict lines that have shown reuse. On the other hand, the percentage of reused data lines loaded twice -the downside of reuse caches-, is exactly 100% minus the percentages above. For instance, the RC-4/1 reloads 4.6% of the data lines, paying twice the main memory accessing cost.

5.4 Fraction of live lines in the data array

It is worth considering how the average lifetime of the lines kept in the reuse cache changes. Figure 7 shows the average fraction of the data lines that are alive for all 100 workloads in the best reuse cache configurations. We also plot data for the 8 MB conventional cache with LRU and DRRIP [12] replacement policies. The average percentage of live lines in the baseline cache is only 16.1% with LRU replacement (*LRU* in Figure 7) and 35.9% with DRRIP, consistent with the analysis presented in Section 2 for the example workload. The reuse cache RC-8/4 increases the percentage of live lines up to 55.1%. That is, with half the lines of the baseline cache, RC-8/4 almost doubles the *number* of live lines compared to the baseline cache (55.1% of 4 MB vs. 16.1% of 8 MB). This is because the combined tag/data replacement algorithm is able to identify the lines being reused.

RC-8/2 and RC-4/1 also achieve high fractions of live lines. However, a similar fraction operating in a smaller data array implies a very significant reduction in the total number of live lines. In spite of that, with a quarter the lines of the baseline cache, with RC-8/2 there are only 9% fewer live lines than with the baseline cache. RC-4/1, while keeping a significantly lower number of live lines, still improves on the performance of the baseline cache because the replacement algorithm of the data array prioritizes lines with the shortest reuse distance, which are the lines that receive the most hits.

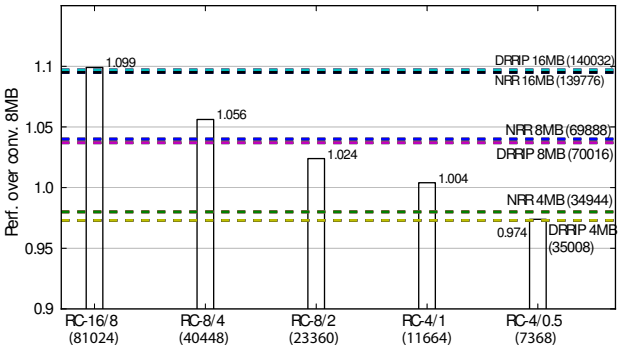


Figure 8: Average speed-ups of reuse caches and conventional caches using both NRR and DRRIP. Hardware storage (in Kbits) for each configuration is shown between parentheses

5.5 Comparison with alternative state of the art proposals

Throughout previous sections, results have been reported as speed-ups relative to a baseline cache with LRU replacement. Performance achieved by LRU replacement may be considered an upper bound for commercial processors as they usually use LRU approximations with slightly poorer performance. However, in this section we also compare the reuse cache with both a conventional cache fitted with state of the art replacement algorithms and an alternative decoupled cache organization that also makes it possible to reduce the data array size.

Comparison with RRIP [12] and NRR [1]. Thread-aware (TA) DRRIP and NRR are two state of the art replacement algorithms for SLLCs (see Section 6 and Section 3.2, respectively, for brief descriptions of these schemes). Figure 8 compares conventional caches operated with TA-DRRIP and NRR (represented as horizontal lines) with several reuse cache configurations (represented as bars). The results plotted are the average speed-ups relative to the 8 MB baseline cache using LRU for all the 100 workloads described in Section 4.

TA-DRRIP replacement improves on the conventional 8 MB cache performance by 3.7% with respect to LRU replacement. Even so, a reuse cache with half the data array (RC-8/4) is 2% better than the conventional cache with TA-DRRIP. Similarly, RC-16/8 is 0.5% better than the conventional 16 MB cache with TA-DRRIP. Finally, the conventional 4 MB cache with TA-DRRIP could be replaced with a reuse cache with one-eighth of the data array (RC-4/0.5) with similar performance.

Figure 8 also shows the hardware storage needed for each configuration. Comparing hardware cost and performance of the reuse cache to a 16 MB conventional cache using DRRIP

or NRR, we can observe how the latter has a hardware cost of 140,000 Kbits to achieve a relative performance of 1.094 relative to the baseline, slightly lower than 1.099 achieved by the RC-16/8, which has a hardware cost of 81,024 Kbits (41% hardware cost savings with respect to the 16 MB conventional cache with DRRIP or NRR). At the same time, an 8 MB conventional cache using DRRIP or NRR has a cost of 70,000 Kbits achieving a relative performance of 1.037; in contrast RC-8/4 achieves 1.056 having a cost of 40,448 Kbits (48% hardware cost savings with respect to the 8 MB conventional cache with DRRIP or NRR). If we focus on a 4 MB conventional cache using DRRIP or NRR, we see it has a cost of 35,000 Kbits and achieves a relative performance of 0.975 which is similar to the performance of RC-4/0.5. However, the latter has a cost of 7,368 Kbits, 80% lower.

Comparison with NCID [35]. NCID involves the adding of tags to each set of a conventional SLLC in order to maintain tag inclusion of the private caches while the data array can be non-inclusive or exclusive. The authors who proposed this approach evaluate the NCID architecture supporting a selective allocation policy to address transient data, and compare it with a conventional cache with a bimodal insertion policy [26] in terms of miss rate reduction. However, NCID with selective allocation could also be used to reduce the data array size maintaining performance.

Following the NCID implementation, the selective mode allocates 5% of lines as most recently used (data and tag) and the remaining 95% as least recently used (only tag). Set dueling selects between selective and normal allocation for each thread.

When specializing NCID to achieve size reduction, we reduce the data array with respect to the tag array. The NCID architecture requires an equal number of sets in the tag and data arrays. Hence, reducing the data array size implies reducing the data array associativity. As an example, an NCID cache with a 16-way, 8 MBeq tag array implies a 1 MB data array with only 2 ways. Therefore, in order to make a fair comparison, we have to choose reuse caches that have the same number of sets and associativities in the data array.

Figure 9 compares reuse cache with NCID for an 8 MBeq tag array and several data array sizes. Each bar represents average performance relative to the baseline cache for the 100 workloads reported in Section 4.

By reducing the data array size, no NCID settings match the performance achieved by the baseline 8 MB cache. For all data array sizes, the reuse cache performs better than NCID, with relative gains of 7.0, 6.4, 5.2 and 5.3% for data array sizes 4 MB, 2 MB, 1 MB and 512 KB, respectively.

5.6 Per-application performance analysis

In order to clarify how the reuse cache affects performance of each application, Figure 10 shows the distribution of speed-ups by application. The number of workloads in which each application appears is shown along the top of the graph. For three reuse cache configurations with an 8 MB tag array and several data array sizes (RC-8/4, RC-8/2 and RC-8/1) five figures are plotted, namely the minimum, the first quartile, the median, the third quartile, and the maximum of the performance with respect to an 8 MB conventional cache using LRU.

RC-8/4 improves the behavior of all the applications in most mixes with respect to the baseline. Only the first quartile of performance for Gems and Calculix is lower than 1 (0.99 and 0.98, respectively), but in both cases the median is greater than or equal to 1.

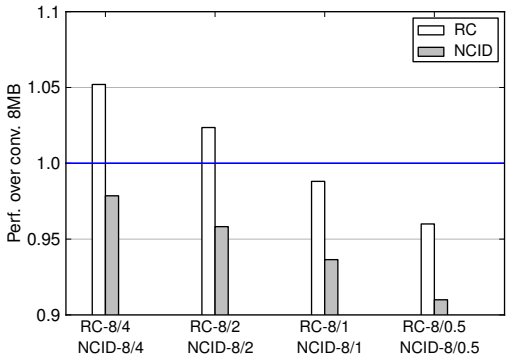


Figure 9: Average speedups of NCID and reuse caches

The number of applications in which performance losses are observed increases as data array size is progressively reduced. When RC-8/1 is used, the median of five applications is clearly lower than 1, the third quartile being lower than 1 in three cases. In these applications, the reuse distance is longer than the available space in the data array. Thus, these lines will not be reused within their stay in the cache and they will be evicted very quickly. Such behavior allows applications with shorter reuse distances to cope with the available data array space.

5.7 Parallel workloads

In this section, we analyze the behavior of our proposal when running parallel applications. We selected the five applications of the PARSEC [3] and SPLASH-2 [32] suites which have more than 1 MPKI in the baseline SLLC. Specifically, the selected applications are blackscholes, canneal, ferret, and fluidanimate from PARSEC and ocean from SPLASH-2; and their MPKIs are 4.5, 3.5, 1.3, 1.7, and 13.4, respectively. We utilized the simmedium input set for PARSEC applications and a 1026x1026 grid for Ocean. For PARSEC applications a checkpoint was created in the parallel phase. The cycle-accurate simulation started at those checkpoints, warming the memory hierarchy for 300 million cycles, and then collecting statistics for the next 700 million cycles. Ocean was run to completion but performance statistics were only taken in the parallel phase.

Figure 11 shows, for the five parallel applications, the performance of the reuse cache with data array sizes from 4 MBytes (RC-8/4) to 512 KBytes (RC-8/0.5) relative to the baseline SLLC. Only ferret suffers a loss in performance, relative to the baseline cache, with a reuse cache. This loss varies from 1% with RC-8/4 to 11% with RC-8/0.5. However, in the other four applications even RC-8/0.5 achieves better performance than the baseline cache (canneal and ocean showing speed-ups of more than 10%).

5.8 Higher memory bandwidth

We have analyzed what would be the impact on the system performance of having a higher available bandwidth to main memory. This higher available bandwidth would make misses to pay, in general, a lower latency. On average, we

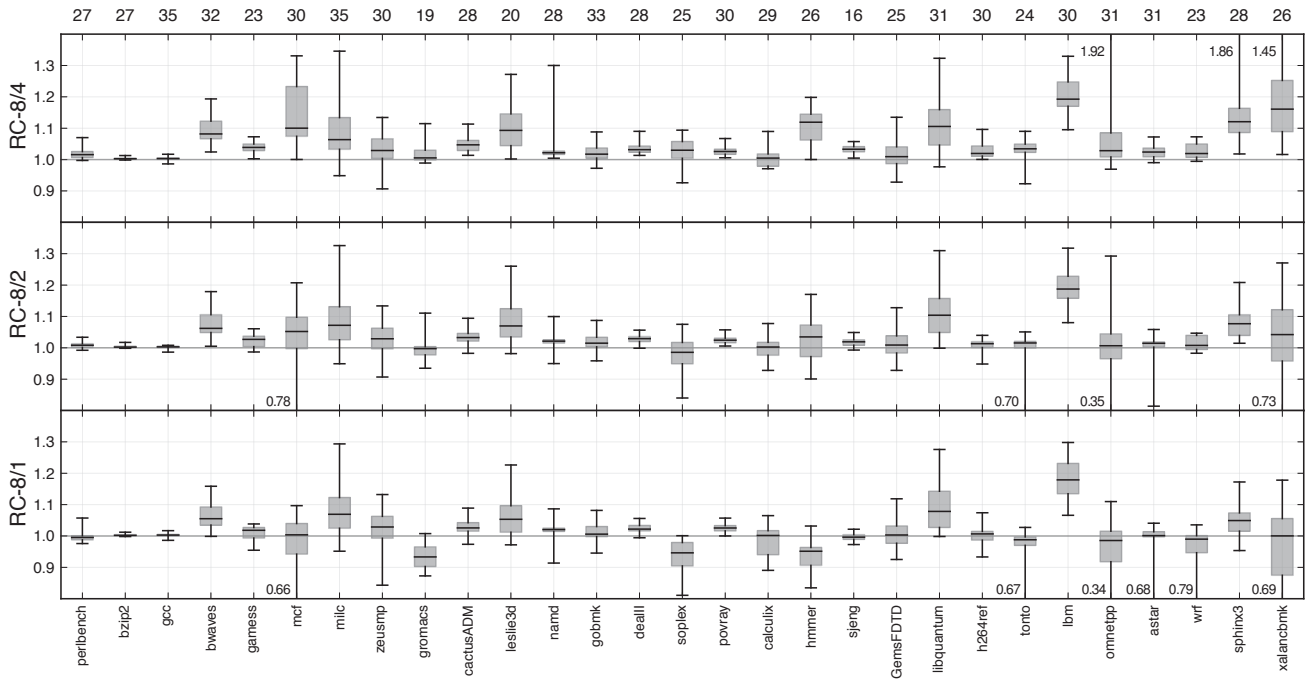


Figure 10: Per-application speedup analysis

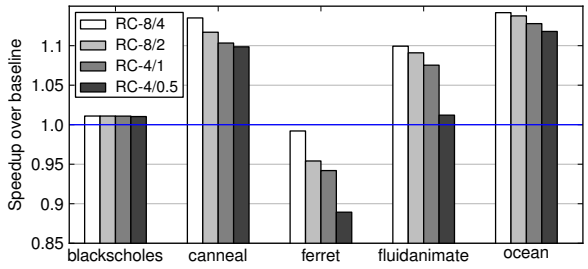


Figure 11: Speedup with reuse cache relative to the baseline for five parallel applications, with data array sizes from 4 MBytes (RC-8/4) to 512 KBytes (RC-8/0.5)

have not observed much contention at the memory controller neither in the conventional nor in the reuse cache. We run simulations including 2 and 4 memory channels and, as it could be expected, simulations showed that the system performance varied less than 1%.

6. RELATED WORK AND CONCLUDING REMARKS

Two recent studies are directly related to our proposal [20, 35]. The cache organization relies on tag/data decoupling to retain tags inclusion property and to use a selective allocation policy of lines in a cache miss.

The NCID architecture [35] offers several tag/data decoupling designs to retain the inclusion property on tags. One such design uses NCID to support a selective allocation policy to address transient data. NCID only allocates tag and data for a randomly chosen 5% of the lines and only tag for the remaining 95%. Set dueling is proposed to select between normal or selective fill policies. This architectural option can be used to reduce the data array size, and although not the purpose for which it was designed, according to the results in Section 5.4, its performance as a reuse cache is quite good.

Lodde *et al.* [20] use cache line state and coherence messages to classify lines as private or shared. Such information is used to selectively allocate only shared lines in the data array in order to reduce data array size. This organization requires tags to be moved within the tag array when the classification of a line changes. The information used by the selective allocation policy does not take reuse into account. Thus, transient private cache lines are put into the shared data array, increasing the required data array size. Our proposal relies on reuse locality detection, independently of which private cache requests the line. Hence, shared lines are implicitly allocated in the data array and private lines are only allocated if reuse locality is detected.

In addition, Chishti *et al.* propose the NuRAPID cache, which decouples tag lookup and data placement in order to reduce the average access latency in dynamic non-uniform cache architectures [6]. Tag/data decoupling has also been proposed in the V-way cache by Qureshi *et al.* to achieve a high associativity and reduce the number of conflict misses in the non-inclusive last level cache [27]. This proposal was evaluated in a single-processor system. The V-way cache stores the same number of items in tag and data arrays and inserts into the cache all the data requested by the lower level caches. In contrast, the reuse cache we propose stores all the tags but only inserts in the data array the small fraction of lines showing reuse.

Several studies explore the prediction of the reuse behav-

ior of the incoming lines when they enter the SLLC, and use of this prediction in an insertion policy [5, 18, 29, 33]. These approaches are complementary to the reuse cache design. For instance, the predictors proposed in [29, 33] could be used to increase the performance of the reuse cache by predicting the reuse behavior of a cache line on a tag miss. The OBM mechanism proposed in [18] signals the first line to be reused in the incoming-victim line pair involved in a miss. Again, this detection scheme could be used to improve the reuse cache, for instance, on a tag array hit missing in the data array.

Chaudhuri et al. propose to track reuse behavior within the private caches and utilize it to estimate reuse in the SLLC when the lines are evicted from private caches [5]. Such a predictor could be used to change the fixed reuse prediction performed in the reuse cache whenever a line is evicted from private caches.

Reuse locality was first observed and exploited in cache memories for disks. Segmented LRU tries to protect useful lines against harmful behaviors (i.e., a burst of single-use accesses) by dividing the classical LRU stack into two different logical lists, the *referenced* and the *non-referenced* list [13]. The boundary between lists is fixed and victims are selected in order to preserve that limit. Recent proposals apply this idea to the replacement policy of the SLLC.

Under the dynamic insertion policy, the maximum size of the non-reference list is one (LRU position). Two replacement policies, LRU and the bimodal insertion policy, are dynamically selected using set dueling [26]. The bimodal insertion policy puts most of incoming lines into the LRU position and the other lines into the MRU position. This idea was used in [35] to evaluate an architectural option with selective allocation. Other proposals such as dynamic segmentation [15] and dueling segmented LRU [9] consider these two logical LRU divisions and try to dynamically find an optimal configuration using set dueling.

Re-reference interval prediction (RRIP) involves a modified LRU that considers a chain of segments, where all the cache lines in a segment are supposed to have the same re-reference interval [12]. In static RRIP, new lines are inserted with an *intermediate* re-reference interval. Bimodal RRIP inserts lines with *long* re-reference intervals but a small fraction of randomly chosen lines are introduced with *intermediate* re-reference intervals. Finally, this proposal includes a dynamic version, DRRIP, that uses *set dueling* to select between static and bimodal RRIP. For a SLLC in a CMP, TA-DRRIP requires a set dueling monitor for each thread.

In inclusive hierarchies the private caches absorb most of the temporal locality and the hot lines may lose positions in the LRU stack of the SLLC, up to the point of being evicted. A recent paper by Jaleel et al. addresses this issue and proposes several mechanisms to identify these lines and to prevent their replacement in the SLLC [10]. The identification of hot lines can be used to introduce more logical lists or segments to improve performance. In a reuse cache, these lines are identified using directory state information. The reuse cache predicts that temporal locality of a line is drained when such a line is evicted from the private caches. Thus, dead block prediction can be used to tune the prediction [16, 17, 19], although it is necessary to send a message to SLLC.

Previous studies have already analyzed how the SLLC replacement algorithm should act with prefetched data. With the aim of avoiding cache pollution, prefetched data should be assigned a lower priority than the data actually demanded by the processors [31, 34]. Moreover, Wu *et al.* still consider prefetched data as low priority contents once the processor have used them, relying on the idea that the prefetcher will be able to put that data into the cache again in advance if it has already achieved this once. The replacement algorithm used in the reused cache is able to adopt these ideas in a straightforward way: simply considering prefetched lines to have a priority as low as the non-reused data.

In this work, we use NRR as the replacement algorithm for the tag array. However, alternative replacement algorithms proposed for increasing hit ratio may govern the tag or data replacement in a reuse cache, in particular those, that for a given data array size, are able to identify lines that will be referenced in the near future. It is not our goal to select the best of them, but rather to show that it is possible to dramatically reduce the SLLC size without compromising performance, even using extremely cheap and simple replacement algorithms.

7. CONCLUSIONS

The reference stream observed by the SLLC of a CMP exhibits little temporal locality but, instead, it exhibits reuse locality. As a consequence, a high proportion of the SLLC lines is useless because the lines will not be requested again before being evicted, and most hits observed by the SLLC come from a small subset of already reused lines.

In this paper, we have proposed the reuse cache, an SLLC with a very selective data allocation policy intended to track and keep that small subset of lines showing reuse. In a reuse cache, the tag and the data arrays are decoupled. On the one hand, the size of data array can be dramatically reduced without negatively affecting performance. On the other hand, the tag array tracks the reuse order of recently referenced lines, and has the size required to store the tag of the lines in the data array and private caches.

We have evaluated our proposal by simulating an eight-core CMP system running multiprogrammed and multithreaded workloads. The results show that a reuse cache can achieve the same performance as a conventional cache with a much lower hardware cost. For instance, a reuse cache with the tag array equivalent to a conventional 4 MB cache but with only a 1 MB data array, gives the same average performance as an 8 MB conventional cache. That reuse cache would require only a 16.7% of the storage budget of the conventional cache.

We have illustrated the usefulness of the reuse cache concept with a case study: reducing space and maintaining performance. Evidently, the reuse cache could also be used in other settings, or for other reasons, for example, seeking to meet design goals in relation to chip area, performance or energy tradeoffs.

8. ACKNOWLEDGEMENTS

Thanks to the GaZ people for their support. We also thank the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by grants TIN2010-21291-C02-01 and TIN2012-34557 (Spanish Gov. and European ERDF), Consolider CSD2007-00050 (Spanish Gov.), gaZ: T48 research group (Aragón Gov. and European ESF), and HiPEAC-2 NoE (European FP7/ICT 217068).

9. REFERENCES

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

[2] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proceedings of the 15th Annual International Symposium on Computer architecture*, ISCA '88, pages 73–80, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] J. Cantin, M. Lipasti, and J. Smith. Stealth prefetching. *SIGOPS Oper. Syst. Rev.*, 40:274–282, October 2006.

[5] M. Chaudhuri, J. Gaur, N. Bashyam, S. Subramoney, and J. Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *PACT*, pages 293–304, 2012.

[6] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *Proc. of the 36th annual IEEE/ACM Int. Symp. on Microarchitecture*, MICRO 36, pages 55–, Washington, DC, USA, 2003. IEEE Computer Society.

[7] F. J. Corbató. A paging experiment with the multics system. *MIT Project MAC Report MAC-M-384*, May, 1968.

[8] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

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

[10] A. Jaleel, E. Borch, M. Bhandaru, S. Steely Jr., and J. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (tla) cache management policies. In *Proceedings of the 43rd Annual IEEE/ACM Int. Symp. on Microarchitecture*, MICRO '43, pages 151–162. IEEE Computer Society, 2010.

[11] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer. Adaptive insertion policies for managing shared caches. In *Proc. of the 17th int. conf. on Parallel architectures and compilation techniques*, PACT '08, pages 208–219. ACM, 2008.

[12] A. Jaleel, K. Theobald, S. Steely, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proc. of the 37th annual int. symp. on Computer architecture*, ISCA '10, pages 60–71. ACM, 2010.

[13] R. Karedla, J. Love, and B. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38 –46, march 1994.

[14] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *Proc. 28th Annual Int Computer Architecture Symp*, pages 240–251, 2001.

[15] S. Khan, Z. Wang, and D. A. Jimenez. Decoupled dynamic cache segmentation. In *Proc. IEEE 18th Int. Symp. High Performance Computer Architecture HPCA 2012*.

[16] S. M. Khan, T. Yingying, and D. A. Jimenez. Sampling dead block prediction for last-level caches. In *Proc. 43rd Annual IEEE/ACM Int Microarchitecture (MICRO) Symp*, pages 175–186, 2010.

[17] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proc. 28th Annual Int Computer Architecture Symp*, pages 144–154, 2001.

[18] L. Li, D. Tong, Z. Xie, J. Lu, and X. Cheng. Optimal bypass monitor for high performance last-level caches. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 315–324, New York, NY, USA, 2012. ACM.

[19] H. Liu, M. Ferdman, J. Huh, and D. Burger. Cache bursts: A new approach for eliminating dead blocks and increasing cache efficiency. In *Proc. MICRO-41 Microarchitecture 2008 41st IEEE/ACM Int. Symp*, pages 222–233, 2008.

[20] M. Lodde, J. Flich, and M. E. Acacio. Dynamic last-level cache allocation to reduce area and power overhead in directory coherence protocols. In *Euro-Par*, pages 206–218, 2012.

[21] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi. Scale-out processors. In *Proc. 39th Annual Int. Symp. on Computer Architecture (ISCA), 2012*, pages 500 –511, june 2012.

[22] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[23] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood. Multifacetś general execution-driven multiprocessor simulator (gems) toolset. *SIGARCH Comput. Archit. News*, 33:2005, 2005.

[24] S. Microsystems. *UltraSPARC T2 supplement to the UltraSPARC architecture 2007. Draft D1.4.3, 19 Sep 2007*.

[25] N. Muralimanohar, R. Balasubramonian, and N. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *Proc. 40th Annual IEEE/ACM Int. Symp. Microarchitecture MICRO 2007*, pages 3–14. IEEE Computer Society, 2007.

[26] M. Qureshi, A. Jaleel, Y. Patt, S. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[27] M. K. Qureshi, D. Thompson, and Y. N. Patt. The v-way cache: Demand based associativity via global replacement. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 544–555, Washington, DC, USA, 2005. IEEE Computer Society.

[28] J. B. Rothman and A. J. Smith. The pool of subsectors cache design. In *Proceedings of the 13th international conference on Supercomputing*, ICS '99, pages 31–42, New York, NY, USA, 1999. ACM.

[29] V. Seshadri, O. Mutlu, M. A. Kozuch, and T. C. Mowry. The evicted-address filter: a unified mechanism to address both cache pollution and thrashing. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 355–366, New York, NY, USA, 2012. ACM.

[30] A. Seznec. Decoupled sectored caches: conciliating low tag implementation cost. In *Proceedings of the 21st annual international symposium on Computer architecture*, ISCA '94, pages 384–393, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[31] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. IEEE 13th Int. Symp. High Performance Computer Architecture HPCA 2007*, pages 63–74, 2007.

[32] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: characterization and methodological considerations. In *Proc. Symp. nd Annual Int Computer Architecture*, pages 24–36, 1995.

[33] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely, and J. Emer. SHiP: signature-based hit predictor for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 430–441, New York, NY, USA, 2011. ACM.

[34] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer. Pacman: prefetch-aware cache management for high performance caching. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 442–453, New York, NY, USA, 2011. ACM.

[35] L. Zhao, R. Iyer, S. Makineni, D. Newell, and L. Cheng. NCID: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. In *Proceedings of the 7th ACM international conference on Computing frontiers*, CF '10, pages 121–130, New York, NY, USA, 2010. ACM.