TESIS DE LA UNIVERSIDAD DE ZARAGOZA

2013

74

Jorge Albericio Latorre

Improving the SLLC Efficiency by exploiting reuse locality and adjusting prefetch

Departamento

Informática e Ingeniería de Sistemas

Director/es

Ibáñez Marín, Pablo Enrique
Llaberia Griñó, José María

ISSN 2254-7606

Prensas de la Universidad Universidad Zaragoza

Universidad Zaragoza
1542

Tesis Doctoral

IMPROVING THE SLLC EFFICIENCY BY EXPLOITING REUSE LOCALITY AND ADJUSTING PREFETCH

Autor

Jorge Albericio Latorre

Director/es

Ibáñez Marín, Pablo Enrique
Llaberia Griñó, José María

UNIVERSIDAD DE ZARAGOZA

Informática e Ingeniería de Sistemas

2013

Universidad Zaragoza

Improving the SLLC efficiency by exploiting reuse locality and adjusting prefetch

Author:
Jorge Albericio Latorre

Supervisors:
Dr. Pablo Ibáñez Marín
Dr. José María Llabería Griñó

Dissertation
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Universidad de Zaragoza

Grupo de Arquitectura de Computadores
Dpto. de Informática e Ingeniería de Sistemas
Instituto de Investigación en Ingeniería de Aragón
Universidad de Zaragoza

Marzo 2013

EXECUTIVE SUMMARY

Chip-Multiprocessors (CMP) are nowadays commonplace from embedded to supercomputer markets. Due to the existing gap, between cpu and main memory speeds, a hierarchy of cache memories is included in the chip with the aim of reducing the average memory access latency.

Commercial CMPs include a hierarchy with two or three levels of cache memories, where the Last-level cache (LLC) is usually shared among all the cores in the system and comprises several megabytes of storage that fill up to a half of the total chip die area. Moreover, the LLC hit ratio critically affects performance since being the last storage inside the chip, any miss at the LLC provokes an expensive off-chip access to the main memory, penalizing the average memory access latency.

The whole CMP efficiency passes through the SLLC efficiency. This thesis makes contributions comprising the SLLC efficiency on two different directions: 1) its performance and 2) its hardware storage.

1) In order to improve SLLC performance, two aspects are tackled during this thesis: hardware prefetching effectivity and replacement policy. The contribution to optimize hardware prefetching is a low-cost controller, called ABS, that relies on a hill-climbing approach to infer the optimal combination of prefetching aggressiveness associated to the different cores of the CMP. This controller achieves better results than state of the art with a lower cost. Regarding to the replacement policy, this thesis proposes to base the SLLC replacement algorithm on a property called *reuse locality* that will be stated during this dissertation. Two new replacement policies are proposed. As it will be shown during the evaluation, our algorithms achieve better performance than the state of the art with a lower hardware complexity.

2) In order to reduce SLLC hardware storage, this thesis proposes a SLLC design called *reuse cache*. This design relies on the reuse locality property to only store data that has shown reuse. The tag array is used to detect reuse and maintain coherence. The experimental evaluation will show that this contribution allows drastic reductions of

the SLLC hardware storage cost while maintaining the overall system performance.

RESUMEN EJECUTIVO

Los chips multiprocesador (CMP) están presentes en la actualidad en todos los segmentos de mercado, desde los teléfonos móviles hasta los superordenadores. Debido a la gran diferencia que existe entre la velocidad del procesador y la de la memoria principal, los CMPs en la actualidad están provistos de una jerarquía de memorias cache que tiene dos o tres niveles.

Cada fallo en el último nivel de esa jerarquía (SLLC) provoca un acceso a la memoria principal que se encuentra fuera del chip. Además la memoria principal está hecha de chips de DRAM. Ambos factores incrementan su latencia de acceso, latencia que se suma a cada uno de los accesos que falla en la SLLC, penalizando a la vez la latencia media de acceso a memoria. Por lo tanto, la tasa de aciertos de la SLLC es un factor crítico para lograr una latencia media de acceso a memoria óptima. Esta tesis fija su atención en la eficiencia de la SLLC y concretamente, en la eficiencia de la prebúsqueda y la explotación de la localidad de reúso.

Para mejorar la eficiencia de la prebúsqueda se propone un controlador de bajo coste llamado ABS capaz de ajustar la agresividad de la prebúsqueda asociada a cada uno de los núcleos de un CMP pero con el ánimo de mejorar el rendimiento general del sistema. El controlador funciona de manera aislada en cada uno de los bancos de la SLLC y recoge métricas locales. Para optimizar el rendimiento global del sistema busca la combinación óptima de valores de la agresividad de prebúsqueda. Para inferir cuál es esa combinación óptima usa una estrategia de búsqueda hill-climbing.

En esta tesis se caracteriza la propiedad de localidad de reúso y se realizan contribuciones que tienen por finalidad última una mayor explotación de dicha propiedad. En concreto, se proponen dos algoritmos de reemplazo capaces de explotar la localidad de reúso, *Least-recently reused (LRR)* y *Not-recently reused (NRR)*. Estos algoritmos son modificaciones de otros dos muy bien conocidos: *Least- recently used (LRU)* y *Not-recently used (NRU)*. Diseñados para explotar la localidad temporal, mientras que los propuestos en esta tesis explotan la localidad de reúso. Las modificaciones propuestas no suponen ninguna sobrecarga hardware respecto a los algoritmos base y al mismo tiempo

muestran ser capaces de incrementar el rendimiento de la SLLC de manera consistente.

Además se propone un diseño para la SLLC llamado *Reuse Cache*. En este diseño solamente se almacenan en el array de datos aquellos bloques que hayan mostrado reúso. El array de etiquetas se usa para detectar reúso y mantener la coherencia. Esta estructura permite reducir el tamaño del array de datos de manera drástica. Como ejemplo, una Reuse Cache con un array de etiquetas equivalente al de una cache convencional de 4MB y un array de datos de 1MB, tiene el mismo rendimiento medio que una cache convencional de 8MB, pero con un ahorro de almacenamiento de en torno al 84%.

CONTENTS

Part I

PRELIMINARIES

In this first part of the thesis, material is provided which will be useful in next parts. Here we justify the presence of the memory hierarchy and present some organizational aspects and properties of the Shared last-level cache. We show previous works that are related with the work exposed on this dissertation, and describe the experimental methodology that has been followed in order to evaluate the contributions of this thesis.

1

INTRODUCTION

1.1 CONTEXT AND BACKGROUND

Chip-Multiprocessors (CMP) are nowadays commonplace from embedded to supercomputer markets. The cores of a CMP have to be constantly fed with data stored in the main memory, but processor memory requests frequency is much higher than memory access latency. This speed gap, called *memory wall*, has been increasing during the last thirty years (Figure 1.1). With the aim of downing the memory wall and getting closer to present the programmer unlimited and fast memory they would want, a hierarchy of cache memories brings profitable data closer to the cores. A cache memory bases its effectiveness on two behaviors that programs show when run in the computer; programs tend to use information (either data or instructions) that was recently used (temporal locality) and is close to other recently used data (spatial locality).

Given each core has to access to the cache nearly every cycle, each core has a first level of cache memories (L1) for data and instructions

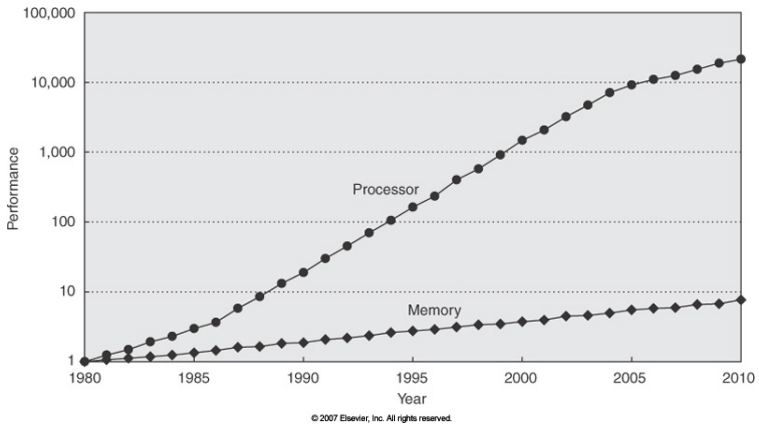


Figure 1.1: The memory wall. This graph shows the evolution of processor and memory performance from 1990 to 2010. The difference between both has always continuously increased. (Note the logarithmic scale)

that are designed small and fast. It is infrequent that more than one core shares the same L1. Misses in the first level will provoke requests to the following level in the hierarchy. Provided that there still exists a big difference between access latencies to L1 and main memory, additional intermediate levels of cache memories are usually installed to mitigate that latency gap. Cache memories are bigger and slower as we move away from the core. Nowadays, commercial processors usually have two or three levels of cache [20]. The aim is to minimize the average access latency with the smallest possible cost. The Last-level cache (LLC) represents a critical component of this hierarchy of caches. Since it is the last storage inside the chip, an LLC miss would provoke an access to the next component, main memory. To access to main memory supposes to exit from the chip, to pass trough the memory controller and to access to the DRAM chips; and all these steps are expensive in latency. Therefore, small improvements in the LLC hit ratio will lead to big improvements in the system performance.

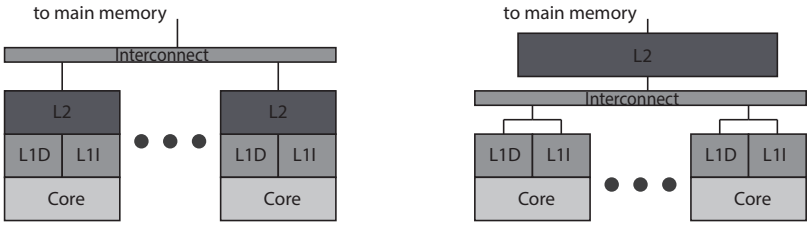
Next, we discuss some aspects about the LLC.

1.1.1 *Last-level cache organization*

The Last-level cache (LLC) can be designed as a collection of private caches (Figure 1.2a) or as a space which is shared among all the cores (Figure 1.2b). Following we will briefly show the pros and cons of both designs.

PRIVATE In a private organization of the Last-level cache, each core has a storage for its own use (Figure 1.2a). The total size of the LLC is equally divided among all the cores in the system. As the size of each LLC is smaller than in a shared design, their average access latency is also shorter. Given only one core accesses each LLC, interference effects between cores are not found.

On the other hand, shared data is replicated in the LLC of each sharer of that cache line, reducing the total effective available space. Moreover, applications running in the CMP may require very different cache sizes; as in this design each core has associated a fixed LLC size, the LLC can not be adapted to the requirements of each application. In addition, a mechanism to maintain coherence in this organization will suffer higher latencies than the same mechanism in a shared design.



(a) Private (b) Shared

Figure 1.2: Last-level cache organizations

SHARED In a shared organization of the Last-level cache (SLLC), all the cores share the complete LLC available storage (Figure 1.2b). The size of the SLLC is bigger than in the private design, so the access latency is also larger. Given the LLC storage is shared among all the cores, the activity of the applications running in the system may harm each other.

On the other hand, only one copy of the data used by more than one core is stored in the SLLC, maximizing the effective SLLC available storage. In addition, a SLLC can suffice at the same time the requirements of two very different applications.

SLLC are internally split in banks which are accessed through one port. This organization provides at the same time layout flexibility and increased access bandwidth.

This thesis follows the general trend in both industry and academy, choosing a shared design (SLLC) as the LLC organization where contributions are proposed and evaluated.

1.1.2 *Inclusivity*

A hierarchy of cache memories (L1-L2) is said to be **inclusive** when the L2 stores the contents of all the L1s. In other words, the L2 contents are always a superset of the L1 caches contents [3]. In order to maintain this property, two actions have to be consistently performed:

1. When a cache line is inserted in some L1, it must be also inserted in the L2 if it is not present.

2. When a L2 line is evicted, invalidation messages are sent to all the L1 copies, if any.

An inclusive hierarchy simplifies coherence maintenance [42]. As the SLLC knows about the contents of the cache levels closer to the processor, a mechanism to maintain the coherence installed at this level is able to take decisions without sending look-up messages to the local caches. Many commercial CMPs present an inclusive hierarchy [20].

A hierarchy of cache memories (L1-L2) is said to be **exclusive** when the contents of the L2 and the L1s are always disjoint sets. Meaning that when a cache line is inserted in some L1, it must be removed from the L2 if it is present. In this type of hierarchy, two basic schemes can be considered to maintain the system coherent. This information can be present at the SLLC as an updated copy of the L1s tags, or be obtained by sending look-up messages to the L1s every time some situation requires it.

When neither inclusion nor exclusion are enforced, the hierarchy is said to be **non-inclusive/non-exclusive**. There exist a broad range of options to define this intermediate scheme [69, 53].

This thesis considers an inclusive SLLC organization as the baseline design where contributions are proposed and evaluated.

1.1.3 *Hardware Data Prefetching*

Hardware prefetching is a technique that tries to load into the cache, contents the processor will use in the future. The decision on what to prefetch has relayed on plenty of prediction schemes, from simple to very sophisticated ones. All of those schemes revolve around a property programs show, called *spatial locality*. This property says that a program will reference in the future, memory addresses that are close to memory addresses it referenced recently in the past. In other words, programs are kind of predictable in their stream of references accessing to memory. The hardware prefetcher installed at some cache level observes the stream of references arriving to that cache with the intention of generating prefetches that bring data into the cache which will be referenced in the near future.

A basic type of prefetcher trying to exploit spatial locality is the *stride* prefetcher. This prefetcher calculates the difference between the addresses of the cache lines referenced by two consecutive misses and it generates a prefetch request based on such difference, e. g. misses are observed over the cache lines corresponding to the ad-

dresses *A* and *A+m*, thus the stride prefetcher will generate a request for the address *A+2m*. The basic case of stride is when m is equal to 1, a prefetcher considering always this kind of pattern is called *sequential prefetcher*. A parameter associated to the prefetcher is its *aggressiveness*. The aggressiveness of a prefetcher defines how much (*degree*), and also, depending on the type of prefetcher, how far (*distance*) the prefetcher will follow its prediction in order to bring data to the cache. E.g. on the stride prefetcher, the degree defines how many cache lines will be brought into the cache per triggering event, for the example before, if we define degree as equal 3, consecutive misses to *A* and *A+m* will generate prefetches of *A+2m*, *A+3m*, and *A+4m*.

Aggressiveness of a prefetcher should adapt to application and memory system characteristics. Adaptive mechanisms to dynamically adjust the prefetching aggressiveness has been previously proposed and are presented in Section 2.2. If a CMP system with a SLLC is considered new tradeoffs, which are presented in the next Section, appear.

1.1.4 *Replacement Policy*

When a miss is observed by the cache and there is no free cache entries, one cache line has to be selected as victim. The replacement policy decides which is the cache line that will be evicted to make place for another incoming. Three basic schemes of replacement traditionally considered are: 1) Random. 2) FIFO. 3) LRU.

RANDOM The random replacement policy, as its name indicates, randomly chooses a victim among the elements present in the set. Its implementations normally uses a pseudo-random periodical function. It does not need any additional storage.

FIRST-IN-FIRST-OUT The *First-In-First-Out (FIFO)* replacement policy orders the elements of a set following their arrival order. Only a pointer is needed to implement this policy, the pointer has to indicate the element entered the furthest in the past. The storage needed to implement this policy is $\log_2$(associativity) bits per set. The lifespan of a line in a *N-way* associative cache is exactly the time for the set to observe N misses. A spread variant of the FIFO policy is the CLOCK algorithm. It was originally proposed for the management of pages in a virtual memory

management system [12]. It has been used in caches or buffers with a high associativity. It uses one bit per cache line to record if the element has been referenced during its stay in the structure or not. When an element has to be evicted, the element entered the first one is selected as victim, if its bit is set, the bit will be reset and the next element will be considered as possible victim.

LEAST-RECENTLY USED The *least-recently used (LRU)* replacement algorithm is oriented to exploit a property called *temporal locality* that programs show when running. Temporal locality assumes that a recently used cache line will be used again in the near future. All the elements in a cache set conform a chain and are ordered following the use order; the most recently referenced elements are at the beginning of the chain, while the end of the chain is occupied by the least recently referenced element. On a miss, a cache line has to be chosen as victim, the last one is selected. Every cache line increases its position by one and the new line is inserted at the beginning of the chain. On a hit, the hit line is situated at the beginning of the chain while all the lines that were occupying positions over the hit line see decremented their position by one.

There is a plenty of options to implement the LRU policy, each one representing different tradeoffs in terms of storage cost and logic complexity. Sudarshan *et al.* made a review of different implementations in [60].

True LRU can be implemented with low cost when small associativities are considered, e.g. to implement LRU in a 4-way associative cache is 8 bits per set. But when higher associativities are considered (16- 32-, or 48-way associative caches), as the ones we can find in a SLLC, true LRU gives way to alternative policies that try to mimic LRU performance with a lower hardware cost. These policies are called *pseudo-LRU* policies. There is a broad variety of this kind of policies and what they have in common is the use of partial information to maintain an approximation to the LRU order inside the set. Normally, all of them try to avoid to evict the most-recently used element while they keep some kind of order among the rest of the elements of a set. An example of pseudo-LRU policy is *Not-recently used (NRU)* [43]. As Figure 1.3 shows, this policy employs one bit for each cache line (NRU-bit), the bit is zero when the line is inserted into the set. All the NRU-bits but the one corresponding

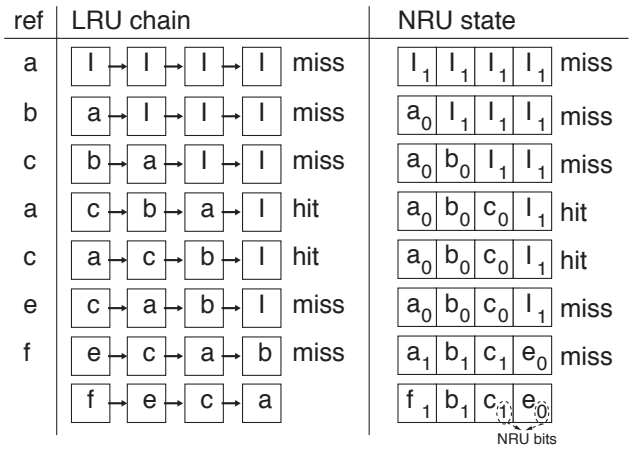


Figure 1.3: Example of LRU and NRU replacement policies

to the just inserted line will be reset in case of all the NRU-bits are equal to zero. When a victim has to be selected, the first cache line with the NRU-bit equal to one is evicted. The search of the line with NRU-bit equal to one can be started from the position indicated by a global pointer (Sun T2) or from the way 0 (intel i7).

In Figure 1.3, we can observe an example of the behavior of LRU and NRU replacement policies. Each line represents the state of the set before the reference that appears on the left column accesses the cache. For LRU, elements are ordered forming a chain, while for NRU each element is at its corresponding way and the NRU bit appears along the address. Next to state, for both algorithms, we can observe if a determined reference is a hit or a miss. Consecutive lines of the figure show consecutive states of the set; the second line of any pair shows the state of the set after the reference (*"ref"* on the figure) of the first line makes its access effective.

Section 2.3 presents recent research on replacement algorithms for the SLLC. These works have shown replacement policies for the SLLC should not be based on temporal locality and the next section will expose the problem.

1.2 PROBLEM

Every SLLC miss provokes an access to the main memory. The main memory is out of the chip and made of DRAM memory chips, adding a long latency to every access that misses in the SLLC and penalizing the average memory access time. Thus, the SLLC hit ratio is a critical factor to achieve an optimal average memory hierarchy access latency. Technological innovations aside, two aspects that can be tackled to improve the SLLC hit ratio are the efficiency of both prefetching and replacement policy.

1.2.1 *Hardware Data Prefetching*

Hardware prefetching tries to load data into the cache time ahead of the processor references it. This technique has been broadly shown as good to reduce the average access memory latency. Prefetching performs specially well in mono-processor memory hierarchies where only one stream of data flows from main memory to the caches closer to the core. However, when prefetching is used in the SLLC of a multicore system where different applications are running at the same time, prefetches associated to one core may interfere with the data placed into the cache by other core, evicting contents of other application and harming its performance. A control mechanism to regulate the prefetching aggressiveness associated to each core is desired. This mechanism should target the overall system performance. In section 2.2, we feature previous work on hardware data prefetching aggressiveness adjustment for mono- and multi-processor systems, and in Section 4.2 we explain in detail the aforementioned inter-core prefetching interference problem.

1.2.2 *Replacement policy*

The replacement policy critically influences the cache memory hit ratio. In a CMP fitted with a hierarchy of cache memories, temporal locality is squeezed by the cache levels closer to the core. Thus, many of the lines inserted in the SLLC are single use, meaning that they will not experiment any hit during their lifespan at the SLLC. However, cache lines that experiment one hit in the SLLC are normally experimenting many hits. Therefore, to assume that the replacement

algorithm has to base its decisions on the temporal locality exploitation is no longer valid at the SLLC. On the contrary, this behavior indicates the SLLC replacement policy should be based on reuse instead of temporal locality. Section 5.2 goes over SLLC replacement problematics with more detail.

1.3 CONTRIBUTIONS

These are the main contributions of this thesis:

1.3.1 *The ABS controller*

A low-cost controller able to adjust the prefetching aggressiveness associated to each core in the CMP with the aim of improving the overall system performance. The controller runs stand-alone at each SLLC bank and gathers local metrics. Using a hill-climbing approach whose target function is the overall system performance, the ABS controller tries to infer the optimal combination of prefetching aggressiveness values for the applications that are running in the system.

1.3.2 *Reuse locality*

Observing the stream of references accessing to the SLLC, we state a property called *Reuse locality* that says that i) Cache lines used more than one time will be highly likely used many times in the future. ii) Cache lines recently reused are more useful than lines reused before. We claim that SLLC access pattern shows reuse locality.

1.3.3 *Replacement algorithms to exploit the reuse locality*

Two replacement algorithms able to exploit the reuse locality are proposed, *Least-recently reused (LRR)* and *Not-recently reused (NRR)*. These algorithms are modifications of two very well known algorithms based on the use, to be based on the reuse. The base algorithms are *Least-recently used (LRU)* and *Not-recently used (NRU)*, our transformations do not add any hardware to them. We will show that

these two new algorithms consistently improve the performance of their predecessors.

1.3.4 *Reuse cache*

A novel design for the SLLC of a CMP is proposed. On this design tag and data arrays are decoupled. Only those cache lines that have shown reuse are stored in the data array. The tag array is used to detect reuse and maintain coherence. This scheme allows to drastically reduce the size of the data array. As an example, a reuse cache with a tag array equivalent to a conventional 4MB cache and a data array of 1MB achieves the same average performance that a conventional 8MB cache

1.4 THESIS ORGANIZATION

This dissertation is organized in four parts. The first part contains Chapters 1, 2, and 3. These chapters introduce the topics of the dissertation, present background and previous works, and discuss the experimental framework followed during this thesis. Second part includes only Chapter 4 that explains our contribution to adjust prefetching aggressiveness. Third part comprises Chapters 5 and 6, which explains our contributions to improve the SLLC efficiency by the better exploitation of reuse locality. Finally, fourth part concludes this dissertation.

CHAPTER 2 presents previous works related with the contributions presented on this thesis. Related work includes research regarding the exploitation of reuse locality, works where decoupled structures are proposed or used, prefetch engines with the aim of improving CMP performance, and in mechanisms to adjust the prefetching aggressiveness in mono- and multi-processors.

CHAPTER 3 explains the experimental setup that has been used to evaluate our proposals. Including the simulator characteristics, the baseline system we consider, and the workloads whose behavior have been analyzed during this thesis.

CHAPTER 4 presents the ABS controller and its evaluation. The evaluation includes the use of particular metrics that will be justified. A comparison with the state of the art is also presented.

CHAPTER 5 states the reuse locality and presents two replacement policies that exploit such property. This chapter also shows the evaluation of our policies and their comparison with both base algorithms and state of art.

CHAPTER 6 presents the reuse cache and its rationale and design are explained. That chapter also shows a broad evaluation from different design points and a comparison with the state of the art. The chapter finishes suggesting further improvements or future lines for the reuse cache.

CHAPTER 7 concludes the dissertation summarizing the work done and discusses about possible future research lines.

2

RELATED WORK

SUMMARY

This chapter about the related work comprises four different sections. First two sections of this chapter are related with our contribution to control the prefetching aggressiveness in a CMP. Concretely, Section 2.1 summarizes previous works about prefetch engines, focusing only on research which targets multiprocessor systems. And Section 2.2 revises the previous work on adjustment of the prefetching aggressiveness in mono and multiprocessor systems. Section 2.3 is about the reuse locality and also about how different insertion and replacement schemes for the SLLC have tried to exploit such property, the material exposed on this section is related with three of the contributions of this thesis, namely, the reuse locality, the policies to exploit it and the reuse cache. Finally, Section 2.4 shows a variety of works that proposed alternative organizations for the cache memory related with our reuse cache.

2.1 PREFETCH ENGINES FOR MULTIPROCESSOR SYSTEMS

The development of prefetching engines able to interpret memory access patterns has been a broadly studied field of research. In this section, we only focus on those proposals with the aim of improving the behavior of the memory hierarchy of multiprocessor systems.

Cantin et al. [6] and Wallin and Hagersten [63] aim to identify private memory regions not shared by the other processors, starting to prefetch only in these memory regions in order to avoid that shared data prefetching may hurt performance. Koppelman [29] uses the instruction history to compute an area around the demanded data, which can be prefetched. Somogyi et al. [56] predicts memory accesses that exhibit a repetitive layout (spatial streaming); it proposes a predictor that correlates the memory access patterns with instructions addresses. Wenisch et al. [64] proposes temporal streaming, which is based on the observation that recent sequences of shared data accesses often recur in the same precise order; therefore it proposes to move data to a sharer in advance of its demand. Somogyi et al. [57] leverages the ideas of the two previous works and proposes a predictor that exploits both temporal and spatial correlations. How to store meta-data off chip for an address-correlating prefetcher has been also evaluated [65]. The concern of that work is how to store the large amount of information that requires their prefetcher to get good results in commercial applications. In contrast, the ABS controller that is proposed in this dissertation is not proposing a new prefetch engine but a control system to set the aggressiveness of a prefetcher installed in the banked SLLC of a Chip Multiprocessor.

2.2 MECHANISMS TO ADJUST PREFETCH AGGRESSIVENESS

In mono-processors, *Adaptive Data Cache Prefetcher (AC/DC)* divides the memory address space into zones of the same size called *Czones*, and it uses global history buffer to track and detect patterns in consecutive miss addresses within each Czone. It is also able to dynamically adjust the size of the Czones and the prefetcher degree. Ramos *et al.* introduce an adaptive policy that selects the best prefetching degree within a fixed set of values, by tracking the performance gradient and following a hill-climbing approach are able to approximate to the prefetcher configuration optimal value [49]. Srinath *et al.* proposed *Feedback directed prefetching (FDP)* [58]; a mechanism that incorpo-

rates dynamic feedback to increase the performance improvement provided by prefetching. The mechanism estimates prefetching accuracy, timeliness, and cache pollution and, depending on a predefined set of thresholds, it adjusts the aggressiveness of the data prefetcher dynamically. FDP also incorporates a mechanism to decide in which position of the LRU stack it inserts depending on the pollution the prefetcher is provoking.

Several works address the problem of adjusting prefetch aggressiveness in CC-NUMA multiprocessos having only private cache memories. Dahlgren *et al.* suggest to determine the prefetch aggressiveness at each private cache by counting the number of useful prefetches every given number of issued prefetches (an epoch) [14]; the prefetch aggressiveness is increased or decreased taking into account two usefulness thresholds. Tcheun *et al.* add a degree selector to a sequential prefetch scheme [61]; when the selector detects useful prefetches along a sequential sub-stream it increases the prefetch aggressiveness of the next sub-stream belonging to the same stream. As these works are not using a shared cache, the interference problems among cores that they found are only related with the available bandwidth when accessing to memory.

To our knowledge, only *Hierarchical Prefetcher Aggressiveness Control (HPAC)* Ebrahimi et al. [15] faced for the first time the problem of reducing the prefetch inter-core interference in a chip multiprocessor with a shared LLC. HPAC monitors several global indexes (prefetch accuracy, inter-core pollution, and memory controller activities) and compares them to a predefined set of thresholds. That comparisons are contrasted against a set of rules to finally adjust the prefetch aggressiveness of each core. In Chapter 4, we have included a deep discussion about HPAC characteristics, and also a comparison in terms of performance and resource consumption with respect to a system using ABS controllers.

2.3 REUSE LOCALITY

Reuse locality says that i) Cache lines used more than one time will be highly likely used many times in the future. ii) Cache lines recently reused are more useful than lines reused before. This property was first observed and exploited in cache memories for disks. Segmented LRU [24] 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. The boundary between the lists is fixed and victims are selected in order to preserve that limit.

Recent proposals have applied this idea to the replacement policy of non-inclusive SLLCs. Both dynamic segmentation [26] and dueling segmented LRU [17] consider these two logical LRU divisions and try to dynamically find an optimal configuration by using set dueling. The former uses a set dueling predictor and a decision tree to dynamically move the border between the reused and non-reused segments. Another level of set dueling chooses between dynamic segmentation and plain LRU. Whenever the size of the reused segment becomes the smallest (one line), bypass is switched on. In spite of the cache being bypassed, one of every thirty-two lines is stored in cache in order to prevent the working set from becoming stale.

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

Recently, *MRU-Tour* based algorithms also propose using reuse to divide the elements of a set into different groups and randomly evict elements with a number of MRU-Tours lower than a given value [62].

2.3.1 *Insertion policy*

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

The dynamic insertion policy (DIP) involves a hybrid cache replacement [47]. Under this scheme, LRU and Bimodal insertion Policy (BIP) are dynamically selected by using set-dueling. LRU always inserts the incoming lines in the MRU position of the recency stack. BIP inserts the majority of incoming lines in the LRU position and with a low probability in the MRU position.

Promotion/Insertion Pseudo-Partitioning (PIPP) allows each thread to insert their lines at a different point in the recency stack

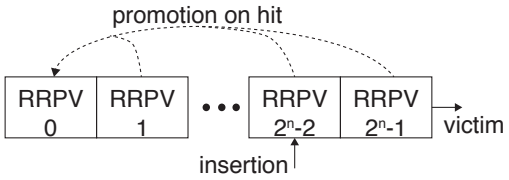


Figure 2.1: Static re-reference interval prediction (SRRIP) replacement algorithm

[68]. It provides the benefits of cache partitioning, adaptive insertion and capacity stealing among threads. However, it requires a utility monitor with shadow tags to detect thread's behavior.

Although industry has widely adopted inclusive SLLC schemes in commercial processors, specific content management for this kind of cache organization has not received much attention from academia. An exception is Re-Reference Interval Prediction (RRIP), that has been proposed for an inclusive hierarchy [23].

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 value (RRPV)* [23] (Figure 2.1). The mechanism is able to insert new lines into the segments corresponding to re-reference intervals either *intermediate* (segment with a RRPV equal to 2^N-2) or *long* (segment with a RRPV equal to 2^N-1), becoming in this way *scan*-resistant and *Thrash*-resistant. A more detailed explanation of this algorithm can be found in Section 5.3.

Recently, a plethora of proposals try to infer reuse properties of cache contents (present and incoming) by using predictors. Later on, these predictions are used to modify the insertion policy.

Signature-based hit predictor (SHip) improves RRIP with re-reference interval prediction [67]. SHiP correlates re-reference behavior with memory region, program counter, and instruction sequence history. SHiP and other work, like ones proposed by Chaudhuri et al. [7], Li et al. [34], Seshadri et al. [51] are complementary to the reuse cache design. For instance, the predictors proposed in [51, 67] 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 [34] signals the first line to be reused between the incoming-victim line pair involved on 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. [7] propose to track the reuse behavior within the private caches and utilize it to estimate reuse in the SLLC when the lines are evicted from private caches . 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.

2.3.2 *Frequency-based replacement*

Frequency-based replacement algorithms classify according to the number of times cache lines have been accessed, giving more priority to the more accessed ones. The *least frequently used (LFU)* approach relies on the access frequency of cache lines to attempt to avoid harmful patterns that may evict useful lines [33]. Although frequency-based replacement improves the performance of applications with frequent *scans*, it is not good if the workload exhibits temporal locality.

In a way, our SLLC replacement proposals are a very simple form of a frequency-based replacement since lines get classified in only two counts: either those accessed one time or more than one time during their stay in the SLLC. However, LRR and NRR also work correctly if the workload exhibits temporal locality because they use recency within each segment.

2.3.3 *Dead-block prediction*

Dead-block prediction tries to identify dead lines by means of hardware predictors [27, 31, 36]. For instance, prediction relies on recording the instruction sequences performing the last touch of a given line, or counting the number of accesses performed on a line before it gets replaced. When a line experiences again some of those past behaviors, it is tagged as dead and becomes a replacement candidate. Dead-block prediction requires expensive hardware to predict and store metainformation. Khan et al. [27] introduce sampling dead block prediction, a technique aimed to reduce the storage of metainformation of the predictor. This mechanism samples program counters (PCs) to determine when a cache block is likely to be dead. Rather than learning from accesses and evictions from every set in the cache, a sampling predictor tracks a small number of sets using partial tags.

Lai et al. [31] proposed Block Predictors, predictors that accurately identify if a cache line is dead. The mechanism evicts those cache lines detected as dead and prefetchs data into them. Instead of using PCs to predict wether a cache line is dead, Liu et al. [36] predict dead blocks based on bursts of accesses to a cache block. A cache burst begins when a block becomes MRU and ends when it becomes non-MRU. The authors claim that cache bursts are more predictable than individual references because they hide the irregularity of individual references.

The underlying behavior of lines assumed in dead-block prediction and reuse-based replacement is the same: a big fraction of cache lines is dead at any moment. However, the opportunity that shows up is exploited in different ways.

Our mechanisms classify a priori all lines entering the SLLC as dead (since we realize most of them are indeed touched once). A line becomes alive once it is referenced a second time during its stay in the SLLC.

2.3.4 *Replacement on inclusive hierarchies*

In inclusive hierarchies, the core caches absorb most of the temporal locality and the hot lines may lose positions in the LRU stack of the SLLC, up to the point of being evicted. A recent paper shows ways to solve this problem by identifying lines in the core caches and preventing their replacement in the SLLC [22]. Three ways are proposed: sending hints to the SLLC about the core accesses (*TLH*), identifying temporal locality by early invalidation of lines in the core caches (*ECI*), or querying the core caches about the presence of the victim lines (*QBS*).

We also address this problem by using the information present in the coherence directory, assuming non-silent eviction of clean blocks in the private caches. However, our main contribution is the design of two replacement algorithms that exploit reuse locality with the same hardware cost than those used in commercial processors.

2.4 HOW TO DECOUPLE THE SLLC TAG AND DATA ARRAYS?

In recent years, there have been many innovations for improving the performance of the SLLC. In spite of this huge quantity of research, most of innovations only achieve to improve system performance within 5%. One of the contributions of this thesis, the reuse cache, targets the SLLC efficiency in a different way, proposing to downsize the SLLC data array while maintaining the average performance. In order to achieve that objective, our proposal relays on a decoupled tag and data arrays. Next we show works used similar types of decoupled designs.

Using pointer indirection appears as a natural solution when tag/-data decoupling is required. Several authors have used this idea for different purposes. Regarding sectored caches, a common idea is to share a number of data subsectors among a set of tag sectors, instead of the conventional 1:1 mapping between a tag sector and its data subsectors. Seznec suggests decoupling to conciliate a low tag implementation cost with a low miss ratio [52], while the decoupling proposed by Rothman et al. aims to reduce cache space requirements [50].

Chishti et al. propose the NuRAPID cache, that decouples tag lookup and data placement in order to reduce the average access latency in dynamic non-uniform cache architectures [9].

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 of a single-processor system [48]. 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. However, V-way relies on additional tag space to reduce conflicts in the set associative tag array. In the data array, V-way requires a global replacement policy based on use frequency. In contrast, the reuse cache objective is to reduce the cache size while keeping performance and inclusion benefits. It stores more tags than data lines and tries to only retain lines showing reuse. The additional tag space in the reuse cache allows to maintain inclusion and track reuse. Furthermore, its data array does not require a global replacement algorithm.

Two recent works are directly related with our reuse cache contribution [37, 69]. The cache organization relies in tag/data decoupling

to retain tags inclusion property and to use a selective allocation policy of lines in a cache miss.

NCID uses additional tags in order to maintain tag inclusion of the private caches, though the data lines of those private caches are not necessarily present in the data array. So, NCID allows SLLC data to be non-inclusive or exclusive while retaining the inclusion property on tags. Moreover, many NCID architectural options are presented and evaluated. One of them uses NCID to support a selective allocation policy to address transient data. Selective allocation 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 fill or selective fill policies.

Instead of random selection, the reuse cache data array selects lines with potential for reuse. In the reuse cache, tags without allocated data are indeed used to maintain inclusion, but also to detect reuse. Once a line experiences such a reuse it is written in the data array. Both, the replacement algorithms of tag and data arrays are designed with the objective of identifying and prioritizing reused lines. Set dueling is not needed.

Lodde et al. [37] 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 to move tags 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 required data array size. Our proposal relies in reuse locality detection, independently of which private cache requests the line. Thus shared lines are implicitly allocated in the data array and private lines are only allocated if reuse locality is detected.

3

EXPERIMENTAL FRAMEWORK

SUMMARY

In order to evaluate each of the contributions of this thesis, the associated mechanisms were faithfully modeled and these models incorporated to a simulator. The simulator was then used to run a set of benchmarks and the results of that simulations were employed to infer the validity of the hypotheses. The experimental framework is explained on this chapter: baseline system, simulator, workloads, and metrics.

3.1 INTRODUCTION

This chapter explains the experimental framework used during this thesis. Here it is shown the common part of this experimental framework, meaning that in the evaluation of some of our contributions the configuration of the considered system varied, workloads were built in an alternative way, or the metrics used to evaluate our proposals were different or enlarged. In such points, this document explains the corresponding changes.

The remaining of this chapter is organized as follows. Section 3.2 shows the baseline CMP system has been considered along this thesis. Section 3.3 presents the *Simics* full-system simulator and *ruby*, a plugin for Simics by the University of Wisconsin Madison which allowed us to model the memory hierarchy of our system. And finally Section 3.4 discusses about the types of workloads used in this thesis and the different metrics used to evaluate them.

3.2 BASELINE SYSTEM

Figure 3.1 shows the CMP system we have used during this thesis. It comprises eight in-order SPARC V9 cores and a hierarchy of memory composed by three levels of cache. The first level is private to each core and it is composed by split instruction and data cache memories. The second level is private and unified (data and instructions). And last, the third level is shared among all the cores in the system. All the three cache levels are write-back write-allocate. Table 3.1 shows remarkable parameters of the considered hierarchy.

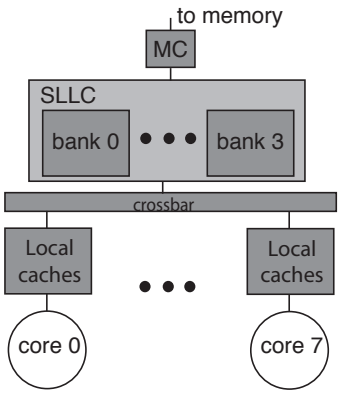


Figure 3.1: Baseline system overview

The third is the last level of our hierarchy of cache memories. We consider an 8 MB LLC, split in four banks which are shared among all the cores in the CMP. The banks are cache line interleaved to facilitate a homogeneous share of accesses, excepting on the ABS prefetching proposal, where they were interleaved using operating system page size to avoid inter-bank interference. The SLLC is accessible through a crossbar that connects the eight L2s to the four SLLC banks. Inclusion is enforced at the SLLC, meaning that a cache line is inserted at the SLLC when it is inserted into some L1 or L2, and when a cache line is evicted from the SLLC, it is also removed from any L1 or L2 that could have a copy.

An invalidation directory-based MOSI coherence protocol is implemented to maintain the private caches coherent. The directory is distributed among the SLLC banks, containing its information along with the tag of each cache line.

The memory system comprises 4GB of DDR3 DRAM per core which are accessed trough one memory channel. The memory runs at one quarter of the frequency of the processor.

3.3 SIMULATOR

In order to perform experimental evaluations of our proposals, we used the *Simics* simulation platform [40]. Simics provides functional models of all the devices included in a system (i/o, network, hard disks, etc...), allowing full-system simulation of multi-processor systems. At the same time it is able to simulate a wide range of cpus

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

Table 3.1: Baseline system configuration

at the instruction set level, e.g. ARM, SPARC, x86, MIPS, and Alpha. Simics run with hypervisor privileges, allowing to execute unmodified real operating systems; during this thesis, the evaluated systems were managed by the *Solaris 10.0* operating system.

To accurately simulate the memory hierarchy of a CMP model we used the *ruby* plugin from the *GEMS Multifacet toolset* [41], publicly available thanks to the University of Wisconsin. This plugin is loaded into the Simics simulator to capture all the memory access operations, modeling their latency with cycle accuracy. Ruby provides faithful models of all the memory hierarchy components from the cache controllers to the switches of the Network-on-chip interconnect. Buffering and blocking of components due to lack of resources are always considered.

During this thesis many of the components of ruby have been modified or enlarged to faithfully model each one of our proposals; including a model of the memory controller that implements the DRAM DDR3 protocol. During the implementation of our ABS controller this model was fundamental in order to accurately model the overhead that prefetching provoked on the memory hierarchy. The model reflected the status of every page of memory, queues, bus occupancy, and the scheduling policy, to observe how they were affecting each memory operation. Other modifications we included in GEMS are clarified and explained at the methodological part of each chapter of this dissertation.

3.4 WORKLOADS AND PERFORMANCE EVALUATION

In order to evaluate how our proposals affect the behavior of a wide range of CMP systems, two different types of workloads have been set up. A first group of workloads is composed by multiprogrammed workloads made of sequential applications from the *SPEC CPU 2006* suite of benchmarks. A second group of workloads is composed by parallel applications from the suites *PARSEC* [5] and *SPLASH-2* [66].

3.4.1 *Multiprogrammed*

In a multiprogrammed workload, as many sequential applications as cores the system has, are running at same time. This kind of work-

loads represents scenarios we can find from our desktop, where a web browser is running at the same time that a video player, to a server, where applications of very different nature run at the same time.

The employed multiprogrammed workloads are composed by applications from among all the 29 included in the SPEC CPU 2006 benchmark suite. Applications were firstly run until completion in a native machine. Their initialization phases where identified by using hardware counters. Groups of eight applications were randomly chosen to form workload mixes. Once in simics, the applications forming a workload mix were executed together, binding each application to a core in order to avoid migration effects. A number of instructions as long as the longest initialization phase of the applications composing the workload mix was fast-forwarded. At that point a checkpoint was taken. The average number of MPKI each application shows at the moment of taking the checkpoint at each cache level of the hierarchy is shown in Table 3.2, it is the average for all the mixes where each application appears.

In order to evaluate a microarchitectural proposal in a given workload mix, its corresponding checkpoint is read from Simics. Then, the ruby module is loaded with the configuration which mimics the pro-

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

Table 3.2: Average MPKI at each cache level

posal to study. The whole hierarchy is then warmed, normally during 300M cycles, after that, statistics are cleaned and performance accounting starts. The duration of the simulation is given in system cycles, during this thesis we have normally used 700M cycles long simulations.

The metric considered to evaluate the behavior of the different proposals when multiprogrammed workloads are employed is the geometric mean of the speedup of each application appearing in the workload mix. Given that duration of simulations is fixed and equal for all the proposals, what is employed in order to calculate speedups are the number of executed instructions. Equation 3.1 formally shows this metric.

$$\text{Speedup} = \sqrt[n]{\prod_{i=1}^{n} \frac{I_i^A}{I_i^{base}}} \tag{3.1}$$

IPC_i^A: Number of executed instructions of program i when run in system A
IPC_i^{base}: Number of executed instructions of program i when run in baseline system

3.4.2 *Parallel*

The parallel applications evaluated in this thesis are five applications of *SPLASH-2* [66] and *PARSEC* [5] suites. The first one is a very well known benchmark suite of scientific parallel programs, while the second focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors.

Table 3.3 gathers the applications we selected for our evaluation. They are those applications from both suites having more than 1 MPKI in a 8-MB SLLC. Each application spawns in its parallel phase as many threads as processors are in the system. We utilize *simmedium* input set for PARSEC applications and a 1026x1026 grid for *Ocean*. Performance statistics are only taken in the parallel phases.

Parallel applications should be run until completion when meaningful results are desired, or at least some work-oriented metric should be considered [1]. In our simulations we have observed that no

	canneal	facesim	ferret	vips	ocean
MPKI	4.48	3.45	1.27	1.74	13.35

Table 3.3: MPKI of the selected parallel applications in the LLC of the baseline system

OS activity appeared when parallel applications were run and moreover the ratio of load of work among the different threads was practically constant between simulations, thus sampling seems to be a reasonable option to evaluate this type of applications. Thus, as only one application has been run until completion (OCEAN) and because of the huge simulation time would be needed to simulate them completely, the rest of the applications were evaluated using a technique similar to the one used for multiprogrammed workloads.

In order to evaluate a microarchitectural proposal in an application can not be run until completion, the corresponding checkpoint and ruby configurations are loaded and the whole hierarchy is warmed during 300M cycles. After that, statistics are reset and performance accounting is performed during 700M cycles.

Processor throughput is used as performance metric, meaning that given that duration is fixed and the same for all of our simulations, in order to compare different proposals, the total number of executed instructions is compared. Equation 3.2 formally shows this metric.

$$\text{Speedup} = \frac{\sum_{i=1}^{n} I_i^A}{\sum_{i=1}^{n} I_i^{base}} \tag{3.2}$$

IPC_i^A: Number of executed instructions of core i in system A
IPC_i^{base}: Number of executed instructions of core i in baseline system

Part II

ADJUSTING PREFETCHING AGGRESSIVENESS

This second part of the thesis comprises only one chapter. Chapter 4 contains our contribution *Low-Cost Adaptive Controller for Prefetching in a Banked SLLC (ABS)*. The aim of this proposal is to increase the SLLC efficiency by adjusting the prefetching aggressiveness associated to each core of a CMP.

4

ABS PREFETCHING

SUMMARY

On a multicore system fitted with a shared Last-Level Cache, prefetch induced by a core consumes common resources like shared cache space and main memory bandwidth. The uncontrolled use of the SLLC space by prefetching could lead to a paradoxical situation where this technique becomes harmful for the system performance. In order to avoid such situation prefetching aggressiveness should be controlled from an overall system performance standpoint. Given it is usual to find a broad spectrum of applications running at the same time in modern CMPs, it seems natural to adjust the prefetching aggressiveness associated to each core independently.

This chapter presents ABS, a low-cost controller that runs stand-alone at each LLC bank without requiring inter-bank communication. Following a hill-climbing approach, the mechanism is able to adapt the prefetching aggressiveness associated to each core in the system gathering only bank-local metrics.

Using multiprogrammed SPEC2K6 workloads, our analysis shows that the mechanism improves both user-oriented metrics (Harmonic Mean of Speedups by 27% and Fairness by 11%) and system-oriented metrics (Weighted Speedup increases 22% and Memory Bandwidth Consumption decreases 14%) over an eight-core baseline system that uses aggressive sequential prefetch with a fixed degree. Similar conclusions can be drawn by varying the number of cores or the LLC size, when running parallel applications, or when other prefetch engines are controlled.

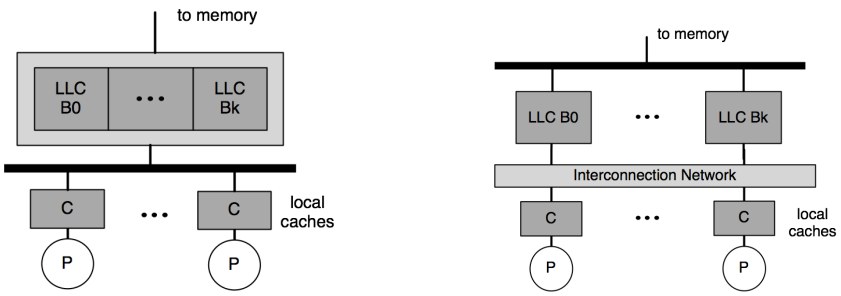
4.1 INTRODUCTION

Hardware data prefetch is a very well known technique for hiding the long latencies involved in off-chip accesses. It tries to predict memory addresses in advance, requesting them to the next level, and loading the lines into the cache before the actual demands take place. Several commercial multicore processors implement some form of hardware data prefetch [11, 32].

Prefetches may be initiated in the first-level caches or directly from events occurring in the SLLC, but in the end the prefetches reach the SLLC and interfere with each other. That is, prefetches issued on behalf of one core may evict LLC lines previously allocated by other cores, either by a memory instruction or a prefetch request. In addition, the prefetch activity originated from a single core can reduce the overall available bandwidth, potentially increasing the latency seen by the demands or prefetches coming from the rest of cores.

Most prefetch proposals in multiprocessors deal with systems having only private caches [6, 14, 29, 57, 61], while prefetch for shared caches has received little attention [15].

Ebrahimi et al. [15] tackle for the first time the problem of reducing the prefetch inter-core interference in a chip multiprocessor with a shared LLC. They propose the *Hierarchical Prefetcher Aggressiveness Control (HPAC)* mechanism, that monitors several global indexes (prefetch accuracy, inter-core pollution, and memory controller activities) to adjust the prefetch aggressiveness of each core. This assumes a centralized implementation of the LLC, internally organized in banks but with a single access port (see Figure 4.1a). Thus, the global aggressiveness control and associated hardware structures are also centralized.



(a) Monolothic (b) Banked

Figure 4.1: Last Level Cache organizations

To the best of our knowledge, our contribution is the first work where prefetch is studied in a multicore system fitted with a *banked* shared LLC, see Figure 4.1b. We assume an LLC organized in independent cache banks with an access port each, and an interconnection network attaching cores to cache banks (a crossbar is assumed, but other topologies can be considered) [28, 30]. Each bank is internally sub-banked in order to provide a higher throughput. This kind of SLLC is already mainstream because independent banks add layout flexibility and increase access bandwidth. Commercial processors from AMD, Sun, Intel, or IBM are using this design [11, 28, 30, 32].

In this scenario, we introduce the *ABS controller*, an Adaptive controller for prefetch in a Banked Shared LLC. ABS controllers are installed in all the LLC banks which are already fitted with a prefetch engine. Each ABS controller runs autonomously and gather local statistics to set the prefetch aggressiveness for each core in the bank it controls in order to maximize the *overall* system performance using a hill-climbing approach. Therefore, a given core is allowed to prefetch with different aggressiveness on different banks of the LLC.

Isolation between banks is a key factor of our proposal, meaning that both the ABS controller and the prefetcher in a bank are not influenced by their peers at other banks. Bank isolation achieves two essential benefits, namely *i)* the prefetches generated from a given bank target itself, and it will always be possible to filter useless prefetches by looking up in the bank, thus saving memory bandwidth, and *ii)* communicating prefetchers or ABS controllers among banks is not required, removing the need for a dedicated interconnection network or extra traffic in the existing one. As discussed in Section 4.5.2, bank isolation can be achieved by selecting a proper address interleaving among banks or by adjusting the prefetch distance.

Our results show that an eight-core system with ABS controllers running multiprogrammed SPEC2K6 workloads improves in both user-oriented metrics and system-oriented metrics over a baseline system with a fixed degree sequential prefetch. The results are consistent when varying the number of cores or LLC sizes. ABS control can be applied to other prefetch engines as long as they are able to operate at different aggressiveness levels. Specifically, we introduce ABS-controlled sequential streams. A comparison with HPAC-controlled sequential streams, such as that proposed by Ebrahimi et al. [15], shows higher performance at a very small fraction of the cost. Furthermore, when running multithreaded workloads from *SPLASH-2*

[66] and *PARSEC* [5], the ABS controllers also reduce the execution time and the consumed bandwidth over the baseline system.

The remaining of the chapter is structured as follows. Section 4.2 describes the motivation behind the work. Section 4.3 gives some background and reviews related work. Section 4.4 introduces the ABS controller. Section 4.5 presents the prefetch framework. Section 4.6 shows the methodological differences respect to what was presented in Chapter 3. Section 4.7 shows the results when ABS controllers are evaluated in a variety of situations, and Section 4.8 discusses and summarizes the contribution.

4.2 MOTIVATION

Figure 4.2 shows instructions per cycle (IPC) for eight SPEC2K6 applications running on a system with eight cores and a 4MB shared LLC. The simulation details are shown in Section 4.6. The figure shows four bars for each application. The first two bars represent programs running alone in the system, either without prefetch or with an aggressive (degree 16) sequential tagged prefetch (see Section 4.3). The last two bars represent the eight applications running together, either all without prefetch or all with the former aggressive prefetch turned on. When comparing the systems with prefetch (second and fourth bars), significant performance losses appear when resources are shared among cores. Note that prefetch involves virtually no performance loss in any application when running alone (first and second bars), while it causes losses in 5 out of 8 applications when running all together (third and fourth bars). Therefore, in order to boost the shared LLC performance by means of prefetch, a mechanism to control aggressiveness is called for. Such a mechanism should consider global metrics to realize when the prefetch activity of a core harms the overall system performance and it should be decreased in spite of the improvement achieved by that core.

As figure 4.2 highlights, the benefit obtained by an application due to prefetch can decrease or even turn into losses if prefetch is simultaneously active in all cores. Therefore, our goal is to design a mechanism that dynamically controls the prefetch aggressiveness of each core in order to maximize system performance.

The impact on system performance can be assessed using global indexes such as aggregated IPC or shared LLC miss ratio. Both are

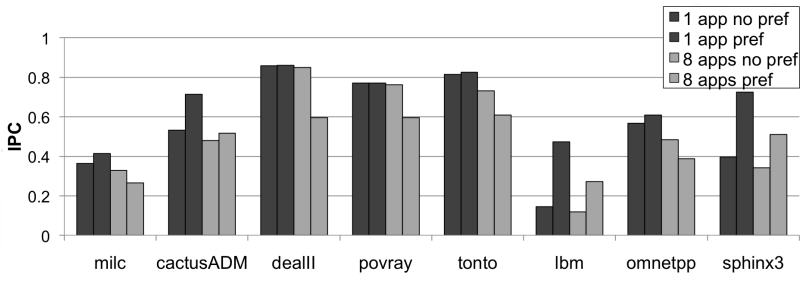


Figure 4.2: IPC for eight SPEC2K6 applications (mix2) running on an 8-core system with a shared LLC

obtained by adding quantities that are distributed in the cores or the cache banks, respectively. So, a centralized design of the prefetch aggressiveness control requires sending information from the places where events are counted to the centralized control point. Alternatively, we propose to place an ABS controller in each LLC bank. The controller uses bank-local information (i.e. bank miss ratio) to improve bank performance. Improving the performance of every bank will thus improve the system performance.

4.3 BACKGROUND AND RELATED WORK

This section provides with material will be useful in next sections. First, Section 4.3.1 explains the concept *prefetching aggressiveness* that will be used in next sections. And second, Section 4.3.2 presents the *Hierarchical Prefetcher Aggressiveness Control (HPAC)*, to the best of our knowledge, the only proposal in the literature with the aim of controlling the prefetching aggressiveness associated to each core in a CMP.

4.3.1 *Prefetch aggressiveness*

Prefetch aggressiveness is often defined in terms of degree and/or distance. Let us consider a stream of references a processor is going to demand $(a_i, a_{i+1}, a_{i+2}, ...)$, where address a_i has just been issued. A prefetcher can be designed to produce the next k addresses following a_i $(a_{i+1}, ...a_{i+k})$, calling k the prefetch degree. Alternatively, or in addition, it can also be designed to produce a single address of a far reference (a_{i+d}), calling d the prefetch distance. As an example, we recall the sequential tagged prefetcher with degree k [14]. If reference

a_i having the line address V is going to trigger a burst of prefetches (a_i misses or is the first use of a prefetched line), then the prefetcher will issue the following request burst $(V + 1, V + 2, ..., V + k)$. Another example of a prefetch engine using aggressiveness is the sequential streams as presented in [58]. In that work, aggressiveness was defined as a combination of distance and degree.

4.3.2 *Hierarchical Prefetcher Aggressiveness Control (HPAC)*

To our knowledge, only the Hierarchical Prefetcher Aggressiveness Control (HPAC) presented by Ebrahimi et al. [15] has faced the problem of adjusting prefetch aggressiveness on a shared LLC.

We notice four main differences between that work and the present one:

1. While HPAC resorts on computing the prefetching aggressiveness by means of a set of rules applied to several system variables (a kind of fuzzy controller), while ABS relies on a local search method (a variant of hill-climbing) to minimize a single system variable, the bank miss ratio.

2. HPAC was proposed for a centralized LLC with a single access port (see Figure 4.1a). We propose ABS for a cache organized in banks, each one with an access port (see Figure 4.1b).

3. HPAC throttles auto-regulated prefetch engines attached to each core. In the original paper, HPAC is evaluated using Feedback Directed Prefetching as the auto-regulated prefetch engine [58]. However, ABS controllers set directly the aggressiveness level of the local prefetchers.

4. HPAC uses four global metrics and FDP (as part of HPAC) uses three more local metrics. All these metrics are monitored and compared to ten thresholds (4 for HPAC + 6 for FDP). In contrast, ABS only samples two system variables and only considers one threshold.

A performance and complexity comparison between HPAC and ABS is presented in Section 4.7.4.

4.4 THE ABS CONTROLLER

An *ABS controller* is an adaptive mechanism that sets dynamically the aggressiveness associated to each core on the prefetcher installed in a bank of a banked shared LLC. Every LLC bank has an ABS controller commanding the prefetcher of that bank. Thus the ABS controller of an LLC bank is able to associate different levels of prefetch aggressiveness to each core, and conversely ABS controllers in different banks can associate to the same core different levels of prefetch aggressiveness.

ABS control relies on a hill-climbing approach for finding the minimum of a function (the miss ratio of a bank[1]) that we assume to be dependent on a set of variables namely, the prefetching aggressiveness of each core in the bank. Time is divided into regular intervals called *epochs*. In each epoch, in order to establish a cause-effect relationship between change in aggressiveness and change in performance, the aggressiveness of only one core (the probed core) is varied. The point is that at each epoch, the observed change in the bank miss ratio is only due to a single aggressiveness change. At the end of the epoch an aggressiveness value is established for the currently probed core and this value remains unchanged until it is probed again. Furthermore, ABS controllers force the prefetch aggressiveness associated to a core to be decreased if its accuracy falls under a given threshold. The operation of ABS controllers involves two aspects: *i)* selection of the core to probe and temporal sampling, and *ii)* adaptive per-core aggressiveness control.

CORE SELECTION AND TEMPORAL SAMPLING At the beginning of each epoch a core is chosen in a round-robin fashion[2] and its current prefetch aggressiveness is changed. Then, at the end of the epoch, the effect of the change is evaluated by comparing the bank miss ratios observed during the current epoch and a reference epoch (Figure 4.3a). The change is undone if the current bank miss ratio is greater than the reference one. Otherwise, the change is confirmed and the current epoch is set as the new reference. So, the core se-

1 Ratio of bank demand misses to demand requests coming from all cores. Other performance indexes were also tested as the target function, such as global miss ratio, MPKI, or IPC. Although results were similar, these other indexes were discarded because they are more expensive to compute in terms of communication and hardware cost.

2 A random order was also tested achieving slightly worse results.

lection and temporal sampling guarantee that there is always only one prefetch aggressiveness change between reference and current epochs. That change corresponds to the probed core at each epoch.

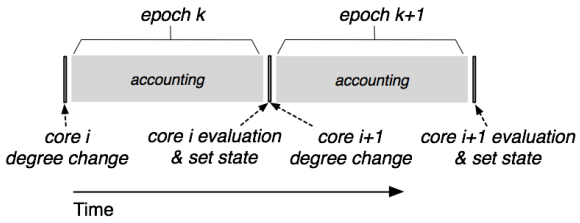
At the end of an epoch, an aggressiveness value is established for the currently probed core and this remains unchanged until the core is probed again. Note that if an application remains in a stable phase the ABS controller reaches a steady state only broken by the glitches involved in testing sub-optimal configurations. Hill-climbing processes usually deals with functions that are not time-dependent. Thus, the process stops when no change can be found to improve the value reached. However, we know miss ratio is time-dependent because applications change their behavior over time. This has two important implications for the design of ABS.

1. Our algorithm never stops. The combination of aggressiveness able to minimize the miss ratio changes over time and ABS continually seeks that combination.

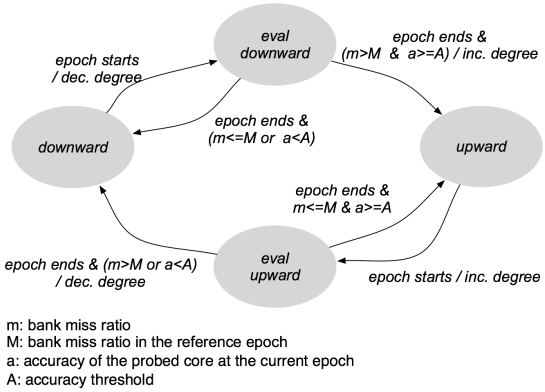
2. When the miss ratio reaches the global minimum in the corresponding program phase, ABS will set the current epoch as the reference epoch and the current miss ratio as the rate to beat. So, as a lower miss ratio will no longer appear, ABS will never change the control actions, and worse, a similar behavior may occur during long program phases after reaching a local minimum. In order to remedy this situation, the number of epochs elapsed without updating the reference epoch is counted. When this count is equal to the number of cores, the mechanism sets the last epoch as the new reference. Updating the reference epoch in this way ensures that a new value is taken after probing all cores without experiencing a miss ratio decrease.

We use epochs of 32K cycles. Other durations were tested without significant variation. Epochs based on counting a given number of events, like cache misses, were also tested without significant variations.

ADAPTIVE PER-CORE MISS-GRADIENT AGGRESSIVENESS CONTROL In an ABS controller, every core has a state which consists of a prefetch aggressiveness *degree* and a prefetch aggressiveness *trend* (downward or upward). At the beginning of the epoch in which a core is being probed, the state changes to eval-downward or eval-upward



(a) Core selection and temporal sampling



(b) Finite state machine controlling the per-core prefetch aggressiveness

Figure 4.3: ABS controller operations

(Figure 4.3b). Note that to probe a core, ABS only changes the aggressiveness following the trend associated with the core, instead of testing both possibilities (downward and upward) as they would do other implementations of hill-climbing.

Four events are locally counted in each LLC bank during each epoch, namely: 1) bank accesses from all cores, 2) misses from all cores, 3) prefetches issued by the core being probed, and 4) hits from the core being probed on prefetched lines. Sequential tagged prefetch uses a bit per cache line to tag the prefetched lines. This bit is set when a line is loaded in the LLC by a prefetch, and it is reset when the line is used for the first time. We use this bit to count the hits of the probed core on prefetched lines. At the end of an epoch two ratios are computed: bank miss ratio (bank misses / bank accesses) and prefetch accuracy for the core being probed (hits from the core in prefetched lines / prefetches from the core). Then the computed bank miss ratio is compared with the bank miss ratio of the reference epoch. If it has increased, the state changes to the reverse trend (from eval-downward to upward or from eval-upward to downward).

Otherwise, the state changes to the initial trend and the reference bank miss ratio is updated. Accuracy is involved in the transitions that leave the eval-x states. It is required that the probed core has an accuracy higher than a threshold to go to the upward state. The rationale of this requirement is to avoid the increase in the aggressiveness of one core whose prefetches are almost useless. We have observed that the optimal threshold depends on the intrinsic accuracy of the prefetch engine. Aggressive prefetch engines such as sequential tagged need a higher threshold (more restrictive) than other more conservative prefetch engines like sequential streams. We use an accuracy threshold of 0.6 when controlling sequential tagged with variable degree and of 0.3 when controlling sequential streams.

4.4.1 *Example*

Figure 4.4 shows an example of how an ABS controller works in an LLC bank of a system with four cores. The degree scale (0, 1, 4, 8, 16) represents the prefetch aggressiveness. Time moves from left to right and is divided into fixed length intervals as shown in the *Epoch* row. The four rows designated as *degree & trend* (P0, P1, P2 and P3) show the level of prefetch degree and trend associated with each core at each epoch. For instance, 4↑ means prefetch degree of 4 and upward trend. The core identifier and its trend over the dashed arrows indicate the change applied between two consecutive epochs. For example, from E0 to E1 the degree of P1 is changed from 4 to 1. The *bank miss ratio accounting* row shows the miss ratio at each epoch. The last row shows the comparison result between the reference and the current bank miss ratios. Next we show the positive and the negative cases.

POSITIVE AGGRESSIVENESS CHANGE At the beginning of E1, aggressiveness of the P1 core is changed from degree 4 to 1 following its downward trend. The question mark next to the degree of P1 at E1 epoch (1?) means that the change is being evaluated. At the end of E1 we observe a decrease in the bank miss ratio (a>b) with respect to the reference epoch (a). Therefore, the change in degree and the current trend of P1 are confirmed. E1 becomes the new reference epoch. The same happens at the epochs E2 and E5.

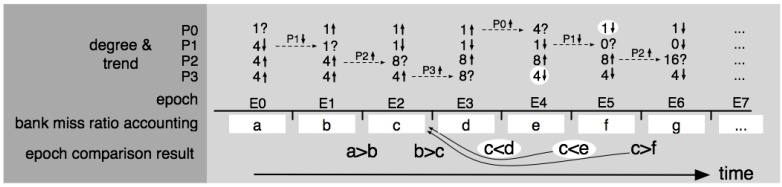


Figure 4.4: Example of ABS controller working in an LLC bank of a 4-core system

NEGATIVE AGGRESSIVENESS CHANGE White circles surround negative evaluations. At the beginning of E3, the P3 degree is changed from 4 to 8 following its upward trend. At the end of E3, an increase in the bank miss ratio (c<d) is observed, therefore the change in the P3 degree is undone, becoming 4, and its trend is reversed to downward. The reference epoch is not changed. At the beginning of E4, the P0 degree is changed from 1 to 4. Note that E2 is the reference epoch in E4. They only differ in the degree of P0, which is the one under evaluation in E4. At the end of E4 the bank miss ratio is also higher than it was at E2. Therefore, the change in the P0 degree is undone, the trend is reversed, and E2 remains as the reference in E5.

4.4.2 *Miss ratio as a good metric to guide aggressiveness*

Prefetch related metrics such as coverage, accuracy, timeliness, pollution or consumed bandwidth have often been proposed to evaluate the quality of a prefetch engine because an aggregate figure such as the miss ratio does not allow the net effect of individual prefetches to be distinguished [46, 63].

These same metrics were subsequently used in other works to guide prefetch aggressiveness [14, 15, 58]. However, these metrics are not directly related with system performance. Moreover, in the context of a banked LLC they pose two important problems: i) some of them are hard to compute online, and ii) they are difficult to aggregate in a single number in order to take a decision.

In this contribution we use the LLC bank miss ratio as the main metric to guide prefetch aggressiveness. The penalty of the off-chip misses is large in processor cycles and so a miss ratio decrease has a great potential to reduce Cycles per Instruction (CPI) and improve performance. Therefore, we expect the LLC miss ratio to be a good measure of performance. Moreover, the ABS controller associated

with each LLC bank can locally count the number of misses in the bank. In consequence, our proposal establishes the feedback loop without requiring communication among LLC banks.

From a performance standpoint, the prefetch related metrics are highly correlated with the miss ratio. In fact, some of these metrics are only valuable when they really correlate with the miss ratio. For instance, a prefetched line is considered useful if it is used along its lifetime in a cache, and useless otherwise (accuracy metric). However, a useful prefetch does not always have a positive impact on performance. Indeed, it will only increase performance if it gets a reduction in the miss ratio. In particular, if the line evicted by the prefetch is referenced before the prefetched line itself, despite having a useful prefetch, the miss ratio does not change and therefore no performance increase will be seen.

4.4.3 *ABS controller hardware cost*

The hardware cost of our proposal is low. In each bank, an ABS controller needs 4 bits per core in order to keep its prefetch state (1 bit for trend + 3 bits for aggressiveness level). It also needs four 16-bit counters (bank misses, bank accesses, prefetch requests, and hits on prefetched blocks). Additionally, it needs a 3-bit counter (4 bits in a 16-core system) to maintain the reference epoch age (number of epochs without changing the reference). The reference and the current miss ratios are stored in 16-bit registers. Finally, a 15-bit counter is needed to divide time into 32K-cycle epochs. For instance, in an 8-core multiprocessor each ABS controller needs 146 bits. Thus, in our baseline system fitted with 4 LLC banks, 584 bits are needed. If we consider a 16-core system with the same memory hierarchy, 716 bits are needed by the four ABS controllers.

Most prefetchers add to every cache line a tag bit in order to detect first use after prefetch and then react based on the prefetchless miss stream[3] [45, 54]. For instance, sequential tagged prefetch uses the tag bit to trigger new prefetches on the first use of a prefetched line. The ABS controller uses the same tag bit to count hits on prefetched lines, and so we ignore that bit in the ABS costs.

3 Given two equal caches, with and without prefetch, note that the miss stream outgoing from the prefetch cache differs from that of the prefetchless cache. However, it is possible to rebuild the prefetchless miss stream in a prefetch cache by joining the actual miss stream with the first-use stream of prefetched blocks.

4.5 PREFETCHING FRAMEWORK

In this section we present the entire prefetch environment. Section 4.5.1 establishes the minimum requirements of a prefetch engine in order to operate under ABS control. Section 4.5.2 discusses how to isolate prefetch in LLC banks, and Section 4.5.3 details the prefetch implementation in each LLC bank.

4.5.1 *Prefetch engine*

The operation of the ABS controller is orthogonal to the prefetch engine used to generate prefetch requests. The only necessary characteristic in this prefetch engine is that it has to be able to operate at different aggressiveness levels. In this work, the base system uses a sequential tagged prefetcher with variable degree (degree varying along the following scale 0, 1, 4, 8, 16). Given an initial address and a degree k, it is asked to generate k sequential references in the next k cycles. In Section 4.7.4 we also evaluate ABS controlling sequential streams as the prefetch engine, using the same model of sequential streams as Ebrahimi et al. [15]. ABS can control prefetch engines generating non-consecutive references, either belonging to a fixed-stride stream, or a stream generated by a context predictor like GHB or PDFCM [45, 49]. Considering such prefetch engines under ABS control is still an open research avenue.

4.5.2 *Bank-isolated prefetch*

A common mapping of memory lines to LLC banks is line-address interleaving. On a miss on line L mapped to bank B, a sequential prefetcher located at bank B generates a prefetch of line L+1, which maps to the next LLC bank. The address is looked up in the destination cache bank and, on a miss, it is forwarded to the main memory. Thus, communication among LLC banks is required in order to send every prefetch request from the bank that generates it to the destination bank. Alternatively, we could send the prefetch request to the memory without a previous lookup. In this case, we can waste memory bandwidth on prefetching lines already existing in the LLC.

In order to avoid expensive communication between LLC banks or waste of memory bandwidth, LLC prefetch is arranged to achieve isolation among banks, i.e. prefetches generated from a bank always target itself. We analyze two bank-isolated prefetch methods: increasing stride and changing the address interleaving among banks. The first consists on increasing the prefetch stride from one to a multiple of the number of banks. As an example, in an LLC with 4 banks and a sequential prefetcher at each bank, a miss on the line L issues the prefetch of the line L+4. Achieving bank-isolation in this way has the drawback that several prefetchers have to learn a fraction of the same stream, and thus the number of prefetch addresses not issued during the learning time is multiplied by the number of banks, this is a serious drawback, especially for short streams.

The second method consists of increasing the address interleaving granularity among banks. The LLC banks are interleaved using operating system pages, where consecutive physical pages map into consecutive LLC banks (mapping logical to physical addresses is performed in our experiments by the simulated operating system). This way an address stream always maps to a single bank while the page boundary is not crossed, and all addresses generated by a bank prefetcher will target the same bank. This is not a problem because in order to avoid translating prefetch addresses, prefetchers do not usually issue addresses beyond the page boundary of the address originating the prefetch [32, 18, 19].

We have evaluated the performance of both interleaving options for the baseline system without prefetch, noting that performance was slightly higher using page interleaving. Similar conclusion was obtained in a previous work [10].

4.5.3 *Prefetch details*

Aggressive prefetchers such as a high-degree sequential tagged prefetcher can generate a significant number of prefetches. We assume the hardware cost of the prefetcher is lowered by sharing the same lookup port for demand and prefetch requests (demand requests have higher priority). Furthermore, only one adder is provided to each LLC bank in order to generate prefetches, and so prefetch addresses are computed at a rate of one per cycle. The generation of a burst of prefetch requests after a cache miss or cache hit to a tagged line is cut off if another event initiates a new prefetch burst.

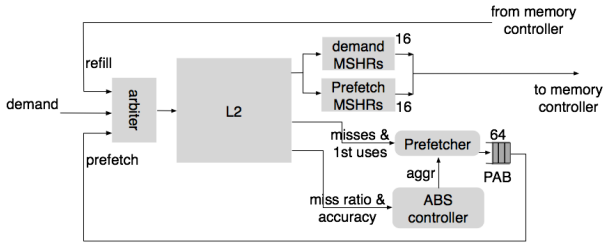


Figure 4.5: Components of an LLC bank

After being generated, each prefetch is sent to a prefetch address buffer (PAB) and waits in a queue until the LLC lookup port is available, see Figure 4.5. Because the PAB has a finite number of entries, it is managed in FIFO order (both for servicing and dropping prefetch requests the oldest one is processed first). Before inserting a prefetch address, the PAB is checked for an already allocated entry with the same address in order to avoid having duplicated requests.

Prefetch and demand Miss Status Holding Registers (MSHRs) keep the requests to the main memory until the arrival of the corresponding cache lines. There are no duplicated entries between MSHRs. When the LLC bank tag port is available, the request at the head of the PAB looks up both the bank tags and the two MSHRs. Only if they all miss, is the request sent to the memory and inserted in the prefetch MSHRs.

Demands have higher priority than prefetch requests at every arbiter in the hierarchy. Moreover, a demand can arrive at an LLC bank asking for a line that is being prefetched but whose data are not yet loaded into the cache. In that case a prefetch upgrade command is sent to the memory controller. If the request is queued at the controller. That command will upgrade the prefetch request giving it demand priority. The rationale of this mechanism is to prevent an aggressive prefetch from damaging regular memory instructions.

4.6 METHODOLOGY

This section explains the differences between the methodology followed in order to evaluate the contribution that this chapter presents and the methodology presented in Chapter 3. Mainly, what is different here is the concrete configuration of the memory hierarchy, the

number of evaluated workloads by simulation, and the metrics considered to evaluate the contribution.

4.6.1 *Experimental setup*

As the general experimental framework description explained, *Simics*, a full-system execution-driven simulator, has been used to evaluate our proposal [40]. The *Ruby* plugin from the *Multifacet GEMS* toolset was used to model the memory hierarchy with a high degree of detail [41], including coherence protocol, on-chip network, communication buffering, contention, etc. The prefetch system has been integrated into the coherence mechanism and a detailed DDR3 DRAM model has been added.

Multiprogrammed *SPECCPU 2K6* workloads running on a Solaris 10 Operating System have been used. In order to discard the less demanding memory applications and locate the end of the initialization phase, all the SPARC binaries were run on a real machine until completion with the reference inputs and hardware counters were used. Eight applications were subsequently discarded and 21 selected, shown in the first column of Table 4.1.

For an eight-core system, a set of 30 random mixes of 8 programs each, were taken from the previously selected 21 *SPECCPU 2K6* programs (no effort has been made to distinguish between integer and floating point). In the next section, we usually show averages over this set of 30 mixes, but in order to gain a deeper insight into individual mix behaviors, sometimes a subset of 10 mixes is shown. This randomly chosen subset of mixes appears in Table 4.1, along with the misses per kilo-instruction (MPKI) of each application in each mix when the application runs alone, that is, it runs using all the shared memory resources and with the remaining seven cores stopped.[4]

4 Notice that in general the same application appearing in two different mixes does not have the same MPKI value. This is because the number of executed instructions before creating a multiprogrammed checkpoint is given by the application with the longest initialization phase in the mix. This means that the results for a particular application, in general, can not be compared between mixes.

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
bzip2					1.8		1.7			
bwaves	20.7		20.7		20.7	20.7	20.7			20.7
mcf	33.9			37.5	30.7	33.2	20.2			
milc	16.4	16.2		34.2						
zeusmp	16.7						8.1	9.1	7.3	13.8
gromacs			3.9						4.0	
cactus.		4.2		4.1		4.2	4.2	4.4		
leslie3d				28.3	32.5	36.4			14.4	
gobmk			1.5	1.5					1.4	1.5
dealII		0.0		0.1				0.2	0.3	
soplex	3.0			4.0				3.6		
povray		0.3	0.3				0.3		0.3	0.3
calculix						0.5			5.9	0.5
gems.	32.5		26.9				32.5	26.8		
libq.					28.8	85.5	65.6			
tonto		3.4	1.5						1.6	
lbm	36.1	47.6	36.1		36.1					36.1
omne.	0.7	5.4			0.7	0.7		0.6		0.7
wrf					3.0			0.5		0.5
sphinx3		12.5		12.9						
xalan.			1.8			1.8		1.5		
MPKI	16.9	4.62	5.05	7.67	10.8	10.37	9.06	3.22	8.03	4.04

Table 4.1: MPKI of the benchmarks in the selected mixes

Private L1 I/D	16KB, 4-way pseudo LRU replacement, 64B line size, 1cycle
Shared L2	4MB inclusive 4 banks of 1MB each. Data array internally sub-banked), 4KB interleaving, 64B line size. Each bank: 16-way pseudo LRU replacement, 2-cycle TAG access, 4-cycle data access. 16 demand + 16 prefetch MSHR
Prefetch engine	Sequential tagged, degree 16
DRAM	1 rank, 16 banks, 4KB page size, Double Data Rate (DDR3 1333Mhz)
DRAM bus	667Mhz, 8B wide bus, 4 DRAM cycles/line, 16 processor cycles/line

Table 4.2: Baseline system configuration

4.6.2 *Baseline system*

The baseline system has eight in-order cores. The shared LLC has four banks interleaved at page granularity (4KB in the simulated operating system). A MOSI protocol keeps the memory system coherent while allowing thread migration among cores[5]. A crossbar communicates the first level caches and the shared LLC banks. There is a single DDR3 memory channel. The DRAM memory bus runs at a quarter of the core frequency. Table 4.2 gives additional implementation details.

4.6.3 *Performance indexes*

As previous work has shown, sequential prefetch delivers good results when only one program is executed in the system [49, 54]. However, as pointed out in Section 4.2, this assumption is no longer true for a multiprocessor system because resources are shared among cores that interfere each other. Thus, the goal of the ABS controller is to control the prefetch aggressiveness on a per-core basis in such a way that programs running together in the multicore system attain similar performance as when running alone with an aggressive sequential prefetch. In consequence, our performance indexes use as reference the IPC of the programs running alone on a system with a sequential tagged prefetcher with a fixed degree of 16.

5 Migration can mainly affect to the operating system threads. Each application thread is bound to a different core for the multiprogrammed workloads.

In order to evaluate the performance of the system when it is controlled by our ABS controllers, we use two system-oriented performance indexes, namely weighted speedup (eq. 4.1) [39], and main memory bandwidth consumption, and two user-oriented performance indexes, namely harmonic mean of speedups (eq. 4.2) [55], and fairness (eq. 4.3) [44]. Next, these metrics are shown with more detail.

First, we define

IPC_i^{SP}: IPC of program i running alone in the system and with fixed degree-16 sequential tagged prefetch.
IPC_i^{MP}: IPC of program i when other applications run in the rest of the cores

WEIGHTED SPEEDUP (*ws*) It quantifies the number of jobs completed per unit of time [16]. Here it represents the sum of slowdowns that each of applications experiments because the competition. It is expected that all the terms in the summation are lower than *1*, thus the value of the metric is between *0* and *n*, and the higher the better.

$$WS = \sum_{i=1}^{n} \frac{IPC_i^{MP}}{IPC_i^{SP}} \tag{4.1}$$

HARMONIC MEAN OF SPEEDUPS (*hs*) It is the inverse of the average normalized turnaround time [16]. Apart from adding the speedups that each of applications experiments, it captures the effects of competition, reflecting fairness more intensely than WS. It is expected that all the terms in the summation are higher than *1*, thus the value of the metric is between *0* and *1*, and the higher the better.

$$HS = \frac{n}{\sum_{i=1}^{n} \frac{IPC_i^{SP}}{IPC_i^{MP}}} \tag{4.2}$$

MEMORY BANDWIDTH It is accounted in order to know how much the different prefetch control schemes stress the links with main memory. We express it as a fraction between *0* and *1* reflecting the average occupancy of the links between the SLLC and the main memory.

FAIRNESS (*fa*) To determine whether the co-execution in the multicore system benefits or harms some programs more than others we use the fairness index (*FA*) [44]. The value of the metric is between *0* and *1*, and the higher the better.

$$FA = \frac{min(IS_1, IS_2, \ldots, IS_n)}{max(IS_1, IS_2, \ldots, IS_n)}, where\ IS_i = \frac{IPC_i^{MP}}{IPC_i^{SP}} \quad (4.3)$$

4.6.4 *Prefetch specific metrics*

Next sections will also show how well prefetching is performing, thus specific metrics will be used. Concretely, our interest will focus on two different metrics: *coverage* (eq. 4.4) and *accuracy* (eq. 4.5). As these metrics are used in an example to gain insight on the prefetch behavior, both will be only used in a per-application way within a multiprogrammed context. Next, these metrics are shown with more detail.

First, we define
$F_i = \text{First hit on lines prefetched by application } i$

COVERAGE (*cov*) It represents the fraction of misses prefetch is able to avoid. Equation 4.4 shows the definition of coverage for the prefetch associated to application *i*. The value of the metric is between *0* and *1*, and the higher the better.

$$Cov_i = \frac{F_i}{F_i + M_i}, \quad (4.4)$$

Where, $M_i = \text{Misses of application } i$

ACCURACY (*acc*) It expresses the fraction of prefetches that are actually used by the processor. Equation 4.5 shows the definition of accuracy for the prefetch associated to application *i*. The value of the metric is between *0* and *1*, and the higher the better.

$$Acc_i = \frac{F_i}{P_i} \quad (4.5)$$

Where, $P_i = \text{number of prefetches issued by application } i$

4.7 RESULTS

In this Section we evaluate the ABS controllers. Section 4.7.1 analyzes ABS controlling sequential tagged prefetch in the baseline system. Section 4.7.2 shows results for 16-core systems. In Section 4.7.3 we increase the LLC size. In Section 4.7.4 ABS controllers are compared with a previous proposal, and in Section 4.7.5 they are evaluated using parallel applications.

4.7.1 *Results for the 8-core baseline system*

Figure 4.6 shows the results of ABS controlling prefetch aggressiveness of the *mix2* described in Section 4.2. The 8 programs run simultaneously and the bars corresponding to execution without prefetch (*8apps no pref*) and with fixed-degree (16) aggressive prefetch (*8apps pref*) are kept. A new bar corresponding to ABS prefetch (ABS) is added.

ABS controlled prefetch increases performance compared with the aggressive fixed degree prefetch in all programs. IPC improvement ranges from 23% in *sphinx3* to 40% in milc. With regard to the system without prefetch, ABS control only slightly affects the performance of *deall* and *povray*, while the aggressive prefetch leads to significant losses in five of the eight programs. In contrast, in six of the eight programs ABS control outperforms the system without prefetch, achieving improvements between 3% in *omnetpp* and 225% in *lbm*.

Global measures such as IPC do not allow us to find out how prefetching is working. To give insight into prefetching behaviour, Figure 4.7a and Figure 4.7b show prefetch coverage and accuracy, re-

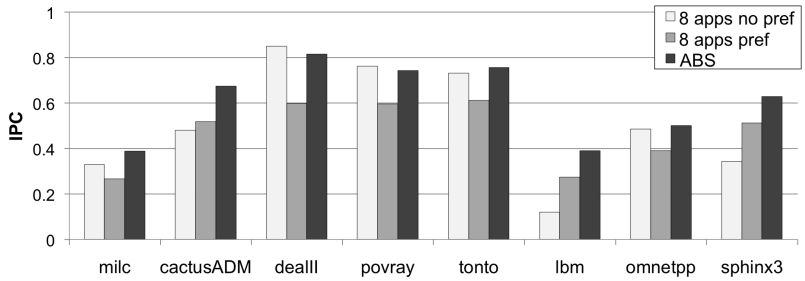
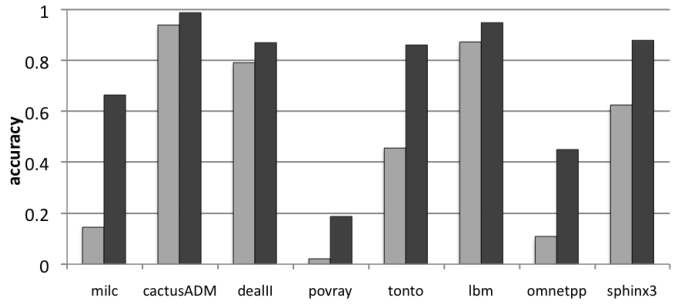
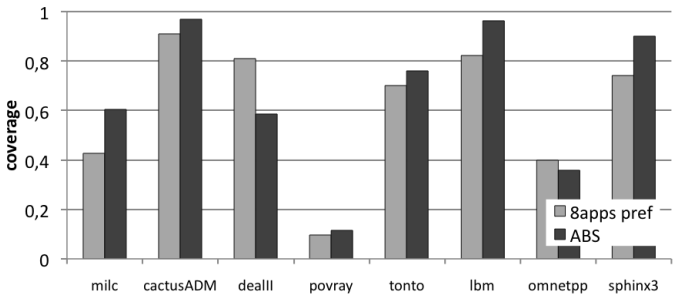


Figure 4.6: IPC for eight SPEC2K6 applications (mix2) running on an 8-core system with a shared LLC



(a) Prefetch accuracy



(b) Prefetch coverage

Figure 4.7: Prefetch accuracy and coverage of sequential tagged prefetching with fixed degree of 16, and variable ABS-controlled degree (mix2).

spectively, comparing fixed degree (*8apps pref*) with ABS-controlled degree (*ABS*). Prefetch coverage is very uneven across applications, ranging from about 0.1 in *povray* to more than 0.9 in *cactus*. ABS, despite being less aggressive, gets coverage similar or even higher than the fixed 16-degree prefetcher. ABS coverage is clearly better in five of the eight applications, and is clearly worse in two. As for accuracy, it is also uneven across applications, being very close to zero in *povray* and very close to one in *cactus*. But here ABS is the clear winner, achieving higher accuracy in all applications, highlighting the cases of *milc*, *povray*, *tonto*, *omnetpp* and *sphinx3*. The combination of both metrics, similar coverage and better accuracy, explains the performance improvement obtained by ABS.

It is interesting to delve into how different the aggressiveness computed in each bank by the replicated ABS controllers can be. Figure 4.8 plots a temporal trace of the prefetch degree for application *milc* in the mix2, in a prefetch system under ABS control in two of the four LLC banks (*bank 0* and *bank 1*). The failed tests (glitches), accounting for 1/ 8 of the total time in the worst case, have been removed to smooth the plot.

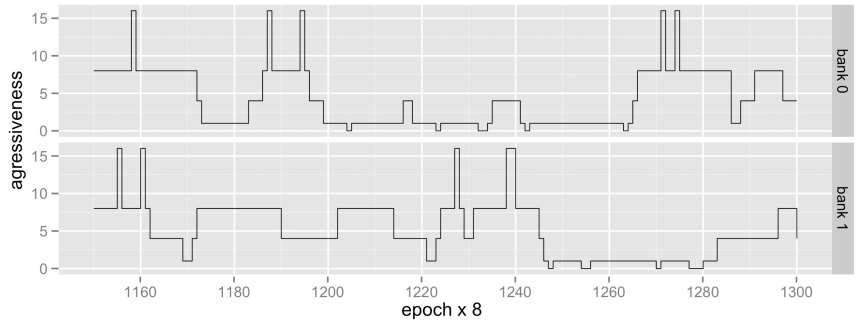


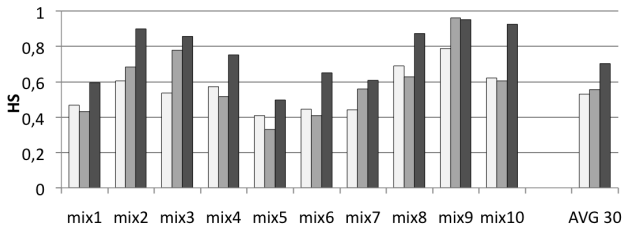
Figure 4.8: Evolution of the prefetching aggressiveness level for the *milc* application (mix2) in two of the four LLC banks during 160 tests

In the plotted sample, both ABS controllers usually take different control decisions; e.g. at time 1180, bank 0 prefetchs with degree 2, while bank 1 uses degree 8. The plot shows the flexibility of the distributed ABS control: a particular core may issue a miss stream with a different pattern into each LLC bank.

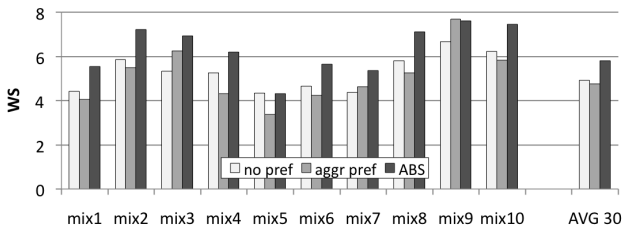
Figure 4.9 shows HS, WS, FA and consumed bandwidth for systems without prefetch (*no pref*), with fixed-degree aggressive prefetch (*aggr pref*), and under ABS control (*ABS*) for the ten mixes shown in Table 4.1. In each plot the rightmost bar group is the average of the 30 mixes (*AVG30*).

The HS values show a nonuniform pattern across the different mixes (Figure 4.9a). Aggressive prefetch increases by 4% the average HS with respect to no prefetch, but causes losses in six of the 10 mixes. Under ABS control, prefetch improves in 9 of the 10 mixes (up to 60% in mix6) and produces small losses in the other mix (0.5% in mix9). Also, ABS control always increases performance compared to no prefetch, between 20% and 50% in mix3 and mix6, respectively. On average, prefetch under ABS control, improves the system without prefetch by 35%.

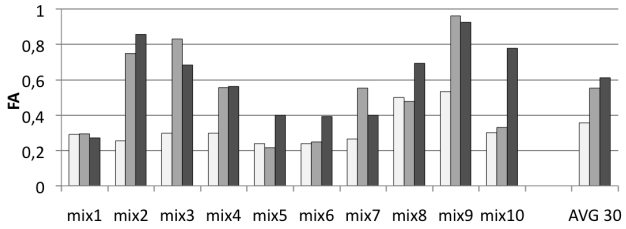
In terms of WS (Figure 4.9b), aggressive prefetch causes losses in seven of the 10 mixes and performs on average 3% worse than the system without prefetch. ABS control improves aggressive prefetch in 9 of the 10 mixes (up to 47% in mix4) and produces negligible losses in the other mix (1% in mix9). On the other hand, prefetch under ABS control improves the system without prefetch in 9 of the 10 mixes, with improvements ranging from 14% in mix9 to 27% in mix3. In mix5,



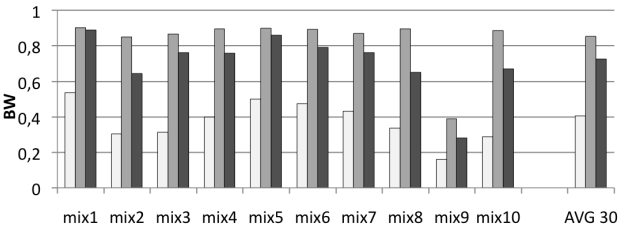
(a) Harmonic mean of speedups



(b) Weighted speedup



(c) Fairness



(d) Bandwidth consumed

Figure 4.9: Results for ten mixes of SPEC2K6 applications running on an 8-core system with a shared LLC

WS results in a reduction of 0.3%. On average, prefetch controlled by ABS improves the system without prefetch by 18%.[6]

Figure 4.9c plots the FA values, showing that the system with aggressive prefetch is significantly more fair than the system without prefetch. This is because the performance indexes use as reference a system with prefetch as we have seen in Section 4.6.3. Therefore, in the system without prefetch we see the unfairness introduced by the lack of prefetch itself, plus the unfairness due to the interferences among the eight cores. In Figure 4.9c, we observe the low fairness of mix2 in the system without prefetch. As the HS index includes some notion of fairness in its definition and WS is a pure throughput index, the previous issue about mix2 becomes clear. On average, the system using ABS controllers is more fair than the system with aggressive prefetch (0.62 and 0.56, respectively). ABS controllers make the system more fair in 6 of the 10 mixes (differences between 1% and 140%), and less fair in the remaining 4 (differences between -3% and -30%).

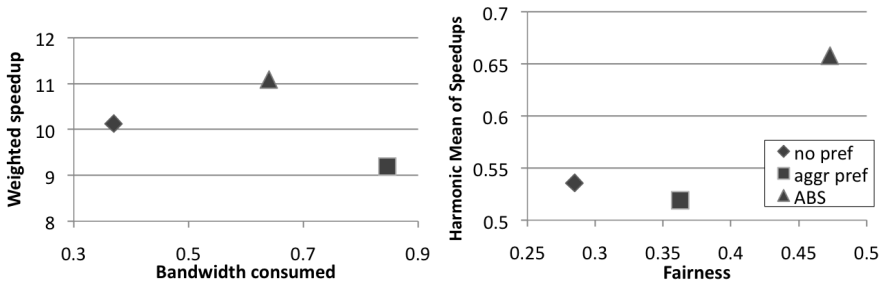
Finally, in Figure 4.9d we see that the main memory bandwidth consumption of the system without prefetch is very uneven among the different mixes, varying between 18% and 55% of the maximum bandwidth. However, the common pattern is that aggressive prefetch greatly increases bandwidth consumption with respect to the system without prefetch (on average, from 40% to 85% of maximum bandwidth), and ABS removes a significant portion of that increase lowering it to 70% of maximum bandwidth.

Summarizing, in an 8-core chip with a shared 4-MB LLC, the use of ABS controllers improves the system that is using uncontrolled aggressive prefetch. On average, throughput (*WS*), the inverse of the turnaround time (*HS*), and fairness (*FA*) increase 27%, 23% and 11%, respectively while memory bandwidth consumption decreases by 18%.

4.7.2 *Results for a 16-core system*

In this section we analyze the behaviour of prefetch in a 16-core system. The LLC is not modified, so that the increase in the number of cores results in an increased pressure on the LLC. However, commu-

6 Notice that the performance indexes HS and WS do not always correlate; in mix2, for instance, the HS index indicates that aggressive prefetch is better than no prefetch while the WS index indicates the contrary. The discussion on fairness will give a deeper insight into what is happening.



(a) System-oriented metrics (b) User-oriented metrics

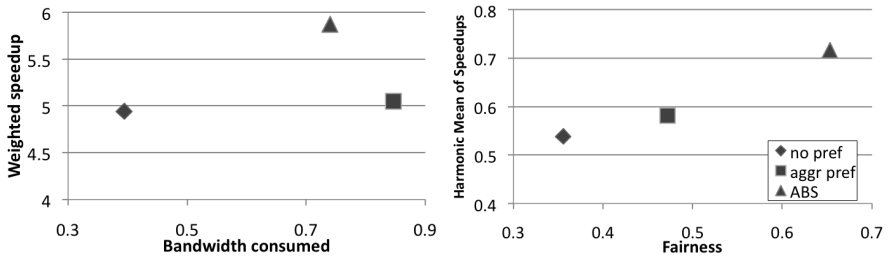
Figure 4.10: ABS performance on a 16-core system

nication with the main memory is expanded from one to two DDR3 channels. We run 30 mixes of 16 applications randomly selected among the 21 SPEC CPU 2006 shown in Table 4.1. We only present the average of each index over the 30 mixes of 16 programs each. Figure 9.a combines in a single Y-X plot the system-oriented metrics, WS and bandwidth, while Figure 9.b combines the user-oriented metrics, HS and FA.

In a system with 16 processors, fixed-degree (16) aggressive prefetch (*aggr pref*) produces losses compared to no prefetch (*no pref*) in terms of throughput (WS decreases 9%) and turnaround time (HS decreases 4%). The memory bandwidth consumption greatly increases from 36% to 85%. Only fairness improves from 0.28 without prefetch to 0.36 with aggressive prefetch.

Controlling aggressiveness leads to improvements in all metrics. Compared to aggressive prefetch, ABS control (*ABS*) increases the HS index by 27% (22% compared to no prefetch), increases the FA index to 0.48, and also improves the system throughput index with a WS increase of 25% (14% compared to no prefetch). The bandwidth consumption decreases significantly compared to aggressive prefetch, from 85% to 62% of the maximum, but it is still greater than without prefetch which only requires 36% of the maximum.

Summarizing, in a 16-core chip with a shared 4-MB LLC, ABS improves the system in all indexes. Comparing between 16 and 8 cores, the increase in the WS index is similar but the improvement in the rest of the indexes, HS, fairness and memory bandwidth, is much higher. This result is consistent because the pressure on the memory hierarchy in a 16-core chip is larger than in an 8-core, resources are more scarce, and therefore controlling the prefetch aggressiveness becomes more important.



(a) System-oriented metrics (b) User-oriented metrics

Figure 4.11: ABS performance on an 8-core system with an 8MB LLC

4.7.3 *Doubling the LLC size*

In this section we analyze the behaviour of prefetch in the 8-core baseline system when doubling the LLC size to 8 MB. We only show average indexes computed over the 30 mixes already used in section 4.7.1. The system-oriented metrics WS and bandwidth, are shown in Figure 10.a, while Figure 10.b shows the user-oriented metrics HS and Fairness.

In an 8-core chip with a shared 8-MB LLC, the use of ABS controllers also improves the behavior of uncontrolled aggressive prefetch. On average, throughput (*WS*), the inverse of the turnaround time (*HS*), and fairness (*FA*) increase 18%, 24% and 38%, respectively while memory bandwidth consumption decreases 14%.

When increasing the cache size, controlling the prefetch aggressiveness becomes less important for improving performance, but it improves fairness and saves bandwidth. Thus, when increasing from 4 to 8 MB, ABS improvements over uncontrolled prefetch change from 27% to 18% in *WS*, from 11% to 38% in *FA*, and from 18% to 14% in *BW*. As for *HS*, the results are similar.

4.7.4 *HPAC comparison*

Next we compare the ABS control with the *Hierarchical Prefetcher Aggressive Control mechanism (HPAC)* introduced in [15]. To the best of our knowledge, this was the only work to date on adjusting prefetch aggressiveness in a shared LLC.

HPAC works in a centralized LLC with a single access port although internally it is organized in banks to support several concurrent accesses. The proposal uses sequential streams as the prefetch engine and a local control of aggressiveness for each core: Feedback-Directed Prefetching (FDP) [58]. HPAC adds a global interference feedback in order to coordinate the prefetchers of the different cores and throttle their aggressiveness.

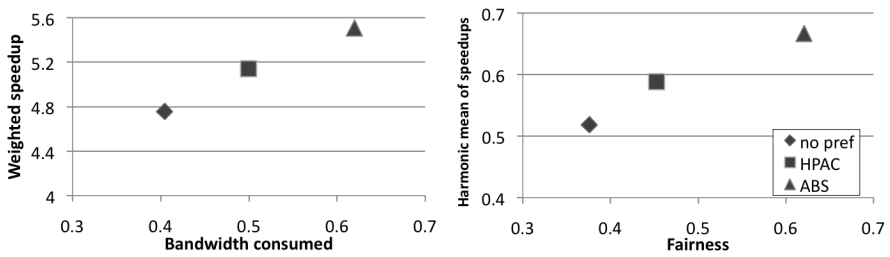
Since we assume autonomous LLC banks, possibly placed at distant die locations, distributing HPAC is not straightforward. We choose a distributed implementation giving HPAC as much knowledge and control as possible, namely each core has an FDP per LLC bank, and each LLC bank has an HPAC controlling the corresponding FDP. So, the distributed HPAC/FDP we test requires 32 FDPs (8 FDPs per bank $\times$ 4 banks = 32 FDPs), and 4 HPACs (1 HPAC per bank $\times$ 4 banks = 4 HPACs).

Besides other local bank metrics, HPACs gather statistics from one/two memory controllers (8/16 core systems), and the communication among HPACs and the memory controllers is modelled in an ideal way (zero-delay/no BW limitations).

We have used the thresholds indicated in the published proposals for both mechanisms. We simulate 32 streams per core and LLC bank (32 streams $\times$ 8 cores $\times$ 4 banks = 1024 streams). Each stream launches sequential prefetches with a degree and distance from a starting address. We implement five levels of aggressiveness that correspond to degrees 1, 1, 2, 4, and 4 and distances 1, 4, 16, 32, and 64, respectively. The aggressiveness control mechanism (HPAC/FDP or ABS) decides the aggressiveness level associated to each core.

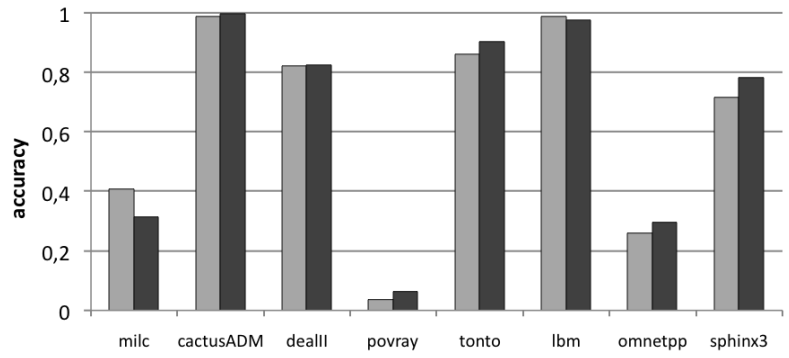
Figure 4.12 plots the results for HPAC and ABS on an 8-core system. Both mechanisms use sequential streams as the prefetch engine. Performance indexes have been computed using as references the IPCs of the programs running alone on a system with a sequential stream prefetcher with a fixed level of aggressiveness (distance 64 and degree 4). We only show average indexes computed over the 30 mixes already used in Section 4.7.1.

Figure 4.12a shows the system-oriented metrics, WS and BW, while Figure 4.12b shows the user-oriented metrics, HS and *Fairness*. ABS control obtains better results than HPAC in all metrics except in the consumed bandwidth. ABS improves WS index by 8%, HS index by

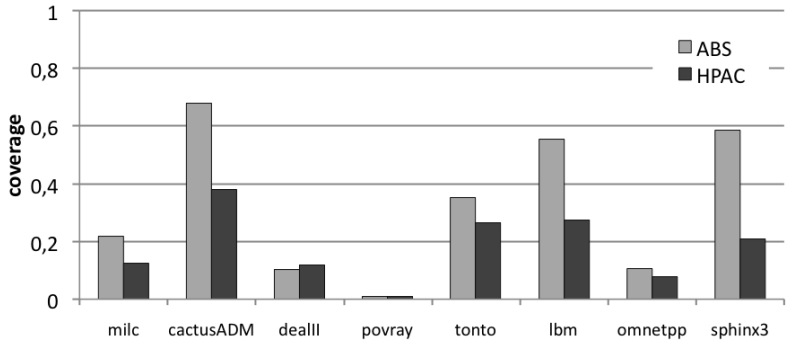


(a) System-oriented metrics (b) User-oriented metrics

Figure 4.12: HPAC and ABS performance on an 8-core system.



(a) Prefetch accuracy



(b) Prefetch coverage

Figure 4.13: Prefetch accuracy and coverage of sequential streams with variable ABS-controlled and HPAC-controlled degrees (mix2).

14% and fairness by 40%. Compared to no prefetch, bandwidth consumption is higher in ABS (62%) than in HPAC (50%).

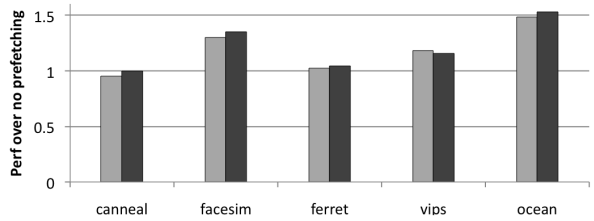
The results for a 16-core system are not shown but are similar. ABS also produces better results than HPAC in all metrics except consumed bandwidth. ABS improves the *WS* index by 7%, the *HS* index by 11%, and fairness by 29%. Compared to no prefetch, bandwidth consumption is higher in ABS (54%) than in HPAC (44%).

Figure 4.13a and Figure 4.13b show prefetch accuracy and coverage, respectively, of sequential streams prefetching, with ABS and HPAC controlling the prefetch aggressiveness. Sequential streams, as a prefetch engine, are less aggressive than sequential tagged, because the former has a long learning period and the latter has not. This explains the low coverage we see compared to that of Figure 4.13b. On the other hand, ABS shows greater coverage than HPAC for most applications in the mix. This may be so because HPAC can be more restrictive than ABS. For instance, in a situation of low accuracy and high pollution, HPAC throttles prefetch aggressiveness regardless of its overall impact on the miss ratio. In contrast, ABS throttles prefetch only if the miss ratio grows. As for accuracy, Figure 4.13a shows that, despite being more aggressive, ABS gets accuracy similar to that of HPAC.

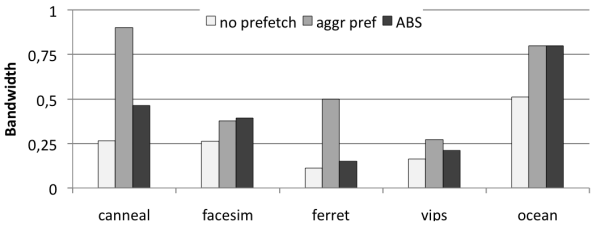
From the referenced papers we can compute accurately the HPAC/FDP hardware costs [15, 58]. In the 8-core system having 4 LLC banks of 1MB each, the total budget to implement HPAC/FDP is 466,624 bits (196,608 + 4 × (33,664+33,840))[7]. The implementation cost of the 1024 sequential streams, and the prefetched bit in each cache line have not been accounted for. In the 16-core system with the same LLC the FDP/HPAC cost is 933,056 bits.

Summarizing, the ABS controller gives a better performance than HPAC at the expense of some increase in consumed bandwidth. In addition, it succeeds with a very low implementation cost.

7 FDP requires a core identifier per LLC block (3 bits × 64K blocks = 196,608 bits). In each bank, FDP also requires seven 16-bit counters per core, and a 1-Kentry Bloom filter per core to detect intra-core prefetching interferences (1 bit and a core id per entry, totaling 33,664 bits per bank). In turn, HPAC requires in each bank eight 16-bit counters per core, three more 16-bits counters, and a 1-Kentry Bloom filter per core (1 bit and a core id per entry) to detect inter-core prefetching interferences (33,840 bits per bank).



(a) Speedup over no prefetch



(b) Bandwidth consumed

Figure 4.14: Results for five parallel applications

4.7.5 *Results with parallel applications*

In this Section we analyze the behaviour of our proposal when running parallel applications in the baseline system presented in Section 4.6.2. A priori, this behaviour should not depend on whether the threads running on each processor are independent or not. ABS varies the prefetch aggressiveness associated with each core seeking to optimize the performance of each LLC bank in terms of bank misses. The decrease in the number of misses in each bank should result in improving system performance.

We have selected the five applications of the *PARSEC* [5] and *SPLASH-2* [66] suites which have more than 1 MPKI in a 4-MB LLC (concretely canneal, facesim, ferret, vips and ocean applications. And their MPKIs are respectively 4.48, 3.45, 1.27, 1.74, and 13.35). In order to avoid migration, the threads are bound to cores using system calls. Binding is not performed if an application spawns in its parallel phase more threads than there are cores in the system. Performance statistics (execution time, memory bandwidth) are only taken in the parallel phases, which are run to completion in all the applications. We utilize the *simmedium* input set for PARSEC applications and a 1026x1026 grid for *Ocean*.

Figure 4.14a shows speedups of a system with fixed-degree prefetch (*aggr pref*) and with prefetch under ABS control (*ABS*) compared to a system without prefetch. The fixed-degree prefetch im-

proves performance with respect to no prefetch in four out of the five applications. Only *canneal* suffers a 5% increase in execution time when under prefetch with fixed-degree. The ABS control achieves higher speedups than fixed-degree prefetch in all the applications except *vips* (-1.8%). In *canneal*, the ABS controllers reduce prefetching looses from 5% to 0.3%.

Figure 4.14b shows the memory bandwidth consumption of the system without prefetch (*no prefetch*), with fixed-degree prefetch (*aggr pref*), and with prefetch under ABS control (*ABS*). Fixed-degree prefetch significantly increases the memory bandwith consumption for all the applications (from 40% in *facesim* to 450% in *ferret*). The ABS control reduces memory bandwidth consumption with respect to fixed-degree in *canneal* (-50%), *ferret* (-70%), and *vips* (-23%), and slightly increases it in *facesim* (+5%). Summarizing, the ABS controllers also improve performance over fixed-degree prefetch when running parallel applications, in spite of the low miss ratios of these applications. Execution time is reduced in four out the five analyzed parallel applications. Besides, significant savings in memory bandwidth arise in three applications.

4.8 CONCLUDING REMARKS

In this contribution we introduce the ABS controller (Adaptive prefetch control for a Banked Shared LLC), an adaptive mechanism to control the prefetch aggressiveness independently for each core in each LLC bank. The ABS controller implements a hill-climbing algorithm which runs stand-alone at each LLC bank, using only local information. Bank miss ratio and prefetch accuracy are sampled at fixed-length epochs and used as performance index. For each bank at each epoch the aggressiveness of only one core is varied in order to establish a cause-effect relationship between the change in aggressiveness and the change in the performance index.

The ABS controllers are evaluated to adjust the aggressiveness level of variable-degree sequential tagged prefetchers. Our analysis using multiprogrammed SPEC2K6 workloads shows that the mechanism improves both user-oriented metrics (Harmonic Mean of Speedups by 27% and Fairness by 11%) and system-oriented metrics (Weighted Speedup increases by 22% and Memory Bandwidth Consumption decreases by 14%) over an eight-core baseline system that uses aggressive sequential prefetch with a fixed degree. Similar re-

sults have been obtained on a sixteen-core system, when doubling the LLC size or running parallel applications.

Besides, ABS is also compared to the mechanism for a centralized shared LLC proposed by Ebrahimi et al. [15] but adapted for a banked LLC. For this comparison, variable-aggressiveness sequential stream prefetchers are used as prefetch engines. ABS performs better in all the performance indexes except in required bandwidth, which is somewhat greater.

Summarizing, ABS controllers are able to control the aggressiveness of prefetch engines in a distributed fashion and with very low implementation costs. Their distributed nature assures scalability in the number of cores and cache banks for future multicore chips.

Part III

EXPLOITING REUSE LOCALITY

This third part of the dissertation treats the reuse locality at the SLLC. It comprises two chapters: Chapter 5 states the reuse locality property and presents two replacement algorithms that exploit such property. Chapter 6 presents the reuse cache, an innovative solution to improve the SLLC efficiency. Employing the reuse locality, the reuse cache is able to dramatically downsize the SLLC data array, but maintaining at the same time CMP average performance untouched.

5

REUSE LOCALITY

SUMMARY

Optimization of the replacement policy used for shared last-level cache (SLLC) management in a chip-multiprocessor (CMP) is critical for avoiding off-chip accesses. Temporal locality is exploited by first levels of private cache memories, thus it is slightly exhibited by the stream of references arriving at the SLLC. Therefore, traditional replacement algorithms based on recency are bad choices to govern SLLC replacement. Recent proposals involve SLLC replacement policies that attempt to exploit reuse either by segmenting the replacement list or improving the re-reference interval prediction.

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

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

5.1 INTRODUCTION

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

Traditionally, each of all the memory hierarchy levels employs algorithms that consider temporal locality in order to select the cache line to replace. In particular, *least-recently used (LRU)* is a widespread replacement algorithm. It predicts that a recently accessed line (either hit or miss) will be used again soon. LRU gives good results on first-level caches because the complete stream of references from the processor is observed but, as many previous authors have shown, it has poor performance as a replacement policy for SLLCs [21, 35, 47, 59].

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

Although the reference stream observed by the SLLC may exhibit little temporal locality in the conventional sense, it does exhibit *reuse locality*. The concept underlying this type of locality can be described as follows: lines accessed at least twice tend to be reused many times in the near future and, moreover, recently reused lines are more useful than those reused earlier. That is, in reuse locality future references are only expected *after the first hit* to a line. In contrast, with temporal locality there is an expectation of future references straight after the first reference to a line, a miss. However, only a few lines in the SLLC have reuse locality. Indeed, most lines in the SLLC are *dead*,

and they will not receive any further references during their lifetime [25, 31, 47, 68].

Reuse locality has been identified and exploited in cache memories for disks. A representative proposal modified the LRU algorithm in order to protect reused pages against access patterns that result in poor performance such as thrashing or scanning [24]. On the other hand, recent research in SLLC replacement policies relies on predicting the re-reference interval [17, 23, 26, 67]. According to the predicted re-reference interval, the utility assigned to each line in these schemes can take one of several values. In contrast, as the reuse locality is a binary property, the derived replacement policies will only require two utility[1] values: to keep or not to keep.

In addition, most proposals consider *non-inclusive* SLLCs [17, 21, 26, 47, 67], meaning that the lines present in the private caches may or may not reside in the SLLC [3]. Several commercial processors have instead an *inclusive* SLLC that always keeps a superset of the contents of private caches [20]. This choice greatly simplifies the cache coherence protocol, and is usually implemented by invalidation. When an SLLC line is evicted, inclusion is enforced by sending invalidation messages to all the copies present in the private caches, if any [3, 8]. However, another replacement inefficiency arises when the replacement of an inclusive SLLC is managed by an LRU-based policy: a heavily referenced line with a short reuse distance may remain in private caches for a long time. During this time this *hot line*, despite being actively accessed by the core, may move down in the LRU stack of the SLLC, to the point of being evicted. This will force invalidation of the line in the private cache, though the processor will request the line again producing a new SLLC miss [22].

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

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

these lines are valuable and, when it is necessary to select a victim among them, the reuse order will provide a basis for the selection.

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

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

The LRR policy has the drawback of the implementation cost increasing with the square of the set associativity. Also based on recency, the *not recently used (NRU)* algorithm is an inexpensive alternative to LRU ordering [43]. Indeed, it is used in the SLLC of commercial processors, such as the Intel Itanium or Sun SPARC T2, and by using only one bit per line, the NRU cost increases linearly with the set associativity. The NRR policy we propose adapts the NRU algorithm for tracking reuse in SLLC sets. Under this NRR policy, every line is provided with a *NRR* bit. In contrast with NRU, the reuse bit will be unset only on hits, not on line refilling. Accordingly, all the not recently reused lines are victim candidates (NRR bit set) and the remaining lines are not.

The proposals are evaluated in an eight-core system with two private cache levels and an inclusive SLLC. By running a rich set of multiprogrammed workloads, we show that LRR outperforms LRU by 4.5%, and NRR outperforms NRU by 4.2% with exactly the same cost. We also show that our mechanisms outperform re-reference interval prediction (RRIP) [23], a recently proposed SLLC replacement policy. Similar conclusions can be drawn for a range of associativity values and SLLC sizes.

The chapter is structured as follows. Section 5.2 presents experimental evidence of reuse locality and the usefulness of not evicting the SLLC lines present in private caches. Section 5.3 presents a state of art proposal that is evaluated along our contributions and compared with them. Section 5.4 explains the LRR and NRR replacement

	milc	wrf	dealII	hmm.	dealII	omn.	libq.	gob.
APKI - 1 app	27.41	0.98	0.15	2.09	0.14	4.26	30.77	0.36
MPKI - 1 app	27.37	0.03	0.01	0.02	0.01	1.90	30.77	0.08
APKI - 8 apps	27.64	1.16	0.33	2.15	0.31	4.31	30.77	0.65
MPKI - 8 apps	27.63	0.53	0.27	1.14	0.27	3.38	30.77	0.46

Table 5.1: Number of SLLC accesses and misses per kilo-instruction of each application in the #91 mix, APKI and MPKI, respectively

policies, giving details of the implementation and associated costs. Section 5.5 reports and discusses the evaluation of our contributions, and finally, Section 5.6 discusses about the contributions exposed on this chapter.

5.2 MOTIVATION

In this section, we analyze the behavior of an example application from one of the evaluated workload mixes (mix #91) running in the hierarchy of a multicore chip made up of an SLLC and private caches. We highlight three effects, namely, *i*) by sharing the SLLC space, the working set of an application spreads towards distances greater than the cache associativity; *ii*) the principle of inclusion may force hot lines in the private levels to be invalidated, but private caches will request the line again straight away; and *iii*) most SLLC hits come from sustained reuse among a small subset of lines.

The selected mix is composed of eight applications of the SPEC CPU 2006 benchmark suite and runs in an eight-core CMP system with an inclusive SLLC. The first two rows of the Table 5.1 show the number of SLLC accesses and misses per thousand instructions, when the programs run alone. We can observe very different behaviors. For instance, *dealII* seems to fit well in the private caches and barely access the SLLC (0.15 APKI) and *hmmer* almost always hits in the SLLC (2.09 APKI and only 0.02 MPKI), while *milc* and *libquantum* access the SLLC many times (27.41 and 30.77 APKI, respectively) and almost always miss.

Figure 5.1 plots the number of *wrf* hits in a set-associative SLLC as a function of the LRU stack distance under three different boundary conditions (gray or black bars). The horizontal axis represents 64 LRU stack distances in the SLLC; the first 16 distances belong to real cache storage while the next 48 distances are tracked using shadow tags.

THE WORKING SET MAY SPREAD BEYOND THE AVAILABLE STORAGE In a CMP, different applications share the SLLC and compete for placing their working set into the cache. The lines of an application are displaced in the LRU stack by the lines inserted into the same set by other applications. Figure 5.1a shows the LRU stack when *wrf* runs alone in the CMP (it has the whole SLLC to itself). It can be observed that all the hits arise at a distance of 1 to 8 (0.95 HPKI overall), and there are no additional hits that a larger cache could capture. Now let us consider Figure 5.1b, which plots the hit distances of *wrf* when it runs along with the other seven applications shown in Table 5.1. We can see that the bars are smaller and spread over much longer LRU distances than before (from distance 1 to 16, 0.63 HPKI overall). This is because other applications such as *libquantum* or *milc* load a large number of lines which in turn displace the *wrf*

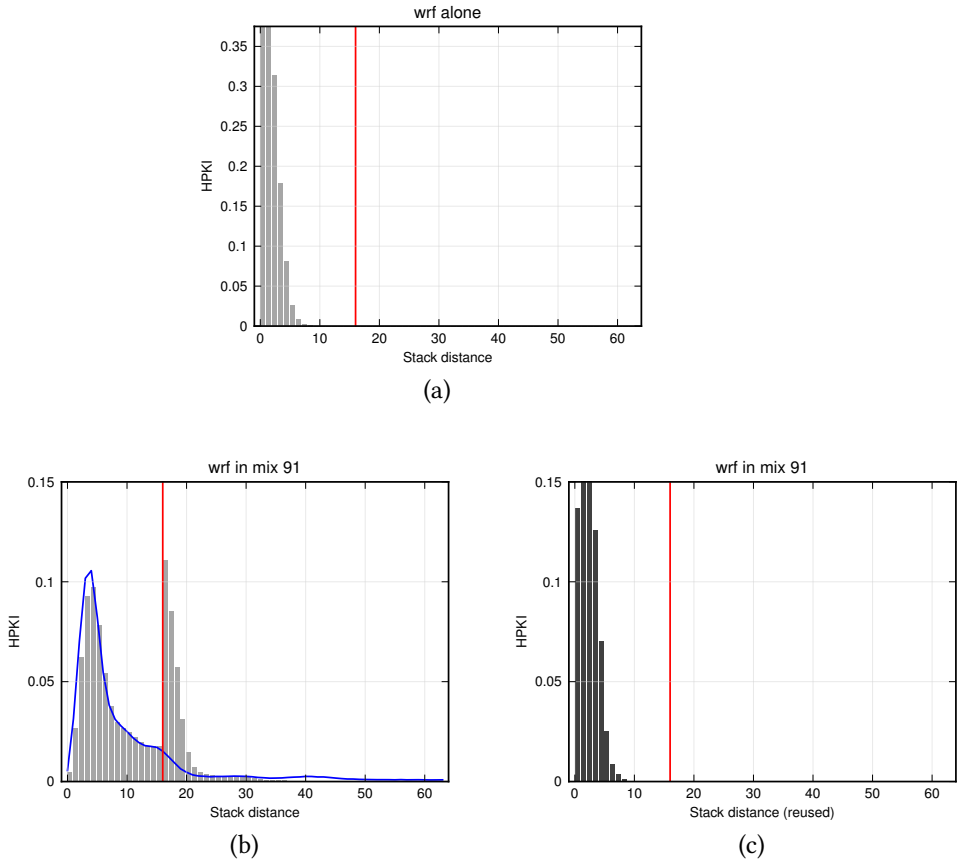


Figure 5.1: Distribution of hits along the LRU stack (HPKI = hits per thousand instructions) in three experiments; the vertical line signals the associativity of the used SLLC cache: a) *wrf* running alone in the CMP, b) *wrf* running with 7 other applications in an inclusive (bars) and in a non-inclusive (solid line) SLLC, and c) *wrf* running with 7 other applications in a non-inclusive SLLC (reuse stack)

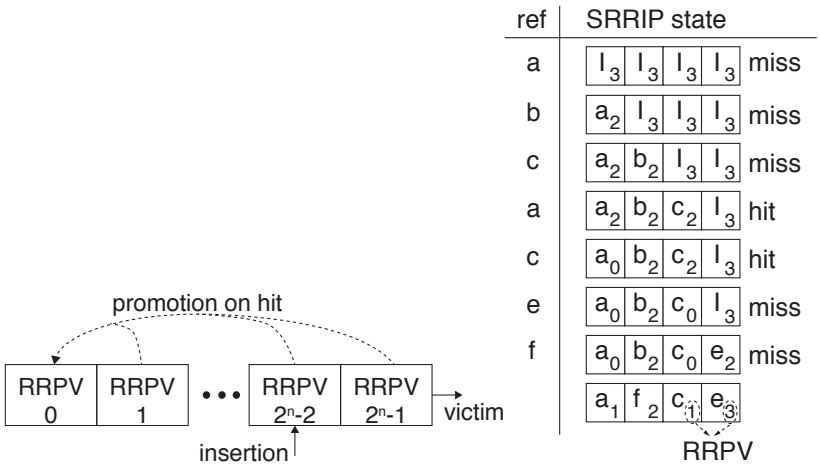
lines (see LLC MPKIs in Table 5.1). Consequently, the MPKI of *wrf* in a 16-way associative SLLC increases from 0.03 when running alone to 0.53 when running together with other applications.

CACHE INCLUSION PLUS HIGH SLLC MISS RATIOS MEAN HOT LINE THRASHING Hot lines, those lines with a sustained core reuse remain silently in private caches for a long time and, therefore, they may become stale in the LRU stack of the SLLC. Before replacing a hot line in the SLLC, the copies present in the private caches are invalidated, but as they are being used by the core, misses will occur and the private caches will request these lines again straight after the invalidation. In Figure 5.1b, we can see a peak at a distance of 17 followed by significant number of references at 17-20, meaning that the core is requesting recently invalidated lines. The solid line crossing the aforementioned peak is the distribution of LRU stack distances for a non-inclusive SLLC (from distance 1 to 16, 0.71 HPKI overall). The non-inclusive SLCC performs better because, even though it may be evicting the same hot lines as the inclusive counterpart, they are not invalidated in the private levels.

THE REUSED LINES FIT WITHIN THE ASSOCIATIVITY Figure 5.1c shows the distribution of the LRU stack distances for a non-inclusive SLLC calculated in the following way. LRU stack entries are tagged as *reused lines* when the first hit occurs, and all the remaining lines (*non-reused*) are ignored when calculating the stack distance. By doing this, we can see how the distance distribution of the reused lines concentrates at the top positions, without exceeding a distance of 10. This indicates that if the SLLC replacement policy were focussed on keeping the cache lines that can be expected to be reused, SLLC performance would be significantly improved.

5.3 RE-REFERENCE INTERVAL PREDICTION (RRIP)

Jaleel et al. [23] proposed *Re-reference interval prediction (RRIP)*; a state of art SLLC replacement policy. It 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 value (RRPV)*. This RRPV is represented by an N-bit counter which is associated to each cache line to classify it into one of 2^N segments.



(a) Overview (b) Example

Figure 5.2: Static re-reference interval prediction (SRRIP) replacement algorithm

The authors of the mechanism proposed two versions of the algorithm: *Static-RRIP (SRRIP)* and *Dynamic-RRIP (DRRIP)*. In Static-RRIP (Figure 5.2a), new lines are inserted into the segment corresponding to *intermediate* re-reference interval (segment with a RRPV equal to 2^N-2) making the algorithm *scan*-resistant. When a line has a to be selected as victim, a line with RRPV equal to 2^N-1 is randomly chosen, if there is no any, the RRPVs of all the elements are increased by one and the search is performed again.

Figure 5.2b shows the behavior of the SRRIP algorithm when an example stream (*a, b, c, a, c, e,* and *f*) of references arrives to a determined set of the cache. Each row represents the state of the set before the reference in column *ref* accesses the cache. We can observe the content of each element of the set along with its RRPV value and if the access was either a hit or a miss. As an example of the algorithm behavior, we will examine the last line of the figure. Let's have a look first at the previous line, the penultimate, *f* accesses to the cache, resulting on a miss. Looking to the last line of the example again, we can observe three different things, i) the RRPV values of all the elements in the set have increased by one, ii) *b* was evicted and iii) *f* has been inserted with a RRPV equal to 2.

Dynamic-RRIP uses *set dueling* to select between SRRIP and Bimodal. Bimodal is a *Thrash*-resistant policy that inserts lines with *long* re-reference interval (segment with a RRPV equal to 2^N-1) but a small fraction of randomly chosen lines that are introduced with

an *intermediate* re-reference interval. For a SLLC in a CMP, Thread-Aware-DRRIP requires a set dueling monitor for each thread.

Sections 5.5.1 and 5.5.3 compare the performance of DRRIP to the NRR and LRR performance. Section 5.5.4 discusses on the hardware cost that DRRIP has. Finally, Sections 5.5.5 and 5.5.6 show how DRRIP improves the system performance when the associativity and the size of the cache, respectively, are varied.

5.4 REUSED-BASED REPLACEMENT

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

Whenever a line is in the private caches, we assume its temporal locality is being well exploited and so it should be among the last ones to be evicted from the SLLC. Among the lines not present in the private caches, replacement should be carried out following the reuse order (total or partial). We consider an SLLC line to be reused if there has been at least one hit on it since it was loaded into the SLLC. The most recently reused lines are the last candidates to be replaced. Conversely, the lines that have not yet been reused will be the first eviction candidates, but as there is no basis for a ordering them, victims are selected randomly.

To record whether a line has been reused or not, we add a bit to each SLLC line, the *reused* bit. The reused bit is unset when a line is loaded into the SLLC (on a miss), and it is set when the line is reused (on a hit). We use another bit to record whether an SLLC line is present in any private cache, the *being-used* bit. When the line is present in a private cache the being-used bit is set and it is unset when the line only resides in the SLLC. Provided that the coherence protocol we use has precise knowledge of the line presence in the private caches[2], the being-used bit is not needed.

2 Non-silent eviction of clean blocks is assumed in the private caches.

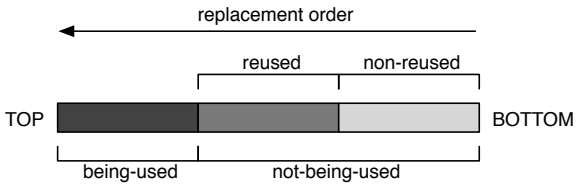


Figure 5.3: The reused-based stack

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

5.4.1 *LRR: Least Recently Reused*

Each cache set is organized as a single logical stack with being-used lines on top, reused lines in the middle and non-reused lines at the bottom (Figure 5.3). The replacement policy searches for a victim starting from the bottom and finishing at the top of the stack. Neither non-reused nor being-used lines are ordered. Thus, random replacement is employed when a line from these groups has to be evicted. Reused lines, on the other hand, are selected following the reuse order. This behavior is shown in Algorithm 1.

The LRR policy does not establish size bounds for any of the three groups. Accordingly, once a line has been classified as reused, it will not be evicted while there are any non-reused lines. This implies that the group of reused lines may grow to occupy all the available space, while the non-reused group may decrease to the point of disappearing. However, the number of non-reused lines increases each time a line without any hits in the SLLC (reuse bit unset) is evicted from the private caches, at which point the line joins the non-reused group. In Section 5.5, we will show how the number of lines evolves in each group over the course of the execution of a given mix of programs, as an example.

5.4.2 *NRR: Not Recently Reused*

LRR and LRU share the same scalability problem: their implementation cost grows quadratically with cache associativity. An alternative to LRU, also based on recency is the *not recently used (NRU)* replacement algorithm [43]. The implementation cost of the NRU approach

ALGORITHM 1: Least-Recently Reused (LRR) algorithm

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

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

grows linearly with cache associativity (1 bit per line). We propose a reuse-based algorithm inspired by NRU that we call *not recently reused (NRR)*. NRR has the same implementation cost as NRU.

As with LRR, NRR organizes each cache set as a single logical stack with being-used lines on top, reused lines in the middle, and non-reused lines at the bottom of the stack (Figure 5.3). However, NRR does not consider lines to have any order within each group.

NRR uses a bit per line, the *NRR* bit, in order to distinguish the recently reused lines from the not recently reused ones. When a line is loaded into the SLLC because of a miss, its NRR bit is set (Algorithm 2). When there is a hit (a reuse), its NRR bit is unset. If the NRR bits of all the lines in the not-being-used group become unset, no useful information about the reuse order remains. Thus, emulating the NRU behavior (but based on reuse), NRR sets the NRR bit of every line in the not-being-used group, except for the line receiving the last hit.

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

In summary, the differences between NRR and NRU are: i) NRR gives an additional level of protection to blocks present in the local caches, and ii) NRR unsets the bit associated with a line only on

hits, while NRU unsets the bit on hits and on misses. Therefore, the hardware cost of NRU and NRR is the same if we consider that the presence information in the local caches is provided by the coherence protocol.

ALGORITHM 2: Not-Recently Reused (NRR) algorithm

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

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

5.4.3 *Hardware complexity*

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

The LRR algorithm, as it is required by LRU, has to keep an order between all the lines present in a set. This objective can be achieved by using different approaches. Being n the associativity of the cache, two common implementations are: *i*) a matrix of n by n bits per cache set, *ii*) a counter of $log_2\ n$ bits per line. The division between reused and no-reused elements that LRR considers is achieved without adding any hardware storage. All the no-reused elements can be represented by one unique state. E.g. n zeroes on alternative *i*) or a counter equal to zero on alternative *ii*).

The NRR algorithm requires one bit per cache line to track reuse (the NRR bit), and another to record if the line is present in the private caches. However, note that this last bit of presence is already

implemented in the directory of an inclusive SLLC.[3] Therefore, NRR requires one bit per cache line, as NRU.

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

5.5 EVALUATION

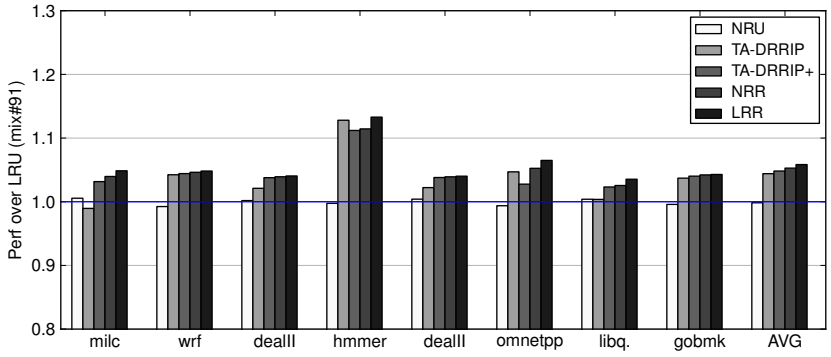
This section shows the evaluation of the LRR and NRR replacement policies which were explained during the previous sections of this chapter. In order to perform the evaluation, we have employed the methodology showed in Chapter 3. Concretely, next sections show results for 100 multiprogrammed workload mixes and the parallel applications were shown in Section 3.4.2.

The remaining of the section is organized as follows. Section 5.5.1 and Section 5.5.2 analyze the performance achieved by the proposed policies and compare them with LRU, NRU, and RRIP [23]. Section 5.3 gives insight into the behavior of LRR and NRR algorithms. Finally, Sections 5.5.6 and 5.5.5 describe the sensitivity of LRR and NRR performance to associativity and cache size, respectively.

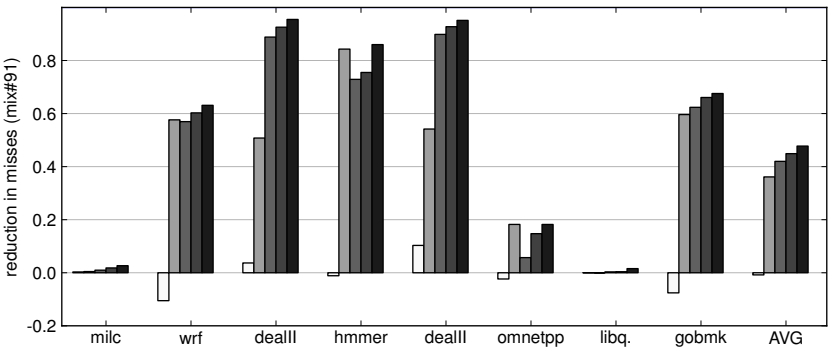
5.5.1 *Results for the baseline system*

Besides LRR and NRR, we also consider the replacement algorithms usually implemented in real processors (LRU and NRU), and a state of the art alternative (RRIP, [23]). We select RRIP as a reference because to the best of our knowledge it is the best proposal for inclusive SLLC caches and one of the few that suggest a reuse-based algorithm for this kind of SLLCs. Specifically, we implement thread-aware dynamic re-reference interval prediction (TA-DRRIP), the extension to shared caches proposed by the authors [23]. We also compare our proposals with an enhanced model of TA-DRRIP that protects SLLC lines while they are in the private caches, which we call TA-DRRIP+.

3 Evictions of clean lines on the private caches are non-silent.



(a) Performance



(b) Reduction in misses

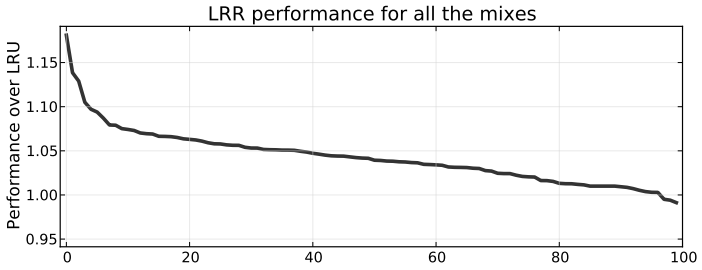
Figure 5.4: Performance improvement and miss reduction in the multiprogrammed workload #91 used as example in Section 5.2

Figure 5.4a shows NRU, TA-DRRIP, TA-DRRIP+, NRR, and LRR performance relative to LRU for the program mix used as an example in Section 5.2. The rightmost group of bars is the geometric mean of individual speedups. LRR improves LRU performance for all the applications in the mix, while NRR has the same effect with respect to NRU. TA-DRRIP is the only algorithm that causes losses in some applications (milc), while TA-DRRIP+ is better than TA-DRRIP in six of the eight applications and eliminates losses. On average in this mix, our proposals perform better than the other algorithms tested.

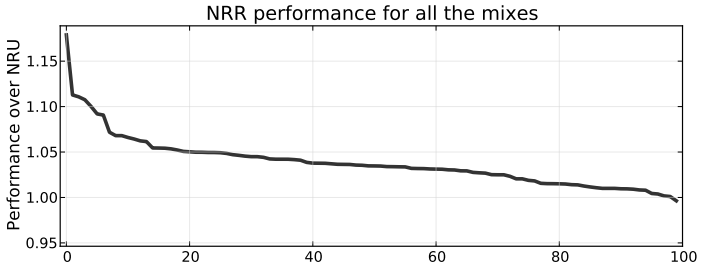
Figure 5.4b shows, for each application in the example mix, the percentage reduction in the number of misses achieved for each mechanism relative to LRU replacement. As it can be observed in the figure, LRR achieves the greatest reduction in all applications. NRR is the second best mechanism in six of the eight applications, and DRRIP in the other two (hmmer and omnetpp). Furthermore, DRRIP is the most irregular mechanism. First, it gets a much smaller reduction in the two instances of the application deall. Also, DRRIP does not achieve any reduction in milc and libquantum. These two applications have high miss ratios, pointing to a possible thrashing behavior. DRRIP correctly recognizes the lack of reuse and acts easing the eviction of their cache lines. However, our mechanisms work with a finer grain, identifying the few lines showing reuse, and giving them higher priority than the rest. As a result, NRR and LRR manage to reduce misses even in these applications (LRR eliminates 1.8% and 2.7% of misses, while NRR eliminates 0.5% and 1.6% in libquantum and milc, respectively).

Figure 5.5a plots all the 100 mixes on the horizontal axis. The different mixes are sorted by the LRR speedup over LRU. In the same way, Figure 5.5b plots NRR performance relative to NRU. LRR outperforms LRU in 97 mixes out of 100, while NRR is better than NRU in all but one of the mixes.

Figure 5.6 shows the mean performance of NRU, TA-DRRIP, TA-DRRIP+, NRR, and LRR relative to LRU for the one hundred workload mixes. On average, LRR improves LRU performance by 4.5%, while NRR outperforms NRU by 4.2%. On the other hand, TA-DRRIP and TA-DRRIP+ increase LRU performance by 3% and 3.3%, respectively.



(a) LRR compared to LRU



(b) NRR compared to NRU

Figure 5.5: Relative performance for all the mixes evaluated

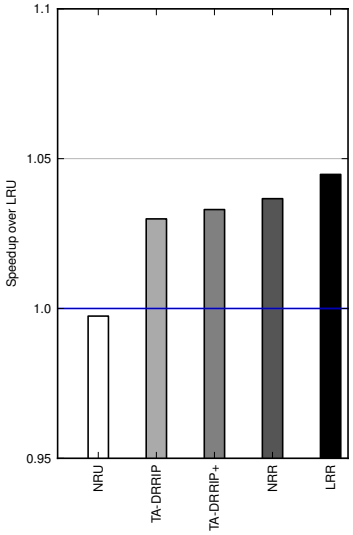


Figure 5.6: Summary of speedups

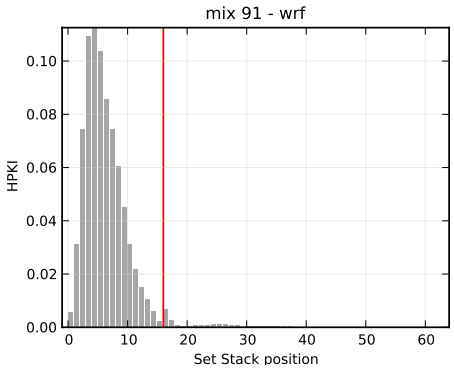


Figure 5.7: Stack profile for *wrf* with LRR as replacement policy

5.5.2 *LRR/NRR behavior*

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

Figure 5.7 shows again the stack profile of Section 5.2 for mix #91 under LRR. As can be seen, hits are concentrated towards the top of the stack, almost always at a distance of less than 16. Thus, by applying LRR, the SLLC is effectively keeping the cache lines likely to be reused, and this is why the SLLC performance improves. The NRR policy, not illustrated in the figure produces a similar behavior.

Figures 5.8 and 5.9 plot the evolution of the number of lines classified as being-used, non-reused, and reused (from bottom to top) in a sample SLLC set, over a short period of execution of mix #91, with LRR and NRR as replacement policies, respectively.

Under LRR, the boundary between reused and non-reused lines moves down each time a line is reused for the first time. As we pointed out in Section 5.4.1, this single-direction movement is counteracted every time a hit occurs on a reused block (the block moves to the being-used group), and each time a new block is loaded into the cache set (miss) and all the non-reused lines in that cache set are also being-used.

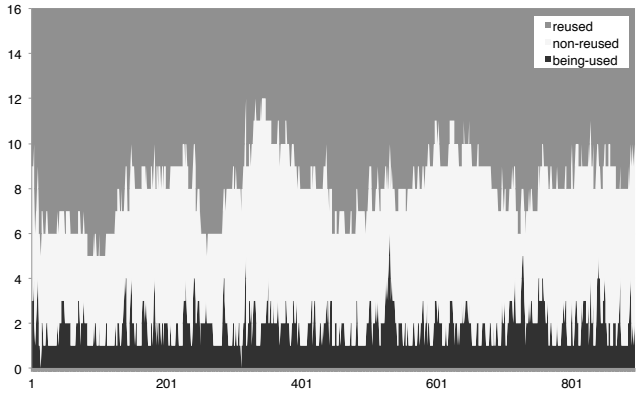


Figure 5.8: Temporal evolution of the group sizes in a cache set when LRR is used

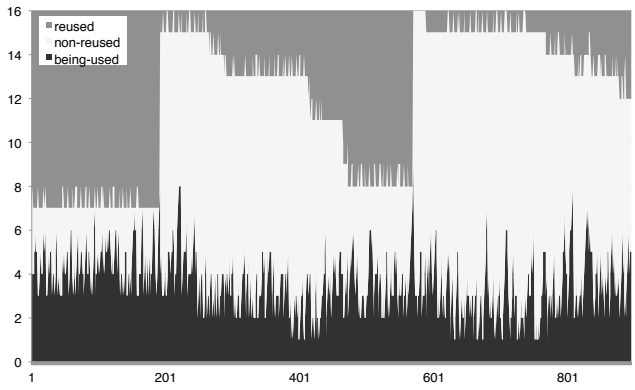


Figure 5.9: Temporal evolution of the group sizes in a cache set when NRR is used

Under NRR, when all the not-being-used lines become reused, the replacement algorithm itself converts all of them but one to non-reused lines. That is to say, when all the not-being-used lines have the reused bit set, then NRR unsets it for all of them except the line receiving the last hit. This behavior can be clearly seen at times 201 and 601 in Figure 5.9.

5.5.3 *Individual applications analysis*

To obtain insight into how the replacement algorithms affect individual applications, Figure 5.10 shows the distribution of speedups by application. The number of mixes in which each application appears is shown along the top of the graph. For each replacement policy (TA-DRRIP, LRR and NRR) five speedups are plotted, namely the minimum, the first quartile, the median, the third quartile, and the maximum.

For the mix #91, we saw (in Figure 5.4) how TA-DRRIP improved the performance of several applications, but also reduced it in one case (milc). In Figure 5.10, we note that this behavior is quite common. In 24 out of the 29 applications, TA-DRRIP performs worse than LRU in some multiprogrammed mixes, whereas LRR and NRR reduce this number to 12. Therefore, it can be concluded that reuse-based replacement is more fair than TA-DRRIP.

The imbalance introduced by TA-DRRIP may be due to the control mechanism deciding which replacement algorithm is used for each application. Specifically, TA-DRRIP uses Set Dueling [47] to identify the best suited replacement policy for each application, dynamically choosing between scan-resistant SRRIP and thrash-resistant BRRIP [23]. That is, if an application greatly reduces its miss rate with a given configuration, even at the cost of increasing the misses of other applications, the configuration that benefits itself will prevail.

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

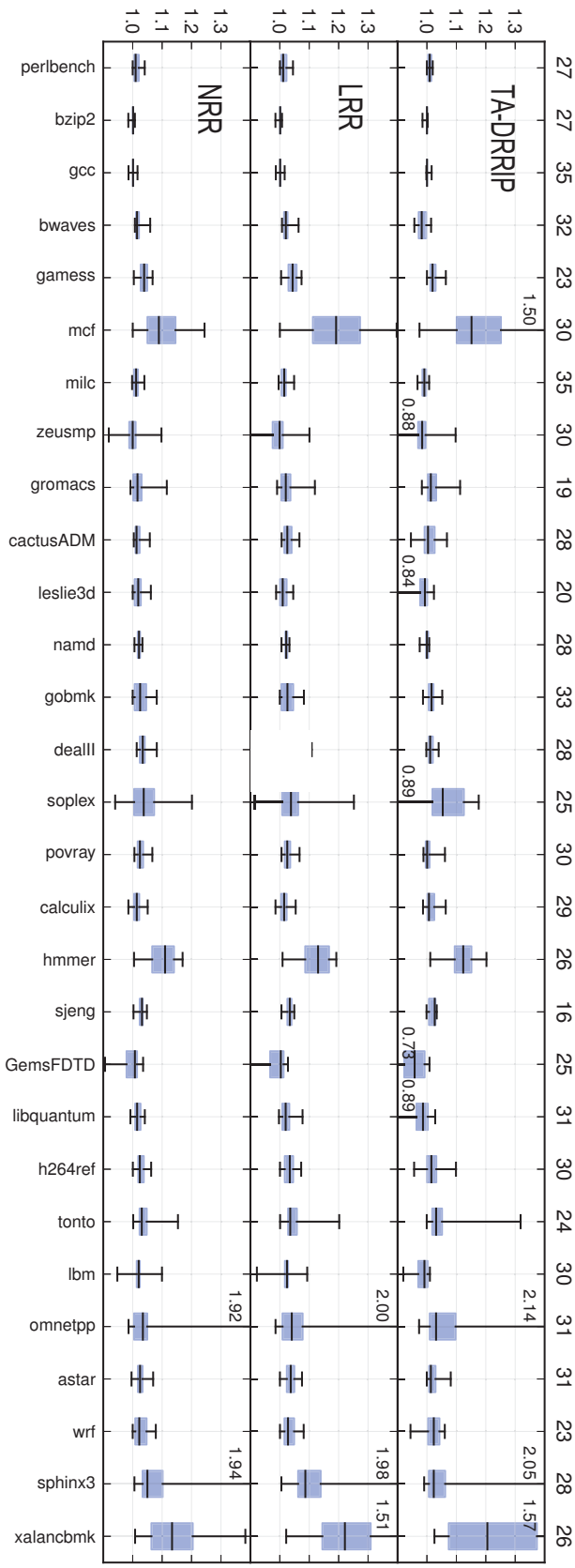


Figure 5.10: Distribution of speedups relative to LRU for all applications

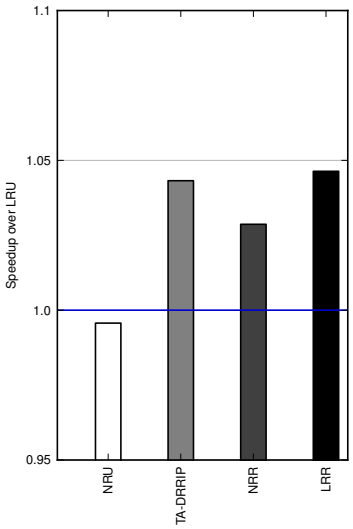


Figure 5.11: Performance comparison for five mixes of a SPEC2K6 subset in a four-core system

from combining the applications of the same SPEC 2006 workload (*cactusADM*, *sphinx3*, *hmmer*, *mcf* and *bzip2*).

Figure 5.11 shows the average speedup of NRU, TA-DRRIP, NRR and LRR over LRU for the five aforementioned mixes. Notably, in comparison to Figure 5.6, TA-DRRIP increases its speedup the most (from 1.02 to 1.04).

5.5.4 *Hardware complexity*

The NRR algorithm requires one bit per cache line (as shown in Subsection 5.4.3), while DRRIP requires N bits per cache line, being N the number of bits required to classify lines in segments. The aging logic for NRR is simpler than that of DRRIP. DRRIP aging requires incrementing the counters of all the lines in a set, whereas NRR aging only requires resetting the reuse bits to the lines not present in the private caches (using the presence bit vector). Moreover, TA-DRRIP requires a per-thread policy selection counter, and the logic for choosing a set dueling monitor. This logic decides whether a miss occurs or not in the (sampled) sets belonging to the monitor of the corresponding thread.

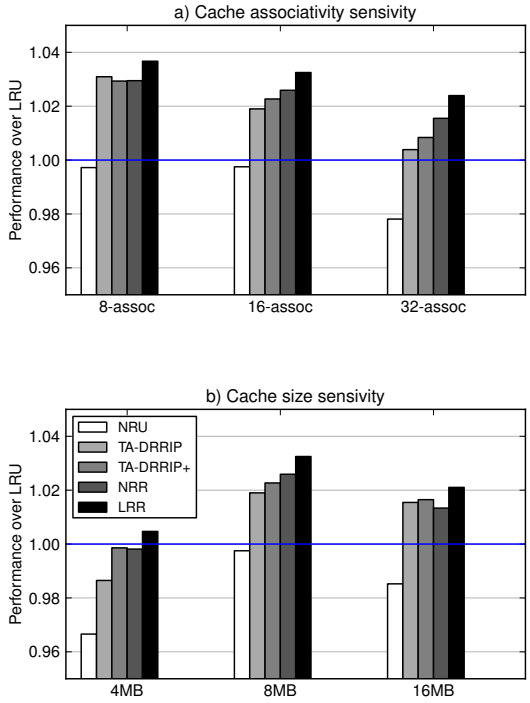


Figure 5.12: Sensitivity to the cache associativity and cache size

5.5.5 *Sensitivity to the SLLC associativity*

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

Figure 5.12a shows three groups of bars for the three cache associativities. Each group have five bars which represent NRU, TA-DRRIP, TA-DRRIP+, NRR and LRR mean performance relative to LRU. The speedup with respect to LRU decreases with increasing associativity. LRR shows the best performance in all the associativities, while the low-cost proposal, NRR, is the second best option for associativities 16 and 32.

5.5.6 *Sensitivity to the cache size*

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

Figure 5.12b shows three groups of bars for the three cache sizes. Each group have five bars which represent the mean performance of NRU, TA-DRRIP, TA-DRRIP+, NRR and LRR relative to LRU. The speedup with respect to LRU decreases for both 4 and 16 MB cache sizes. Moreover, for a 4 MB cache, all replacement algorithms lead to poorer performance than LRU except LRR. LRR gives the best performance in all the cache sizes, while the low-cost proposal, NRR, is better than TA-DRRIP for 4 MB and 8 MB caches.

5.6 CONCLUDING REMARKS

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

Further, a heavily referenced line with a short reuse distance may remain in private caches for a long time, steadily losing position in the LRU stack and eventually being evicted. Thus, if an SLLC follows an inclusive scheme such hot lines will be invalidated and fetched again and again. Consequently, traditional replacement algorithms based on recency such as LRU and NRU are poor choices for inclusive SLLCs.

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

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

For an eight-core system with two private cache levels and an inclusive SLLC, we have found that LRR outperforms LRU by 4.5% (97 out of 100 mixes) and NRR outperforms NRU by 4.2% (99 out of 100 mixes). A detailed comparison with RRIP [23], a recent SLLC replace-

ment proposal, indicates that LRR and NRR give 1.5% and 0.89% better performance, respectively. Additionally, we have shown that our algorithms are more fair than TA-DRRIP and that similar conclusions can be drawn considering a range of different associativity values and SLLC sizes.

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

6

REUSE CACHE

SUMMARY

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 contribution, we propose the reuse cache, a decoupled tag/data SLLC which tries 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 non-selective data allocation policy to exploit the 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 will show that a reuse cache with the 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.

6.1 INTRODUCTION

Shared-memory chip multiprocessors (CMPs) reached high-performance and high-throughput computing markets past decade. Nowadays they enjoy widespread acceptance through all the market segments: cloud servers, desktop, embedded and mobile SoCs. 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. But this trend is unlikely to continue in the future, obligating many-core systems to include lower cache-to-core ratios. [38]

It is common that all the cores in the CMP share a last level of cache memory (SLLC), this SLLC is called to perform two key tasks, managing coherence and storing valuable data. Regarding the second task, the SLLC pursues to feed with low latency the misses of the private cache levels, getting the most of the scarce off-chip bandwidth with the external main memory.

Previous work point to conventional SLLCs as effective but inefficient because their contents are mostly useless. In fact, a dominant fraction of the SLLC lines is dead, meaning they will not be requested again before being evicted, to the point that some lines are just used once, being useless during their entire stay in the SLLC [27, 31, 47]. Managing SLLC contents efficiently is difficult because the classic properties of temporal and spatial locality that govern private cache levels become dissolved in the reference stream observed by the SLLC [23, 47]. A great deal of research has addressed the problem from several angles, with the aim of improving the SLLC hit ratio. Many proposals on decreasing the number of conflicts in the SLLC sets [48], improving replacement decisions [21], and prefetching useful data [6], among other techniques, have been proposed.

Despite the effort, 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 [23, 2]. Moreover the fraction of live blocks in the SLLC does not increase much with all the previous techniques. Such facts encouraged us to look for a different objective: to draw a solution where the SLLC area is drastically reduced without compromising performance. A solution in this direction would be very interesting, since the saved area could contribute to cut manufactur-

ing costs, reduce power consumption, or decrease the cache per core ratio, allowing an increase on the core count 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 cache. However, recent proposals note that temporal locality is not always followed and consequently the precise insertion point in the LRU stack replacement is dynamically varied [21, 23].

Other recent works show that is not a good idea to base SLLC replacement only on temporal locality exploitation, given that local caches are already doing that job [22, 23, 7]. In fact, a recent study explicitly notes the reference stream entering the SLLC has reuse locality instead of temporal locality. In short, reuse locality states the second reference to a line is a good indicator of forthcoming reuse, and the recently reused lines are more valuable than other lines reused a long time ago [2].

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 less entries than the tag array, because they only contain 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 a 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 6.2 provides experimental evidence of the small fraction of live lines in the SLLC and the concentration of hits in the reused lines. Section 6.3 explains the reuse cache organization, replacement algorithms and coherence protocol modifications, giving implementation details and costs. Section 6.4 presents the experiments, discusses results and compares with two state of the art proposals such as DRRIP [23] and NCID [69]. Finally, conclusions are drawn in Section 6.5.

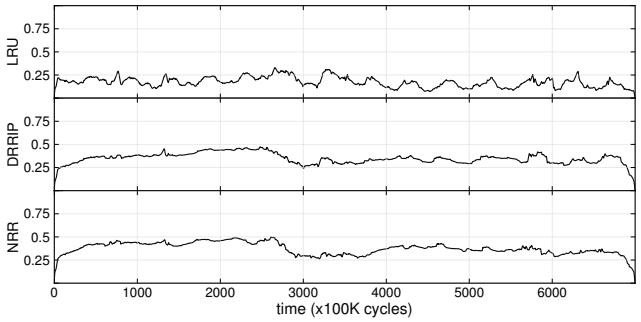
6.2 MOTIVATION

In this section, we analyze the behavior of a representative multiprogrammed SPEC CPU workload[1] running in the hierarchy of an eight-core chip 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).

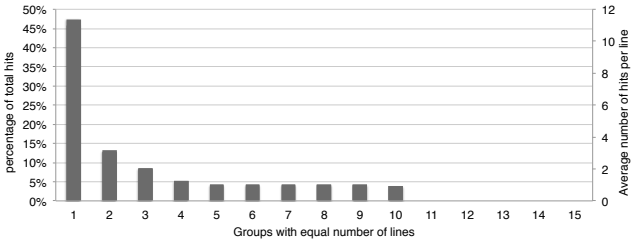
THE FRACTION OF LIVE SLLC LINES IS VERY SMALL. Figure 6.1a shows the fraction of the SLLC lines that are live over the course of the execution of the example workload. We simulated 700 million cycles and took a sample every 100K cycles. In each sample, we tracked the future SLLC references for every line, computing 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. By using Dynamic Re-Reference Interval Prediction (DRRIP)[23] and Not Recently Reused (NRR)[2], two state of the art replacement policies, this average increases to 34.8% and 37.9% respectively. The average for the 100 workloads used in the experiments in Section 6.4 is 16.2%, 35.9% and 40.0% for LRU, DRRIP and NRR respectively. In other words, 83.8% of the lines stored in the SLLC do not provide any benefit, and replacement algorithms proposed in the literature only are able to reduce this percentage to about 64.9%. Hence,

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



(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 a 0.5% of the loaded lines

Figure 6.1: Line usage patterns in a conventional 8 MB SLLC during a simulation period of 700 Mcycles

it can reasonably be argued that the extra space, energy and time associated with those dead lines could be saved without compromising performance.

HITS COME FROM A SMALL SUBSET OF LINES. Figure 6.1b shows the contribution to the total number of hits of a given group of lines during their stay in the SLLC. In order to obtain this distribution, we run the simulation and, just after each line is evicted from the SLLC, we insert 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 is loaded multiple times, i.e., it has multiple generations [25], its hit count will be inserted multiple times in the sorted list.

Once the simulation ends, we break 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 6.1b 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.

6.3 THE REUSE CACHE DESIGN

Starting from a conventional cache, we can try 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. Besides, the tag array should also record the lines present in the private levels (directory inclusion), so that coherence management is made simple [4, 22, 69]. Finally, in order to detect reuse, the tag array should maintain some usage history of the recently used lines, lines that may not be in the data array nor in the private levels.

Having more entries in the tag array than in the tiny 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 data associated. A hit in the tag array with no data associated detects a reuse. Thus, the line is read again from main memory and loaded in the private cache and SLLC data array at the same time. In order to take advantage of reuse locality, replacement in the data array is based on recency. When a line is from the 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. On the other hand, tag replacement is designed to protect both private cache lines and recently reused lines.

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

6.3.1 *Organization*

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

Previous work proposes to decouple tag and data arrays having the same [9], or different number of entries [48, 50, 52, 69]. Decoupling with the same number of entries allows both arrays to be shaped differently, for instance enabling the concept of distance associativity [9]. In any case, all proposals rely on relating the entries of both arrays by means of pointers. Some proposals need only *forward pointer*s from the tag array to the corresponding data lines, if any [9, 50, 69]. Other proposals need only a *reverse pointer* which links each data line to the corresponding tag [52], while some others require both kind of pointers [48].

An alternative organization, which allows to eliminate 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 an associated data). This organization involves moving tags between the ways with and without data [37, 69].

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 one other the entries of both arrays. Figure 6.2 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 the replacement in the data array. By increasing associativity, the replacement algorithm has more options to choose a victim. The data array associativity also has a small impact

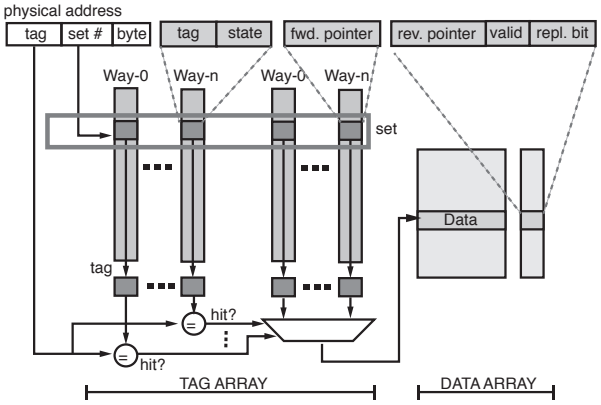


Figure 6.2: Reuse cache overview

on hardware cost. By increasing associativity, the size of the pointers stored in tag and data arrays also increases. Section 6.3.3 will detail the data array organization and Section 6.3.5 and Section 6.4.1 analyzes the influence of the data array associativity in the reuse cache size and performance, to conclude that the impact of data array associativity is very limited both in cost and performance.

6.3.2 *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 tradeoffs it brings [22]. In this contribution we adopt *not recently reused (NRR)* (Section 5.4.2), a replacement policy based on *not recently used (NRU)* [43]. 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 among 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.

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

6.3.3 *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 or equal than 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 ($\log_2$ the number of tag array sets bits - $\log_2$ the number of data array sets bits). For instance, a data array with only one set (fully associative) requires for each forward pointer $\log_2$ the number of data array entries bits. 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, the NRU performance decreases for high associativities. Thus, for the fully associative case, a suitable alternative we have tested is the low-cost Clock algorithm introduced by Cor-

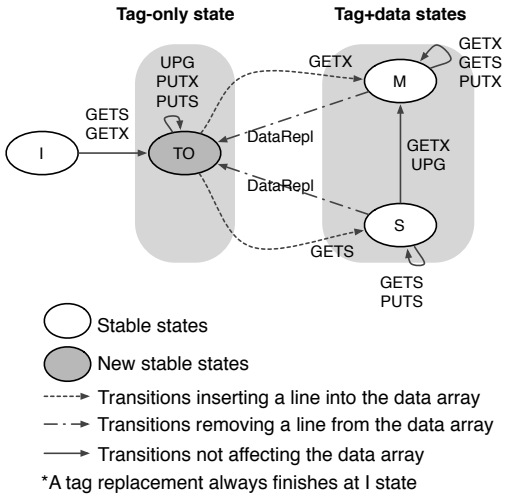


Figure 6.3: Functional description of the TO-MSI example coherence protocol

bató [12]. The implementation cost of both NRU and Clock is one bit per line.

Under Clock replacement the data array of the reuse cache works as a circular queue[2]. Each line in the circular queue has an associated bit called the *used* bit. This bit is unset when the line is inserted into the data array and it is set when the line is read (the line receives a *hit*). When a line is inserted into the data array, the replacement algorithm selects the oldest line as the victim. If the circular queue is full and the victim line is valid and has the used-bit set, this line is promoted to the first position and its *used* bit is unset. This operation only requires the queue pointer to be incremented and it can be carried out in the background after an insertion. We observed that normally there is a small number[3] of victim lines to promote but the victim search can be stopped if an insertion arrives before finding a victim.

When evicting a data line, the corresponding forward pointer in the tag array has to be invalidated. This corresponding tag array entry is located by following the reverse pointer of the just invalidated line (Figure 6.2).

2 Given that tag array replacement may force the invalidation of data array entries, "holes" may appear in our simple circular queue implementation. The effect of these has been measured and is minimal.

3 This number is, on average, lower than 2.

6.3.4 *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 6.3 outlines an example coherence protocol based on the MSI protocol[4] [13], which is able to work with decoupled tag and data arrays. Table 6.1 explains the states and events of the protocol. In this description, neither replacement nor the external requests are represented. In every state except *I*, private caches may have copies of the line or not. 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 contain lines in the data array; and *tag-only* (a single state in this simplified version of the protocol) contains lines that are not present in the data array.

Transitions between both groups always imply lines getting 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 labelled with the *DataRepl* event, coming out of M and S, represent these state transitions.

4 For the sake of clarity, a simple protocol is shown here. In our evaluation, we rely on 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.

Component	Conv. 8MB	RC-4/1 Fully-assoc.	RC-4/1 16-assoc.
Tag	21	22	22
Coherence	4	5	5
Full-map vector	8	8	8
Replacement	1	1	1
Fwd. pointer	-	14	4
Total tag entry (bits)	**34**	**50**	**40**
Data	512	512	512
Valid	-	1	1
Replacement	-	1	1
Reverse pointer	-	16	6
Total 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 6.2: Hardware cost

6.3.5 *Hardware Cost*

In this section, we compute the reduction in the total number of bits, 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 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 a 16-way and a fully associative data array organizations for the reuse cache.

The conventional cache is 16-way associative, 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[5]). The data array requires 512 bits per line. Overall, the conventional cache needs 69888 Kb (see Table 6.2).

5 Although LRU has been used as the replacement policy of the conventional cache in Section 6.4, NRU has been considered here to not bias the comparison.

The reuse cache has a 1 MB data array and a tag array with the same number of entries than a 4 MB conventional cache (RC-4/1 in the Table 6.2 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[6]. The forward pointer requires 14 bits for the fully associative (16K-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 a 6.8% less bits than the fully associative. Regarding the 8MB 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).

6.4 EVALUATION

In order to perform the evaluation, we have employed the methodology showed in Chapter 3. Concretely, next sections show results for 100 multiprogrammed workload mixes and the parallel applications were shown in Section 3.4.2.

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 6.4.3 and Section 6.4.4 we give insight into the reuse cache behavior by analyzing the percentage of lines not entering in the data array and the number of live lines when reducing the reuse cache size. Next, in Section 6.4.5 we compare the reuse cache with DRRIP [23], a state of the art replacement algorithm, and NCID [69], a recent proposal of a decoupled tag-data cache. Finally, in Section 6.4.6 we analyze the behavior of the reuse cache when running parallel applications.

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

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 64K tags (4 MB / 64) organized in 4K sets (64K 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 is the reuse cache outlined in Table 3, having a tag array equivalent to a 4 MB conventional cache with 1 MB data array.

With respect to a conventional cache with the same number of sets, the access time of the tag array increases due to the added forward pointer and the mux to select the pointer, see Figure 6.2. However, when comparing reuse and conventional caches, both the tag and data arrays of reuse caches are always smaller than those of conventional caches[7]. 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. Also, 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.

6.4.1 *Data array size and associativity*

Figure 6.4 shows performance of a reuse cache with 8 MBeq tag array and varying the data array size from 4 MB (RC-8/4) to 512 KB (RC-8/.5) and the data array associativity among 16, 32, 64, 128 and fully associative. Each bar represents average performance relative to the baseline for the 100 workloads mixes.

In general, performance varies very slightly and unevenly for associativities between 16 and 128. The reuse cache with fully associative data array achieves better results for all sizes. However, the differences are not significant. For instance, the difference between a 16-

7 In the performance comparisons we vary the number of tag sets, and only one of the cases shows a number of sets equal for the reuse and conventional caches. However, this case has never been chosen as a suitable design point.

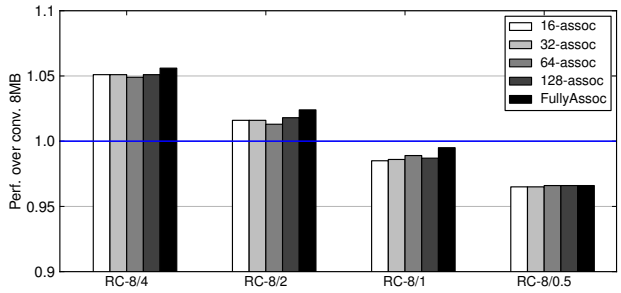


Figure 6.4: Average speedup over the baseline for reuse caches of various data array sizes and associativities. Tag array size and associativity are 8 MBeq and 16, respectively.

way associative and fully associative varies from -0.1% for RC-16/8 and +1% for RC-4/1. We can conclude that the fully associative and the set-associative organizations 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 noted, the remaining experiments are carried out with fully associative data arrays.

Regarding the size of the data array, a reuse cache with 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 worse than the baseline cache (-0.5%).

6.4.2 *Tag array size*

In this section we study the tag array size that achieves the best performance for each size of the data array. Figure 6.5 shows the relative performance of a reuse cache with respect 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.

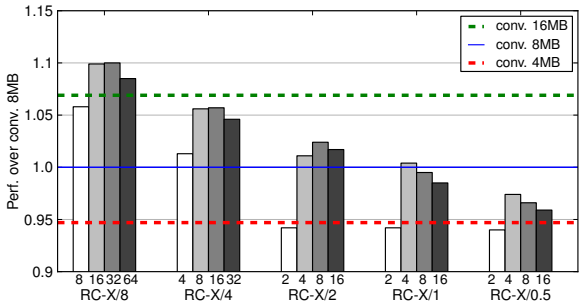


Figure 6.5: Average speedup over the baseline for reuse caches variyng the tag and data array sizes. Tag array associativity is 16.

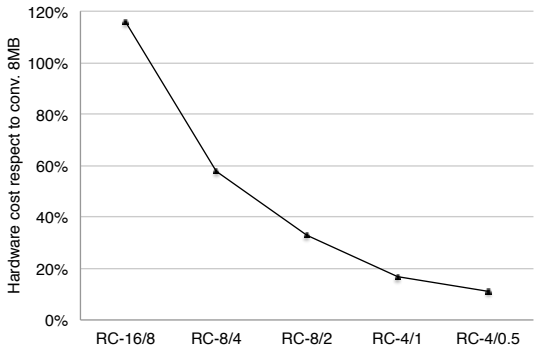


Figure 6.6: Storage budget of the best reuse caches, relative to a conventional 8 MB cache.

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, whose size 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. Besides, 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 between RC-16/4 and RC-8/4. Hence, for the remaining sections, the reference sizes of the reuse cache replacing a conventional 8 MB cache will be: RC-8/4, RC-8/2, RC-4/1 and RC-4/0.5.

Figure 6.6 shows the total number of bits of those reuse caches relative to a conventional 8 MB cache; RC-16/8 has been added for completeness. We can choose RC-4/1 as the smaller reuse cache performing better than a conventional 8 MB cache; indeed, RC-4/1 requires half the tags, one-eighth the data, and only spends 16.7% storage of the conventional 8 MB cache.

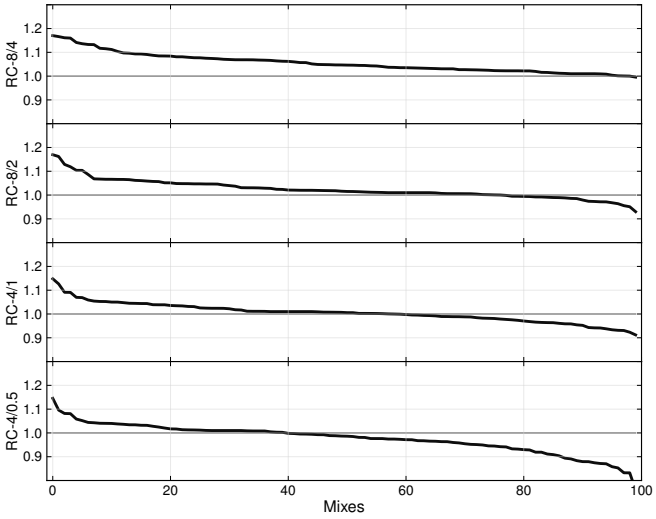


Figure 6.7: Performance of the selected reuse cache configurations for all the 100 multiprogrammed workloads, relative to the 8 MB baseline. The data array ranges from 4 MB (RC-8/4) to 512 KB (RC-4/0.5).

Figure 6.5 also shows relative performance of 16-way LRU in 4 MB and 16 MB caches with respect to the 8 MB baseline (red and black lines). We could replace both conventional caches with the smaller reuse cache giving some performance advantage. So, 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).

In order to analyze the result variability, Figure 6.7 plots the reuse cache speedups for every workload, varying the reuse cache size. We only show the previously selected configurations having the best performance/size trade-off. In each plot, the different workloads are ordered along the horizontal axis according to their speedup.

As can be seen, the speedup variability 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 in 64 out of 100 workloads, reaching speedups from 0.82 to 1.14, only four and two workloads suffer losses and improve their performance over a 10%, respectively.

	RC-8/4	RC-8/2	RC-4/1	RC-4/0.5	Conv.
Mean (%)	93	93	95	95	0
Minimum (%)	81	81	89	89	0

Table 6.3: Mean and minimum percentage of lines not entering in the data array with respect to tags entering in the tag array for different reuse cache and conventional configurations.

6.4.3 *Data lines entering in the data array*

Table 6.3 shows, for the one hundred workloads, the mean and the minimum percentage of lines not entering in the data array with respect to tags entering in the tag array. Of course, in a conventional cache this percentage is zero because tags and data are always allocated together. However in a reuse cache the selective allocation policy discards most of data allocation. 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 in average 93% and 95.4%, respectively. Even the most demanding workloads discard more than 80% of the data lines. So we can conclude that the reuse cache is very effective in 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.

6.4.4 *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 6.8 shows the fraction of live lines during the execution of the example workload in the best reuse cache configurations. We also plot data for the 8 MB conventional cache with LRU, DRRIP [23] and NRR replacement policies.

As we see, the fraction of live lines varies greatly as a function of the size of the reuse cache. It is apparent that the reuse working set varies over time, and each reuse cache configuration is able to house it to a greater or lesser extent, depending both on storage capacity (data array size) and ability to capture reuse (tag array size).

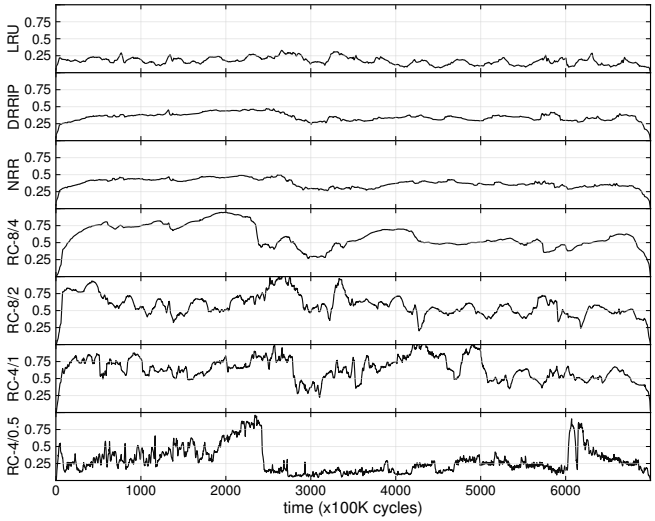


Figure 6.8: Instantaneous fraction of live lines during the execution of the example workload, for 8 MB LRU, DRRIP, NRR, and for the selected reuse cache configurations.

As we will show below in more detail, lines live longer in reuse caches but the instantaneous fraction shows a more variable shape. As we consider lower capacities, the fraction of live lines shows higher-frequency variations. This is because smaller reuse caches become more sensitive to the line bursts coming from individual cores, simply because the relative impact of allocating a given number of lines in a row may be higher as the storage capacity decreases.

Another effect we notice concerns the limitations imposed by the data array size for a given tag array. As an example we can consider the first sharp decline in RC-4/0.5 around 250 Mcycles; here a burst of lines apparently showing reuse may have entered into the reuse cache, probably overflowing the limited capacity of RC-4/0.5 and causing a net drop in the duration of lives. In contrast, when increasing the data array (RC-4/1) this decline is not experienced.

Figure 6.9 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 LR, DRRIP [23] and NRR replacement policies. The average percentage of live lines in the baseline cache is only 16.1% with LRU replacement (*LRU* in Figure 6.9), 35.9% with DRRIP and 40.0% with NRR, consistent with the analysis presented in Section 6.2 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

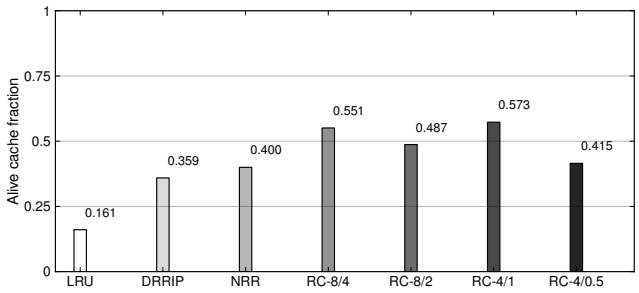


Figure 6.9: Average fraction of live lines for 8 MB LRU, DRRIP, and NRR conventional caches, and for the selected reuse cache configurations.

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, RC-8/2 obtains a number of live lines only a 9% lower than the baseline cache. RC-4/1, while keeping a significantly lower number of live lines, still improves 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 more hits.

For RC-4/0.5, the percentage of live lines drops to 41.5%. Here the tag array is oversized regarding the data array. Many lines are tagged for a potential reuse but they can not be stored in the data array, leading sometimes to line thrashing. As we saw in Section 6.4.2, 4 MBeq is the minimum tag array size greater than the aggregated capacity of the private levels. Certainly we could test a smaller size by decreasing associativity, but this change would make the comparison harder with the other reuse cache configurations.

6.4.5 *Comparison with alternative state of the art proposals*

Throughout previous sections, results have been reported as speedups 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 worse performance. However, in this section we also compare the reuse cache with both a conventional cache fit-

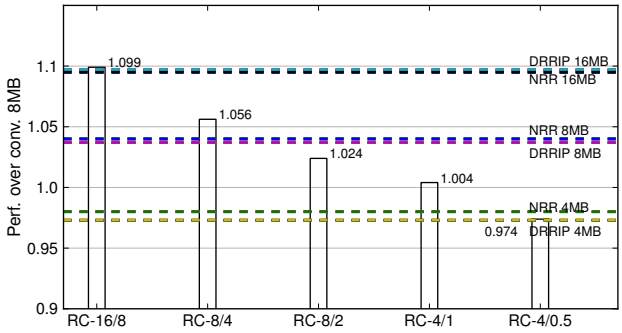


Figure 6.10: Average speedups of DRRIP and reuse caches.

ted with state of the art replacement algorithms and an alternative decoupled cache organization that also allows to reduce the data array size.

COMPARISON WITH RRIP [23] AND NRR [2]. Thread-aware-DRRIP is a state of the art replacement algorithm for SLLCs (see Section 5.3). While NRR was presented in Section 5.4.2 as one of the contributions of this thesis. In this experiment we want to observe how much benefice the reuse case is still bringing compared to a conventional cache, despite the conventional cache is using a state of the art replacement algorithm. Figure 6.10 compares conventional caches operated with TA-DRRIP and NRR (represented as horizontal lines) with several reuse cache configurations (represented as bars). The shown results are the average speedups over the 8 MB baseline cache using LRU for all the 100 workloads mixes.

TA-DRRIP replacement improves the conventional 8 MB cache performance in a 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-eight the data array (RC-4/0.5) with similar performance.

COMPARISON WITH NCID [69]. NCID adds 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. Their authors evaluate the NCID architecture supporting a selective allocation policy to address transient data, and compares it with a conventional cache with bi-modal fills [47] in terms of miss rate reduction.

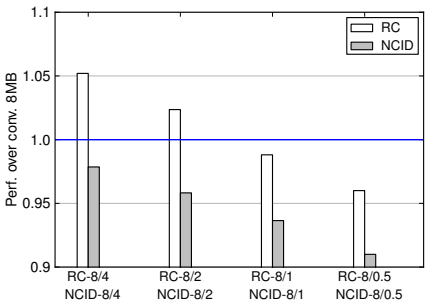


Figure 6.11: Average speedups of NCID and reuse caches.

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 organization requires an equal number of sets in the tag and data arrays. So, reducing the data array size implies reducing the data array associativity. As an example, a NCID cache with a 16-way, 8 MBeq tag array implies a 1 MB data array with only 2 ways. So, in order to perform a fair comparison we have to choose reuse caches having the same number of sets and associativities in the data array.

Figure 6.11 compares reuse cache against 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 mixes.

By reducing the data array size, any NCID settings match the performance achieved by the baseline 8 MB cache. For all data array sizes, the reuse cache get better performance than NCID with relative gains 7.0%, 6.4%, 5.2% and 5.3% for data array sizes 4 MB, 2 MB, 1 MB and 512 KB, respectively.

6.4.6 *Parallel workloads*

In this section, we analyze the behavior of our proposal when running parallel applications. As it was explained in Section We have selected the five applications of the PARSEC [5] and SPLASH-2 [66]

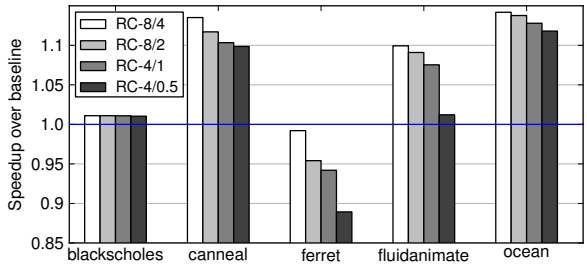


Figure 6.12: Speedup of reuse cache over the baseline for five parallel applications, with data array sizes from 4 MBytes (RC-8/4) to 512 KBytes (RC-8/0.5)

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; their MPKIs are 4.5, 3.5, 1.3, 1.7, and 13.4, respectively. We utilize the *simmedium* input set for PARSEC applications and a 1026x1026 grid for Ocean. For PARSEC applications a checkpoint is created in the parallel phase. The cycle-accurate simulation starts at those checkpoints, warming the memory hierarchy for 300 million cycles, and then collecting statistics for the next 700 million cycles. Ocean is run to completion but performance statistics are only taken in the parallel phase.

Figure 6.12 shows, for the five parallel applications, the relative performance of the reuse cache with data array sizes from 4 MBytes (RC-8/4) to 512 KBytes (RC-8/0.5) with respect to the baseline SLLC. Only ferret suffers a loss in performance when using a reuse cache with respect to the baseline cache. This loss varies between 1% with RC-8/4 and 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* show speedups of more than 10%).

6.5 CONCLUDING REMARKS

The reference stream observed by the SLLC of a CMP exhibits little temporal locality but, instead, it exhibits reuse locality. As a consequence, a dominant fraction 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 propose the reuse cache, a 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 1 MB of 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, seeking to meet other design goals in relation to some chip area, performance or energy tradeoff.

Part IV

CONCLUSION

This last part of the thesis only includes one chapter. This chapter includes the general conclusions of this thesis, future work continuing the contributions presented during this dissertation and the publications where such contributions appeared.

7

CONCLUSIONS

SUMMARY

This last chapter exposes general conclusions about the contributions shown during the present dissertation. The chapter also shows the publications where the contributions of this thesis have been published and highlights future directions to continue with work of this thesis.

7.1 CONCLUSIONS

The speed gap between cpu and main memory has been increasing during the last forty years and nothing says this tendency is going to change in the near future. In order to mitigate such speed gap a hierarchy of cache memories has been traditionally included between the main memory and the cpus. This hierarchy relies on temporal and spatial locality to provide the cpus with data and instructions, the goal is to offer this information with a low average latency.

In the multi-/many- core scenario where we are now, hierarchies of memory are more complex than ever, including some levels of cache accessible only by each core and, as we have been seeing during this dissertation, a last level that is shared among all the cores present in the system. Moreover, Moore's law is still strictly complied providing the architects with a bunch of new transistors to spend on the next processor generation. New silicon is precisely employed in many of the new designs to amply each of all the levels of the memory hierarchy. This hierarchy is already occupying an important part of the chip die, e.g. the last-level cache of an *intel i7* processor takes away roughly the 50% of the total chip die area.

Until now, last-level caches were including a constant, or even increasing, number of megabytes of shared last-level cache per core, but such tendency is not likely to continue in designs with some dozens of cores if the access latency to this last level wants to be maintained reasonable. Thus the way to design the hierarchies of the future devices has to pass through improving the efficiency of such hierarchies. This thesis has focused on improving the efficiency of the shared last-level cache, exploring the way on two different directions: 1) improving performance and 2) reducing hardware storage.

1) Regarding performance, this dissertation makes contributions to improve the performance of two mechanisms that critically affect the cache performance: hardware prefetching and replacement.

Hardware prefetching may harm the system performance when it is used in an uncontrolled manner in the hierarchy of memory of a CMP. Prefetches issued by one core can evict contents of other cores affecting their performance. Given the broad ecosystem of applications can be found running in a CMP, prefetching should be controlled but controlled in a per-core fashion, always with maximum fairness and overall system performance as objectives. A low-

cost controller has been proposed with such objectives. This controller is able to improve system performance by 27% (harm. mean of speedups) respect to a system with uncontrolled prefetching.

This dissertation has stated a property called *reuse locality*, which is linked to the stream of references arriving to the SLLC and says that i) if a line receive a hit on the SLLC is highly likely that line will receive another hit in the future, and ii) recently reused lines are more useful than lines reused before.

Reuse locality is used to propose two new replacement algorithms for the SLLC, which are transformations of two standard ones. These standard algorithms, LRU and NRU, were designed to exploit temporal locality while ours, LRR and NRR, are designed to exploit reuse locality. This dissertation has shown during evaluation sections that our contributions are consistently better than both base algorithms and a proposal, DRRIP, from the state of the art.

2) Finally, this dissertation proposes a new SLLC organization that relying on the reuse locality implements a very restrictive data array insertion policy. This insertion policy only stores into the SLLC data array, data that has shown reuse. Evaluation has shown this design is able to offer drastic reductions on the SLLC hardware storage requirements, achieving to reduce up to 84% of the SLLC hardware storage while maintaing average performance untouched.

7.2 PUBLICATIONS

This section shows the publications of this thesis, which have already been referenced in the corresponding sections.

- J. Albericio, R. Gran, P. Ibáñez, V. Viñals, and J.M. Llabería. "ABS: A low-cost adaptive controller for prefetching in a banked shared LLC". ACM Transactions on Computer Architecture and Optimization (TACO): Special Issue on "High-Performance and Embedded Architectures and Compilers". Volumen: 8, Issue 4. Pp. 1-19. January, 2012.

- J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería. "Exploiting reuse locality on inclusive shared last-level caches". ACM Transactions on Computer Architecture and Optimization (TACO). Volumen: 9, Issue 4. Article 38. January, 2013.

- J. Albericio, P. Ibáñez, V. Viñals, and J. M. Llabería. "The Reuse Cache: downsizing the shared Last-level cache". University of Zaragoza, technical report TR-12-02.

7.3 FUTURE WORK

7.3.1 *Replacement*

INCLUSIVENESS In spite of everything suggests reuse locality property will be present in all kind of SLLC, the behavior of our replacement policies, NRR and LRR, would have in a hierarchy with different inclusion properties is a evaluation still to perform.

PARTIAL INFORMATION Our replacement policies have been only evaluated in a hierarchy with a full-map directory. An interesting study would be to observe if the performance of the replacement policies is affected because of the use of a coherence system that considers partial information, meaning that directory information is not necessarily updated. This interaction arises due to our replacement schemes take information about the presence of lines directly from the full-map directory.

PREFETCH AND REPLACEMENT Only a recent work [Martonosi + Jean Wu] has studied the interaction between replacement and prefetch. Moreover, this previous work only focuses on a multiprogrammed scenario, thus a scenario where parallel applications run is still something to study,

SHARING AND REPLACEMENT The interaction between sharing and replacement is still something to study. Most of research about replacement policies has focused on multiprogrammed scenarios where unbalance and competition between different applications are the problem to solve. In a different scenario where parallel applications run (alone or running along other), proposed techniques may not be valid anymore or at least they should be reviewed and evaluated.

7.3.2 *Reuse Cache*

Regarding to future work relying on the reuse cache design:

DYNAMIC DATA ARRAY SIZE Dynamic total size To have a decoupled organization offers an additional flexibility that could be employed to different things. One of these things is that data array becomes a candidate to switch some of its parts off. Instead of being the total SLLC storage the target, a reuse cache without data array downsizing (or minimum) could be employed as base design. Then, parts of the data array could be arbitrarily switch on/off depending on the system load. The level of system load could be easily detected by using the SLLC tags. Dimensions of the data array could be downsized by cutting the set associativity or by reducing the total number of sets.

DYNAMIC TAG ARRAY SIZE The evaluation of the reuse cache showed that different applications require a different reuse detection capacity. The optimal reuse detection capacity depends not only on the application but also on the data array size where the reused data are stored. This topic has still to be studied but classical dynamic optimization approaches like hill-climbing or/and set-dueling can be applied. These dynamic schemes may be employed to vary the number of elements in tag array sets.

TWO MISSES Given that in the reuse cache, if not predictor is used, all the hits will pay two misses in advance, some kind of reuse predictor could be used. Wu et al. [67] proposed some predictors to improve the re-reference interval prediction of the RRIP mechanism [23]. Taking into account those policies and ours have the same last goal, it seems reasonable that such predictors are highly likely to work well on the reuse cache scheme.

VARIABLE GRAIN A recent work [Amoeba] proposes to implement the cache as a contiguous vector of non-constant-size elements. Each element has included along its tag, what is the size of the data field. Taking this concept to the end, such elements could include empty data fields and only to include the data when the element has shown reuse. At the same time, the reuse locality property could be studied in a sub-block fashion.

The reuse cache as it is proposed on this dissertation considers inclusion enforcement in both tags and data. Given that, the reuse

cache is proposed for a low cache-per-core ratio, in order to maximize SLLC storage, it seems reasonable to think in an exclusive reuse cache scheme. This inclusion enforcement could be relaxed in one or both tags and data arrays. On the reuse cache proposed in previous sections, once a data has been classified as reused, it will be inserted at the same time into the SLLC reuse data and into the local caches.

EXCLUSIVITY: DATA ARRAY If we considered to implement local caches and SLLC to be disjoint sets, a cache line would be only put into the SLLC data array once this data has shown reuse at the SLLC and such cache line has been evicted from the local caches. Considering this scheme, tags array would still maintain inclusion, simplifying coherence but maximizing at the same time the SLLC data array storage. Besides, an exclusive data array would require some mechanism to send information from one local cache to another when a sharing behavior is found.

EXCLUSIVITY: TAG AND DATA ARRAYS A further step would be to implement exclusivity at the same time on both tags and data arrays. This scheme would increase the reuse detection capacity of the reuse cache. At the same time this scheme would complicate coherence maintenance and its worthiness would have to be carefully studied.

(DYNAMIC) EXCLUSIVITY As most of previously shown possible future lines, exclusivity implementation may be considered in a dynamic manner. Sim et al. [53] have recently shown how to adapt this property and such kind of adaptation could be also employed in any of the exclusive schemes here proposed.

CONCLUSIONES

La diferencia de velocidad entre las *CPUs* y la memoria principal ha venido aumentando durante los últimos cuarenta años y no hay ningún signo aparente que indique que dicha tendencia va a remitir en un futuro próximo. La jerarquía de memorias cache surgió con la intención de maquillar la citada diferencia. Esta jerarquía se apoya en las localidades temporal y espacial para proporcionar datos e instrucciones a las cpus de manera eficaz, teniendo como objetivo ofrecer dicha información con una latencia media reducida.

En el escenario *multi-/many- core* en el que nos hallamos ahora, las jerarquías de memoria son más complejas que nunca. Incluyen uno o varios niveles accesibles solo por uno (o un subconjunto reducido) de los núcleos y, como hemos visto a lo largo de esta tesis, un último nivel que es compartido por todos los núcleos del sistema. Además, la ley de Moore se sigue cumpliendo a rajatabla, por lo que los arquitectos cuentan con un número mucho mayor de transistores para cada nueva generación de sus diseños. Precisamente, una gran parte de esos nuevos transistores se emplean en muchos de los diseños para ampliar cada uno de los niveles de la jerarquía de cache. La jerarquía ocupa una parte importante del área total del chip, como ejemplo, el último nivel de cache de un *intel i7* representa aproximadamente el 50% del área total de dicho procesador.

Hasta ahora el número de megabytes por core que incluía el último nivel de cache aumentaba en cada nueva generación de procesadores, pero no es probable que esta tendencia continúe en el futuro si se pretende que la latencia de acceso se mantenga dentro de lo razonable en diseños con docenas de núcleos. Por lo tanto, el diseño de las jerarquías de los dispositivos del futuro ha de pasar por una mejora en la eficiencia de dichas jerarquías. Esta tesis se ha centrado en mejorar la eficiencia del último nivel de cache, prestando especial atención a dos aspectos diferentes: 1) la mejora de su rendimiento y 2) la reducción de la cantidad de hardware empleado en su diseño.

1) Esta tesis hace contribuciones en relación a dos mecanismos que afectan al rendimiento del último nivel de cache de manera crítica: la prebúsqueda hardware y el reemplazo.

La prebúsqueda hardware no controlada puede llegar a dañar el rendimiento del sistema cuando se usa en un multiprocesador donde diferentes aplicaciones se están ejecutando al mismo tiempo. Las prebúsquedas asociadas a un núcleo podrían interferir con los datos cargados en la cache por otro núcleo, provocando la eliminación de los contenidos de otra aplicación y dañando su rendimiento. Es necesario por tanto un mecanismo para regular la prebúsqueda asociada a cada uno de los núcleos. Este mecanismo debería tener por objetivo el mejorar el rendimiento general del sistema. Esta tesis propone un autómata de bajo coste que es capaz de lograr dicho objetivo, mejorando un 27% el rendimiento de un sistema con prebúsqueda no controlada.

Establecemos una propiedad llamada *localidad de reúso* que está ligada a la secuencia de referencias que acceden al último nivel de cache y que dice que (i) si un bloque de cache recibe un acierto es altamente probable que reciba más aciertos en el futuro. Y (ii) los bloques de cache recientemente reúsados son más útiles que otros reúsados previamente. Se ha demostrado en esta tesis que el patrón de acceso a la SLLC muestra localidad de reúso.

En esta tesis se proponen dos algoritmos de reemplazo capaces de explotar la localidad de reúso, *Least-recently reused (LRR)* y *Not-recently reused (NRR)*. Estos dos nuevos algoritmos son modificaciones de otros dos muy bien conocidos: *Least- recently used (LRU)* y *Not-recently used (NRU)*. Dichos algoritmos fueron diseñados para explotar la localidad temporal, mientras que los nuestros explotan la localidad de reúso. Las modificaciones propuestas no suponen ninguna sobrecarga hardware respecto a los algoritmos base. Durante esta tesis se ha mostrado que nuestros algoritmos mejoran consistentemente el rendimiento de los originales así como el de DRRIP, algoritmo perteneciente al estado del arte.

Finalmente, proponemos un novedoso diseño para la SLLC llamado *Reuse Cache*. La reuse cache se apoya en la localidad de reúso para implementar una política de selección de contenidos muy estricta. Dicha política solo almacena en el array de datos de la cache de último nivel aquellos bloques que han mostrado reúso, mientras que el array de etiquetas se usa de una manera convencional. Este diseño permite reducir el tamaño del array de datos de manera drástica. Como ejemplo, una Reuse Cache con un array de etiquetas equivalente al de una cache convencional de 4MB y un array de datos de 1MB, tiene el mismo rendimiento medio que una cache convencional de 8MB, pero con un ahorro de almacenamiento de en torno al 84%.

BIBLIOGRAPHY

[1] A.R. Alameldeen and D.A. Wood. IPC considered harmful for multiprocessor workloads. *Micro, IEEE*, 26(4):8 –17, july-aug. 2006. ISSN 0272-1732. doi: 10.1109/MM.2006.73.

[2] Jorge Albericio, Pablo Ibáñez, Víctor Viñals, and Jose María Llabería. Exploiting reuse locality on inclusive shared last-level caches. *ACM Trans. Archit. Code Optim.*, 9(4):38:1–38:19, January 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400697. URL http://doi.acm.org/10.1145/2400682.2400697.

[3] J.-L. Baer and W.-H. Wang. On the inclusion properties for multi-level cache hierarchies. In *Proc. 15th Annual Int Computer Architecture Symp*, pages 73–80, 1988. doi: 10.1109/ISCA.1988.5212.

[4] 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. ISBN 0-8186-0861-7. URL http://dl.acm.org/citation.cfm?id=52400.52409.

[5] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[6] J. Cantin, M. Lipasti, and J. Smith. Stealth prefetching. *SIGOPS Oper. Syst. Rev.*, 40:274–282, October 2006. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/1168917.1168892. URL http://doi.acm.org/10.1145/1168917.1168892.

[7] Mainak Chaudhuri, Jayesh Gaur, Nithiyanandan Bashyam, Sreenivas Subramoney, and Joseph Nuzman. Introducing hierarchy-awareness in replacement and bypass algorithms for last-level caches. In *PACT*, pages 293–304, 2012.

[8] X. Chen, Yu Yang, G. Gopalakrishnan, and C. T. Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Proc. Formal Methods in Computer Aided Design FMCAD '06*, pages 81–88, 2006. doi: 10.1109/FMCAD.2006.28.

[9] 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. ISBN 0-7695-2043-X. URL http://dl.acm.org/citation.cfm?id=956417.956577.

[10] S. Cho and L. Jin. Managing distributed, shared l2 caches through os-level page allocation. In *Proceedings of the 39th International Symposium on Microarchitecture*, 2006.

[11] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Ieee micro. 30(2):16–29, 2010. doi: 10.1109/MM.2010.31.

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

[13] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. ISBN 1558603433. The Morgan Kaufmann Series in Computer Architecture and Design.

[14] Fredrik Dahlgren, Michel Dubois, and Per Stenstrom. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In *Proc. Int. Conf. Parallel Processing ICPP 1993*, volume 1, pages 56–63, 1993. doi: 10.1109/ICPP.1993.92.

[15] E. Ebrahimi, O. Mutlu, Chang Joo Lee, and Y. N. Patt. Coordinated control of multiple prefetchers in multi-core systems. In *Proc. MICRO-42 Microarchitecture 42nd Annual IEEE/ACM Int. Symp*, pages 316–326, 2009.

[16] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE MICRO*, 28(3):42–53, 2008. doi: 10.1109/MM.2008.44.

[17] 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*, 2010.

[18] J.L. Hennessy, D.A. Patterson, and A.C. Arpaci-Dusseau. *Computer architecture: a quantitative approach*. Number v. 1 in The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2007. ISBN 9780123704900. URL http://books.google.com/books?id=57UIPoLt3tkC.

[19] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2011.

[20] Intel. *Intel Core i7 Processor. http://www.intel.com/products/processor/corei7/ specifications.htm*, 2011.

[21] 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. ISBN 978-1-60558-282-5.

[22] 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. ISBN 978-0-7695-4299-7.

[23] A. Jaleel, K.B. Theobald, S.C. 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. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1815971. URL http://doi.acm.org/10.1145/1815961.1815971.

[24] R. Karedla, J.S. Love, and B.G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38 –46, march 1994. ISSN 0018-9162. doi: 10.1109/2.268884.

[25] 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. doi: 10.1109/ISCA.2001.937453.

[26] S. Khan, Z. Wang, and D. A. Jimenez. Decoupled dynamic cache segmentation. In *Proc. IEEE 18th Int. Symp. High Performance Computer Architecture HPCA 2012*, 2012. URL http://dx.doi.org/10.1109/HPCA.2012.6169030.

[27] 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. doi: 10.1109/MICRO.2010.24.

[28] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded sparc processor. *IEEE MICRO*, 25(2):21–29, 2005. doi: 10.1109/MM.2005.35.

[29] D. M. Koppelman. Neighborhood prefetching on multiprocessors using instruction history. In *Proc. Int Parallel Architectures and Compilation Techniques Conf*, pages 123–132, 2000. doi: 10.1109/PACT.2000.888337.

[30] S. Kottapalli and J. Baxter. Nehalem-ex cpu architecture. In *Hot Chips*, 2009.

[31] An-Chow 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. doi: 10.1109/ISCA.2001.937443.

[32] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. Ibm power6 microarchitecture. *IBM Journal of Research and Development*, 51(6):639–662, 2007. doi: 10.1147/rd.516.0639.

[33] D. Lee, J. Choi, J.-H. Kim, S.H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *Computers, IEEE Transactions on*, 50(12):1352 –1361, dec 2001. ISSN 0018-9340. doi: 10.1109/TC.2001.970573.

[34] Lingda Li, Dong Tong, Zichao Xie, Junlin Lu, and Xu 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. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370862. URL http://doi.acm.org/10.1145/2370816.2370862.

[35] W. Lin and S. K. Reinhardt. Predicting last-touch references under optimal replacement. Technical report, CSE-TR-447-02, U. of Michigan, 2002.

[36] Haiming Liu, M. Ferdman, Jaehyuk 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. doi: 10.1109/MICRO.2008.4771793.

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

[38] 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. doi: 10.1109/ISCA.2012.6237043. URL http://dl.acm.org/citation.cfm?id=2337217.

[39] Kun Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in smt processors. In *Proc. ISPASS Performance Analysis of Systems and Software 2001 IEEE Int. Symp*, pages 164–171, 2001. doi: 10.1109/ISPASS.2001.990695.

[40] 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. doi: 10.1109/2.982916.

[41] M. Martin, D. Sorin, B. Beckmann, M. Marty, Min 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.

[42] M. Martin, M. Hill, and D. Sorin. Why on-chip cache coherence is here to stay. To appear in Communications of the ACM, July, 2012, 2012.

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

[44] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proc. 40th Annual IEEE/ACM Int. Symp. Microarchitecture MICRO 2007*, pages 146–160, 2007. doi: 10.1109/MICRO.2007.21.

[45] K. J. Nesbit and J. E. Smith. Data cache prefetching using a global history buffer. 25(1):90–97, 2005. doi: 10.1109/MM.2005.6.

[46] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proc. 21st Annual Int Computer Architecture Symp*, pages 24–33, 1994. doi: 10.1109/ISCA.1994.288164.

[47] 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. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250709. URL http://doi.acm.org/10.1145/1250662.1250709.

[48] Moinuddin K. Qureshi, David Thompson, and Yale 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. ISBN 0-7695-2270-X. doi: 10.1109/ISCA.2005.52. URL http://dx.doi.org/10.1109/ISCA.2005.52.

[49] Luis M. Ramos, José Luis Briz, Pablo E. Ibáñez, and Víctor Viñals. Multi-level adaptive prefetching based on performance gradient tracking. *J. Instruction-Level Parallelism*, 13, 2011.

[50] Jeffrey B. Rothman and Alan Jay 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. ISBN 1-58113-164-X. doi: 10.1145/305138.305156. URL http://doi.acm.org/10.1145/305138.305156.

[51] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd 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. ISBN 978-1-4503-1182-3. doi: 10.1145/2370816.2370868. URL http://doi.acm.org/10.1145/2370816.2370868.

[52] 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. ISBN 0-8186-5510-0. doi: http://dx.doi.org/10.1145/192007.192072.

[53] Jaewoong Sim, Jaekyu Lee, Moinuddin K. Qureshi, and Hyesoon Kim. Flexclusion: balancing cache capacity and on-chip bandwidth via flexible exclusion. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 321–332, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-1-4503-1642-2. URL http://dl.acm.org/citation.cfm?id=2337159.2337196.

[54] Alan Jay Smith. Cache memories. *ACM Comput. Surv.*, 14:473–530, September 1982. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/356887.356892. URL http://doi.acm.org/10.1145/356887.356892.

[55] A. Snavely and D. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. *SIGARCH Comput. Archit. News*, 28:234–244, November 2000. ISSN 0163-5964. doi: http://doi.acm.org/10.1145/378995.379244. URL http://doi.acm.org/10.1145/378995.379244.

[56] Stephen Somogyi, Thomas F. Wenisch, Anastassia Ailamaki, Babak Falsafi, and Andreas Moshovos. Spatial memory streaming. In *Proceedings of the 33rd annual international symposium on Computer Architecture*, ISCA '06, pages 252–263, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2608-X. doi: http://dx.doi.org/10.1109/ISCA.2006.38. URL http://dx.doi.org/10.1109/ISCA.2006.38.

[57] Stephen Somogyi, Thomas F. Wenisch, Anastasia Ailamaki, and Babak Falsafi. Spatio-temporal memory streaming. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 69–80, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: http://doi.acm.org/10.1145/1555754.1555766. URL http://doi.acm.org/10.1145/1555754.1555766.

[58] S. Srinath, O. Mutlu, Hyesoon 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. doi: 10.1109/HPCA.2007.346185.

[59] R. Subramanian, Y. Smaragdakis, and G. H. Loh. Adaptive caches: Effective shaping of cache behavior to workloads. In *Proc. MICRO-39 Microarchitecture 39th Annual IEEE/ACM Int. Symp*, pages 385–396, 2006. doi: 10.1109/MICRO.2006.7.

[60] T. S. B. Sudarshan, Rahil Abbas Mir, and S. Vijayalakshmi. Highly efficient LRU implementations for high associativity cache memory.

[61] Myoung Kwon Tcheun, Hyunsoo Yoon, and Seung Ryoul Maeng. An adaptive sequential prefetching scheme in shared-memory multiprocessors. In *Proc. Int Parallel Processing Conf*, pages 306–313, 1997. doi: 10.1109/ICPP.1997.622660.

[62] A. Valero, J. Sahuquillo, S. Petit, P. López, and J. Duato. Combining recency of information with selective random and a victim cache in last-level caches. *ACM Trans. Archit. Code Optim.*, 9(3):16:1–16:20, October 2012. ISSN 1544-3566. doi: 10.1145/2355585.2355589. URL http://doi.acm.org/10.1145/2355585.2355589.

[63] D. Wallin and E. Hagersten. Miss penalty reduction using bundled capacity prefetching in multiprocessors. In *Proc. Int. Parallel and Distributed Processing Symp*, 2003. doi: 10.1109/IPDPS.2003.1213088.

[64] T. F. Wenisch, S. Somogyi, N. Hardavellas, Jangwoo Kim, A. Ailamaki, and Babak Falsafi. Temporal streaming of shared memory. In *Proc. 32nd Int. Symp. Computer Architecture ISCA '05*, pages 222–233, 2005. doi: 10.1109/ISCA.2005.50.

[65] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos. Practical off-chip meta-data for temporal memory streaming. In *Proc. IEEE 15th Int. Symp. High Performance Computer Architecture HPCA 2009*, pages 79–90, 2009. doi: 10.1109/HPCA.2009.4798239.

[66] 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.

[67] 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. ISBN 978-1-4503-1053-6. doi: 10.1145/2155620.2155671. URL http://doi.acm.org/10.1145/2155620.2155671.

[68] Y. Xie and G. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 174–183, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0. doi: 10.1145/1555754.1555778. URL http://doi.acm.org/10.1145/1555754.1555778.

[69] Li Zhao, Ravi Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. NCID: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. In *Pro-*

ceedings of the 7th ACM international conference on Computing frontiers, CF '10, pages 121–130, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0044-5. doi: 10.1145/1787275.1787314. URL http://doi.acm.org/10.1145/1787275.1787314.