Departamento de Informática
de Sistemas y Computadores

Adaptive Prefetching and Cache Partitioning for Multicore Processors

A thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Engineering)

Author

Vicent Selfa Oliver

Advisors

Prof. Julio Sahuquillo Borrás
Prof. María E. Gómez Requena

València, September 2018

Agraïments

Esta tesis és el resultat de molt d'esforç i dedicació. Un esforç i dedicació que no haguera sigut possible sense l'ajuda i suport de moltes persones que han estat al meu costat tots estos anys.

En primer lloc vull agrair als meus directors Julio i Maria Engracia la seua confiança i el haver-me donat l'oportunitat de realitzar este treball, així com els seus consells i idees, sense els quals aquesta tesi no haguera sigut possible. També a Salva, que tot i no ser director, ha estat sempre molt present, aconsellant i ajudant en la revisió dels treballs.

En l'elaboració d'esta tesi ha influït molt l'estança de tres mesos que vaig fer al grup d'investigació del professor Lieven Eeckhout, a la Universiteit Gent, a Gent, Bèlgica. Les reunions setmanals del grup, en les que cada membre presentava els seus avanços per a rebre *feedback* dels demés van ser molt enriquidores. A banda, les idees del professor Eeckhout sens dubte han millorat este treball.

També vull agrair als meus pares, al meu germà i el seu suport, ja que han estat sempre al meu costat, i no haguera sigut possible sense ells. Per suposat, també a Irene, per recolzar-me sempre (incloent l'estada a Gent) i per tindre tanta paciència durant estos anys, en els que separar vida laboral i personal ha sigut de vegades complicat.

Finalment, donar les gràcies als companys del GAP, especialment a Albert, Diego, Fran, Joan, José María, José Vicente i Migue, sense els quals tot ha-

guera sigut molt menys entretingut, i amb els que he aprés molt. I com no, també agraïr el suport i consells de Joana i Gema, *advisors* a l'ombra.

Abstract

Accessing main memory represents a major performance bottleneck in current processors, since the different cores compete among them for the limited off-chip bandwidth, aggravating even more the so called *memory wall*. Several techniques have been applied to deal with the core-memory performance gap, with the most preeminent ones being prefetching and hierarchical caching.

Hierarchical caches leverage the temporal and spacial locality of the accessed data, mitigating the huge main memory access latencies. To limit the number of accesses to the off-chip DRAM memory, current processors feature large Last Level Caches. These caches are shared between all the cores to improve the utilization of the cache space and reduce cost. This approach significantly improves the performance of most applications compared to using smaller private caches. Cache sharing, however, presents an important shortcoming: the interference between applications. Prefetching, on the other hand, brings data blocks to the caches before they are requested, hiding the main memory latency. Unfortunately, since prefetching is a speculative technique, inaccurate prefetches may pollute the cache with blocks that will not be used. In addition, the prefetches interfere with the regular memory requests, both the ones from the application running on the core that issued the prefetches and the others.

This thesis focuses on reducing the inter-application interference, both in the shared cache and in the access to the main memory. To reduce the inter-application interference in the access to main memory, the proposed approach regulates the aggressiveness of each core prefetcher, and selectively activates or deactivates some of them, depending on their individual performance and

the main memory bandwidth requirements of the other cores. With respect to interference in shared caches, this thesis proposes two LLC partitioning techniques that give more cache space to the applications that have their progress diminished due inter-application interferences. The first cache partitioning proposal requires dedicated hardware not available in commercial processors, so it has been evaluated using a simulation framework. The second proposal dealing with cache partitioning presents a family of partitioning policies that overcome the limitations in the number of partitions and the number of available ways by grouping applications and overlapping cache partitions, so multiple applications share the same ways. Since it has been implemented using the cache partitioning features of modern Intel processors it has been evaluated in a real machine.

Experimental results show that the proposed selective prefetching mechanism reduces the number of main memory requests by 20%, which translates to improvements in unfairness, performance, and energy consumption. On the other hand, regarding the proposed partitioning schemes, compared to a system with no partitioning, both reduce unfairness more than 25% on average, regardless of the number of applications running in the multicore, and this reduction in unfairness does not negatively affect the performance.

Resum

L'accés a la memòria principal en els processadors actuals suposa un important coll d'ampolla per a les prestacions, ja que els diferents nuclis competeixen pel limitat ample de banda de memòria, agreujant la bretxa entre les prestacions del processador i les de la memòria principal. Diferents tècniques ataquen aquest problema, sent les més rellevants l'ús de jerarquies de memòria cau multinivell i la prebusca.

Les memòries cau jeràrquiques aprofiten la localitat temporal i espacial que en general presenten els programes en l'accés a les dades per mitigar les enormes latències d'accés a memòria principal. Per limitar el nombre d'accessos a la memòria DRAM, fora del xip, els processadors actuals compten amb grans caus d'últim nivell (LLC). Per millorar la seva utilització i reduir costos, aquestes memòries cau solen compartir-se entre tots els nuclis del processador. Aquest enfocament millora significativament el rendiment de la majoria de les aplicacions en comparació amb l'ús de caus privades més menudes. Compartir la memòria cau, no obstant, presenta una problema important: la interferència entre aplicacions. La prebusca, per altra banda, porta blocs de dades a les memòries cau abans que el processador els sol·licite, ocultant la latència de memòria principal. Desafortunadament, donat que la prebusca és una tècnica especulativa, si no té èxit pot *contaminar* la memòria cau amb blocs que no fan falta. A més, les prebusques interfereixen amb els accessos normals a memòria, tant els del nucli que emet les prebusques com els dels altres.

Aquesta tesi es centra en reduir la interferència entre aplicacions, tant en les cau compartides com en l'accés a la memòria principal. Per reduir la in-

terferència entre aplicacions en l'accés a la memòria principal, el mecanisme proposat en aquesta dissertació regula l'agressivitat de cada prebuscador, activant o desactivant selectivament alguns d'ells, en funció del seu rendiment individual i dels requisits d'ample de banda de memòria principal dels altres nuclis. Pel que fa a la interferència en caus compartides, aquesta tesi proposa dues tècniques de particionat per a la LLC, les quals atorguen més espai de memòria cau a les aplicacions que progressen més lentament a causa de la interferència entre aplicacions. La primera proposta per al particionat de memòria cau requereix hardware específic no disponible en processadors comercials, per la qual cosa s'ha avaluat utilitzant un entorn de simulació. La segona proposta de particionat per a memòries cau presenta una família de polítiques que superen les limitacions en el nombre de particions i en el nombre de vies de memòria cau disponibles mitjançant l'agrupació d'aplicacions en clústers i la superposició de particions de memòria cau, de manera que diverses aplicacions comparteixen les mateixes vies. Atès que s'ha implementat utilitzant els mecanismes per al particionat de la LLC que ofereixen alguns processadors Intel moderns, aquesta proposta s'ha avaluat en una màquina real.

Els resultats experimentals mostren que el mecanisme de prebusca selectiva proposat en aquesta tesi redueix el nombre de sol·licituds a la memòria principal en un 20%, cosa que es tradueix en millores en l'equitat del sistema, el rendiment i el consum d'energia. Per altra banda, pel que fa als esquemes de particiónat proposats, en comparació amb un sistema sense particions, ambdues propostes redueixen la iniquitat del sistema en més d'un 25% de mitjana, independentment de la quantitat d'aplicacions en execució, i aquesta reducció en la iniquitat no afecta negativament el rendiment.

Resumen

El acceso a la memoria principal en los procesadores actuales supone un importante cuello de botella para las prestaciones, dado que los diferentes núcleos compiten por el limitado ancho de banda de memoria, agravando la brecha entre las prestaciones del procesador y las de la memoria principal. Distintas técnicas atacan este problema, siendo las más relevantes el uso de jerarquías de caché multinivel y la prebúsqueda.

Las cachés jerárquicas aprovechan la localidad temporal y espacial que en general presentan los programas en el acceso a los datos, para mitigar las enormes latencias de acceso a memoria principal. Para limitar el número de accesos a la memoria DRAM, fuera del chip, los procesadores actuales cuentan con grandes cachés de último nivel (LLC). Para mejorar su utilización y reducir costes, estas cachés suelen compartirse entre todos los núcleos del procesador. Este enfoque mejora significativamente el rendimiento de la mayoría de las aplicaciones en comparación con el uso de cachés privados más pequeños. Compartir la caché, sin embargo, presenta una problema importante: la interferencia entre aplicaciones. La prebúsqueda, por otro lado, trae bloques de datos a las cachés antes de que el procesador los solicite, ocultando la latencia de memoria principal. Desafortunadamente, dado que la prebúsqueda es una técnica especulativa, si no tiene éxito puede *contaminar* la caché con bloques que no se usarán. Además, las prebúsquedas interfieren con los accesos a memoria normales, tanto los del núcleo que emite las prebúsquedas como los de los demás.

Esta tesis se centra en reducir la interferencia entre aplicaciones, tanto en las caché compartidas como en el acceso a la memoria principal. Para reducir la interferencia entre aplicaciones en el acceso a la memoria principal, el mecanismo propuesto en esta disertación regula la agresividad de cada prebuscador, activando o desactivando selectivamente algunos de ellos, dependiendo de su rendimiento individual y de los requisitos de ancho de banda de memoria principal de los otros núcleos. Con respecto a la interferencia en cachés compartidos, esta tesis propone dos técnicas de particionado para la LLC, las cuales otorgan más espacio de caché a las aplicaciones que progresan más lentamente debido a la interferencia entre aplicaciones. La primera propuesta de particionado de caché requiere hardware específico no disponible en procesadores comerciales, por lo que se ha evaluado utilizando un entorno de simulación. La segunda propuesta de particionado de caché presenta una familia de políticas que superan las limitaciones en el número de particiones y en el número de vías de caché disponibles mediante la agrupación de aplicaciones en clústeres y la superposición de particiones de caché, por lo que varias aplicaciones comparten las mismas vías. Dado que se ha implementado utilizando los mecanismos para el particionado de la LLC que presentan algunos procesadores Intel modernos, esta propuesta ha sido evaluada en una máquina real.

Los resultados experimentales muestran que el mecanismo de prebúsqueda selectiva propuesto en esta tesis reduce el número de solicitudes de memoria principal en un 20%, cosa que se traduce en mejoras en la equidad del sistema, el rendimiento y el consumo de energía. Por otro lado, con respecto a los esquemas de partición propuestos, en comparación con un sistema sin particiones, ambas propuestas reducen la iniquidad del sistema en un promedio de más del 25%, independientemente de la cantidad de aplicaciones en ejecución, y esta reducción en la injusticia no afecta negativamente al rendimiento.

Contents

Agraïments iii

Abstract v

Resum vii

Resumen ix

Contents xi

1 Introduction 1

1.1 Problem Description 1
1.2 Objectives. 9
1.3 Contributions of the Thesis 10
1.4 Thesis Outline 11

2 Related Work 13

2.1 Prefetching 13

2.2 Cache Interference Analysis and Fairness . . . 15
2.3 Cache Partitioning . . . 16

3 Experimental Framework 19

3.1 Simulation Framework . . . 19
3.2 Real Machine: The Intel Xeon E5 2658A v3 . . . 22
3.3 Benchmark Suites . . . 26

4 Selective Prefetching under Limited Memory Bandwidth 31

4.1 Introduction . . . 31
4.2 Characterization Study . . . 33
4.3 ADP Mechanism . . . 37
4.4 Experimental Setup . . . 42
4.5 Evaluation . . . 44
4.6 Summary . . . 51

5 Improving Fairness with LLC Partitioning 53

5.1 Introduction . . . 53
5.2 Analysis of the Inter-Application Cache Interference . . . 57
5.3 Analysis of Progress Estimation Approaches . . . 58
5.4 FPCP Partitioning Approach . . . 61
5.5 Experimental Setup . . . 67
5.6 Evaluation . . . 68
5.7 Summary . . . 74

6 Improving Fairness with LLC Partitioning using Intel CAT 77

6.1 Introduction . . . 77
6.2 Progress Characterization and Estimation . . . 79
6.3 To Overlap or Not To Overlap Cache Ways . . . 83
6.4 Cluster-Based Partitioning Policies . . . 85
6.5 Experimental Setup . . . 87

6.6 Evaluation 89
6.7 Summary 96

7 Conclusions 99

7.1 Contributions 99
7.2 Future Directions 102
7.3 Publications 102

Bibliography 107

Chapter 1

Introduction

This chapter introduces important concepts to ease the understanding of this dissertation and the motivation for the work done. It first describes the problems associated with the so called *memory wall*, the difference between the rates at which the processor cores consume data and the rates at which these data come from main memory. Then, it discusses how modern processors tackle this problem, focusing on two main techniques, caching and prefetching, presenting their benefits and drawbacks, especially when used in multicore processors where the different cores compete for the shared memory resources. This topic, the interference between cores when competing for the shared memory resources, is the main problem tackled by this thesis.

1.1 Problem Description

Following the Moore's law, the amount of transistors in the processors roughly doubles every two years. As a corollary of Moore's law, microprocessor performance has steadily increased year by year. DRAM technology, in contrast, has had more modest performance improvements, but it has experienced important capacity increases. This trend has favored the so called *memory wall*, the difference in performance between the cores and the main memory, which represents the main bottleneck in current processors.

Several techniques have been applied to deal with the core-memory performance gap. The most preeminent one is hierarchical caching. Since, in general, computer programs tend to access data blocks multiple times (temporal locality), and to accessed addresses usually are consecutive (spatial locality), storing recently used data in faster but smaller caches greatly improves the performance. Because of this, current processors feature several cache memory levels, where the closest to the core are the smallest and fastest, and they increase in size and become slower as they get closer to main memory. The current trend in high performance processors of different manufacturers is to have per core private L1 and L2 caches and a much bigger, shared, L3 cache. For example, the recent Intel Xeon CPU E5-2620 v4 features 32KB L1 instruction and data caches, 256KB L2 caches, and a huge 20MB L3, shared between 8 cores.

While due to the locality principle most of the accesses find the data they are looking for in L1 and L2 caches, the Last Level Cache (LLC) plays a key role in the effective performance of the processor. The reason is that the huge main memory latencies (in the order of 100s of cycles) are too important to be hidden by the out-of-order execution engine, so eventually, the LLC misses block the Re-Order Buffer (ROB), stopping the instruction issuing and stalling the processor. For this reason, the trend is to increase the size of the LLC, in order to maximize its hit ratio. However, the space in the chip is limited, so to improve its cost/effectiveness the LLC tends to be shared. Unfortunately, cache sharing also presents shortcomings, with the most prominent one being the interference between cores, which will be analyzed in the next sections.

Nevertheless, caching is only useful if the data is accessed again and it does not solve the latency of the first access that brings the block to the cache from main memory (i.e. cold misses). To deal with this, current processors use prefetching, i.e. fetching the data from lower levels of the memory hierarchy (i.e. main memory) before it is requested by the processor. To this end, a per-core engine detects patterns in the data and instruction accesses and speculatively requests new cache blocks. While this mechanism can dramatically improve the performance for most applications, its predictions may fail or it may be too aggressive in some cases, causing the so called *cache pollution*. Since the prefetchers of all the cores bring blocks to the shared LLC and compete for the limited off-chip memory bandwidth, in some cases they can destructively interfere and significantly damage the performance of specific applications.

1.1.1 Chip Multiprocessors

The demand for more and more computing power drove the microprocessor industry to produce faster chips, exploiting the reductions in the integration scale to both increase the processor frequency and to increase its complexity. As a consequence, new designs introduced larger reorder buffers, more complex cache structures, added more physical registers for register renaming and improved the support for speculative execution. However, increasing the complexity of a circuit tends to increase its switched capacitance, which, in turn, increments the dynamic power consumption, that is proportional to capacitance (C), frequency (f), and squared voltage (V^2): $P_{dyn} = CV^2f$.

Since power consumption is a growing design concern, in order to reduce energy consumption and thus, power dissipation, processor manufacturers moved to multicore designs, that provide more aggregated performance with less power consumption. Nowadays, multicore processors are the standard for high-performance computing, but also in the battery-constrained mobile market.

A multicore processor is usually a single processor that contains multiple *cores*, i.e. computing units with private caches, typically L1 and L2. These cores are linked by an interconnection network and may share the LLC, usually distributed, and the main memory bandwidth.

Multicore designs help with power consumption and performance, but with this power and performance benefits also come some major challenges. One important challenge with multicores is that applications do not automatically run faster with more cores. An application needs to be specifically designed (i.e. parallelized) to make use of the additional cores and benefit from them. However, new compilers that auto-parallelize code, new libraries and frameworks, and a slow change in programming paradigms are contributing to make this issue less critical. Another challenge that is gaining importance as the core count increases is the management of the shared processor resources: namely the cache hierarchy and the main memory bandwidth. As the number of cores sharing resources increases, so does the risk of destructive inter-core interference.

1.1.2 The Last Level Cache

Since the introduction of multicores, which further stressed the already limited main memory bandwidth, current high-performance processors tend to include huge Last Level Caches (LLC) of the order of tens of megabytes [39] to mitigate capacity misses. While the percentage of accesses served by these caches is small compared with the L1 and L2 cache levels, a miss in the last level of cache implies an onerous access to main memory, so they tend to have an important impact on the overall system performance. In addition to their size, they are usually implemented with a high degree of associativity, with more than 16 ways, to keep conflict misses low. The reason is the same: accessing main memory is expensive.

The LLCs, as implemented in recent processors, are usually shared between the different cores, as this improves their utilization (i.e. the space not used by one core can be used by others). However, as all the cores access the same cache, they can interfere and evict data that is being used by others, damaging the performance. As a consequence, the performance of individual applications running on a multicore processor with co-runners (i.e. other applications) is usually significantly worse than when executed alone in the same system.

Nowadays, a significant amount of computer power is devoted to cloud computing services, where virtual machines or containers are used to provide a platform to allow customers to develop, run, and manage applications. These customers usually share physical machines, but their workloads have to be as isolated from other customers' workloads as possible. A shared LLC complicates this issue and can impact the whole system performance and fairness.

1.1.3 Cache Partitioning

A lot of effort has been invested in finding ways to mitigate cache interference in multicores, with the most prominent one being cache partitioning. The idea behind cache partitioning is to retain the benefits of a monolithic and shared cache and, at the same time, avoiding the disadvantages of cache interference. By mitigating cache interference, a well-crafted cache partitioning policy can improve performance [87], fairness [78], and isolate applications for security [96] and quality of service (QoS) reasons [8].

Cache partitioning allows system software to divide the cache space between cores, threads or applications. There are different ways to partition the cache, doing set-partitioning from software with page coloring [101] way partition-

ing from hardware [78, 70] or using probabilistic cache replacement mechanisms [58].

Regarding page coloring, it works as follows. When there is a cache access, the address is divided into three parts: tag, index and offset, as shown in Figure 1.1. The *offset* is the position in the cache block, the *index* indicate the set, and the *tag* is compared with the blocks in the set to decide if the access is a hit or a miss. The idea behind page coloring is simple: use the remaining index bits after the page offset (see Figure 1.1) to control to which set the blocks of a given page are placed to. This approach has several limitations, though. It is incompatible with superpages, which provide important performance benefits; it is common for the LLC of modern systems to use hash-based addressing[1]; and repartitioning usually implies copying memory pages, which is expensive.

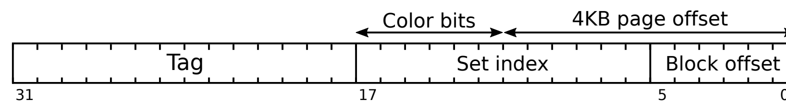


Figure 1.1: Physical address bits used for partitioning a 4MB 16-way associative LLC with page coloring and 64 colors.

Way partitioning, on the other hand, works by keeping a registry of the ways used in each set per core. Then, when there is a miss, if the core has more cache lines occupied than it has been assigned, it evicts one of them. If the core is occupying less lines than assigned, it evicts one from another core.

The probabilistic cache partitioning policies work similarly to way partitioning approaches but, unlike way-partitioning, where the evicted block depends on the number of blocks each core has, they choose an eviction candidate based on probability distributions. The rationale is that controlling the eviction probability of the lines of each core, one can control the cache space they occupy with a finner granularity than with way-partitioning.

Since hardware-based cache partitioning techniques (e.g. way-based partitioning, probabilistic partitioning) require specialized hardware structures or performance counters not available in commodity processors, most research has been conducted using detailed simulation frameworks [7, 91, 5]. Due to the promising simulation results of these works, some companies like Intel [32], ARM [2], and Cavium [95] have introduced in some of their most recent products way-partitioning capabilities. This dissertation leverages both ap-

[1]The LLC in Intel processors is divided into multiple slices, each operating as a typical set-associative cache. Intel uses an undisclosed hash function to map memory locations to cache slices [100].

proaches, simulation and experimentation in commercial Intel machines, to design, implement and evaluate the different cache partitioning schemes. Each approach has its pros and cons: experimenting on a real machine provides arguably more reliable results and the experiments usually run orders of magnitude faster, but it is constrained to the capabilities provided by the hardware.

1.1.4 Hardware Prefetching

As already stated, data caching is not enough, by itself, to solve the challenge that high latency memory accesses pose. To further deal with this problem, the cache controller can try to predict future accesses and bring the data from main memory to the cache before it is requested. One can prefetch data to any cache level and kind (i.e. instructions or data), but each one has its peculiarities, which have to be taken into account when deciding which prefetching mechanism is the most adequate. For example, L1 caches are smaller than other cache levels, so unrestricted prefetching can cause undesired evictions of blocks that would be accessed by the processor, damaging the performance. This effect is referred to as *cache pollution*. L3, being considerably larger, tolerates better a more aggressive prefetching scheme. On the other hand, from the L3 cache level one does not have access to useful information (from the prefetching point of view) like the Program Counter (PC) of the loads and stores accessing the cache or their virtual addresses, and this limits the kind of prefetching techniques that can be used. Another example would be instruction caches. Due to their access patterns, it may be enough to use simple prefetchers that bring the next cache line after one is accessed (i.e. One Block Lookahead). Other caches may benefit from more convoluted prefetching mechanisms.

To be effective, the prefetches must be both *accurate* and *timely*. A prefetch is accurate when the prefetched data is requested by the core before it is evicted from the cache. A prefetch is timely if the data has been already brought to the cache when requested, and the demand access does not have to wait for it. It may be difficult to improve both metrics at the same time, since to be timely a prefetcher may have to aggressively fetch a lot of data, and fetching big amounts of data usually sacrifices the prefetcher accuracy. Inaccurate prefetches waste energy and bandwidth, and can pollute the cache and negatively impact the performance. Also, CMPs share the main memory bandwidth, the interconnection network and some levels of cache hierarchy, so an aggressive prefetcher may be accurate and improve the performance of its associated core but, at the same time, decrease the performance of other cores executing applications concurrently.

For this reason, usually the prefetchers first detect a pattern and then launch prefetches following it. Three common prefetchers that follow this approach are stride prefetchers, Markov prefetchers and delta correlation prefetchers. Stride prefetchers first group accesses by Program Counter or by some of the higher bits of the address, and then compute the strides between the accesses. When two equal strides are found in a row, a prefetch is triggered.

Markov prefetchers store in a FIFO buffer the cache accesses, indexed by address. When a miss occurs, the address that immediately followed that access in the past is prefetched. Delta correlation prefetchers index accesses by Program Counter and compute the deltas with previous accesses with the same index. When there is a miss, it searches the delta history to find the last time the two last deltas occurred, and prefetches the next delta or deltas, depending on the prefetcher aggressiveness.

1.1.5 Pros and Cons of Resource Sharing

Both prefetchers and huge last-level shared caches are used to hide long memory latencies. Each core has its own prefetchers that bring blocks to the shared LLC, and consume part of the shared main memory bandwidth. This resource sharing between CMP cores is a common way to reduce costs and improve the utilization of the system resources. Also, when only an application is running, it can entirely utilize the shared resources. The same applies when there are several applications running but some of them have a low resource usage. Since each core could potentially have all the shared resources for itself, it avoids having to overprovision them to face demand spikes.

However, resource sharing has, unfortunately, an important drawback: unfairness among co-running applications. In this dissertation a system is considered to be fair when all the applications in execution *progress* at the same pace. In this context, the *progress* of an application is computed [23, 22, 92, 27] as the ratio of its execution time while running with other applications, relative to its execution time in isolation[2], as shown in Equation 1.1; Progress is, therefore, a value between 0 and 1, or between 0 and 100, if expressed as a percentage.

$$Progress = \frac{ExecCycles_{alone}}{ExecCycles} \tag{1.1}$$

[2]Note that progress is considered on a per-application basis, regardless of the application type, i.e., single-threaded or multi-threaded.

Another related metric is the *slowdown*, computed as the inverse of *progress*, as shown in Equation 1.2. The *slowdown* of an application is always a value equal to or larger than 1. Both metrics, *progress* and *slowdown*, are used to estimate how interference between applications degrades performance. When they are equal to 1, the performance of the application is not affected by the other competing applications. When considering them to estimate interference, one can define a system as completely fair when all the tasks in the system experience the same *progress* or *slowdown* [8, 17, 27, 62].

$$Slowdown = \frac{ExecCycles}{ExecCycles_{alone}} \tag{1.2}$$

Based on this statement, some works [14] propose the ratio between the progress of the application that progresses the most and the application that progresses the least as a way to quantify unfairness. However, this metric only considers extreme values. To overcome this limitation, an alternative metric was proposed in [92], which uses the coefficient of variation (CoV) [20] of the progresses of the running applications with respect to the mean to measure how unfair the system is. This metric is shown in Equation 1.3, where σ refers to the standard deviation and μ to the mean *progress*:

$$Unfairness = \frac{\sigma_{Progress}}{\mu_{Progress}} \tag{1.3}$$

Unfairness is a major concern in current CMP design, which threatens scalability and causes critical undesirable behaviors: (i) it makes execution time unpredictable, which complicates the analysis of both hardware and software implementations, (ii) it complicates priority-based Operating System scheduling, and (iii) it enables denial of service attacks. Despite this, unfairness is still a pending problem in current microprocessors.

1.1.6 Performance metrics

While fairness is important, it should never be attained at the cost of performance. Through this dissertation the system performance is measured using the System Throughput (STP) Average Normalized Turnaround Time (ANTT) metrics, shown in equations 1.4 and 1.5. Both are two recent and well-defined metrics extensively used in the literature [23, 22], which compute aggregated progress and average slowdown, respectively. Since they quantify different aspects of multiprogram performance (overall system performance

versus per-application performance), both should be considered when evaluating a system.

$$STP = \sum_{t \in Tasks} Progress_t \tag{1.4}$$

$$ANTT = \mu_{Slowdown} \tag{1.5}$$

1.2 Objectives

The main objective of this thesis is to deal with a major design concern of multicore processors: the interference at the shared resources when multiple concurrently running applications compete for a limited amount of shared main memory bandwidth and LLC space. The goal is to achieve a good trade off between performance and system unfairness (i.e. improving performance without increasing the unfairness or improving unfairness while sustaining the performance).

Regarding main memory bandwidth, this dissertation pursues a prefetch managing mechanism that acts in each core prefetcher, taking decisions regarding prefetching aggressiveness that consider not only core-specific metrics but also global information, like the memory bandwidth requirements of other cores. This mechanism regulates the aggressiveness of each prefetcher and selectively activates and deactivates some of them. It estimates when the prefetches are going to be useful and not harm other cores, rising the aggressiveness in such a case, and saving memory bandwidth otherwise.

With respect to interference in shared caches, this dissertation focuses on its most nefarious effect, the inequalities between the progresses of the applications running on the different cores of a CMP. To mitigate this problem, this thesis employs LLC partitioning techniques to provide additional cache space to the applications that have their progress diminished due to inter-application interference, effectively reducing unfairness.

1.3 Contributions of the Thesis

This thesis presents three major contributions: a selective prefetching mechanism targeting multicore processors with constrained main memory bandwidth, and two proposals that improve system fairness by partitioning the LLC, one that requires dedicated hardware, not currently available in commercial processors, and another that uses commodity hardware. The first cache partitioning approach presented in this work has been evaluated in a simulation framework, and the second one has been implemented and evaluated in a real machine with an Intel processor featuring Intel Cache Allocation Technology. Below these contributions are listed with more detail.

- **Selective prefetching under limited memory bandwidth.** Current multicore systems implement multiple hardware prefetchers to tolerate the large main memory latencies. However, memory bandwidth is a scarce shared resource which becomes critical with the increasing core count. To deal with this fact, recent work has focused on adaptive prefetchers, which control the prefetcher aggressiveness to regulate the main memory bandwidth consumption. Nevertheless, in limited bandwidth scenarios with memory-hungry workloads, keeping active the prefetcher may damage the system performance and increase energy consumption. This dissertation introduces a *selective prefetcher* that deactivates, throttles and reactivates individual prefetchers to better distribute main memory bandwidth, improving energy consumption and performance.

- **Unfairness reduction with LLC partitioning.** Shared caches have become the common design choice in the vast majority of modern multicore and many-core processors, since cache sharing improves throughput for a given silicon area. Sharing the cache, however, has a downside: the requests from multiple applications compete among them for cache resources, so the execution time of each application increases over isolated execution. The degree in which the performance of each application is affected by the interference becomes unpredictable yielding the system to *unfairness* situations. To mitigate this effect, this dissertation proposes the Fair-Progress Cache Partitioning (FPCP), a low-overhead hardware-based cache partitioning approach that addresses system fairness by partitioning the LLC cache. To adjust partitions, our approach estimates during multicore execution the time each application would have taken in isolation, relying on additional hardware to estimate the inter-core interference. Since the required hardware is not yet available in com-

mercial processors, this proposal has been evaluated using a simulation framework.

- **Unfairness reduction with LLC partitioning using Intel CAT.** Existing proposals targeting unfairness reduction require extra hardware, which makes them impractical in commercial processors. Recent Intel Xeon processors feature Cache Allocation Technology (CAT), a hardware cache partitioning mechanism that can be controlled from userspace software and that allows to create partitions in the LLC and assign different groups of applications to them. This contribution consists of a family of clustering-based cache partitioning policies that target unfairness reduction in real systems that feature Intel's CAT. They act at two levels: applications showing similar amount of core stalls due to LLC accesses are first grouped into clusters, after which each cluster is given a number of ways using a simple mathematical model. To the best of our knowledge, this is the first attempt to address system fairness using the cache partitioning hardware in a real product.

1.4 Thesis Outline

The remaining of this dissertation is organized into 6 chapters as follows:

- Chapter 2 discusses previous works.
- Chapter 3 describes the experimental frameworks used to evaluate the proposals.
- Chapter 4 presents the selective prefetcher for multicore processors.
- Chapter 5 proposes Fair Progress Cache Partitioning, a LLC partitioning technique for unfairness reduction.
- Chapter 6 introduces a family of cache partitioning policies that also target unfairness reduction but have been implemented in commercial Intel processors featuring Intel Cache Allocation Technology.
- Finally, Chapter 7 draws the conclusions, discusses future work, and enumerates the related publications.

Chapter 2

Related Work

This chapter discusses the state-of-the-art of the topics covered by this dissertation. First, it is introduced the related work regarding prefetching techniques to tackle main memory congestion. Then, the related work on cache partitioning is revised, covering proposals implemented and evaluated both with simulators and with real machines.

2.1 Prefetching

This section describes previous work focusing on the prefetcher aggressiveness and the reduction in the number of memory requests in multicores.

In [64] the AC/DC adaptive method for prefetching data from main memory to the L2 cache is proposed. Like the selective prefetcher presented in this dissertation, AC/DC uses concentration zones (also called *CZones*) [67] that divide memory into fixed size zones. The mechanism is enhanced to make use of delta correlations to find access patterns. They propose an adaptive algorithm that dynamically adjusts the prefetch degree ranging from 2 to 16 cache blocks. The mechanism provides the opportunity to turn off the prefetcher but only in those cases where prefetching hurts the system performance, and no policy is devised to turn on the prefetcher again. More recently, PATer [52] has been proposed. It uses a prediction model based on machine learning with the

aim of dynamically tuning the prefetcher parameters of the IBM POWER8, which has 2^{25} possible configurations. This prediction is based on the value of performance monitoring counters. Unlike our proposal, this approach is only applicable to POWER8 processors.

The FDP adaptive approach, presented in [85], dynamically selects among five different levels of aggressiveness, ranging from very conservative to very aggressive. The baseline prefetcher is a stream prefetcher like the one used in this dissertation for the proposal presented in Chapter 4. Similarly to this dissertation selective prefetcher, FDP selects at the end of each sampling interval the aggressiveness for the next interval. For this purpose, accuracy, lateness, and pollution metrics are used to throttle up or down the aggressiveness level. This mechanism was extended for multicores in another proposal, the Hierarchical Prefetcher Aggressiveness Control (HPAC) [19]. In HPAC each core implements a FDP prefetcher, but local decisions to change the aggressiveness can be overridden by the memory controller, which collects global information about the memory requirements of each application. Unlike our work, these proposals always keep the prefetcher enabled.

Other previous proposals are based on prefetch filtering techniques. In [6] it is proposed a weighted majority filter to predict the usefulness of the prefetch addresses. Other works, like [104, 13] estimate *a priori* if a given prefetch will be useful or not, discarding it in the latter case. While their goal is the same as ours, the metrics used to guide the prefetcher are not. For example, our proposal considers the memory contention for decision making, and not just if a prefetch is useful for the core that has issued it. Additionally, the way the goal is achieved is also different. Our approach throttles and disables the prefetcher, but filtering techniques dynamically decide whether to issue or not a prefetch, based on prefetch history. However, both techniques are orthogonal to each other, so using them together could further reduce the amount of wasted bandwidth.

A prefetcher that classifies prefetches according to their impact on performance is proposed in [57]. This impact is estimated with a history table indexed by the program counter that collects the stall cycles caused by each load. The mechanism prioritizes the prefetches associated to loads that have caused more Reorder Buffer (ROB) stalls. That is, the prefetcher is mainly guided by core performance instead of prefetcher performance.

In [69], Sandbox Prefetching, a mechanism to select at runtime the most appropriate prefetcher from a set of different prefetchers is proposed. Rather than actually fetching data into the cache to evaluate the prefetcher accuracy,

the tags of the blocks that the evaluated prefetcher would fetch are stored in a Bloom Filter. On each memory access, the Bloom Filter is checked to estimate the expected accuracy of the prefetcher. The candidate prefetchers are evaluated one at a time, in a multiplexed fashion, with the sandbox being reset in between evaluations. The main weakness of this mechanism, unlike ours, is that prefetch decisions are taken locally in the core, without considering global system conditions. More recently, Best-Offset Prefetching [60] has been proposed as a Sandbox improvement by considering prefetch timeliness to calculate the prefetch offset.

Prefetching performance can be also improved by enhancing the policies managing memory requests at the shared resources, that is, the arbiter at the NoC or the scheduling policy at the memory controller. Regarding the NoC, some interesting approaches [10, 50] implement virtual channels and dynamically use them to adjust the priority between regular and prefetch requests coming from multiple cores. With respect to memory controller policies, recent proposals [18, 54, 53] have also focused on multicores. These policies take into account the prefetcher performance to dynamically select the priority of both regular and prefetch requests. NoC and memory controller works are orthogonal to our proposals and can be applied together to achieve further improvements.

2.2 Cache Interference Analysis and Fairness

Inter-application interference at the shared caches, both in CMPs and SMTs, is a well-known effect that rises as the number of processing units sharing the resource increases and that can yield the system to important unfairness scenarios. A key challenge to address unfairness is to be able to estimate the interference at the shared cache. Eyerman *et al.* [15] and Du Bois *et al.* [21] propose an approach to measure this interference by duplicating a fraction of the shared LLC cache tags. Ebrahimi *et al.* [17] propose to keep track of the lines evicted by the different competing cores using a hash table to estimate interference and use this estimation to enforce fairness. Subramania *et al.* [87] employ an approach similar to Eyerman *et al.* to estimate application slowdown due to inter-application interference.

Most techniques aimed at achieving fair system performance, are software-based and rely on OS scheduling [24, 98, 99]. Ebrahimi *et al.* [17] improve system fairness by dynamically adapting the rate at which different cores inject requests into the memory subsystem. The approach proposed by Sharifi *et*

al. [80] is based on changing the cache replacement policy to focus on fairness among cores by penalizing the core with the highest IPC in favor of the others.

Different metrics have been proposed in the literature to quantify unfairness. Some works [14] propose the ratio between the progress of the application that progresses the most and the application that progresses the least as a way to quantify unfairness. However, this metric only considers extreme values. An alternative metric without this handicap was proposed in [92], which uses the coefficient of variation [20] of the progresses of the different applications running in the system. This is the metric used throughout this dissertation to estimate unfairness.

2.3 Cache Partitioning

A large body of research has focused on LLC partitioning during the last years, with different goals and techniques. In order to facilitate their understanding, the cache partitioning proposals are classified according to three axes, depicted in Figure 2.1. The first axis is the goal of the proposal, which can range from performance or fairness to QoS or security. The second axis is how the partitions are enforced, separating proposals that enforce partitions using only pure software methods (i.e. no hardware support is required) from others which require hardware support (e.g. way-partitioning, changes in replacement policy). The last axis is how the proposal has been implemented and evaluated, if using a real machine or a simulation framework. Pure software approaches tend to be easier to evaluate in real systems, while approaches that require hardware support have been traditionally evaluated in simulators. The reason is that, until the last few years, there was little or no hardware that supported cache partitioning. This is slowly changing, with some Intel and ARM designs featuring hardware support for cache partitioning in their LLCs [36, 2]. Following this trend a growing number of cache partitioning proposals that require hardware support are being evaluated in real systems.

Below, we discuss cache partitioning approaches, classifying them into two main groups according to the third characterization axis; that is, if a given proposal has been implemented and evaluated on a real machine or on a simulator. For each group, further refinements are done, separating proposals depending on the second axis. Finally, the goal of each proposal (first axis) is briefly explained.

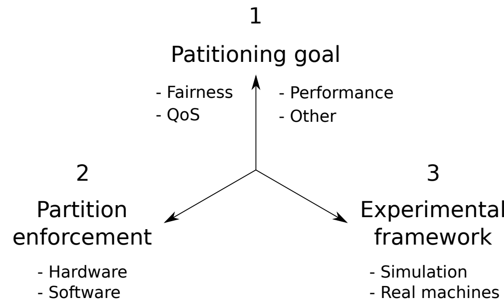


Figure 2.1: Three axes taxonomy of cache partitioning proposals.

Dynamic LLC management in real machines. This approaches can be divided in two categories depending on if they are hardware- or software-based. Regarding hardware-based proposals, this section focuses on the ones that use Intel's Cache Allocation Technology, a feature of some recent Intel processors that provides cache partitioning support. These proposals use CAT to control the LLC usage, targeting different goals. The first work using a prototype of this technology is the one by Cook *et al.* [12], where they measure the energy and performance benefits of a LLC-partitioning scheme and propose a simple dynamic partitioning policy. Later proposals, like Heracles [56] and Dirigent [103], utilized more mature versions of CAT. They both focus on maximizing utilization in large-scale datacenters without affecting user-perceived latency in latency-critical applications. To do so, they classify applications *a priori* as *batch* or *latency-critical*, and use CAT to limit the amount of cache resources that batch applications can consume. Another proposal, Ginseng [26], focuses on cloud computing providers that rent virtual machines, and uses a market-driven auction system to partition the LLC into non-overlapping partitions depending on how much each guest is willing to pay and how that affects the rest. El-Sayed *et al.* [76] has a more broad goal, targeting performance. They group applications into clusters, assigning them to different CLOSes. While it manages to significantly improve throughput in selected workloads, it uses a detailed profiling, which introduces overhead.

Regarding software-based partitioning proposals, a significant amount of work has focused on this cache partitioning technique. Most of it is based on page-coloring [101, 89, 102, 81]. With page-coloring one can control into which cache sets the application data is mapped, effectively partitioning the cache with set granularity. Some of them, such as [55], perform application profiling at runtime and choose the page coloring used. However the overhead of this profiling can be high. Solari *et al.* [77] use page coloring, but with the novelty

of considering an LLC addressing scheme similar to the one used by Intel Sandy Bridge processors.

Partitioning proposals using simulation frameworks. Approaches in this group propose the use of extra hardware was not available in commercial processors, so their evaluation was performed using simulation frameworks.

UCP [70] and ASM-Cache [87] use set sampling and duplicate cache tags to gather information that is later used to partition the cache with way level granularity. Gupta and Zhou [29] also use way-partitioning to divide the cache while increasing spatial locality with aggressive prefetching. Kim *et al.* [49] improve system fairness by partitioning the shared L2 cache. However, their approach requires offline profiling, which makes it impractical. These proposals work by keeping a count of the blocks that each core has in a given set. Therefore, when there is an eviction, if the core that causes it is occupying more lines than what the partitioning policy states, one of its blocks is selected as the victim. On the contrary, if it has less lines than assigned, the victim block is selected from another core.

Other proposals use a different approach to enforce cache partitions. This is the case of PriSM [58], which manages the cache occupancy of the different cores by directly controlling the eviction probabilities of their cache lines. Similarly, Kahn *et al.* [48] modify the replacement policy to dynamically create two logical partitions, one for clean lines and another for dirty lines, with different eviction probabilities. Futility Scaling (FS) [94] is yet another replacement-based cache partitioning scheme. Its goal is to precisely partition the cache while still maintaining high associativity even with a large number of partitions.

A different approach is used in Vantage [75] and Ubik [46]. Both employ ZCaches [74] to partition the cache with block granularity. Vantage optimizes partitions to enhance the performance, while Ubik ensures QoS and improves the performance of batch applications. Iyer [42] presents the CQoS cache management framework, which provides prioritized service to multiple heterogeneous threads sharing a cache structure. Chang and Sohi [9] select multiple partitions and enforce them in a time-sharing manner across multiple epochs within a stable program phase. They propose a QoS metric that modulates the allocated cache space for a given thread.

Chapter 3

Experimental Framework

This chapter describes the experimental framework used to perform the experiments presented in this thesis. The results presented in chapters 4 and 5 have been obtained with the Multi2Sim simulation framework, while the results of Chapter 6 have been gathered implementing the approach on a real machine. This chapter presents the characteristics of both systems, the benchmark suites used, and the metrics studied. The workloads employed to evaluate both systems, have been the same: sets of applications from the SPEC CPU 2006 and the NASA Advanced Supercomputing benchmark suites.

3.1 Simulation Framework

The proposals presented in chapters 4 and 5 of this dissertation propose the use of additional hardware that is not available in commercial processors. For this reason, they have been implemented and evaluated using the simulation software packages described in this section.

3.1.1 Multi2Sim

Multi2Sim [91] is a cycle accurate event driven simulation framework for CPU-GPU heterogeneous computing written in C. It includes models for superscalar, multithreaded, and multicore CPUs, as well as GPU architectures.

The CPU simulation framework consists of two major interacting software components: the functional simulator and the architectural simulator. The functional simulator (i.e. emulator) mimics the execution of a guest program on a native x86 processor, by interpreting the program binary and dynamically reproducing its behavior at the ISA level. The architectural simulator (i.e. detailed or timing simulator) obtains a trace of x86 instructions from the functional simulator, and tracks execution of the processor hardware structures on a cycle-by-cycle basis.

The experimental results of this dissertation have been obtained using version 4.2 of Multi2Sim, which supports the execution of a number of different benchmark suites without requiring any porting, including SPEC CPU 2006, as well as custom self-compiled user code. The architectural simulator models many-core superscalar pipelines with out-of-order execution, a complete memory hierarchy with cache coherence, interconnection networks, and can be easily extended to model additional components.

A key shortcoming of Multi2Sim is that it does not implement a detailed model of the main memory or the memory controller. Consequently, main memory requests have no contention at all and latencies are constant values. This is not realistic [44] and a serious limitation when evaluating proposals that deal with the memory subsystem, like the ones presented in this dissertation. To overcome this, we have integrated DRAMSim2 [72], a dedicated main memory simulator, into the Multi2Sim framework.

3.1.2 DRAMSim2

DRAMSim2 is an open source cycle accurate simulator that implements detailed timing models for a variety of existing memories, including DDR3. It models commercial DRAM devices, as the ones used in commodity DIMM modules. DRAMSim2 faithfully models the internal organization of the DRAM devices, which are composed of multiple independent banks that can be accessed in parallel.

As in real hardware, DRAMSim2 allows to group the devices into ranks working in lockstep, ensuring that all the required timing constraints are meet.

Note that DRAMSim2 models the memory refresh and its timing constraints and the In addition to the structure of the DRAM memory system, this simulator accurately models the command-based protocol to access the DRAM banks which, in its most simple form, is composed of three commands: i) a precharge command to precharge the row bitlines, ii) an activate command to open the row corresponding to the row address into the row buffer, and finally iii) a read/write command to access the row buffer at the position indicated by the column address. DRAMSim2 also models the memory controller. Specifically, it defines the Row-Buffer-Management Policy, the Address Mapping Scheme, and the Memory Transaction and DRAM Command Ordering Scheme.

The high level of detail of the model allows DRAMSim2 to accurately estimate the power consumption of the modeled DRAM devices from their configuration parameters and the memory requests served.

3.1.3 Baseline Machine Parameters of the Simulated Systems

The baseline system used throughout the simulation related chapters (i.e. Chapter 4 and Chapter 5) consists of four major components: the cores, the cache hierarchy, the interconnection network and the main memory. This system is an Out-of-Order multicore with 2, 4, or 8 cores, depending on the specific experiment performed, running at 3GHz. Each core has private 32KB 8-way L1 caches, while the LLC is private and has 256KB when evaluating the proposal presented in Chapter 4, and is shared and has a capacity of 1MB per core when evaluating the proposal presented in Chapter 5.

For the evaluation of the selective prefetcher proposal presented in Chapter 4 a prefetcher engine that brings blocks to the LLC is added to this baseline system. This prefetcher engine is a CZone-based stride prefetcher that detects strided accesses by keeping a record of previous cache accesses in a Global History Buffer [65, 64].

The basic idea behind this prefetcher is to dynamically partition the physical address space in different zones, referred to as CZones, and to detect strided references within each of these zones. The processor sets the size of the zones by storing a mask in memory-mapped references. Usually CZones have the same size as pages. Strided references within each partition are dynamically detected by using a finite state machine, which checks whether the last three accesses are offsetted by a fixed stride. If so, a pattern has been detected and the engine triggers new prefetches. Prefetches based on history buffers

indexed by CZones have the desirable property of not needing the program counter value of the load or store instruction that accessed the cache to predict the following accesses, which is interesting for L2 and lower cache levels. The maximum aggressiveness of the prefetcher has been set to 4 cache blocks (i.e. 4 blocks are brought to the cache when a stride is detected).

In addition, the NoC and the memory controller have been also modeled in detail for the sake of accuracy. The NoC, which connects all the cores and memory controllers has been modeled as a ring, as in Intel multicores. Congestion and contention in the NoC are realistically modeled, since they may contribute to the latency perceived by the core when accessing the memory subsystem [79, 68, 34].

The system has a main memory controller with an unified queue for both demand requests and prefetches. The parameters of the DRAM devices modeled with DRAMsim2 have been set according to a commercial MICRON DDR3 memory device [90]. In addition, requests to main memory are scheduled on a first come first served (FCFS) basis with a closed-page row buffer policy. The address mapping used to access the DRAM memory is shown in Figure 3.1. Note that the simulated processor's maximum address space is 4GB, so addresses are 32 bits wide. In addition, the number of bits for row and column are fixed (device dependent) and the number of banks in DDR3 is 8. Since the bus is 64 bits wide [45] the number of bits for the byte offset is also fixed. That limits the number of ranks and banks that can be addressed.

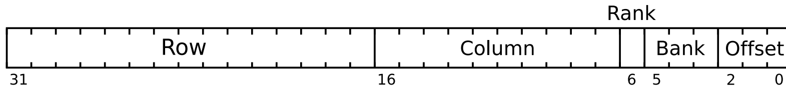


Figure 3.1: Main memory address mapping.

Table 3.1 summarizes the main configuration parameters of the cores, the cache hierarchy, the interconnection network and the main memory of the system.

3.2 Real Machine: The Intel Xeon E5 2658A v3

All the experiments presented in Chapter 6 have been performed on the Intel Xeon E5 2658 v3 processor [39]; one of the first Intel processors to support Cache Allocation Technology (CAT).

This processor implements HyperThreading and is deployed with twelve cores supporting up to two simultaneous threads each. However, to avoid intra-core

Core characteristics	
Core count	2/4/8 cores at 3GHz
Issuing policy	Out-of-order
Issue/Commit width	4 instructions/cycle
ROB size	128 entries
Load/Store queue	64/48 entries
Cache hierarchy	
IL1 (private)	32KB, 8ways, 64B-line, 2cc
DL1 (private)	32KB, 8ways, 64B-line, 2cc
LLC (private) (4 cores, Chapter 4)	256KB, 16ways, 64B-line, 11cc, 16 MSHR
LLC (shared) (2 cores, Chapter 5)	2MB, 16ways, 64B-line, 11cc, 16 MSHR
LLC (shared) (4 cores, Chapter 5)	4MB, 32ways, 64B-line, 11cc, 16 MSHR
LLC (shared) (8 cores, Chapter 5)	8MB, 32ways, 64B-line, 11cc, 16 MSHR
Interconnection network	
Topology	Ring
Input/ouput buffer size	128B
Link bandwidth	64B/cycle
Main memory & memory controller	
Memory controllers	1
DRAM bus freq.	1066MHz
DRAM device	DDR3 (2133 Mtransfers/cycle)
Latency	t_{RP}, t_{RCD}, t_{CL} 13.09ns each
DRAM banks	8
Page size	8KB
Burst length (BL)	8
Scheduling policy	FCFS

Table 3.1: Simulated system configuration.

interference, experiments have been conducted allocating a single thread per core. The base core frequency its 2.20 GHz, up to 2.90 GHz with Turbo Boost. Each core includes a 32 KB L1 data cache and a 256 KB L2 cache, both of which are private. All the cores share an L3 cache, the LLC in the system, with 30 MB and 20 ways (which gives an average of 2.5 MB per core). The entire cache hierarchy is inclusive [36]. The processor cores, memory controllers and data buses are connected using a NoC composed of two rings, as depicted in Figure 3.2.

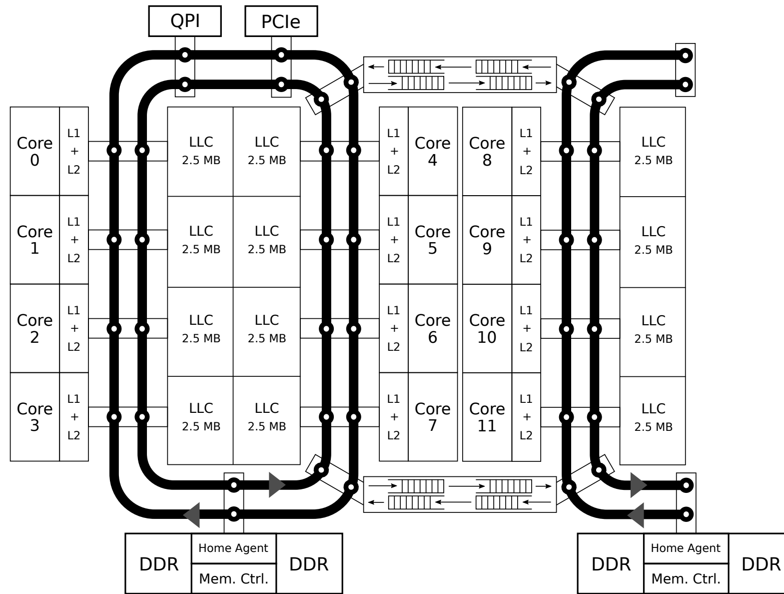


Figure 3.2: Block diagram for the Intel Xeon E5 2658A v3.

3.2.1 Intel Cache Allocation Technology

Intel's Cache Allocation Technology provides primitives that can be used to control the maximum amount of cache space that a given application, container, virtual machine or hardware thread can consume. It works by partitioning the cache with cache way granularity. With Intel CAT, the operating system, a hypervisor or even user code can use CAT to dynamically isolate or prioritize specific applications to improve their performance, for security reasons or to provide QoS.

CAT has three key concepts: Class of Service (CLOS), Resource Monitoring ID (RMID), and Capacity Bitmask. The RMIDs are IDs assigned to applications, containers, virtual machines, etc. that will have its cache space monitored and controlled by CAT. A CLOS encompasses a set of RMIDs and a Capacity Bitmask, where the bitmask marks the cache ways that can be *written* by the RMIDs. The Capacity Bitmasks can overlap, which means that some ways can be shared by different classes of service. A limitation of CAT is that the writable ways defined in a Capacity Bitmask have to be contiguous. For instance, a CBM like `1111-0000-1111-0000` would not be valid. Note that CAT has no effect by default, since all the applications start mapped to

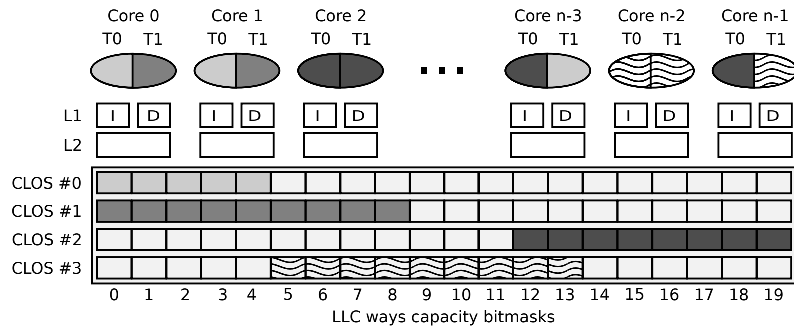


Figure 3.3: Cache Allocation Technology example.

CLOS #0, which, if not modified, has a CBM that allows full access to all the LLC ways.

This technology is available in a subset of the Intel Xeon E5-2600 v3 processors and in all the processors of the Intel Xeon E5 v4 family. The Xeon E5 v4 family provides more CLOSes than prior-generation processors, moving from 4 CLOSes to 16. While this allows for a greater flexibility, the partitioning is done in a per-way granularity, so the limiting factor is usually the LLC associativity, usually 20 ways.

There are three different ways of configuring Intel CAT. The first one is to use Machine Specific Registers (MSR), as stated in the Intel's Software Developer's Manuals [37]. Another option is to use the Intel-provided user space library [41], and the last option is to use the Linux kernel interface to CAT (*resctrl*), that first appeared in Linux 4.10. This dissertation makes use of the first approach, the Intel-provided user space library.

Figure 3.3 shows an example of a possible cache partitioning scheme in a processor of the Xeon E5 2600 v3 family. Each of the four possible classes of service (CLOS 0 to CLOS 3) has a subset of the 20 ways of the LLC assigned, and each thread is assigned to a CLOS. Each CLOS is identified by a color/pattern which marks both the threads that belong to the CLOS and the ways they can write. For instance, thread 0 of core $n-3$ is assigned to CLOS 2 and thread 1 to CLOS 0. Note that all the CBMs are contiguous and that CLOS 3 shares some of its assigned ways with CLOS 1 and CLOS 2.

3.3 Benchmark Suites

The proposals presented in this dissertation have been evaluated using a wide set of benchmarks from the SPEC CPU 2006 benchmark suite [86] with the ref input set and the NASA Advanced Supercomputing Parallel Benchmark suite [63] (single-threaded runs). A brief description of the benchmarks used is presented below.

3.3.1 SPEC CPU 2006

The SPEC CPU 2006 benchmark suite is a CPU-intensive benchmark suite, stressing a system's processor, memory subsystem and compiler, developed by the Standard Performance Evaluation Corporation. This suite contains different applications from the High Performance Computing field and is divided between integer and floating point benchmarks. The integer benchmarks are listed next.

perlbench: C, Programming Language. Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).

bzip2: C, Compression. Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.

gcc: C, C Compiler. Based on gcc Version 3.2, generates code for Opteron.

mcf: C, Combinatorial Optimization. Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.

gobmk: C, Artificial Intelligence. Plays the game of Go, a simply described but deeply complex game.

hmmer: C, Search Gene Sequence. Protein sequence analysis using profile hidden Markov models (profile HMMs).

sjeng: C, Artificial Intelligence: chess. A highly-ranked chess program that also plays several chess variants.

libquantum: C, Physics / Quantum Computing. Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.

h264ref: C, Video Compression. A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2.

omnetpp: C++, Discrete Event Simulation. Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.

astar: C++, Path-finding Algorithms. Pathfinding library for 2D maps, including the well known A* algorithm.

xalancbmk: C++, XML Processing. A modified version of Xalan-C++, which transforms XML documents to other document types.

The remaining benchmarks of the suite make intensive usage of floating point arithmetics.

bwaves: Fortran, Fluid Dynamics. Computes 3D transonic transient laminar viscous flow.

gamess: Fortran, Quantum Chemistry. Gamess implements a wide range of quantum chemical computations. For the SPEC workload, self-consistent field calculations are performed using the Restricted Hartree Fock method, Restricted open-shell Hartree-Fock, and Multi-Configuration Self-Consistent Field.

milc: C, Physics / Quantum Chromodynamics. A gauge field generating program for lattice gauge theory programs with dynamical quarks.

zeusmp: Fortran, Physics / CFD. ZEUS-MP is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, University of Illinois at Urbana-Champaign) for the simulation of astrophysical phenomena.

gromacs: C and Fortran, Biochemistry / Molecular Dynamics. Molecular dynamics, i.e. simulate Newtonian equations of motion for hundreds to millions of particles. The test case simulates protein Lysozyme in a solution.

cactusADM: C and Fortran, Physics / General Relativity. Solves the Einstein evolution equations using a staggered-leapfrog numerical method.

leslie3d: Fortran, Fluid Dynamics. Computational Fluid Dynamics (CFD) using Large-Eddy Simulations with Linear-Eddy Model in 3D. Uses the MacCormack Predictor-Corrector time integration scheme.

namd: C++, Biology / Molecular Dynamics. Simulates large biomolecular systems. The test case has 92,224 atoms of apolipoprotein A-I.

dealII: C++, Finite Element Analysis. C++ program library targeted at adaptive finite elements and error estimation. The testcase solves a Helmholtz-type equation with non-constant coefficients.

soplex: C++, Linear Programming / Optimization. Solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military airlift models.

povray: C++, Image Ray-tracing / Image rendering. The testcase is a 1280×1024 anti-aliased image of a landscape with some abstract objects with textures using a Perlin noise function.

calculix: C and Fortran, Structural Mechanics. Finite element code for linear and nonlinear 3D structural applications. Uses the SPOOLES solver library.

GemsFDTD: Fortran, Computational Electromagnetics. Solves the Maxwell equations in 3D using the finite-difference time-domain (FDTD) method.

tonto: Fortran, Quantum Chemistry. An open source quantum chemistry package, using an object-oriented design in Fortran 95. The test case places a constraint on a molecular Hartree-Fock wavefunction calculation to better match experimental X-ray diffraction data.

lbm: C, Fluid Dynamics. Implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D.

wrf: C and Fortran, Weather. Weather modeling from scales of meters to thousands of kilometers. The test case is from a 30km area over 2 days.

sphinx3: C, Speech recognition. A widely-known speech recognition system from Carnegie Mellon University.

3.3.2 NAS Parallel Benchmarks

The NAS Parallel Benchmarks (NPB), designed by the NASA Advanced Supercomputing Division, are a small set of programs designed to help evaluate the performance of supercomputers. The benchmarks are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudo-applications. Problem sizes in NPB are predefined and indicated as different classes. The five kernels and pseudo-applications are listed below.

IS: Integer Sort, random memory access.

EP: Embarrassingly Parallel.

CG: Conjugate Gradient, irregular memory access and communication.

MG: Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive.

FT: discrete 3D fast Fourier Transform, all-to-all communication.

BT: Block Tri-diagonal solver.

SP: Scalar Penta-diagonal solver.

LU: Lower-Upper Gauss-Seidel solver.

Chapter 4

Selective Prefetching under Limited Memory Bandwidth

This chapter introduces the concept of *selective prefetching*, where individual prefetchers are activated, throttled or deactivated to improve both main memory energy consumption and performance. It proposes ADP, a prefetcher that deactivates local prefetchers in some cores when they present low performance and co-runners need additional main memory bandwidth. Then, based on heuristics, an individual prefetcher is reactivated or throttled up when performance enhancements are foreseen.

4.1 Introduction

Addressing memory latencies is a major design concern in modern multicores. In this regard, hardware prefetching plays a key role in modern high-performance processors with deep cache hierarchies. Modern microprocessors [82, 84, 66, 1] implement multiple prefetchers, which work along the different levels of the cache hierarchy. For example, the IBM POWER8 [82, 4] has an instruction cache prefetcher and a data cache prefetcher. The instruction prefetcher fetches up to three sequential cache lines in single thread mode. The stream based data prefetcher detects strides in load requests and

optionally store requests, issuing prefetches in all the three levels of the cache hierarchy. Modern Intel processors [1], have four prefetchers per core two L1-data cache prefetchers and two L2 prefetchers. The L1 prefetchers are an One Block Lookahead prefetcher (OBL), that fetches the next cache line, and a prefetcher that detects strides in the load history by indexing the loads with the program counter. The two L2 cache prefetchers are an L2 adjacent cache line prefetcher, and another one that fetches additional cache lines.

In current processors, prefetching requests from multiple cores (i.e. applications) compete with regular memory requests for off-chip main memory bandwidth. Therefore, since prefetching is a speculative technique, it increases the total number of accesses to main memory [6, 52, 25]. In scenarios of high memory bandwidth consumption, this fact can turn into significant performance losses in some individual applications, which are affected by the prefetches of their co-runners.

A straightforward solution to increase bandwidth availability would be to turn off all the individual prefetchers and, therefore, remove all the speculative prefetches. However, this is not an acceptable solution from a performance perspective, since aggressive hardware prefetching can bring large performance improvements in some applications. A solution recently proposed [85] to keep prefetches under control is to implement throttling up/down mechanisms that regulate the aggressiveness of the prefetchers. An individual prefetcher is then throttled down when no performance benefits are expected. However, due to limited bandwidth, keeping active the prefetchers when no benefits are expected, even with a low aggressiveness, may damage the performance of some applications, due to inter-application interference.

To provide further insights on the impact of prefetching on performance and bandwidth consumption, this chapter characterizes the relation between main memory accesses (prefetches and regular accesses) and IPC, for the applications in the SPEC CPU 2006 benchmark suite. The study shows that some applications exhibit different execution phases from the main memory and prefetching points of view. Some of these phases are highly benefited by prefetching, while some others are negligibly benefited or even negatively affected. The amount of main memory bandwidth consumed also is phase-dependant: some phases consume significant amounts of memory bandwidth, some others consume much less bandwidth. This study suggests that, in multi-core execution, throttling the memory bandwidth consumed by the prefetches of applications in prefetch unfriendly phases may improve the performance of other co-runners.

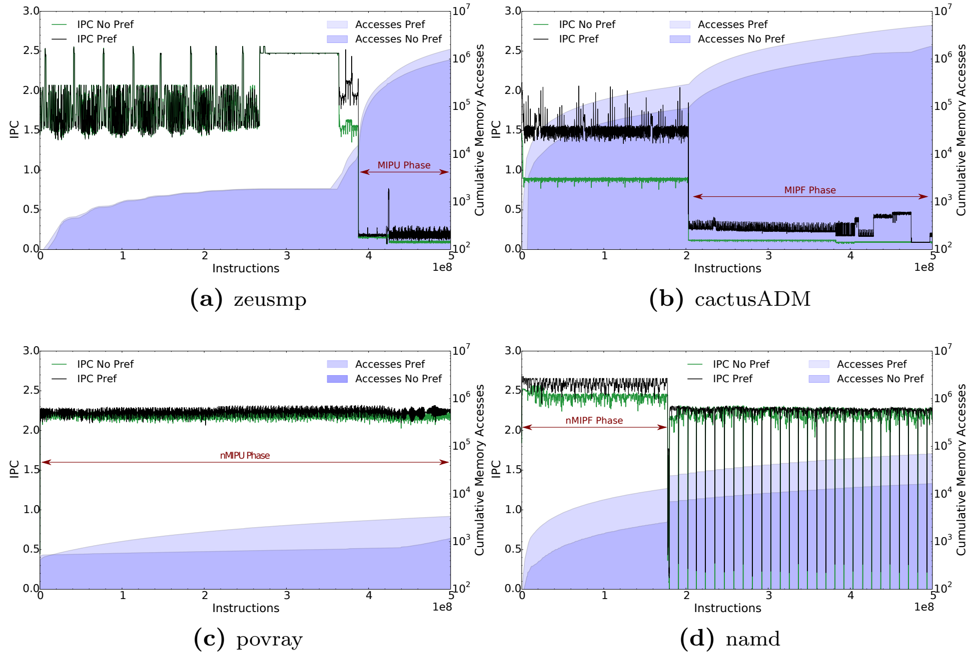
This chapter proposes the Activation/Deactivation Prefetcher (ADP), which in addition to throttle up/down the aggressiveness, activates and deactivates individual per-application prefetchers considering both local and global information (i.e. inter-application interference). Deactivation policies turn off the prefetcher in specific cores, thereby increasing the available bandwidth for those prefetchers that require it to improve their cores' performance. Activation policies rely on activation conditions that estimate when an individual prefetcher could improve the performance. The key challenge of activation conditions is that they cannot have updated information about the prefetcher activity, since it is disabled. Therefore, either data from previous phases or other metrics should be used.

4.2 Characterization Study

The aim of this section is to study the relation between memory activity, prefetching, and performance (i.e. IPC) in order to provide insights in the design of selective prefetchers. For this purpose, first, all the benchmarks are analyzed in isolation with the aim of identifying those execution phases where prefetching can bring benefits, paying special attention to phases with high memory activity, since in such cases, a selective prefetcher can potentially provide extra main memory bandwidth for the co-runners.

4.2.1 Characterizing Benchmark Phases

To analyze the benefits of prefetching in individual applications, all the benchmarks have been run in isolation in a system with and without the baseline prefetcher described in Section 3.1.3. Based on the gathered results, the phases of the applications have been classified according to two main axes: bandwidth consumption (intensive or not), and the effect of the prefetcher on IPC (friendly or not), which gives four possible categories. For illustrative purposes, Figure 4.1 shows examples of benchmarks showing execution phases belonging to each category. Each graph shows the IPC evolution of an application (left Y axis) with and without prefetching. To analyze the relationship with the memory activity, the cumulative amount of memory accesses (right Y axis) is also shown in the same plot using a logarithmic scale, broken down into regular and prefetch requests, labeled as *Accesses No Pref* and *Accesses Pref*, respectively. Below, the four categories are presented, highlighting the main characteristics of each of them:



(a) zeusmp

(b) cactusADM

(c) povray

(d) namd

Figure 4.1: Characterization study. Examples of benchmarks in the different categories. No prefetching compared to aggressive prefetching.

- *Memory Intensive, Prefetch Unfriendly (MIPU).* This category refers to memory intensive phases where prefetching does not improve the performance. This kind of phase can be observed at the end of the execution of `zeusmp` (see Figure 4.1a).
- *Memory Intensive, Prefetch Friendly (MIPF).* This category includes memory intensive phases where prefetching brings important performance benefits. This kind of behavior dominates through the entire execution of some benchmarks, like `cactusADM`, as shown in Figure 4.1b.
- *non Memory Intensive, Prefetch Unfriendly (nMIPU).* These phases refer to those parts of the execution where the main memory activity is rather low and prefetching brings scarce or no benefit. Examples of phases of this category can be observed in `povray` across all its execution (see Figure 4.1c) and at the beginning of the execution of `zeusmp`.

- *non Memory Intensive, Prefetch Friendly (nMIPF)*. This category refers to non memory intensive phases in which prefetching still boosts the performance. This behavior can be observed in Figure 4.1d during the first part of the execution of `namd`.

An execution phase is defined as an interval of the execution of an application where the IPC behavior is homogeneous or follows the same pattern. Therefore, a new phase of the execution starts when the IPC changes its trend, so a given phase does not has a predetermined length but it varies according to the application behavior. We found that these changes are usually related to variations in memory activity.

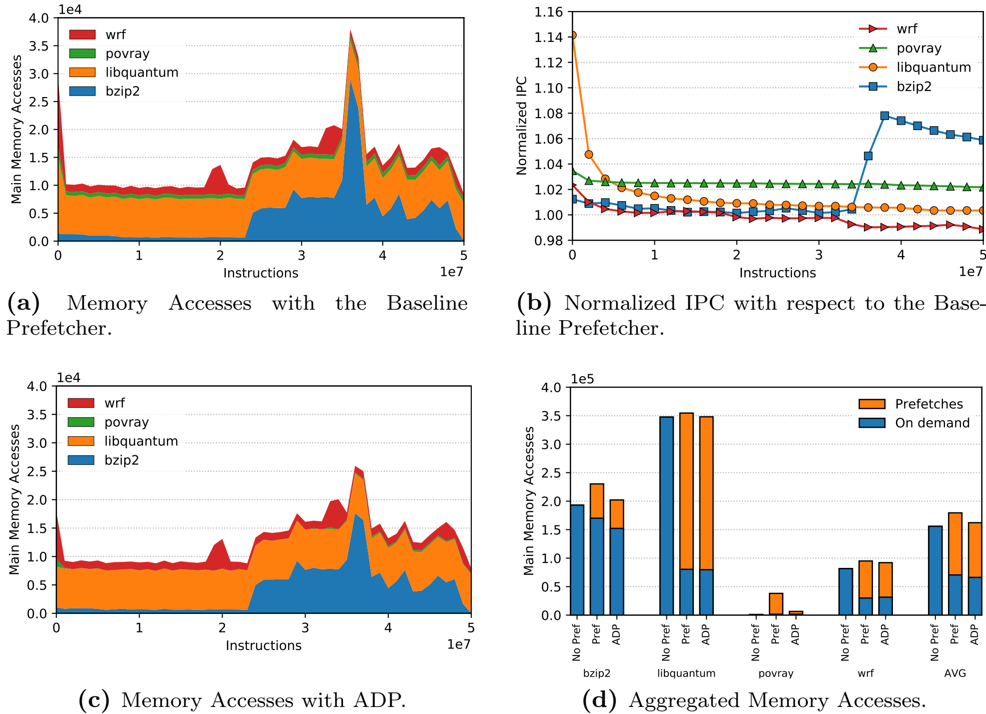
During prefetch unfriendly phases (categories MIPU and nMIPU), the prefetcher could be turned off or throttled down with minimal impact on performance. This claim can be observed at the end of the execution in Figure 4.1a, where a significant amount of prefetches (notice the log scale) brings minor performance benefits. Therefore, deactivating the prefetcher could result in important main memory energy savings, especially in MIPU phases where a high number of accesses can be reduced.

In prefetch friendly phases (categories MIPF and nMIPF), the prefetcher should be enabled to enhance the performance; however, its aggressiveness should be adjusted. This would reduce wasted energy and it is particularly useful when executing multiprogram workloads, in order to leave more bandwidth available to the co-runners. Especial care should be taken in the case of nMIPF phases, since the potential energy savings might not justify the possible performance losses.

4.2.2 Analysis in Multiprogram Execution

As mentioned above, prefetching brings scarce or null benefits in *MIPU* applications/phases in spite of having high memory activity. This observation is especially relevant when individual prefetching requests compete among them for main memory resources. Therefore, if prefetching were selectively disabled in specific cores (and enabled when required), then, an extra amount of bandwidth would become available for those co-running applications that really benefit from it. Moreover, important savings in main memory energy could be achieved.

This claim can be observed in Figure 4.2, which compares the memory activity and performance of a selective prefetcher (ADP, the prefetcher presented in Section 4.3) with respect to an aggressive prefetcher when executing a 4-



(a) Memory Accesses with the Baseline Prefetcher.

(b) Normalized IPC with respect to the Baseline Prefetcher.

(c) Memory Accesses with ADP.

(d) Aggregated Memory Accesses.

Figure 4.2: Selective ADP prefetcher vs. baseline prefetcher.

application workload. The IPC of the execution with ADP is normalized over the baseline aggressive prefetcher. The value for each point has been computed from the start of the execution. Therefore, the last value represents the final normalized performance. It can be appreciated that the selective prefetcher reduces the number of memory accesses (compare figures 4.2a and 4.2c) by detecting phases in the individual benchmarks where the prefetchers can be disabled. Not only does the reduction in prefetchers not decrease the performance, but instead, as shown in Figure 4.2b, it increases it.

There are two points in Figure 4.2b of the execution of the applications that are interesting to analyze. The first one, at the beginning of the execution, is a significant increase in performance in comparison with the baseline prefetcher. It appears because the selective prefetcher reduces the amount of prefetches issued by `libquantum` and `wrf`. This reduction unclogs the main memory access, improving the IPC across all the benchmarks. The other point of in-

terest occurs when approximately $3.5e7$ instructions have been executed. At this point, the main memory suffers an important congestion that bottlenecks the system performance. Consequently, the reduction of prefetches alleviates the congestion, reducing the perceived memory latency, which turns into performance enhancements. This claim can be observed in the 8% rise in the IPC of `bzip2` at the same point of the execution in Figure 4.2b. This application is the most affected one, since it is the most memory intensive at that point.

Note that the minor performance loses exhibited by `wrf` (around 1%) are because the selective prefetcher prioritizes unclogging the main memory access and reduces the aggressiveness of the core where `wrf` is being executed. However, this slight reduction in the performance of this application is clearly compensated by the increase in performance that the co-runners experience.

The reduction in main memory accesses can be also appreciated in Figure 4.2d, which presents, for each approach (not prefetching, baseline prefetcher and ADP), the total amount of main memory reads, classified in two main groups: prefetch requests and on demand accesses. Looking at the figure, two important observations can be drawn: i) the baseline prefetcher significantly increases the total amount of main memory accesses in some applications over not prefetching, while ADP does not suffer this drawback; and ii) an important fraction of on demand accesses are replaced by prefetches, which indicates that the prefetches are useful and their timeliness is adequate, since the block is already in cache when requested, and thus, the main memory access is not performed.

In summary, this analysis has shown that selective prefetching can provide a good trade-off between main memory accesses reduction (and therefore, energy savings) and performance. Moreover, a good design could enhance both of them. The key challenge that designers must face is devise policies to decide when individual prefetchers should be either activated or deactivated, and the adequate prefetching aggressiveness.

4.3 ADP Mechanism

The proposed ADP approach, apart from throttling up or down core prefetchers, also selectively activates and deactivates them, considering both local and global information. This information is used by the devised mechanism to better distribute the available memory bandwidth among the competing cores, leading to a more effective prefetching scheme.

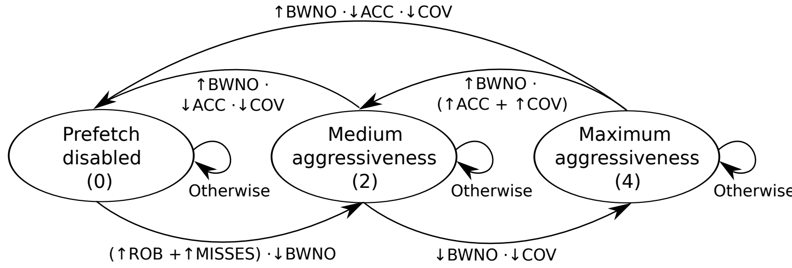


Figure 4.3: ADP aggressiveness states and transition rules.

This section introduces the Finite State Machine (FSM) that governs the behavior of the selective prefetcher and discusses the throttling/deactivation and activation policies.

4.3.1 Finite State Machine of the ADP Prefetcher

Existing prefetchers generally use two metrics, accuracy (ACC) and coverage (COV), to quantify the prefetcher performance. They are depicted in Equation 4.1 and Equation 4.2, respectively.

$$Accuracy = \frac{Useful_Prefetches}{Total_Prefetches} \tag{4.1}$$

$$Coverage = \frac{Useful_Prefetches}{Misses + Useful_Prefetches} \tag{4.2}$$

Based on these performance metrics, the aggressiveness is throttled accordingly. A recent metric that is being considered in multicore execution is the memory bandwidth needed by others (BWNO) [19]. It is computed as follows. First, we define the Bandwidth Consumed by Core i (BWC_i) on a given cycle as the number of DRAM banks servicing requests from core i. Therefore, for a system with DDR3 memory modules and only one main memory rank, this value is between 0 and 8. Additionally, we define the Bandwidth Needed by Core i as the number of banks that are busy serving a request from a core other than i, and that have requests pending from core i. Based on these definitions, the Bandwidth Needed by Others (BWNO) from the perspective of core i is computed as the sum of the bandwidth needed by the cores other than i. While BWNO values are computed every cycle, they are averaged for 50K-cycle intervals.

Thresholds				
Low Accuracy	Low Coverage	Rise in misses	High % ROB stall	High BWNO
$< 40\%$	$< 30\%$	$> 15\%$	$> 60\%$	> 2.75 banks

Table 4.1: Thresholds used in ADP.

ADP uses the three mentioned metrics not only to throttle down the prefetcher aggressiveness but also to completely deactivate the mechanism when it is estimated that the prefetcher is not obtaining performance benefits. In addition, an extra set of performance metrics has been explored to reactivate the prefetcher. The final design uses two metrics to activate the prefetcher (see Section 4.3.3): the percentage of time the Reorder Buffer (ROB) is stalled due to a long latency memory access (referred to as ROB condition) and the increase in the number of L2 misses (MISSES condition).

Notice that simple hardware is required to implement the ADP prefetcher and much of it is based on performance counters already available in current processors [38]. Both the *per process* number of cache misses and stall cycles can be gathered on most current commercial processors with the available performance counters. ACC can be calculated as the ratio of two hardware counters that keep track of the number of useful prefetches and the total number of prefetches in each core. This can be done by adding a single bit to each cache entry to indicate that the block has been prefetched [83, 28]. The first counter is updated when there is a hit in a prefetched block, and the second one is increased each time a prefetch is issued. COV requires a counter for misses and a counter for keeping track of useful prefetches, already described. With respect to BWNO, it requires three simple counters in the memory controller which are updated every cycle.

Figure 4.3 depicts the FSM of the ADP prefetcher. The number between brackets within each node represents a prefetcher aggressiveness level and the arcs represent transitions between states. Transitions are labeled with the condition driving the corresponding state change. Upward and downward arrows in the labels mean high or low values (e.g. high or low accuracy), respectively, compared to a threshold (see Table 4.1). These parameters were empirically determined using a limited number of simulation runs and optimized to reduce the number of memory accesses. Therefore, further performance improvements could be achieved, but at the cost of increasing the number of main memory accesses.

if *co-runners need more bandwidth (BWNO)* **then**
  if *low accuracy* **and** *low coverage* **then**
    disable prefetch $(2 \rightarrow 0 \parallel 4 \rightarrow 0)$
  else
    reduce local aggressiveness $(2 \rightarrow 2 \parallel 4 \rightarrow 2)$
else
  if *low coverage* **then**
    increase local aggressiveness $(2 \rightarrow 4 \parallel 4 \rightarrow 4)$

Figure 4.4: Deactivation/throttling algorithm.

The devised policies adjust the prefetchers depending on the values of the mentioned performance metrics, which are gathered during fixed-length intervals of 50K processor cycles. At the end of each interval, the hardware logic determines the machine state for the following interval. Below, the deactivation, throttling and activation policies are discussed in detail.

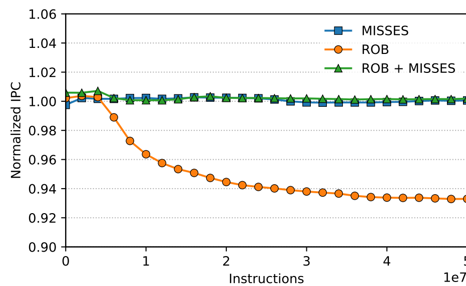
4.3.2 Deactivation and Throttling Policy

While the prefetcher is active, this policy is applied at the end of each interval to decide, using the gathered data, if a prefetcher state change is required. Three decisions can be taken: throttle up the aggressiveness, throttle down the aggressiveness, or turn off the prefetcher. Algorithm 4.4 depicts the conditions that must be satisfied to carry out such actions. On the right side of each action, the associated transition in the FSM (Figure 4.3) is presented.

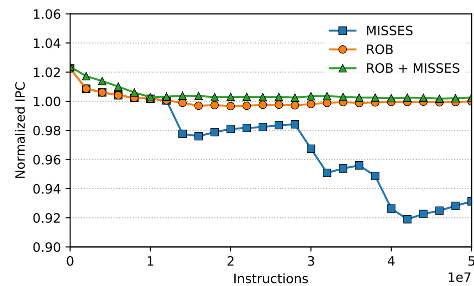
When some co-runners need more bandwidth, the option to reduce or even deactivate the local prefetcher is checked. In case the local prefetcher is performing poorly (low accuracy and coverage), then the prefetcher is completely disabled. Otherwise, the aggressiveness is set to a medium level (remember that ADP aggressiveness levels are to 0 –disabled–, 2, and 4) to increase memory bandwidth availability for the co-runners. On the other hand, if BWNO is not a constraint and the local prefetcher is not saving enough cache misses (low coverage), then the mechanism speculatively increases the aggressiveness (aggressiveness level is set to 4, the maximum value) to improve its performance. If this increase in aggressiveness does not work well, the algorithm returns to the previous aggressiveness in the subsequent interval.

if *sudden rise in misses (MISSES)* **or** *high ROB stalls due to memory instructions (ROB)* **then**
  if *co-runners do not need more bandwidth (BWNO)* **then**
    activate prefetcher $(0 \rightarrow 2)$

Figure 4.5: Activation algorithm.



(a) Performance of mcf in execution with namd, omnetpp and soplex.



(b) Performance of cactusADM in execution with gcc, hmmer and libquantum.

Figure 4.6: Normalized IPC of ADP over the baseline prefetcher using different activation conditions.

4.3.3 Activation Policy

This policy is applied to cores with a disabled prefetcher to decide whether it should be reactivated for the next interval. When a local prefetcher is disabled, all the prefetcher related structures like stride and pattern detection are disabled, thus no information about the prefetcher activity is available.

The proposed activation policy estimates if there were noticeable performance losses with respect to the last interval the prefetcher was enabled, and based on this information it decides if the prefetcher should be reactivated. The rationale for this is that execution phases tend to exhibit the same behavior for a relatively long time, as found in Section 4.2.1. The challenge is choosing the adequate performance metrics.

Several metric were explored to drive the activation policy. Among them, the most promising were the IPC, the percentage of ROB stalls, and the MISSES in the LLC. With respect to IPC and MISSES, the activation policy takes into account the difference (in percentage) between the value in the current

interval and in the last interval the prefetcher was enabled. Experimental results proved that IPC and the percentage of time ROB is stalled are inversely correlated, since when the ROB is blocked the core cannot keep decoding instructions, and consequently, the IPC drops. In addition, the ROB is mainly blocked because of long latency memory instructions, therefore a ROB based metric can be used to provide insights about when the prefetcher can improve performance. On the other hand, a primary goal of prefetching is saving cache misses. Therefore, a sudden rise in misses since the last time the prefetcher was enabled can be used as a hint to reactivate the prefetcher. The final design activates the prefetcher when the ROB stalls or MISSES surpass a threshold.

Of course, using only one of them would be more restrictive and would provide additional bandwidth savings, but at the cost of performance. Experimental results show that the use of any of them alone yields to significant performance losses in some applications of the studied workloads. This can be appreciated in Figure 4.6. Each graph shows the IPC of an individual benchmark in multi-core execution (normalized over the IPC obtained by an aggressive prefetcher) for the proposed ADP approach using both activation conditions jointly (ROB + MISSES) and individually. As observed, at the end of the execution, using the ROB condition alone drops the performance by 6% in `mcf` (Figure 4.6a) and using only the MISSES condition penalizes performance around 7% in `cactusADM` (Figure 4.6b).

Algorithm 4.5 summarizes the devised activation mechanism. If any of the metrics (ROB stalls and cache MISSES) suffers a sudden rise, that is, if any of the two conditions is meet, then the local prefetcher is activated, provided that the co-runners do not need more bandwidth. The reason for this restriction is that reactivating the prefetcher when bandwidth is scarce could rise the congestion and damage the global performance.

4.4 Experimental Setup

As explained in Chapter 3 this proposal has been evaluated with the Multi2sim [91] simulation framework. The Table 3.1 shows the configuration parameters of the core, the interconnection network, and the main memory. In addition to the hardware described in the table, the system has, per core, a stream prefetcher that detects constant stride patterns in the accesses to the cache, described in Section 3.1.3.

The experiments have been performed with 4-application multiprogram workloads composed of applications from the SPEC CPU 2006 benchmark suite.

Mix type	Benchmarks (categories)			
Combined	tonto (4)	h264ref (3)	hmmer (2)	omnetpp (1)
	bwaves (2)	gamess (3)	GemsFDTD (3)	sjeng (3)
	astar (1)	bzip2 (1)	gcc (2)	GemsFDTD (3)
	gamess (3)	GemsFDTD (3)	leslie3d (2)	wrf (2)
Memory intensive	xalancbmk (1)	gcc (2)	gobmk (2)	dealII (1)
	leslie3d (2)	dealII (1)	soplex (2)	gromacs (2)
	mcf (2)	soplex (2)	perlbench (2)	xalancbmk (1)
	dealII (1)	soplex (2)	xalancbmk (1)	gobmk (2)

Table 4.2: Mix composition. Numbers 1, 2, 3 and 4 between brackets correspond to categories MIPU, MIPF, nMIPU, and nMIPF, respectively, defined in Section 4.2.

Each application runs until it commits 300M instructions after executing 500M instructions to warm up the system. To avoid performance differences caused by early finalization of the execution of some benchmarks, all the applications are kept running until the slowest benchmark commits the targeted number of instructions. This implies that some benchmarks will execute more instructions than the targeted number. Consequently, metrics are only gathered for the first 300M committed instructions.

4.4.1 Mix Design

The characterization study presented in Section 4.2, classified application phases in four categories. The same rationale, however, can be used to classify applications. Following this approach, the applications in the SPEC CPU 2006 benchmark suite have been classified, and a set of workloads mixes has been designed to study the effects of prefetching on performance and energy in two main scenarios: under normal conditions and in more demanding conditions that stress the main memory.

To evaluate the first scenario, four mixes were designed with benchmarks randomly chosen from all the identified categories. To evaluate the second scenario, the designed mixes only contain memory intensive applications from MIPU and MIPF categories. The first type of mixes are referred as *combined* and the second as *memory-intensive*. Table 4.2 details all the mixes. Mixes from m0 to m3 are combined and mixes from m4 to m7 are memory-intensive.

4.5 Evaluation

To evaluate the proposed prefetcher, ADP is compared to the same system without prefetching and two other prefetchers: HPAC [19] and an aggressive prefetcher. HPAC is an adaptive prefetcher that implements throttling up and down policies to control the aggressiveness. The aggressive prefetcher is always working at the maximum aggressiveness level (it brings 4 extra blocks of data each time a strided access pattern is detected). The adaptive prefetchers (HPAC and ADP) use 2-block and 4-block as middle and high aggressiveness levels, respectively. The lowest aggressiveness is set to 1-block for HPAC (HPAC does not deactivate any prefetchers), while it is 0 for ADP, since it may completely deactivates some prefetchers. The ADP threshold values used in the experiments for Algorithm 4.4 and Algorithm 4.5 are shown in Table 4.1. In the case of HPAC, the configuration parameters where derived from the original publication [19].

4.5.1 Performance and Unfairness Analysis

To ensure that the proposed approach does not interfere in the performance of the applications when they run without co-runners, and to measure the effect of prefetching without interference in the main memory, Figure 4.7 shows the IPC of the applications of the SPEC CPU 2006 suite when they run alone in the multicore. Labels *No pref* and *Pref* in the figure refer to no prefetching and the aggressive prefetcher, respectively, while HPAC and ADP are the adaptive prefetchers. As can be observed, aggressive prefetching (*Pref*) brings important performance benefits to most applications. Performance improves on average by 12% over no prefetching and up to 30% in `gromacs`. Notice that the adaptive prefetchers perform better than the aggressive prefetcher in some applications like `bwaves`, in which *ADP* achieves 7% more IPC than *Pref.* This means that aggressive prefetching may be suboptimal even when an application is executed alone, so it may become a problem when executing with co-runners. Therefore, adaptive prefetchers may be used in multicores to sustain the performance.

After analyzing the applications in isolation, the rest of this section focuses on multicore execution, evaluating the performance and unfairness of the proposal with the metrics defined in Section 1.1.6. Figure 4.8 shows the STP achieved by the studied approaches for each workload. The three last columns (*gm 0–3*, *gm 4–7* and *gm*) represent the geometrical mean for the combined workloads (0 to 3), the memory intensive workloads (4 to 7), and all of them, respectively.

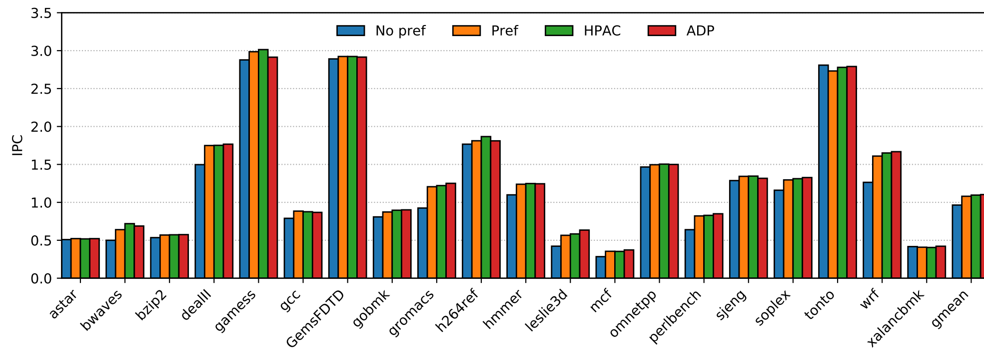


Figure 4.7: Performance of prefetchers running benchmarks in isolation.

As can be observed, the aggressive prefetcher improves the performance compared to no prefetching for combined mixes, but decreases the performance in most of the memory intensive mixes. On the other hand, the adaptive approaches, which stress less the memory hierarchy, perform significantly better than the aggressive prefetcher in both combined and memory intensive mixes. The performance drop that the aggressive prefetcher experiences in memory intensive workloads is mainly due to timeliness. That is, the prefetched data comes too late from main memory because of the longer latencies experienced in memory intensive mixes, so the data is not ready when needed [97]. This exacerbates when all the cores prefetch aggressively, since inter-core interference rises.

Compared to the other approaches, ADP achieves, on average, better performance regardless of the type of mix. ADP increases STP by 6% on average with respect to no prefetching considering both types of mixes while HPAC only improves it by 4%. The aggressive prefetcher, on the other hand, has more modest gains, improving STP by 1% on average. An important observation is that in memory intensive mixes, ADP is the only approach whose performance is on par with or higher than no prefetching. However, in workload 0 HPAC is the best performing approach, but only by a slim margin. The reason is that in this case ADP is too conservative, keeping the prefetcher disabled for too much time. Despite that, it performs better than the baseline prefetcher, and the difference with HPAC is smaller than 1%.

Since ADP reduces interference when accessing the main memory, this section also quantifies its effect on the system unfairness, with Figure 4.9 comparing the unfairness results of the evaluated approaches. As the figure shows, all the prefetching mechanisms increase the unfairness with respect to no prefetching.

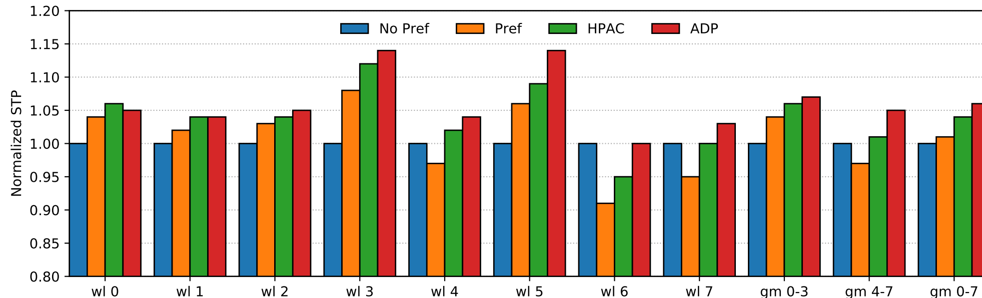


Figure 4.8: STP normalized with respect to no prefetching.

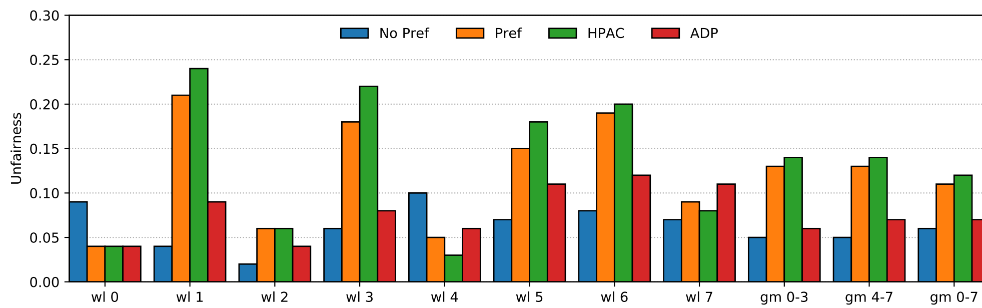


Figure 4.9: Unfairness of the different approaches.

While this is an interesting finding, it These results were expected, since main memory contention increases due to prefetch requests. However, ADP clearly has less unfairness than the other approaches, with only 0.07. In comparison, HPAC has 0.12 and Pref 0.11. Notice that, in spite of HPAC having less main memory accesses than Pref, it is more unfair. The reason is that it grants more bandwidth to the applications that are already performing well, increasing the differences between applications. As a consequence, unfairness rises. On the other hand, ADP performance gains are better distributed between co-runners, so it presents less unfairness.

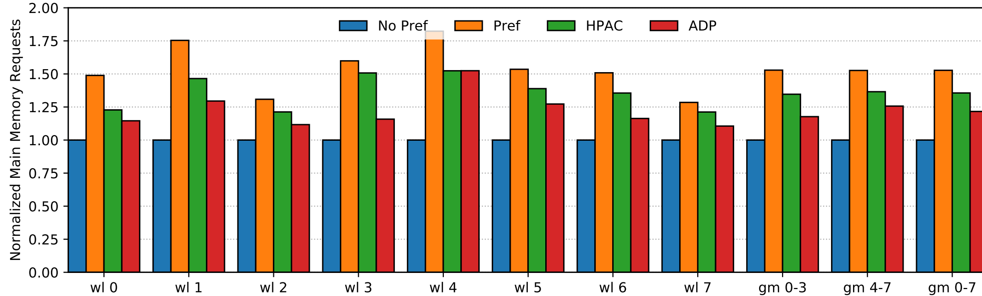


Figure 4.10: Main memory requests normalized with respect to no prefetching.

4.5.2 Prefetch Activity Reduction Analysis

This section shows how ADP saves memory traffic by reducing the amount of prefetches with respect to HPAC and aggressive prefetching. Figure 4.10 shows the number of main memory requests of the studied prefetchers normalized to no prefetching. The aggressive prefetcher increases the amount of memory requests compared to no prefetching by 53%. In contrast, HPAC and ADP reduce this amount to 36% and 21%, respectively. Therefore, considering these results jointly with the ones presented in the previous section, one can conclude that ADP improves the performance by significantly reducing the amount of useless prefetches, saving energy and bandwidth.

4.5.3 Main Memory Energy Analysis

This section compares the main memory energy consumption of the studied schemes. Figure 4.11 presents the energy results of the DDR3 DRAM memory subsystem. Unlike IPC and memory requests, which are gathered when a benchmark commits 300M instructions (see Section 4.4), energy consumption at the main memory is gathered at the end of the execution of the mix for simplification purposes. Therefore, energy results also consider those memory accesses issued after a benchmark executes 300M instructions (where IPC and memory requests metrics measured) until the slowest benchmark of the mix finishes its execution. This means that the presented energy results are conservative and it is the reason why the differences in Figure 4.11 are not so wide as in Figure 4.10.

Energy results are broken down in four components depending on the memory activity that consumes the energy: i) activation and precharge, ii) background energy, iii) data bursts, and iv) refresh. The first component accounts for

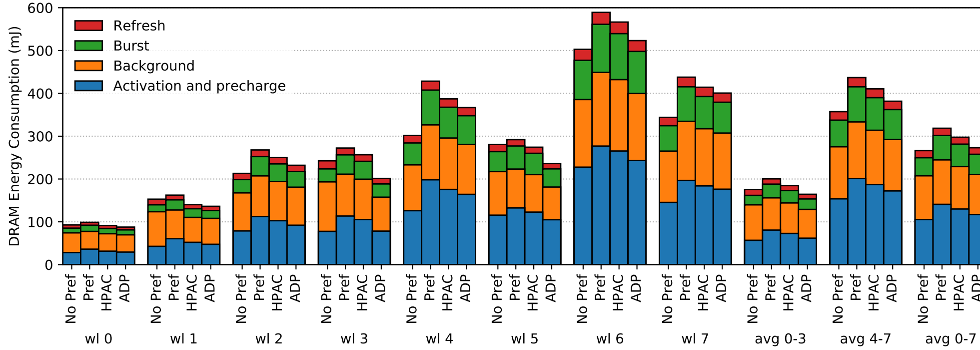


Figure 4.11: Energy consumption of the prefetching mechanisms.

the energy consumed activating rows for reads and writes, plus the energy consumed due to precharging the bitlines. The second component refers to the energy consumed in background to keep memory devices powered on. Burst energy is consumed when data are transferred on the memory bus write and read operations. Finally, refresh energy is required to avoid capacitors loose the stored value.

The studied prefetchers differ in the number of memory accesses they perform, so this section focuses on energy consumed by DRAM memory modules. Nevertheless, the important reduction in the number of prefetches is also expected to save dynamic energy in the LLC and in the NoC, since there are less accesses and traffic.

As expected, the prefetching schemes consume more activation and precharge energy as well as burst energy than no prefetching. The reason is that more main memory requests are served, as can be observed in Figure 4.10. On the other hand, prefetching helps to reduce both background and refresh energy, especially in combined workloads, because of the reduction in the execution time.

The aggressive prefetcher increases the total energy consumed by the DRAM by 20% over no prefetching, on average. This increase in energy consumption may be unacceptable, especially taking into account that aggressive prefetching can damage the performance in memory intensive workloads. In summary, ADP achieves the performance gains presented in Section 4.5.1 with a minimal impact on the consumed main memory energy, a 3% increase. In comparison, HPAC has a much large impact, with a 12% increase in energy consumption.

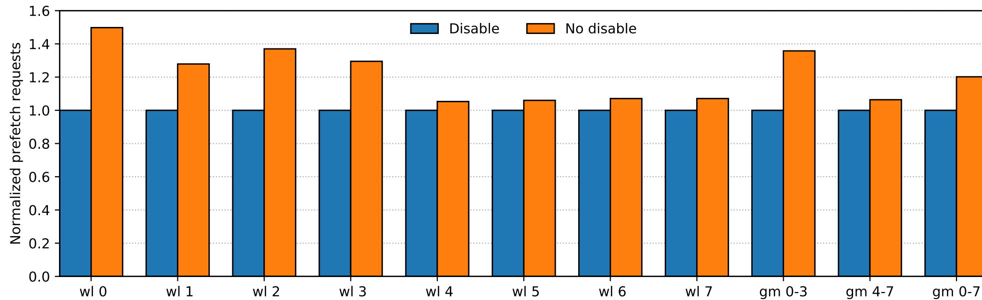


Figure 4.12: Effect of changing the minimum aggressiveness of ADP from 0 (completely disabled) to 1 in the number of prefetch requests.

4.5.4 Benefits of Deactivating the Prefetcher

To quantify which part of the benefits come from completely deactivating the prefetcher, we increased the minimum aggressiveness of ADP from 0 (disabled) to 1, keeping unchanged the remaining state machine (see Figure 4.3). In other words, the only difference is that this approach transits to a minimum aggressiveness level instead of turning off the prefetcher.

Figure 4.12 shows how the number of prefetches increases (in percentage) when the minimum aggressiveness is set to 1 instead of completely turning off the prefetcher. As observed, keeping activated the prefetcher even with minimum aggressiveness increases the number of prefetches on average by 36% in combined mixes and by 7% in memory intensive mixes. Moreover, this reduction is achieved with minor performance differences (less than 1% on average, not shown in the figures). If the total number of memory requests (prefetches and on demand accesses) are taken into account, the overall amount of requests increases by 6% and by 2% for combined mixes and memory intensive mixes, respectively, when the prefetcher is not deactivated. These results show that i) deactivating the prefetcher plays a key role in the reduction of memory accesses, and ii) the devised activation/deactivation policies work properly, as performance is not damaged.

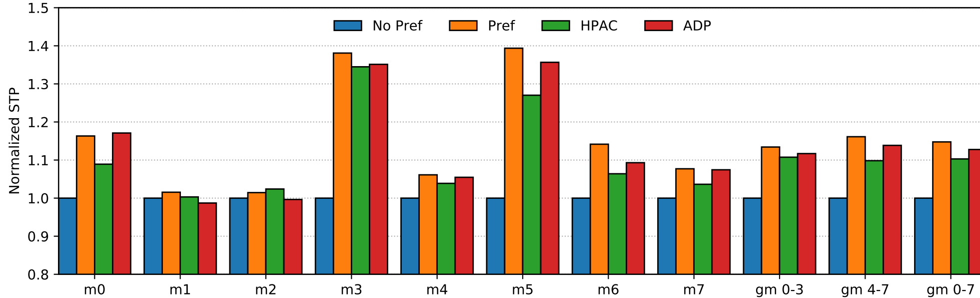


Figure 4.13: Normalized STP for the studied mixes with an Intel-like PC-based stride prefetcher.

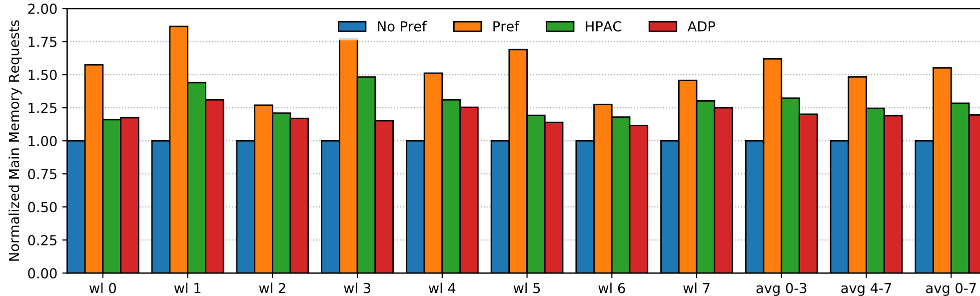


Figure 4.14: Relative increase in memory accesses, compared with no prefetching, using an Intel-like PC-based stride prefetcher.

4.5.5 Analysis with a Different Underlying Prefetcher

The results presented in the previous sections have all been obtained using a CZone-based stream prefetcher that detects strides in the load requests. Nevertheless, both ADP and HPAC, and in general other techniques that smartly adapt the prefetching aggressiveness, are orthogonal to the underlying prefetcher. To back this claim, this section presents some results obtained with a different prefetcher, inspired in one used by current Intel processors [1], that detects strided patterns in load streams, classifying them using the Program Counter (PC).

Figure 4.13 and Figure 4.14 show the results. Although we do not see the performance increases 4.13 shown in previous sections, ADP's performance is only slightly lower than the aggressive prefetcher, with much less main memory accesses, and it is better than HPAC. The reason for the lack of performance gains is that the thresholds used to transition to the different states of the

activation/deactivation mechanism were not tuned for this prefetcher, so they need refinement, which is out of the scope of this work. The reduction in main memory accesses, however, is similar to what was shown in Section 4.5.2.

4.6 Summary

This chapter has characterized the behavior of hardware prefetching for multi-program workloads running on multicore processors in limited memory bandwidth scenarios. The characterization study has shown that applications usually exhibit different execution phases from the prefetching perspective. In some of these phases, the core prefetcher has a minimal impact on the application performance, so it could be disabled making more bandwidth available to the other co-running applications. This approach often improves their co-runners' performance and saves energy, especially in main memory modules.

In this regard, this chapter has proposed the ADP selective prefetcher, that dynamically deactivates, activates and throttles individual core prefetchers. A core prefetcher is deactivated when the co-runners need more bandwidth, provided that the local prefetcher presents low accuracy and coverage. ADP smartly reactivates the prefetcher based on activation conditions that estimate if prefetches will improve the system performance, but only if the co-runners do not need more memory bandwidth. ADP increases the system performance without increasing the system unfairness, and since it reduces the amount of requests sent to main memory, it significantly reduces the energy consumption.

Chapter 5

Improving Fairness with LLC Partitioning

This chapter proposes Fair-Progress Cache Partitioning (FPCP), a low-overhead hardware-based cache partitioning approach that addresses system fairness. FPCP reduces the inter-application interference in the shared LLC by allocating to each application a cache partition and adjusting the partition sizes at runtime. To adjust partitions, our approach estimates during multicore execution the time each application would have taken in isolation, which is challenging.

5.1 Introduction

Shared caches can be found in the vast majority of modern multicore and many-core processors. The main reason is that *cache sharing* improves throughput for a given silicon area. As a consequence, recent microprocessors incorporate shared caches in almost all, if not all, levels of the cache hierarchy. In this regard, all cache levels (e.g. L1, L2 and L3) are shared in simultaneous multithreading (SMT) processors, e.g. the multicore IBM Power8 processor [82] and the many-core Knights Landing Intel Xeon Phi [84]. The benefits of cache sharing are also exploited in the embedded market, e.g., the L2 cache in the

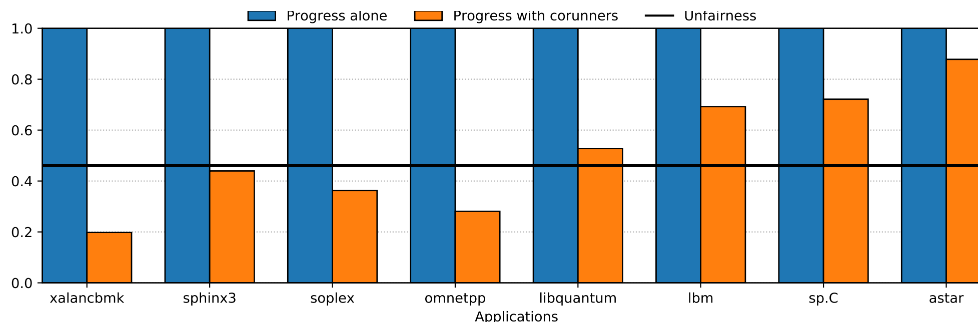


Figure 5.1: Progress and unfairness for 8 applications running concurrently during 120 seconds.

ARM Cortex-A53 processor [33]. Cache sharing, however, introduces interference when the co-running threads dynamically contend for cache resources. Consequently, the performance of individual applications can be worse than when executed alone, depending on how severe the contention is. This causes an *unfairness* problem, which is a major concern in current multicores.

To illustrate the unfairness problem, Figure 5.1 shows the *progress* of eight applications running concurrently. Each bar of the figure is labeled with the application it belongs to, and the horizontal line represents the system unfairness when executing this workload. It can be appreciated that `astar` is the most progressing application, with a progress rate of 0.88 (i.e. 1.14 slowdown) and `xalancbmk` is the least progressing application, with a progress rate of 0.20 (5.05 slowdown). The unfairness is, therefore, 0.46, which means that progress differences in this workload are quite significant and that they may be an issue.

To address unfairness in modern multicore processors, this chapter proposes *Fair-Progress Cache Partitioning* (FPCP), a low-overhead hardware-based cache partitioning approach that reduces cache interference by allocating a cache partition to each application and adjusting its size dynamically at run-time, since the cache requirements of each application vary during its execution. Thus, FPCP acts periodically, modifying the number of ways allocated to each partition, giving more cache space to those applications suffering more slowdown. Section 5.4 explains the proposal in detail.

A key characteristic of FPCP is that the number of cache ways assigned to a given application can only vary in one-unit steps between two consecutive intervals. This particularity makes the hardware simpler, the mechanism more

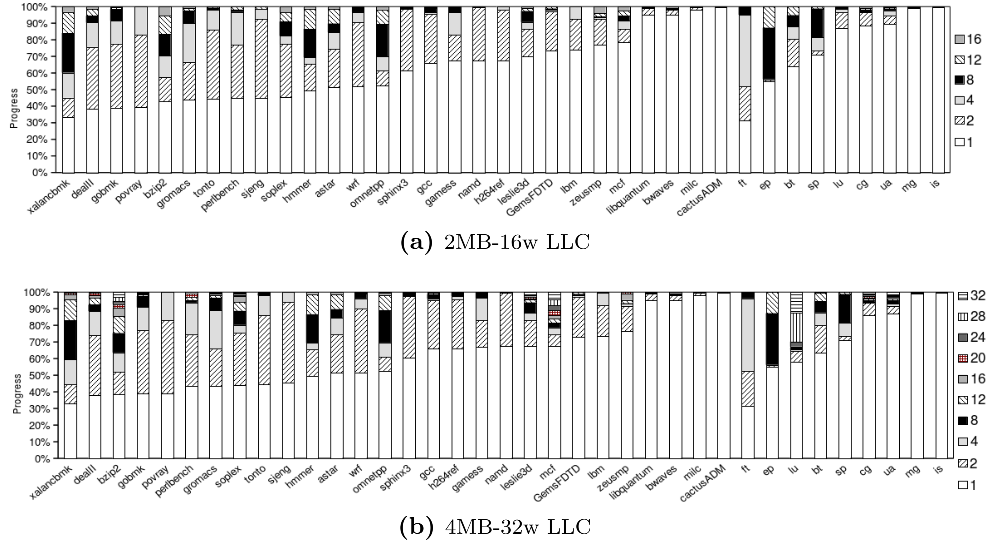
resilient to deviations in the estimations and, as experimental results will show, allows the system to achieve the best cache distribution. However, the slowdown an application suffers is unknown at runtime, so a key challenge to deal with fairness is the estimation of the execution time each application would experience in isolation. To deal with this issue one needs a performance model that, taking the performance of each application in concurrent execution with other applications and the inter-application interference as inputs, provides accurate performance estimations of isolated execution as output.

Two approaches have been recently proposed to estimate isolated execution performance. The first one, *Per-Thread Cycle Accounting* (PTCA) [15], identifies at run-time the cache misses that would have not occurred in isolation (i.e. inter-application misses). Then, the amount of cycles the reorder buffer (ROB) is blocked due to these misses is subtracted from the total execution time to obtain an estimation of the execution time the application would experience if executed without co-runners. The other, *Application Slowdown Model* (ASM) [87], is conceptually similar. However, ASM uses the Cache Access Ratio (CAR), cache accesses per time unit, as a proxy for performance. In this work, both approaches have been implemented in order to check and evaluate our proposal. Experimental results show that the ASM approach is slightly more accurate than PTCA. Section 5.3 explains the differences between these two approaches and compares them.

Finally, Section 5.6 compares FPCP with two state-of-the-art cache partitioning mechanisms, Utility Cache Partitioning (UCP) [70], by Qureshi and Patt, and ASM-Cache, a proposal by Subramanian *et al.* [87] that works on top of the ASM performance model. While both help to reduce unfairness, they present several drawbacks: i) since finding the optimal partitioning is a NP-hard problem [71], both UCP and ASM-Cache employ a $\mathcal{O}(n^2)$ greedy algorithm, compromising scalability; ii) they strongly depend on the accuracy of the estimation of the execution time in isolation, so errors in this estimation can have a big impact on application progress and system unfairness, and iii) both rely on the cache replacement policy obeying the stack property [59] (as with LRU) and in being able to track the number of hits in each frame of the replacement policy stack. In contrast, FPCP avoids these limitations by featuring an incremental cache partitioning algorithm, which is both less complex and more resilient to estimation errors.

This chapter makes the following contributions.

- We characterize the applications from SPEC CPU2006 and NAS benchmark suites according to their relationship between progress and shared



(a) 2MB-16w LLC

(b) 4MB-32w LLC

Figure 5.2: Effect of the available number of ways on progress, sorted by 1-way progress, for applications of the SPEC CPU 2006 and NAS benchmark suites.

cache interference, analyzing how unfairness may be affected depending on both the co-running applications and the available cache resources.

- We implement and compare two different models to estimate isolated execution performance, PTCA and ASM, concluding that ASM is slightly more accurate.
- We propose FPCP, a simple, cost-effective and scalable cache partitioning mechanism that improves system fairness regardless of the number of contending applications.
- We show that FPCP achieves better fairness than two state-of-the-art cache partitioning mechanisms, UCP and ASM-Cache, across a wide range of workloads and system configurations.

5.2 Analysis of the Inter-Application Cache Interference

The interference an application suffers when contending for the shared resources with other applications has an unpredictable impact on its progress (Equation 1.1), and thus in the unfairness of the system, measured as in Equation 1.3.

To help to understand the causes of unfairness, this section analyzes how the progress of each individual application is affected when the amount of assigned cache ways varies from 1 way to the total available, simulating other applications competing for the same cache resources[1].

Since results depend on the cache geometry, to focus the analysis we first consider a 2MB-16w LLC cache. Figure 5.2a shows the progress rate (from 0% to 100%) achieved by the different applications (sorted by ascending progress with 1 cache way) when varying the number of assigned cache ways. For instance, `povray` achieves a progress rate of around 39% with just 1 cache way and requires 2 and 3 ways to increase its progress up to 82% and 100%, respectively. These results have been obtained using the simulation environment described in Section 5.6.

According to the number of ways required to achieve a significant progress (e.g. 80%), three main categories of applications can be distinguished: *cache-insensitive*, *highly cache-sensitive* and *moderately cache-sensitive*. The former group contains those applications whose progress is barely affected by the number of cache ways assigned to the application, that is, a single cache way is enough to achieve significant progress. The second and third categories group those applications whose progress is sensitive to the number of ways. However, while applications in the second group require a high number of cache ways (e.g. half the number of the total cache ways), applications in the last group require a relatively low (e.g. from 2 to 4) number of ways to achieve significant progress.

Next, we illustrate how this information can be used to provide an overview analysis about system unfairness through two examples: one with low unfairness and another one with a high level of unfairness.

Low-unfairness example. Assume that `cactusADM` and `milc` run together. In this case, unfairness would not be a concern since both applications are

[1]The progress of a task when running in the cache with a reduced number of ways, say w, is computed as the execution time of the task when it has the entire cache for itself (i.e. 16 ways and no interference) divided by the execution time taken in the constrained cache.

classified as *cache-insensitive progress*, and the progresses of both applications would be higher than 97% regardless of the cache way distribution.

High-unfairness example. Assume now that the applications running together are `cactusADM` and `xalancbmk`. The former is an application with *cache-insensitive progress* while the latter is classified as *highly cache-sensitive progress*. Thus, when executed together, `cactusADM` only requires one way to achieve a notable progress, while `xalancbmk` requires significantly more cache resources to have an adequate progress rate. However, due to its poor cache locality under the typical LRU replacement algorithm, which is the reason its progress is so cache insensitive, `cactusADM` uses around 10 out of 16 cache ways, leaving to `xalancbmk` only the remaining 6. Consequently, the progress of `xalancbmk` drops below 80%, which yields the system to an unfairness level of around 18%.

An analogous characterization study has been performed for a 4MB-32w cache (see Figure 5.2b). It can be noticed that, although using the same sorting approach, the relative order of the workloads in Figure 5.2a and in Figure 5.2b is slightly different. This can occur when the working set fits in the larger cache or when the larger cache improves the hit ratio enough to significantly reduce cache trashing.

5.3 Analysis of Progress Estimation Approaches

FPCP employs auxiliary circuitry to estimate the execution time each application would have experienced if executed without co-runners, since this information is required to estimate the progress of the application, see Equation 1.1. Then, the progress results are used to select cache partition sizes. Therefore, if estimations are not accurate enough, it is likely that the cache partitioning will perform poorly.

There are are two recent approaches to estimate performance without co-runners: Per-Thread Cycle Accounting (PTCA) by Du Bois *et al.* [15] and Application Slowdown Model (ASM) by Subramanian *et al.* [87].

To implement and evaluate FPCP, we first implemented and compared the PTCA and ASM models to obtain progress estimations, based on the guidelines discussed in the original works. To make this chapter self contained, some key guidelines are discussed below. Please, refer to the original work for further details.

Both approaches make use of an Auxiliary Tag Directory (ATD) per core, which is a structure that keeps track of what the status of the shared cache would have been if it were private to the core [70]. Note that if SMT is used and one wants to distribute cache resources per-application instead of per-core, then an ATD per thread is required and the mentioned models need to take into account the interference between threads at the shared cache levels above the LLC. The key challenge lies on obtaining accurate estimates of performance in isolation by using information gathered during execution with co-runners. This can be done by subtracting the cycles an application makes no progress due to interference caused by co-runners from the concurrent execution time (see Equation 5.1, where $C_{t,alone}$ is the execution cycles when executed alone, $C_{t,multicore}$ the execution cycles when executed with co-runners, and $I_{t,multicore}$ represents the stall cycles due to interference).

$$C_{t,alone} = C_{t,multicore} - I_{t,multicore} \tag{5.1}$$

Using the ATD, PTCA identifies at run-time the LLC cache misses that would have not occurred in isolation (i.e. inter-application misses). Then, the amount of cycles the Reorder Buffer (ROB) is blocked due to these misses is accounted as interference cycles.

The approach followed by ASM is conceptually similar. However, ASM uses the Cache Access Rate to the LLC (CAR, accesses per cycle) as a proxy for performance. $CAR_{t,multicore}$ is obtained during execution with co-runners and it is defined as in Equation 5.2. On the other hand, $CAR_{t,alone}$ is estimated dividing the number of cache accesses to the LLC by the cycles elapsed minus the cycles lost due interference (see Equation 5.3). Notice that the fraction's denominator of the latter equation matches $C_{t,alone}$ when applying Equation 5.1.

$$CAR_{t,multicore} = \frac{\#LLC_Accesses}{C_{t,multicore}} \tag{5.2}$$

$$CAR_{t,alone} = \frac{\#LLC_Accesses}{C_{t,multicore} - I_{t,multicore}} = \frac{\#LLC_Accesses}{C_{t,alone}} \tag{5.3}$$

Considering these metrics, progress is approximated in ASM as shown in Equation 5.4. Note that the original ASM paper estimates slowdown instead of

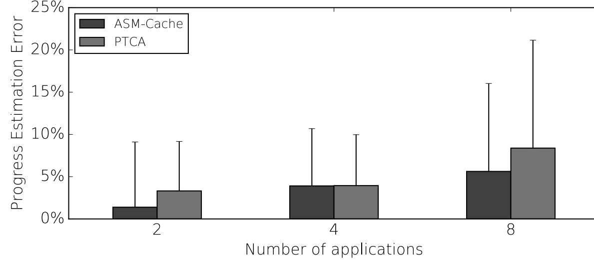


Figure 5.3: Progress estimation error with ASM and PTCA.

progress but, since one is the inverse of the other, both can be used with the same aim.

$$Progress_t \approx \frac{CAR_{t,multicore}}{CAR_{t,alone}} = \frac{\frac{\#LLC_Accesses}{C_{t,multicore}}}{\frac{\#LLC_Accesses}{C_{t,alone}}} = \frac{C_{t,alone}}{C_{t,multicore}} \tag{5.4}$$

ASM uses a different method than PTCA to compute the interference cycles. Instead of tracking the time the ROB is stalled due to interference, they multiply the number of inter-application misses (obtained with the ATD) by the average cache miss service time. Note that both ASM and PTCA provide mechanisms to separate and identify interference coming from different parts of the system so, while in this work we are only targeting LLC-originated unfairness, additional sharing policies, orthogonal to our proposal, could be implemented in other shared parts of the system, like the memory controller or the NoC [61, 51, 93] to further reduce unfairness.

We compared the accuracy of both ASM and PTCA and we found that, in our experimental setup, ASM was slightly more accurate than PTCA. Thus, results will be presented only with the ASM model. Figure 5.3 shows the average and the standard deviation of the estimation error across the studied workloads varying the number of applications running concurrently.

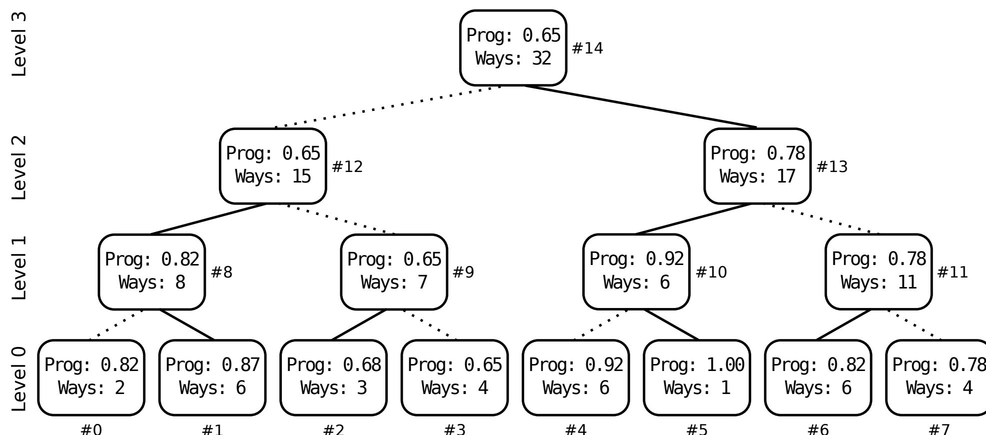


Figure 5.4: Logical tree structure used by FPCP.

5.4 FPCP Partitioning Approach

As mentioned above, a system is totally fair if all the co-running tasks progress at the same pace with respect to isolated execution. The proposed partitioning approach pursues to minimize system unfairness by narrowing progress differences among co-executing tasks. FPCP gathers interference data during multicore execution at regular intervals, and then estimates progress at the end of each interval according to Equation 5.4 and distributes cache ways.

We evaluated different interval lengths and found that the best results for FPCP were obtained with intervals of 5K misses. The reason we use misses to define the interval length instead of cycles is that this way responsiveness is increased during execution phases with lots of cache accesses.

Next, we analyze the scalability of FPCP with the number of cores, we discuss the reasons behind using binary trees as underlaying structure, and finally, we estimate the proposal overhead.

5.4.1 Algorithm and Hardware Implementation

FPCP implements a hardware tree-based algorithm, which requires little extra logic [3]. The set of n applications running in the processor is subdivided recursively until the subsets have only two applications, and a *binary tree* with

the n applications at the leafs is constructed. Figure 5.4 shows the resulting tree for 8 applications sharing a cache in an 8-core CMP.

Each leaf node contains the number of ways assigned to the corresponding application as well as the Modified Moving Average (MMA) [47] of its progress (see Equation 5.5).

$$MMA_i = MMA_{i-1} + f(Progress_{i-1} - MMA_{i-1})|_{f=0.1} \tag{5.5}$$

The reason why we use the MMA is that it approximates a conventional moving average, but only requires the previous average and the progress from the current interval to compute the average for the next. In addition, it retains enough previous information to make solid partitioning decisions while allowing a fast reaction time to changes in the workload behavior.

Each non-leaf node, on the other hand, stores i) the total number of ways assigned to its children and ii) the minimum progress (MMA) between its children. Note that, to ease the understanding of the example, in Figure 5.4 we use dashed lines to indicate which child node has the minimum progress.

Each Level i in the tree except Level 0 has an associated number of intervals I_i, where i indicates the level depth. Every I_i intervals the algorithm listed in Figure 5.5 is applied to the nodes in Level i ($i \geq 1$). For each node in that level, the minimum progress between its children is determined and stored in the node by checking *only* the lower level. Then, there are three possible cases: ① The node has the same number of ways as its children combined. When this occurs, the most progressing child relinquishes a way (provided it has more than one) in favor of the least progressing child. ② The node has more ways assigned than its children. In this case no child has its ways reduced, but the least progressing node gains a way. ③ The node has less ways assigned than its children. If this happens, then no child receives a way, and a way is subtracted from the most progressing child, if possible, or its brother, if not.

In Figure 5.4, node `#8` is an example of case ①. When the algorithm is applied to this node, a way is transferred from node `#1` to `#0`. On the other hand, node `#11` is an example of case ②. It has 11 assigned ways, but one of these ways is not assigned either to node `#6` or node `#7`. This can happen if node `#10` gave a way to node `#11` in a previous application of the algorithm in the upper level (node `#13`). So, when the algorithm is applied to `#11`, `#7` gains a way, and `#6` is not affected. Finally, node `#10` is an example of case ③, since

```
foreach n ∈ N_level do
    if n.left.progress < n.right.progress then
        least ← n.left
        most ← n.right
        n.prog ← n.left.progress
    else
        least ← n.right
        most ← n.left
        n.prog ← n.right.progress
    if n.ways == most.ways + least.ways then
        if most.ways > 1 then
            most gives 1 way to least
    else if n.ways > most.ways + least.ways then
        least gains 1 way
    else
        if most.ways > 1 then
            most loses 1 way
        else
            least loses 1 way
```

Figure 5.5: FPCP algorithm.

it has less ways assigned than its combined children. Again, this situation can occur due to a previous application of FPCP in an upper level.

From now on, we assume $I_1 = 1$, $I_2 = 4$, and $I_3 = 8$, which are the values we used to obtain the experimental results in Section 5.6. Therefore, way transfer at Level 1 will occur every 1×5K misses; at Level 2 every 4×5K misses, and at Level 3 every 8×5K misses. Additionally, progress is computed every 1×5K misses. While we transfer ways one at a time, this value can be increased, but there is a trade-off between responsiveness and a higher penalty due estimation errors. In addition, a threshold could be used to only transfer ways if the progress difference is considered significant.

Item	General cost	Cost for a 4-core CMP	Cost for a 8-core CMP
Core ID per tag	Cache blocks × $\log_2$ cores	131072 bits	393216 bits
Alternate Tag Directory	Sampled sets × associativity × (tag + replacement bits) × cores	237568 bits	475136 bits
Per-core interference cycle counter	32 bits × cores	128 bits	256 bits
Counters for the total number of accesses, sampled accesses and inter-application misses	3 × 20 bits	60 bits	60 bits
Per-core counters for hit and miss service times	16 bits × 2 × cores	128 bits	256 bits
FPCP tree cost	(2 × cores - 1) × (32 bits + $\log_2$ associativity)	259 bits	555 bits
Total		369215 bits	869479 bits
Percentage area overhead w.r.t. shared cache		1.10%	1.30%

Table 5.1: Detailed FPCP hardware overhead.

5.4.2 Rationale for Using Binary Trees as the Underlying Structure

FPCP's goal is to ensure that applications progressing the most relinquish cache resources, and that the freed cache resources are assigned to those applications progressing the least. A simple approach could be to only transfer resources from the most progressing application to the least progressing application, since finding the minimum and maximum elements in a set has a $\mathcal{O}(n)$ cost, being n the size of the set. However, the benefits of this approach do not scale as the number of applications increases, since we are considering only two applications and ignoring the rest. The distribution of cache resources would be, therefore, not responsive enough. Other approach could be to sort the applications by progress and adjust all the partitions according to this information. Although the idea seems appealing, the sorting cost is $\mathcal{O}(n\Delta log(n))$, which could make prohibitive the hardware implementation cost. Using a binary tree and making updates by levels maintains the benefits of potentially exchanging ways between all the applications, while keeping complexity $\mathcal{O}(n)$.

The reason for using different delays to update the levels of the tree is to leave some time for the changes to settle in the lower levels, which lead to better overall results. Arguably, other data structures could be used instead of binary trees, but binary trees have been used because both their simplicity and straightforward scalability to greater numbers of applications.

5.4.3 Overhead Analysis

This section analyzes the FPCP overhead in terms of hardware and timing complexity for 4- and 8-core systems.

The proposed approach assumes a thread-aware LRU replacement algorithm [42, 70, 88]. Therefore, each cache block is tagged with the associated thread it belongs to (i.e. a 2-bit tag for a cache shared among four threads). As depicted in Table 5.1, this implies around 35% and 45% of the total overhead for 4 and 8 cores, respectively, which accounts for 1.10% and 1.30% of the area of the entire shared cache (4/8 MB). Notice, however, that some processors already implement this capability; for instance, recent Intel Xeon processors [37, 32] feature cache utilization monitoring and cache partitioning, referred to as Cache Monitoring Technologies (CMT) and Cache Allocation Technologies (CAT), respectively, that can be used for this purpose. Although CAT could be, in principle, used to this end, notice that CAT is designed to be controlled by software, while our approach works at hardware level, with a granularity orders of magnitude smaller (i.e. ns vs ms). Anyway, if the processor implements these features, the total hardware overhead would drop to around 0.71% for both configurations, assuming that we still use the ATD to estimate progress.

The ATD is the other key hardware structure incurring overhead. As mentioned above, this component is used, one per core, to track inter-application interference. However, to reduce hardware costs we only monitor a subset of 64 cache sets, and the results are extrapolated with minimal impact on accuracy [15]. ATDs account for 64% and 54% of the total overhead, for 4 cores and 8 cores, respectively. The remaining components, included the tree structure required by FPCP, incur in a minimal hardware overhead (less than 1%) and the other values used by the proposal (e.g. CPI and number of cache misses) that do not appear in the table can be gathered from performance counters available in most multicores, so they do not incur in additional overhead.

Each $I_1 \times 5$K misses interval, the progress of all the applications is updated in the leaf nodes. Additionally, each I_i intervals, the algorithm listed in Figure 5.5

(which has a constant execution time) is applied to the Level i of the tree. Therefore, the cost of FPCP is of $\mathcal{O}(n)$ since each interval a number of nodes that depends linearly on the number of applications (n) must be traversed. Further timing analysis are discussed in the next section.

5.4.4 Main Differences with the ASM-Cache Approach

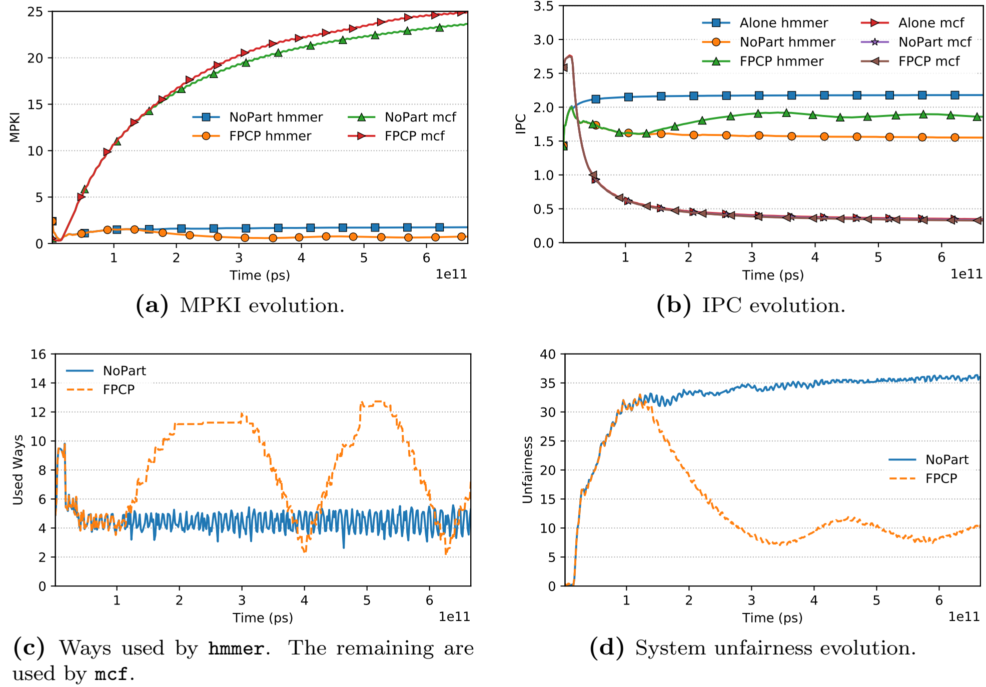
FPCP differs from ASM-Cache both in the criteria applied to distribute cache ways and in the hardware implementation complexity.

ASM-Cache distributes cache ways among applications in a greedy way, according to the estimates of the execution time in isolation. Thus, the number of assigned ways to a given application can highly vary between two successive intervals. For instance, an application could have assigned a few ways (e.g. two out of sixteen) in a given interval and almost all the cache ways (e.g. fourteen or fifteen) in the subsequent interval, or vice versa. As a consequence, inaccurate estimations can severely impact on the system fairness. Unlike this approach, what we propose is to distribute cache space in steps of a single cache way between consecutive intervals. Notice that our proposal is, in essence, based on relative estimates instead of absolute ones. This way makes our approach more resilient to possible inaccuracies in the estimation process.

Regarding complexity, ASM-Cache relies on the cache replacement policy obeying the stack property [59] (as with LRU) and in being able to track the number of hits in each frame of the replacement policy stack. Moreover, since finding the optimal partitioning is a NP-hard problem [71], ASM-Cache employs a $\mathcal{O}(m^2)$ greedy algorithm (where m is the cache associativity) to search for an adequate partitioning. All these reasons difficult the design of a viable hardware implementation. Instead, as discussed above, the cost of FPCP is only $\mathcal{O}(n)$ (with n being the core count).

We experimentally measured the time taken by both the FPCP and other approaches, i.e. ASM-Cache, on a 2.2GHz Xeon, as an approximation of the time taken by the hardware. Regarding FPCP, the algorithm takes between $100 - 800$ cycles, depending on the depth of the specific level of the tree being considered. The algorithm is triggered when a given number of cache misses is reached, which translates to around 500K cycles on average, but it is highly dependent on the workload. On the other hand, the time taken by ASM-Cache is about 120K cycles, and the algorithm is triggered each 5M cycles. This means that the overhead of FPCP falls in between 0.02% and 0.16%,

while the overhead of ASM-Cache is at least one order of magnitude higher, i.e. by 2.4%. This was expected, since ASM-Cache has quadratic complexity while FPCP's complexity is linear.



(a) MPKI evolution.

(b) IPC evolution.

(c) Ways used by `hmmer`. The remaining are used by `mcf`.

(d) System unfairness evolution.

Figure 5.6: Dynamic `hmmer-mcf` evolution.

5.5 Experimental Setup

We modeled all the studied approaches and performed a microarchitectural, cycle-by-cycle simulation by extending the Multi2Sim simulation framework. The proposal has been evaluated varying the number of applications (i.e. cores in our system) sharing the cache. We have studied a cache shared by two, four, and eight applications, which covers a representative range of shared caches in current multicores (e.g. ARM processors).

Each processor core has private 32KB 8-way L1 caches, while the shared cache has a capacity of 1MB per core in the system (e.g. 8MB for the 8-core multi-

core). The shared cache for the 2-core processor has 16 ways, while the others have 32 ways. Section 3.1.3 describes the main architectural parameters.

FPCP was evaluated and compared against a baseline shared cache using the LRU replacement policy without partitioning (referred to as NoPart) and against two state-of-the art approaches (i.e. UCP and ASM-Cache). In the UCP scheme, each core has a small utility monitor based on dynamic set sampling (UMON-DSS) with 64 sets. This monitor estimates how well each core makes use of cache capacity, and distributes cache resources to minimize the overall number of misses.

With respect to ASM-Cache, it employs a mechanism similar as the used by UCP, but with a different aim. Instead of trying to minimize misses, the mechanism estimates how the slowdown of a given application will be affected according to the number of ways allocated to it, so the optimal partitioning is the one that minimizes the per application slowdown.

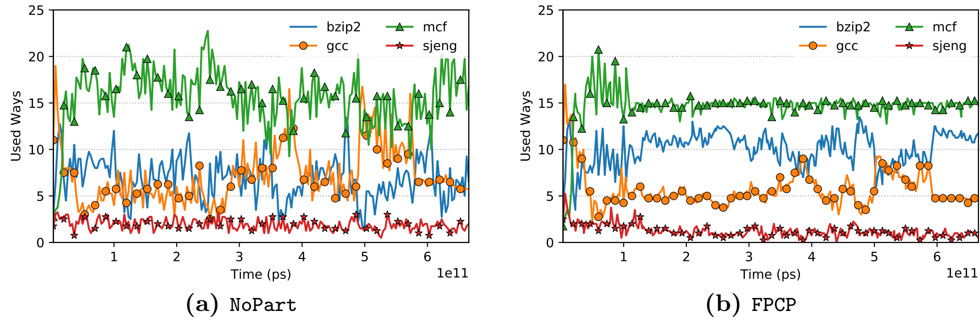
Experiments were run with multiprogram workloads from both the SPEC CPU 2006 and NAS benchmark suites. Three different sets of workloads were considered, varying the number of applications sharing the cache. We used 100 2-application, 175 4-application and 50 8-application workloads to consider a wide range of scenarios. All the workloads were randomly generated, and results were collected simulating each workload for 2-billion cycles after skipping the initial 500M instructions of each individual application.

5.6 Evaluation

This section analyzes the dynamic run-time interactions among applications, considering used cache ways, system performance, progress and unfairness. To help the understanding of these interactions and on how our proposal works, we start with a simple 2-application example.

Figure 5.6 presents the results for two benchmarks, `hmmer` and `mcf`, running concurrently. As we already observed in Figure 5.1, without cache partitioning this workload exhibits significant unfairness during its execution. This can be also appreciated in Figure 5.6d, which shows the unfairness evolution for both the baseline approach (NoPart) and for FPCP.

It can be noticed in Figure 5.6a that the MPKI (Misses Per Kilo-Instruction) at the shared cache of both applications is considerably different; while the MPKI of `mcf` is over 20 for most of its execution, the MPKI of `hmmer` is only



(a) NoPart (b) FPCP

Figure 5.7: Run-time way partitioning under NoPart and FPCP for the {**bzip2**, **gcc**, **mcf**, **sjeng**} workload.

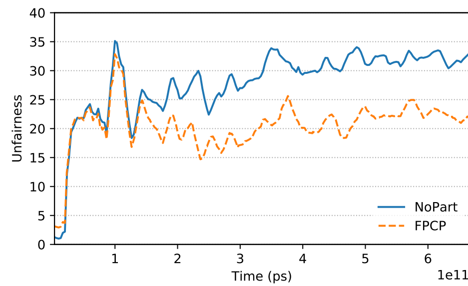


Figure 5.8: Unfairness for the {**bzip2**, **gcc**, **mcf**, **sjeng**} workload.

around 2. The reason for such a difference is that this fragment of the **hmmer** execution is CPU-bound, and therefore very sensitive to cache misses. This can be appreciated in Figure 5.6b, where the IPC of **hmmer** drops from above 2 to about 1.6 due to the interference of **mcf**. On the other hand, this phase of **mcf** is memory bound and experiences a high amount of cache misses, so this benchmark shows little sensitiveness to cache miss ratio variations and is a cache hog.

Without any intervention in the shared cache (i.e. without partitioning), this fact translates into noticeable differences in the progress rate experienced by both applications. As shown in Figure 5.6c, NoPart assigns between 3 and 5 ways to **hmmer** and the remaining cache ways (13 to 11) to **mcf**. Notice that, despite **mcf** holding around two thirds of the cache ways, its IPC is still below 0.5. Moreover, this IPC is similar to what the application achieves

in standalone execution. This means that mcf exhibits an excellent progress when co-running with hmmer (see Figure 5.1). Therefore, the only way to reduce unfairness is to accelerate the progress of hmmer (the least progressing application).

FPCP correctly identifies these progress differences and borrows ways from mcf, assigns them to hmmer, and improves its progress. As a result, unfairness is reduced from 36% to 10%.

At a first glance, it could seem counterintuitive, since FPCP takes ways from the application with the highest MPKI and assigns them to the one with the lowest MPKI, thus widening even more the huge MPKI differences. However, it makes sense when we realize how little the progress of mcf is affected by the number of assigned ways (see Figure 5.2a). In fact, mcf only needs 2 ways to achieve a progress rate of around 90%, while hmmer requires around 12 ways to achieve the same progress. This brings two important findings: i) *blindly* reducing cache misses does not necessarily address system fairness, and ii) accurate progress estimations are critical to address fairness.

This analysis can be extrapolated regardless of the number of applications. Lets see another simple working example for a 4-application workload. Figure 5.7 compares the distribution of ways per application between the non-partitioned baseline approach and the proposal. According to the analysis performed in Section 5.2, to achieve a progress rate of around 80%, bzip2 requires about 12 ways, gcc 2 ways, mcf 8 ways, and sjeng 2 ways. However, without cache partitioning, Figure 5.7a shows that gcc occupies at run-time similar or even more ways than bzip2, while mcf exceeds 16 ways for most of the execution time. FPCP, in contrast, distributes ways much more accordingly (compared to NoPart) to what the progress analysis suggested, giving more ways to bzip2 and fewer ways to the remaining co-runners, as can be seen in Figure 5.7b. As a result, FPCP significantly improves system fairness for this workload (see Figure 5.8), reducing the final unfairness around one third (from 33% with NoPart to 23%).

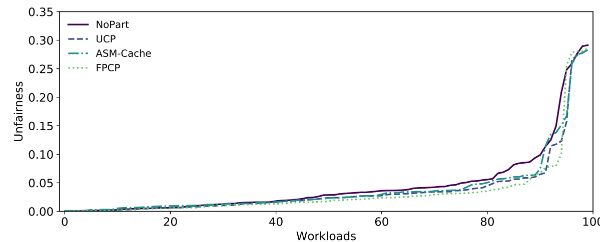


Figure 5.9: System unfairness results over 100 2-application workloads.

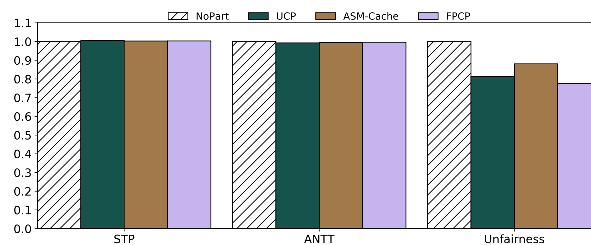


Figure 5.10: Average results normalized against NoPart for the 2-application workloads.

5.6.1 Comparison with the State-of-the-Art

FPCP has been evaluated against NoPart, UCP and ASM-Cache in terms of system unfairness and performance, using the metrics defined in sections 1.1.5 and 1.1.6. Figure 5.9 presents the unfairness results for 100 pairs of benchmarks sorted in increasing unfairness order (the highest, the worst). Since each application only suffers interference from one co-runner, average system unfairness for this case is relatively low, averaging 0.05% for the baseline (NoPart), 0.04% and 0.04% for UCP and ASM-Cache and 0.03% for FPCP. In spite of this fact, UCP, ASM-Cache and FPCP significantly improve unfairness over LRU for some of the workloads, as can be seen in the figure.

Regarding performance, Figure 5.10 shows average STP, ANTT and unfairness normalized over NoPart. As can be seen, the reduction in unfairness does not negatively impact the performance, as STP and ANTT are unaffected.

The interference grows with the number of applications running together. Figure 5.11 shows the results for 175 4-application workloads. Three major observations can be drawn. First, FPCP significantly improves unfairness over NoPart and both state-of-the art approaches. On average, NoPart presents

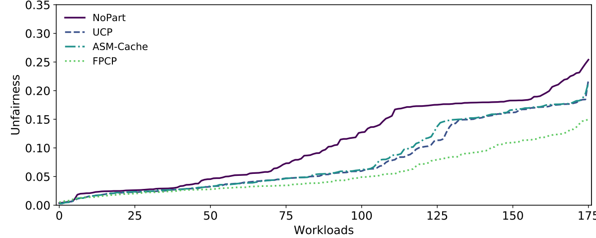


Figure 5.11: System unfairness results over 175 4-application workloads.

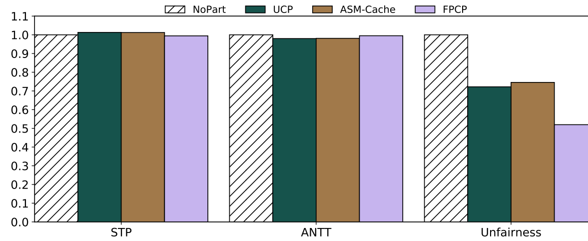


Figure 5.12: Average results normalized against NoPart for the 4-application workloads.

an unfairness level of 0.107; UCP and ASM-Cache of 0.078 and 0.080, respectively; and FPCP reduces it to only 0.056. Second, FPCP highly reduces the maximum unfairness compared to the second best approach (i.e. 0.15 vs 0.21 of ASM-Cache). As with 2-application workloads, the reduction in unfairness in 4-application workloads does not negatively impact STP and ANTT (see Figure 5.12).

To cover a wider range of scenarios, we also evaluate the proposal with 8-application workloads using a shared LLC with 8MB and 32 ways. Again, as shown in Figure 5.13, FPCP is the approach that presents the lowest unfairness, reducing it from the average 0.11 of NoPart to 0.08. UCP and ASM-Cache, while reducing unfairness compared to NoPart in most cases, do not achieve this goal in some specific workloads (as depicted in the right side of the figure). This is because these approaches strongly depend on the accuracy of the estimations, which degrades with the number of applications [87, 15] (see Section 5.3). Our approach, however, only exchanges one way at a time, so it is more forgiving with progress estimation inaccuracies. Figure 5.14 shows normalized performance and unfairness results for the 4 different approaches. It shows how UCP and ASM-Cache, on average, actually increase unfairness,

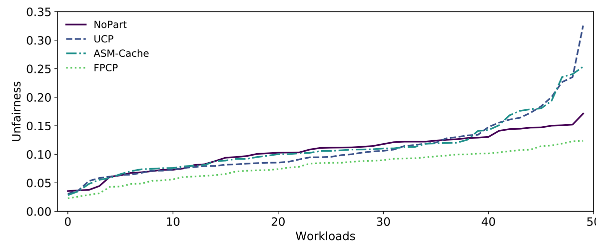


Figure 5.13: System unfairness results over 50 8-application workloads.

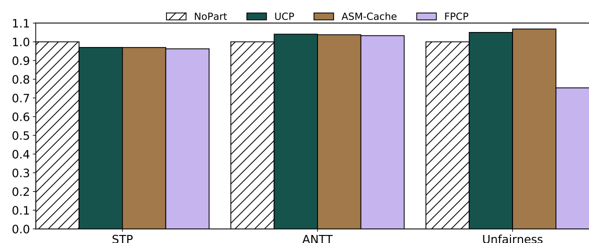


Figure 5.14: Average results normalized against NoPart for the 8-application workloads.

due to their behavior on the workloads on the rightmost part of Figure 5.13. Additionally, UCP, ASM-Cache and FPCPslightly damage STP and ANTT, but this effect is small, less than 3%.

There are multiple reasons why system unfairness rises with the number of cores, even assuming a perfect *progress estimation approach* (i.e. perfect interference estimation). On the one hand, while the per core cache space remains constant (1 MB), the number of ways does not, so the partitioning policy presents comparatively less flexibility. Therefore, under certain circumstances, it cannot provide the high number of cache ways some applications require to have an adequate progress rate. On the other hand, as seen in Figure 5.2, there are applications whose progress is not affected by the number of cache ways allocated to them (e.g. `CactusADM` or `is`). Thus, when these applications are combined with other applications that require a high number of ways (e.g. `mcf` or `lu`), the system will show an inherently high unfairness level. The reason is that, while the partitioning approaches can equalize the progresses of moderately or highly cache-sensitive applications, the differences between these progresses and the progresses of cache insensitive applications will be important.

5.6.2 Sensitivity to the Aggressiveness of Way Redistribution

In order to check the unfairness benefits coming from assigning cache ways conservatively either in one-way steps or considering longer intervals, we performed three additional experiments considering 4-application workloads. In the first experiment, we slightly modified FPCP to allow redistributing 2 and 3 ways at a time instead of a single one to check its impact on unfairness. In the other two experiments, we checked the impact of redistributing ways conservatively in other existing approaches. First, we modified the original ASM-Cache to redistribute only one way at a time, and finally, we compared different versions of ASM-Cache by increasing the interval length, thus slowing down way redistribution.

With respect to the first experiment, increasing the number of ways exchanged in each operation, we found that either trading 2 or 3 ways increases unfairness compared to trading only one cache way at a time. This increase is, on average, of around 5%. In the second experiment, we found that limiting to a single one the amount of ways traded each time ASM-Cache is triggered, helped to reduce unfairness. The improvement was, on average, by 6% compared to original ASM-Cache scheme. While this seems significant, note that FPCP has 35% less unfairness, on average, than ASM-Cache. Regarding the third experiment, we compared three versions of ASM-Cache, with 5M-cycle, 10M-cycle and 50M-cycle interval lengths, respectively. While the longest interval slightly reduced unfairness, differences are lower than 3%.

In short, two meaningful findings can be drawn from the aforementioned experiments. First, care must be taken when basing decisions on estimations that can be inaccurate or present transient errors, so slowly redistributing cache ways tends to perform better. And second, conservatively distributing cache ways is not a panacea, but the partitioning algorithm plays a key role, since there are still important unfairness differences between FPCP and the limited ASM-Cache version.

5.7 Summary

Cache sharing must be properly managed to avoid system unfairness in current multicore processors. With this aim, this chapter has presented three main contributions: i) an application characterization study from the LLC point of view, ii) a comparison of two recent approaches to estimate performance in isolation during multicore execution, and iii) a novel cache partitioning approach that improves fairness over the state-of-the-art schemes.

Regarding the characterization analysis, it studies the impact of the available LLC space on the performance of the applications when executed without co-runners, analyzing the relationship between progress and number of assigned cache ways. This analysis can be used to estimate the progress that each application would achieve when executed with co-runners had it a fixed number of exclusive cache ways assigned. Thus, it provides insight about what would be a reasonable fairness level to aim for when executing a set of applications concurrently.

A major challenge to address system fairness is the estimation of what would be the execution time of an application had it been executed without co-runners. This chapter implements and evaluates the two most well accepted approaches addressing this issue, concluding that, while both provide accurate estimations, the Application Slowdown Model is the most precise one.

The key contribution of this chapter is FPCP, a simple and effective hardware cache partitioning algorithm that balances the progress of the applications running on a system. Compared to previous approaches, the algorithm complexity is reduced from $\mathcal{O}(n^2)$ to only $\mathcal{O}(m)$, where n is the cache associativity and m the number of cores sharing the cache.

Chapter 6

Improving Fairness with LLC Partitioning using Intel CAT

This chapter proposes a family of clustering-based cache partitioning policies to address fairness in systems featuring Intel's CAT. The proposals act at two levels: applications showing similar amount of core stalls due to LLC accesses are grouped into clusters, and then, each cluster is given a number of ways using a simple mathematical model.

6.1 Introduction

Recent multicore processors typically implement a huge LLC to hide the large main memory access latencies. The size of these caches ranges from several tens of MBs to hundreds of MBs in recent processors like the IBM Power8 or the Intel Xeon Phi Knights Landing. Because of their large storage capabilities, as well as their high associativity (e.g., more than 16 ways), these caches are typically shared among all the cores in the processor. By default, all the running applications compete among each other for LLC space, which is governed by a single replacement policy. As a consequence, the applications replace blocks that belong to other applications, which can seriously degrade their performance. Moreover, it is difficult to predict the effect of

these inter-application interactions, since depending on their characteristics, some applications are affected more than others. That is, while the performance of some applications can be highly degraded, other applications may be unaffected, creating a fairness problem in the system.

Much research has focused on cache sharing over the past decade. Some of these works concentrate on performance [8, 43, 70, 75]; others target LLC cache fairness [17]; and yet others focus on providing system fairness [92, 24, 98]. The latter works consider system components other than the LLC (e.g., the memory controller). The vast majority of these works, however, present four main drawbacks that render their conclusions either invalid or inapplicable to recent processor generations. First, most of these works consider the L2 cache as the LLC, mainly because their research is performed in simulators, in which filling up a huge LLC of tens of MBs would require a prohibitive amount of simulation time. That means that since neither the cache geometry nor the data locality match, results cannot be easily extrapolated to recent commercial machines. Second, most of these works do not take into account the impact of hardware prefetchers or do not model the prefetchers employed in commercial machines, often not well documented. Third, these works either do not model the memory controller or model a simplified version. Fourth, some of these approaches require the use of extra hardware to obtain their inputs (e.g., the number of cache misses specifically caused by other co-runners). Since any runtime approach that deals with fairness has to estimate the slowdown the applications are suffering, most of previous research targeting fairness has this problem. Notable examples are the Per-Thread Cycle Accounting Architecture [21, 15] and the Application Slowdown Model [87]. Both approaches require extra hardware that is not readily available in any commercial processor.

The results of the discussed research regarding cache partitioning, however, were so promising that some processor manufacturers have implemented cache partitioning capabilities in their products. This is the case for recent Intel processors that feature the Cache Allocation Technology (CAT), which provides primitives to limit the amount of cache space a hardware thread can occupy in the LLC.

More precisely, CAT allows for a given number of ways to be assigned to a specific set of processes, a *Class of Service* or CLOS in Intel terminology. As there can be much more processes than classes of service, processes must be mapped to classes following a given policy. Therefore, a policy providing a limited number of cache ways to each application as done in previous works [70, 87] is unsuitable, since it would require a different CLOS for each application.

While this problem could be solved by assigning multiple applications to the same CLOS, we have characterized the slowdown each application experiences over isolated execution varying the number of LLC ways, and the results of this study show that assigning an exclusive subset of ways to each CLOS significantly reduces both system throughput and fairness compared to allowing different classes of service to share LLC ways.

This work proposes a family of clustering based cache partitioning policies that leverage the capabilities of Intel's Cache Allocation Technology to deal with system fairness. Although the proposal has been evaluated on an Intel Xeon E5 2658A v3, it is straightforward to port to any processor supporting CAT. It works by applying clustering techniques to group applications suffering from similar core stall cycles due to L2 misses into the same CLOS, and giving each CLOS an adequate number of LLC ways.

In this chapter we make the following key contributions:

- We propose a family of cache partitioning policies based on application clustering to improve system fairness on recent real machines.
- To the best of our knowledge, our proposal is the first to leverage state-of-the-art cache partitioning technologies, i.e., Intel's Cache Allocation Technology (CAT), to improve system fairness.
- We comprehensively evaluate the devised policies against the original system with no cache partitioning, and demonstrate improvements in system fairness by up to 80% (39% on average) for 8-application workloads and by up to 45% (25% on average) for 12-application workloads for a range of multiprogram workloads on modern hardware. This is done without significantly affecting the performance for 8-application workloads and improving it for 12-application workloads.

6.2 Progress Characterization and Estimation

The main aim of the cache partitioning scheme proposed in this chapter is to balance the progress among applications to improve system fairness. In concurrent execution, the cache interference caused by other applications reduces the *effective* number of cache ways a given application can use. To explore the sensitiveness of individual applications to the number of available cache ways, we conduct several experiments in which we use CAT to adjust the number of ways available to the application from 2 to 20 (i.e., the total cache

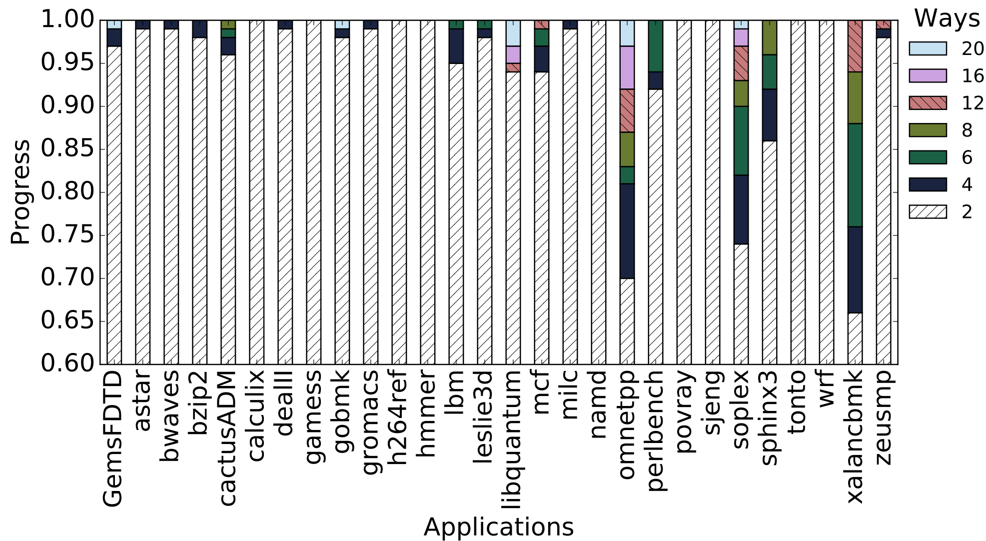


Figure 6.1: Effect of the number of cache ways on progress.

space). Using CAT, we prevent the application under study from using non-allocated ways. This approach allows modeling the reduction in the available cache space for a given application due to the cache interference induced by co-runners. This provides a reproducible way to study how progress is affected by co-runners competing for cache space.

Figure 6.1 shows the progress results for different applications of the SPEC CPU2006 benchmark suite. As observed, for a given number of assigned cache ways, applications achieve different progress levels. That is, the progress of each application exhibits a distinct sensitiveness to the available cache space. There are *highly cache sensitive* applications, like `xalancbmk`, `soplex` or `omnetpp`, whose progress is significantly harmed when the number of assigned ways drops below 6. In contrast, some applications are not affected at all; that is, they achieve 100% progress with only 2 cache ways. The rest of the applications fall somewhere in between, with different degrees of cache space sensitiveness. Note that each way represents 1.5 MB of the total cache space (i.e., 30 MB), so two ways are equivalent to 3 MB of LLC cache space, which is already a considerable amount.

Another way to look at this issue is to analyze how the number of assigned ways affects the slowdown. For this purpose, we plot the slowdown of each application as a function of the number of assigned ways. Figure 6.2 illustrates

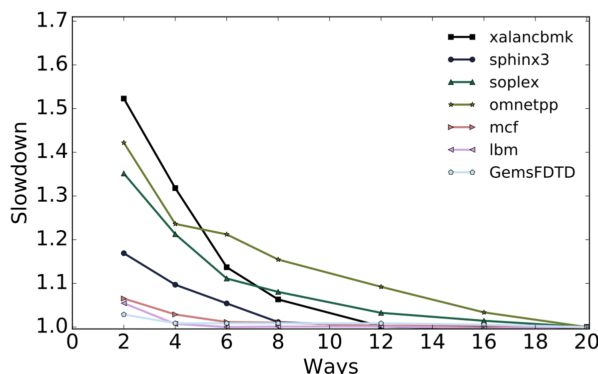


Figure 6.2: Slowdown when varying the available ways with respect to a 20-way cache.

the results for some applications of the SPEC suite. Looking at *highly cache sensitive* applications like `xalancbmk`, we find that the slowdown when varying the cache space can be modeled using an exponential function $a \cdot e^{-x} + b$, where a and b are constants that depend on the application and x is the cache space. We also explored other approximations, including a linear, quadratic and cubic function, however, we find the exponential function to yield the best fit. For instance, `soplex` follows the equation $5.24 \cdot e^{-x} + 0.026$.

The previous finding suggests that the slowdown grows exponentially as the number of assigned cache ways is reduced. Or, inversely, for having a linear reduction in slowdown we need an exponential increase in cache space. Note that previous research has found a square root relationship between cache space and hit ratio [11, 31].

Theoretically, we could use these equations to determine the assignment of cache ways. However, the *real* slowdown that a given application is suffering at runtime cannot be directly calculated since the execution time of the applications in isolation is unknown. Several previous works have focused on estimating this execution time. However, most of them require additional hardware [87, 21, 15, 17] to calculate the number of cycles the processor is stalled due to interference in the shared resources, or need to modify the OS scheduler [98, 24].

Our work targets unmodified commercial processors running a vanilla Linux kernel, so since we could not use previous research, we looked into the available performance events related to processor stalls due to shared resources, and studied the correlation between them and overall application slowdown.

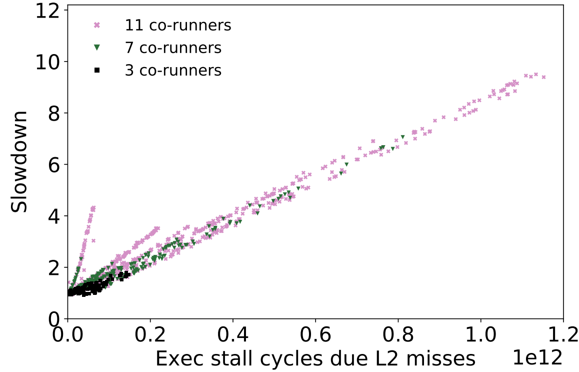


Figure 6.3: Correlation between L2 miss stall cycles and slowdown on the Intel Xeon E5 2658A v3.

We find that the $STALLS_L2_PENDING$ performance counter is the one that best correlates with application slowdown. This counter gathers the number of cycles during which the execution of an application is stalled due to L2 cache misses. Figure 6.3 plots slowdown versus the number of stalls (in trillions) gathered by the mentioned performance counter for several runs of SPEC applications executed with different numbers of co-runners (3, 7 and 11 random co-runners). As observed, there is a strong positive correlation (r=0.982, N=833, p=0.000) between the slowdown metric and the $STALLS_L2_PENDING$ count.

This correlation can be explained with the following rationale. The aforementioned L2 stalls counter is affected by the interference in all the shared resources in our experimental platform, which are the LLC, the main memory, and the on-chip interconnects (two rings connecting the L3 slices and the memory controllers), as depicted in Figure 3.2. This performance counter does not differentiate between stall cycles caused by *normal* misses and interference misses, but as the number of concurrently running applications grows, the stall cycles due to interference start to dominate (the ANTT for 8 and 12 concurrently running applications is, on average, over 3 and 4, respectively) so this drawback becomes less important.

Another way to understand this correlation is to examine the following equation:

$$Slowdown = \frac{ExecCycles}{ExecCycles_{alone}} =$$

$$= \frac{CoreCycles + STALLS_L2_PENDING}{ExecCycles_{alone}}, \quad (6.1)$$

where $CoreCycles$ represents the execution time minus the cycles stalled due to L2 misses. For high enough slowdowns, the result of this equation is dominated by STALLS_L2_PENDING.

Finally, as observed in Figure 6.3, there are some points that deviate from the main trend. These deviations are due to L3 block replacements that cause L2 invalidations to keep the inclusion principle (notice that the L2 cache is private and the whole cache hierarchy is inclusive [36]). In turn, these invalidations produce additional L2 misses that increase the number of $CoreCycles$. We verified this hypothesis by analyzing the data used for Figure 6.1 in which we varied the number of available ways for each application. As expected, we find that the applications that present a deviant behavior in Figure 6.3 have a sudden increase in L2 evictions when the cache space is reduced (2 or 3 ways in the LLC). This only happens for the applications that exhibit such behavior and not the rest.

This effect could be taken into account using a different performance event that considers L2. For instance, there is a performance counter that gathers the number of execution stalls due to L1 misses ($STALLS_L1D_PENDING$). Unfortunately, this counter does not behave correctly in our experimental platform (its value is always zero).

6.3 To Overlap or Not To Overlap Cache Ways

Most previous cache partitioning approaches work by assigning cache ways to applications to be used *exclusively*. That is, a given cache way can be only used by one application.

When using CAT capabilities, however, a wider design space opens. In addition to assigning ways exclusively to individual applications, cache ways can be: (i) allocated to a single CLOS hosting a set of applications (i.e., limited sharing), and (ii) allocated to multiple classes of service. To the best of our knowledge, only the first design choice has been considered in previous research [56, 103, 26]. Moreover, unlike this work which targets fairness, the focus of these approaches is on Quality-of-Service, and CAT capabilities are used to isolate latency-critical applications.

Assigning all the ways as private to individual applications becomes quite unrealistic on real hardware using CAT, mainly due to the high number of potentially running applications, the limited number of supported classes of service, and the limited number of cache ways. For instance, the Intel Xeon E5 2600 v3 family has 20 ways in the LLC and supports 4 CLOS. Therefore, no more than four applications could have exclusive ways assigned.

Even in the most recent Intel Broadwell EP machines that support up to 16 classes of service, the number of concurrently running applications can be higher, and the number of ways in the LLC is still limited to 20. Thus, if a workload with eight applications is executed, each application would have on average 2.5 ways, which clearly is insufficient to reach reasonable performance for most of them. Other partitioning schemes could be tried, giving more ways to some applications and less to others, but since there are applications that require a high number of ways to perform well (see `xalancbmk`, `omnetpp` and similar applications in Figure 6.1) this approach does not scale well and cannot be generalized.

Although grouping applications in classes of services that can access disjoint sets of ways may seem like a viable solution, it has the same problems as the previous approach, because while it partly solves the limitation in available classes of service, the number of ways in a CLOS is still too small for some applications to meet reasonable performance goals.

Two experiments were carried out to verify this claim. In one, the cache is divided in partitions of the same size for each CLOS and we try different clustering schemes to map 8 applications to 4 classes of service. In the other, we try several partitioning schemes with each partition being of a different size. In each case we also try different approaches to group applications (e.g., KMeans clustering, complementary cache requirements, similar cache requirements, etc.). In both experiments, throughput is significantly degraded without improving fairness with respect to no partitioning.

Consequently, all the partitioning approaches proposed in this work allow for overlapping LLC cache ways (i.e., CBMs) among classes of service.

6.4 Cluster-Based Partitioning Policies

As mentioned in Section 6.2, applications running on a multicore processor suffer from slowdown due to interference that arises from resource sharing. The interference varies during execution time depending on the run-time resource requirements of the applications and, since not all the applications have the same requirements and are not affected equally, unfairness arises. Our goal is to smartly partition cache resources to counteract the slowdown inequalities, which leads to a fairer system.

A cache partitioning mechanism can be characterized by three main design aspects [35]: target, evaluation metric and policy metric. Our proposal targets fairness, and the evaluation metric we use is the one defined in Section 1.1.5, the CoV of the slowdowns. This metric cannot be computed online, because the execution time alone cannot be measured at run-time, so using the insights presented in Section 6.2, we use the CoV of the $STALLS_L2_PENDING$ as the metric that guides our policies. Specifically, the goal is the minimize the CoV of the $STALLS_L2_PENDING$ counter.

Additionally, when using CAT for partitioning the cache three main design decisions must be taken: (i) the number of partitions in the LLC, (ii) which applications are assigned to each partition, and (iii) the amount of resources assigned to each partition.

Each design decision can, in turn, be either statically established or dynamically adjusted at run-time. These three axes open a new design space, summarized in Table 6.1, which greatly affects the performance and fairness that a policy provides. In this work we explore it and present the most relevant results. Although policies for each type that made sense have been devised and evaluated, for the sake of clarity only results for the best performing ones are presented in this chapter.

Taking all of this into account, we propose a family of application clustering algorithms, based on the $STALLS_L2_PENDING$ event. They target fairness and cover the key issues of the design space. The family consists of three main policies, namely SFn-mK, mK, and Dunn, where n and m are parameters of the policies, whose meaning is described below. The differences between policies mainly arise due to two aspects: the number of clusters the policy builds at run-time, and the form in which cache ways are assigned to clusters (i.e., fixed or dynamic).

Num. Clusters	Cluster sizes	Ways per cluster	Policies
S	S	S	
S	S	D	
S	**D**	**S**	**SFn-mK**
S	**D**	**D**	**mK**
D	S	S	
D	S	D	
D	D	S	
D	**D**	**D**	**Dunn**

Table 6.1: Design space and evaluated policies. Legend: S = static and D = dynamic.

All the proposed policies group applications in clusters using the KMeans algorithm [30] according to the number of core stalls due to L2 misses. Given n one-dimensional data points (only one variable — core stalls — is being considered per application), this algorithm distributes them into k clusters, assigning each application to its closest cluster, where the *closeness* is calculated as the Euclidean distance between the data point and the cluster centroid. A major advantage of using one-dimensional data (as in this case) is that an optimized version of KMeans can be used, with $\mathcal{O}(k \cdot n \cdot \log n)$ complexity. Once the clusters have been obtained, all the applications in a given cluster are assigned to the same CLOS. The number of LLC ways assigned to each CLOS and the number of clusters used depend on the specific policy.

SFn-mK Policies. In these policies, applications are grouped in m clusters using the KMeans algorithm. After the clustering process is done, the m clusters are sorted according to their centroid values in descending order and mapped to different classes of service. The cluster whose applications are suffering the highest slowdown (i.e., the most critical one) is given the highest priority and it is allowed write access to all the cache ways (i.e., its CBM is set to `0xFFFFF`). The following clusters receive a decreasing number of ways in steps of n according to their criticality. For example, for $n = 3$ and $m = 4$, the four clusters, sorted in critical order, receive 20, 17, 14 and 11 cache ways, respectively. Different values of n and m, ranging from 2 to 4, have been evaluated. In this work, we only show results for the policies of the form SFn-4K, which were the best performing.

mK Policies. These policies also group applications using the same criterion as the previous group of policies, but the number of ways assigned to each partition is not static but computed using a simple exponential function. We chose an exponential function because looking at Figure 6.2, to have a lin-

ear reduction in slowdown an exponential increase in cache space is required. Other functions were explored, such as linear and quadratic functions, but the exponential approach was the one providing the best results. The input to the exponential function is the normalized stalls of each cluster with respect to the most critical one, a value in the interval [0..1]. The output of the function is the number of assigned cache ways for the cluster, a value in the interval [2..20]. Figure 6.4 depicts the behavior of this policy for m clusters. In the example of the figure, cluster $m-1$ is the most critical one, and the number of assigned ways is 20, 10, 4, and 2, for the clusters $m-1$, $m-2$, 1, and 0, respectively.

Dunn Policy. This policy follows the same approach as the mK policy regarding clustering and the assignment of cache ways to classes of service. The number of classes of service to use is, unlike previous policies that consider a fixed number of clusters, dynamically determined at runtime to adapt to the different phases of the workload execution. To this end, two indices to evaluate clustering validity and determine the optimal number of clusters (Silhouette [73] and Dunn [16]) have been evaluated. In this chapter, results are only shown for the policy using the Dunn index, since it yielded slightly better results on our experimental platform, thus we refer to this approach as the Dunn policy.

The Dunn index is defined as follows. Assuming k denotes the number of clusters, d_{min} the minimal distance between points of different clusters, and d_{max} the largest within-cluster distance; the Dunn index for k clusters is then computed as

$$Dunn_k = \frac{d_{min}}{d_{max}}.$$

The larger the Dunn index, the better the clustering. As a result, the k value that maximizes the Dunn index is selected.

6.5 Experimental Setup

All the experiments have been performed on the Intel Xeon E5 2658 v3 processor described in Section 3.2, one of the first Intel processors to support Cache Allocation Technology.

The proposal focuses on the LLC, where the space is distributed according to the different devised policies using CAT. In order to make the experiments repeatable, important system details are provided next. The processor

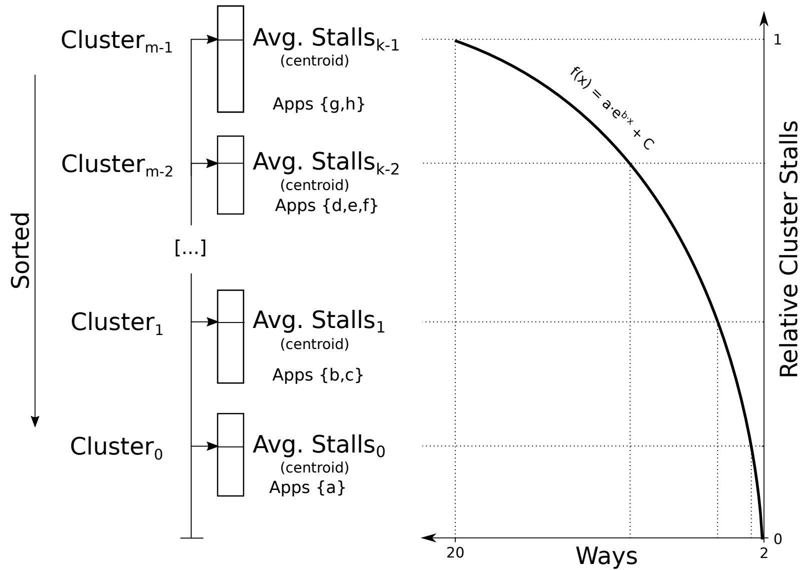


Figure 6.4: Policy with clustering and a model for the ways.

frequency was fixed to 2.20 GHz with the Linux performance governor. The main memory of the system has a maximum theoretical bandwidth of 68 GB/s across 4 channels. We experimentally measured memory latencies of 75 ns for an idle machine and 570 ns for a saturated machine. The machine has 4 types of hardware prefetchers: 2 prefetchers associated with the L1-data cache and 2 prefetchers associated with the L2 cache. All of them were kept enabled during the experiments.

All the devised schemes, explained in Section 6.4, have been evaluated and compared against a baseline that performs no partitioning (referred as NoPart).

The experiments have been conducted with two sets of 45 multiprogram mixes from the SPEC CPU 2006 benchmark suite using the reference input set. The first set contains 8-application workloads and the second 12-application workloads. To compose the application mixes, we first classify the applications in the SPEC benchmark suite into two categories, cache sensitive and cache insensitive, based on the offline evaluation performed in Section 6.2. Then, we create workloads varying the ratio of sensitive to insensitive applications. All the workloads are randomly generated, and results are collected executing each workload until all the applications have completed the same number of instructions they execute when running alone on the machine for 60 seconds.

Our experimental environment consists of a *manager* program that reads a configuration file with a list of workloads and the partitioning policy that will be used. The *manager* then *forks* and *execs* as many times as necessary to launch the applications in the workload. At regular intervals of 500 ms the *manager* reads the required performance counters and uses this information to properly size the partitions and assign applications to classes of service. We tried other two different intervals for adjusting the partitioning: 100 ms and 1000 ms. In the first case there was no significant difference in the results, but the overhead was higher, since the manager was active more frequently. In the second case the results were worse compared to the same configuration with a smaller interval.

When an application executes as many instructions as it would run in isolation during 60 seconds, the *manager* restarts it. However, only the results from the first run of each application are taken into account. Note that due to limitations in the libraries employed for performance monitoring [40] and cache partitioning [41], applications need to be pinned to cores[1].

Each experiment has been repeated *a minimum* of 10 times, until the margin of error was less than 1%, with a confidence of 95%. This applies to all the plots and data shown in this chapter, so no confidence intervals are drawn on the figures.

6.6 Evaluation

This section is aimed at providing insights and quantify the benefits of the devised policies. To this end, the evaluation focuses on three main design concerns: (i) Can a simple static policy provide significant fairness? (ii) Should the number of clusters match the maximum number of CLOSes supported by the machine all the time? (iii) Would dynamically adapting the number of classes of service to the *optimal* number of clusters further improve the results? The three devised policies and the experiments discussed below were designed to answer these three questions.

[1]Starting from version 4.10, the Linux kernel has native support for CAT, and it does not have this limitation. Unfortunately, it was not available when the experiments were performed.

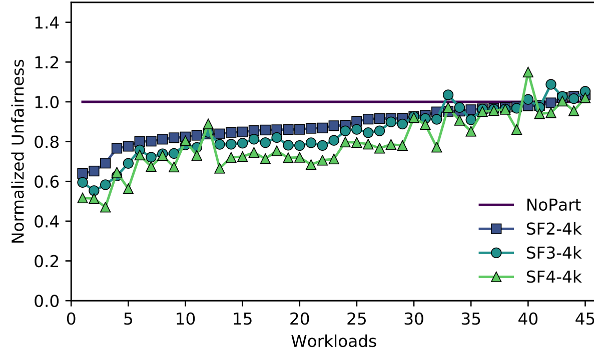


Figure 6.5: Normalized unfairness across the 45 8-application workloads with the SF2-4K, SF3-4K and SF4-4K policies.

6.6.1 Exploring Unfairness Enhancements with a Simple Policy

To explore the potential of CAT to help facing unfairness, we evaluate the simple SFn-4K policy across the studied 45 8-application workloads. Remember that this policy assumes four clusters, and the number of ways assigned to each cluster is decreased from 20 (assigned to the highest priority cluster) in steps of n. In this experiment, we vary the value of n from 2 to 4.

Figure 6.5 shows the results. To plot the curves, we first have sorted the 45 workloads in ascending order depending on the normalized unfairness they present using the SF2-4K policy. Then, SF2-4K and SF3-4K have been plotted following the same order. As observed, this policy, by merely assigning a static number of ways to each CLOS and using 4 classes of service improves unfairness significantly, by 12%, 16% and 21% on average for SF2-4K, SF3-4K and SF4-4K, respectively. Also note that not all the mixes obtain the best results with the same value of n. Although SF4-4K presents the best results, in around 10% of the workloads, another value of n yields better results.

Important conclusions can be drawn from this experiment. First, the amount of stalls caused by missing in the L2 acts as a good criterion to group applications in clusters. Second, regardless of the value of n, unfairness is significantly improved over the non-partitioning approach. Third, CAT presents high potential to improve system fairness even with a simple policy based on clustering.

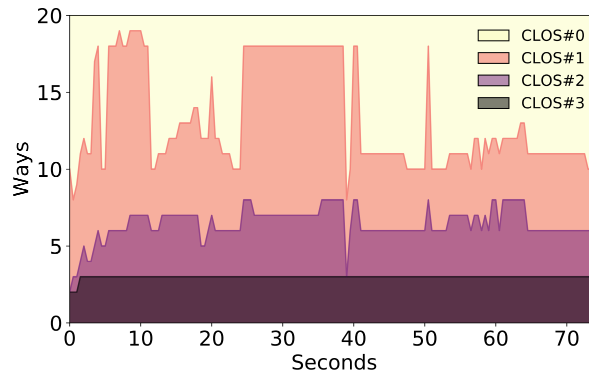


Figure 6.6: Ways assigned to each CLOS during the execution of the 4th 8-application workload.

6.6.2 Determining the Number of Cache Ways Assigned to Each Cluster Dynamically

Section 6.6.1, using a simple policy and exploring only three values of n, has shown that there is not a number of ways that can be assigned statically to clusters to provide the best results for every workload. In other words, there is not a single optimal n value for all the workloads. Additionally, although not measured in the previous experiment, cache space requirements of the applications in a workload can change during the execution so being able to adapt to these changes can provide further unfairness improvements.

To deal with the shortcomings of a static way allocation, the mK policy was devised, which dynamically assigns the number of ways that best fits the cache space requirements for each of the application clusters. This policy allows greater flexibility, since the number of ways assigned to each cluster is adjusted dynamically at runtime. Figure 6.6 illustrates how the number of ways assigned to each CLOS varies during the execution of the 4th 8-application workload running under the 4K policy (i.e., grouping applications into 4 clusters). As observed, unlike the previous policies (the SFn-mK group of policies), the number of cache ways assigned to each cluster varies at runtime, which allows this policy to bring important benefits over the SFn-4K, as our results will show.

The mK policy has been evaluated while setting the value of m to 2, 3 and 4 clusters, see Figure 6.7. The 45 8-application workloads have been sorted in increasing unfairness order, according to the results of the 4K policy. The

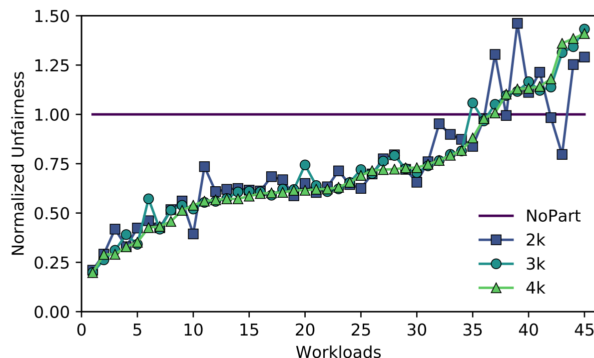


Figure 6.7: Normalized unfairness across the 45 8-application workloads with the 2K, 3K and 4K policies.

2K and 3K policies have been plotted following the same order. Counter-intuitively, not always the maximum number of clusters (i.e., 4) shows the best results; even more, on average, the policy with two clusters (2K) yields slightly better results than the one with 4 clusters (4K). The reason is that changing the clustering (i.e., the number of clusters and the applications in them) displaces the centroids and thus can have a significant impact on the number of assigned cache ways to each cluster. Moreover, less clusters oftentimes means more ways per CLOS, which improves the performance of applications with high cache space requirements.

Taking into account these results and comparing them with the ones presented in Figure 6.5, two important conclusions can be drawn. First, the major fairness benefits come from the fact that cache ways are dynamically assigned to clusters at run-time, rather than the number of clusters itself. Notice that in Figure 6.7, the normalized unfairness starts from below 0.25 while in Figure 6.5 it starts from around 0.5. Second, additional benefits could be brought by dynamically selecting the proper number of clusters.

Note that some workloads experience an increase in unfairness (see rightmost data points in Figure 6.7). This is the case for workloads composed mostly of applications that are not cache-sensitive, so the absolute (non-normalized) unfairness is low. Their absolute unfairness is around 0.07, which is significantly less than the global average around 0.23. So, while in some corner cases we see a small increase in unfairness in absolute terms, this is compensated for by large reductions in unfairness (both in absolute and relative terms) in the general case. The explanation for this behavior is that in low unfairness

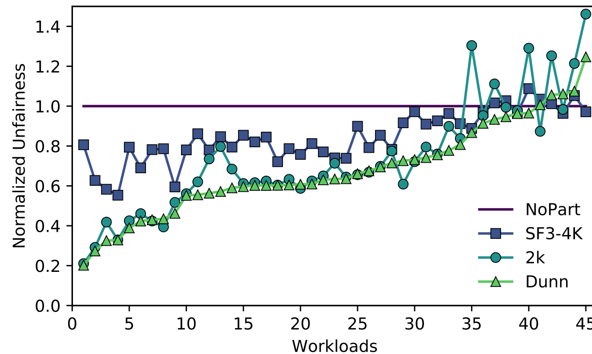


Figure 6.8: Normalized unfairness results with respect to NoPart for the 45 8-application workloads with the different partitioning policies.

scenarios, and for some specific applications, our metric slightly overestimates the cache space required. This could be addressed by employing an unfairness threshold to enable or disable the policy, but since the increase in unfairness is so small, we decided to keep the policy as simple as possible.

6.6.3 Putting It All Together: the Dunn Policy

The previous discussion indicates that a dynamic policy selecting the optimal number of clusters for each workload, and the optimal number of ways for each cluster would be the best approach. This claim led us to design the Dunn partitioning policy, which is the one that provides the best results overall.

Figure 6.8 shows normalized unfairness over no partitioning for each of the studied 8-application workloads and compares the Dunn policy with the 2K and SF3-4K policies, since both policies achieve a significant reduction in unfairness, with performance results within the same range as Dunn. The workloads are sorted in ascending unfairness order according to the Dunn policy results, and the 2K and SF3-4K results have been plotted following the same order. Clearly, the 2K and Dunn policies are the ones that reduce unfairness the most over no partitioning, by on average 36% and 39%, respectively, and by up to 80%. The other policy, SF3-4K, has less effect on unfairness, reducing it by 16% on average.

Figure 6.9 presents the average results for different metrics (see Section 1.1.5 and Section 1.1.6) across all the 8-application workloads achieved by the Dunn policy. The results have been normalized against the NoPart baseline. Note

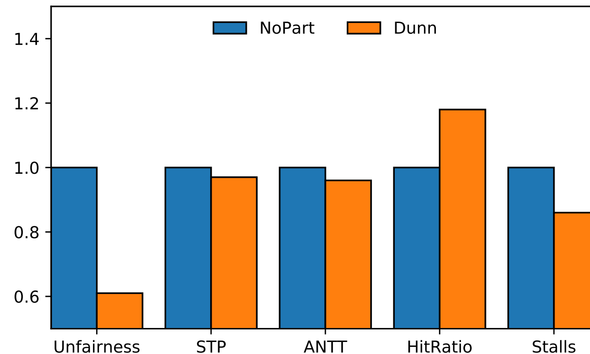


Figure 6.9: Average Dunn results normalized against NoPart for the 8-application workloads.

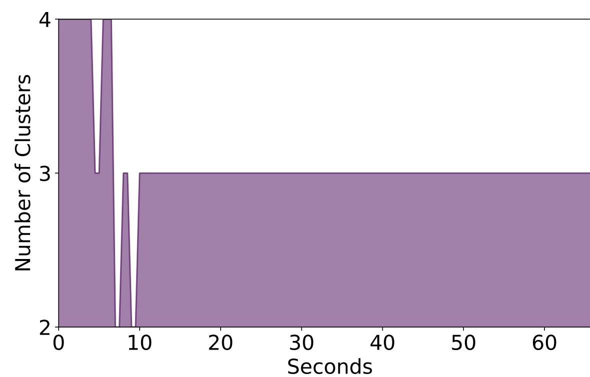


Figure 6.10: Number of clusters used during the execution of the 12th 8-application workload.

that there is a clear inverse correlation between LLC hit ratio and unfairness. The reason is that most of the time unfairness is caused by the slowest applications, which frequently access the cache but miss due to lack of enough cache space. As a consequence, these applications stall for long periods. As the Dunn policy gives more cache ways to the slowest applications, their accesses start to hit the cache, which reduces the number of stalls and improves system fairness. This behavior also improves per-application performance (ANTT), although the STP metric is slightly reduced because the fastest applications are given fewer resources.

As explained above, the Dunn policy dynamically chooses the optimal number of clusters. To analyze if the improvements that Dunn provides over the mK

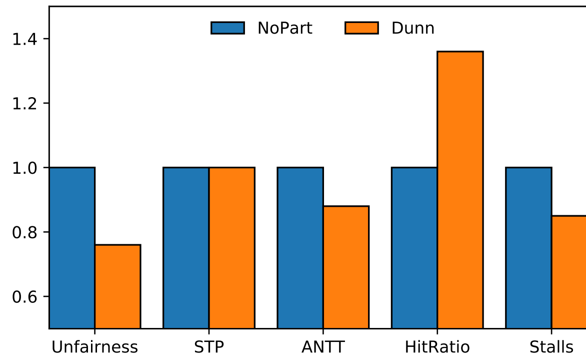


Figure 6.11: Average Dunn results normalized against NoPart for the 12-application workloads.

policies come, in fact, from choosing the *correct* number of clusters, Figure 6.10 plots the number of clusters used during the execution time of a workload (workload number 12 in Figure 6.8 and number 11 in Figure 6.7[2]). According to Figure 6.7, it seems clear that two clusters does not work well for this workload, and that three and four clusters perform similarly. Figure 6.8 shows that for this workload, Dunn performs significantly better than 2K, so it must be picking three and four clusters as the optimal number of clusters. As expected, Figure 6.10 corroborates this.

In summary, a dynamic policy to adapt the number of clusters and the number of ways per cluster can, in fact, provide meaningful fairness improvements compared to static policies. Another insight is that the potential of CAT seems to be limited more by the number of available ways in the LLC than by the number of classes of service.

6.6.4 Results with 12 Cores

To evaluate the scalability potential of the Dunn policy we now consider 12-application workloads created following the approach described in Section 6.5. Figures 6.11 and 6.12 provide insight regarding how the Dunn policy affects system performance, unfairness and LLC hit ratio.

Looking at Figure 6.11 it is clear that the important unfairness reduction achieved for the 8-application workloads is also accomplished for the 12-applica-

[2]While the workloads in both figures are the same, they have been sorted following a different criterion.

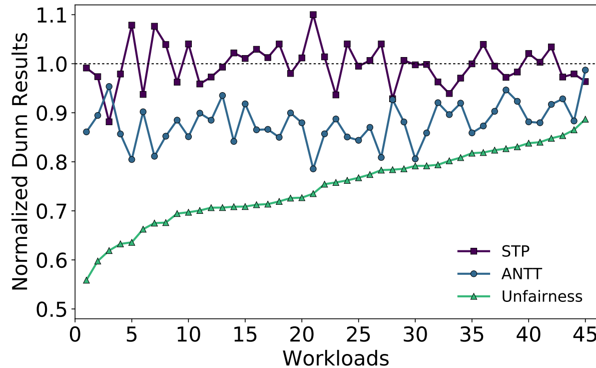


Figure 6.12: Dunn results normalized against NoPart for the 45 12-application workloads.

tion workloads, where unfairness is reduced by 25% on average. This unfairness reduction does not affect STP, which on average remains the same, but comes accompanied by a significant improvement in ANTT, which decreases by 12% on average. As with 8-application workloads, the unfairness and performance (ANTT) improvements are because the Dunn policy greatly increases the LLC hit ratio, which in turn reduces the number of cycles that cores are stalled (by 15%, on average).

Figure 6.12 shows detailed results for each one of the workloads. Unfairness is reduced for all the workloads by at least 12%, and up to 45%. ANTT also improves for all the workloads. Finally, depending on the workload, the Dunn policy presents an STP that deviates from NoPart by less than 10%.

6.7 Summary

This chapter has presented a family of cache partitioning policies to address system fairness on commercial Intel processors using Intel CAT. This chapter has proven that *STALL_L2_PENDING* event counter behaves as a good proxy for per-application slowdown. Therefore, all the devised policies employ this counter to cluster applications into classes of service. Once the number and composition of the classes of service have been established, each class is given a number of ways according to a simple mathematical function. The proposed partitioning policies drastically reduce the system unfairness for 8- and 12-application workloads, without damaging the performance.

Regarding the number of classes of service and the number of cache ways assigned to each of them two main conclusions can be drawn. First, in most of the evaluated workloads, using only two classes of service during the whole execution time allows achieving outstanding system fairness results. In other words, counterintuitively, using additional classes of service does not always result in further system fairness enhancements. Notice that this observation contrasts with the current Intel trend, which has increased the number of supported classes of service from 4 to 16 in the latest Intel microprocessor generation. Second, instead of using a large number of classes of service, the key issue to deal with system fairness lies on the function employed to assign cache ways to classes of service. We find that an exponential distribution of cache ways is the one that best improves system fairness.

Chapter 7

Conclusions

This dissertation addressed the issues with resource sharing in multicore processors from the performance and fairness perspective, focusing on two major shared resources: the main memory bandwidth and the LLC. It proposed techniques to reduce the main memory bandwidth usage of the core prefetchers, and devised LLC partitioning schemes that improve system fairness.

In this chapter the main contributions of these proposals are summarized, followed by a discussion about future work and an enumeration of the scientific publications related with this dissertation.

7.1 Contributions

Multicore processors emerged as a solution to continue increasing performance while facing the power consumption wall of complex and monolithic single-threaded processors. To improve resource utilization and the performance per area, current designs share important off-core resources among the processor cores, the most significant of them being the main memory bandwidth and the LLC, which may be L2 or L3, depending on the system.

Regarding main memory bandwidth, Chapter 4 of this thesis focused on the impact of prefetching on this limited resource. The reason for the interest

in this topic is that, while current prefetchers work well to improve per-core performance, they do not take into account the shared nature of main memory bandwidth, potentially damaging the performance of applications running on other cores if the prefetches consume a large amount of bandwidth.

To deal with this problem, a characterization study of the effect of prefetching on memory bandwidth consumption and performance was conducted. It showed that the applications experience different phases of execution, that can be classified in four categories: memory intensive and prefetch unfriendly, memory intensive and prefetch friendly, non memory intensive and prefetch unfriendly, and finally, non memory intensive and prefetch friendly. From this study it was concluded that, during prefetch unfriendly phases, a prefetcher can be disabled or its aggressiveness reduced with little or no impact on performance. Moreover, it also showed that it may be worthy to throttle down individual prefetchers during memory intensive phases if other cores require more bandwidth, to improve overall system performance.

Building on these findings, this dissertation proposed ADP, a selective prefetcher that dynamically throttles or disables the core prefetcher, taking into account both its performance and the bandwidth requirements of other cores. Results showed that ADP increases STP on average by 6% over no prefetching, considering both memory intensive and combined workloads, while HPAC (a state-of-the-art prefetcher) improves this metric by 4%. Moreover, these performance gains are achieved improving unfairness over the other two approaches. Compared to no prefetching, the aggressive baseline prefetcher and HPAC increased the amount of memory requests by 53% and 36%, respectively, while ADP increased them by 21%. This reduction and the faster execution time resulted in important main memory energy savings. On average, energy consumption increased by 12% and 20% in HPAC and the aggressive baseline prefetcher, respectively, over no prefetching. In contrast, with ADP energy just increased by 3%. To prove its independence with respect to the underlaying prefetcher, ADP was also evaluated with a different prefetching mechanism, one that tracks the sequence of accesses of individual load instructions to determine whether to prefetch additional lines, and the results were promising.

With respect to cache partitioning, this dissertation presented the results of two different approaches, both targeting unfairness reduction. One was developed in a simulated system and the other in a real machine. Both are conceptually similar: they determine which applications are suffering more slowdown and try to compensate it by providing them with more cache space. However, in a real machine one is limited to what the hardware allows, so

the slowdown estimation and the partition scheme used need to be adapted to these hardware constraints.

The first approach, presented in Chapter 5 estimates an application slowdown by identifying at run-time the cache misses that would have not occurred had the application been executed alone (i.e. without co-runners). This is done by using an Alternate Tag Directory or ATD, a per core private structure that keeps track of what the status of a shared cache would be if it were not shared with other cores. To keep its size small, it only stores tags and replacement bits, not data, and only for a subset of the cache sets. In this approach the Cache Access Ratio, estimated using the ATD, is used to estimate the slowdown per application. This slowdown estimation is then used by FPCP to distribute cache ways. This approach proposes adding additional hardware structures not present in commercial processors, it was tested in a simulated system, varying the number of applications running concurrently. Experimental results showed that, compared to a system with no partitioning, FPCP reduced unfairness by 48% in four-application workloads and by 28% in eight-application workloads, without harming the performance.

Building on the experience gained developing the work presented in Chapter 5, in Chapter 6 we proposed a family of cache partitioning algorithms targeting commercial machines. They all target unfairness reduction and cover the key issues of the design space. The differences between policies mainly arise due to two main aspects: the number of clusters the policy builds at run-time, and the form (i.e. fixed or dynamic) in which cache ways are assigned to clusters.

All the proposed policies group applications in clusters using the KMeans algorithm according to the number of core stalls due to L2 misses. Given n one-dimensional data points (only one variable, core stalls, is being considered per application), this algorithm distributes them into k clusters, assigning each application to its closest cluster, where the *closeness* is calculated as the Euclidean distance between the data point and the cluster centroid. A major advantage of using one-dimensional data (as in this case) is that an optimized version of KMeans can be used, with $\mathcal{O}(k \cdot n \cdot \log n)$ complexity. Once the clusters have been obtained, all the applications in a given cluster are assigned to the same CLOS. The number of LLC ways assigned to each CLOS and the number of clusters used depend on the specific policy. Experimental results showed that for 8-application workloads, our best performing policy reduced system unfairness by up to 80% (39% on average) for 8-application workloads and by up to 45% (25% on average) for 12-application workloads compared to a non-partitioning approach without harming overall system performance

(STP) and even significantly improving per-application performance (ANTT) for 12-application workloads.

7.2 Future Directions

New Intel processors allow to partition the memory bandwidth analogously as how the LLC can be partitioned in those systems. As future work, we plan to develop a partitioning scheme for bandwidth, targeting both fairness and performance.

We also plan to adapt our selective prefetcher ideas to a real machine, and combine all our proposals in a unified scheme: an approach that dynamically adjusts core prefetchers, LLC space, and main memory bandwidth to meet fairness, performance or other targets.

In addition, we are developing custom scheduling algorithms that make use of the proposed partitioning and prefetching policies to further enhance both performance and fairness.

7.3 Publications

Below we list in reverse chronological order the publications related with this thesis, broken down in international conferences, international journals, and domestic conferences.

7.3.1 International Conferences

- Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit and María Engracia Gómez. "Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology". In *Proceedings of the 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Portland, OR, USA. 2017, pp. 194–205.
- Vicent Selfa, Julio Sahuquillo, Salvador Petit and María Engracia Gómez. "Student Research Poster: A Low Complexity Cache Sharing Mechanism to Address System Fairness". In *Proceedings of the 25th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Haifa, Israel. 2016, p. 455.

- Vicent Selfa, Crispín Gómez, María Engracia Gómez and Julio Sahuquillo. “A Simple Activation/Deactivation Prefetching Scheme for Chip Multiprocessors”. In *Proceedings of the 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Heraklion, Crete, Greece. 2016, pp. 143–150.
- Vicent Selfa, Julio Sahuquillo, Crispín Gómez and María Engracia Gómez. “Methodologies and Performance Metrics to Evaluate Multiprogram Workloads”. In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Turku, Finland. 2015, pp. 150–154.

7.3.2 International Journals

- Vicent Selfa, Julio Sahuquillo, Salvador Petit and María Engracia Gómez. “A Hardware Approach to Fairly Balance the Inter-Thread Interference in Shared Caches”. In *IEEE Transactions on Parallel and Distributed Systems* 28(11) (2017), pp. 3021–3032
- Vicent Selfa, Julio Sahuquillo, María Engracia Gómez, Crispín Gómez. “Efficient Selective Multicore Prefetching under Limited Memory Bandwidth”. In *Journal of Parallel and Distributed Computing*, DOI: 10.1016/j.jpdc.2018.05.002.

7.3.3 Domestic Conferences

- Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit and María Engracia Gómez. “Particionado de cache mediante Intel Cache Allocation Technology para mejorar la equidad del sistema”. In *Actas de las XXVIII Jornadas de Paralelismo.* Málaga, Spain. 2017, pp. 231–240.
- Vicent Selfa, Julio Sahuquillo, María Engracia Gómez and Salvador Petit. “Particionado dinámico de cachés compartidas para maximizar la equidad entre tareas”. In *Actas de las XXVII Jornadas de Paralelismo.* Salamanca, Spain. 2016, pp. 455–460.
- Vicent Selfa, Julio Sahuquillo, Maria Engracia Gomez and Crispín Gómez. “Metodologías y Métricas de Prestaciones Para Evaluar Cargas Multiprogramadas”. In *Actas de las XXVI Jornadas de Paralelismo.* Córdoba, Spain. 2015, pp. 318–323.

- Vicent Selfa, Paula Navarro, Crispín Gómez, María Engracia Gómez and Julio Sahuquillo. "Diseño de mecanismos de prebúsqueda adaptativa bajo gestión eficiente de memoria para procesadores multinúcleo". In *Actas de las XXIV Jornadas de Paralelismo*. Madrid, Spain. 2013, pp. 43–48.

All the works listed above are exclusively related with this thesis. The contributions of the Ph.D candidate to them have been the implementation of the proposed techniques, the setup and execution of the experiments, the writing of the paper drafts, and the presentation of the different proposals in conferences. The co-authors have collaborated in the paper writing, helped in the design of the proposals, and provided invaluable general advice.

7.3.4 Indirectly Related Conferences and Journals

The following journal and conference papers are indirectly related with this dissertation.

- Lucía Pons, Vicent Selfa, Julio Sahuquillo, Salvador Petit and Julio Pons. "Improving System Turnaround Time with Intel CAT by Identifying LLC Critical Applications". In *Proceedings of the 24th International Conference on Parallel and Distributed Computing (Euro-Par)*, Turin, Italy, August 2018.
- Salvador Petit, Julio Sahuquillo, María Engracia Gómez and Vicent Selfa. "A research-oriented course on Advanced Multicore Architecture: Contents and active learning methodologies". In *Journal on Parallel and Distributed Computing* 105 (2017), pp. 63–72.
- Paula Navarro, Vicent Selfa, Julio Sahuquillo, María Engracia Gómez and Crispín Gómez Requena. "Row Tables: Design Choices to Exploit Bank Locality in Multiprogram Workloads". In *Proceedings of the 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, Turku, Finland. 2015, pp. 22–26.
- José Duro, José Puche, Vicent Selfa, María Engracia Gómez, Salvador Petit and Julio Sahuquillo. "Explotación de la Localidad de Banco en Cargas Multiprogramadas". In: *Actas de las XXVI Jornadas de Paralelismo*. Córdoba, Spain. 2015, pp. 305–312.
- Julio Sahuquillo, Salvador Petit, Vicent Selfa and María Engracia Gómez. "ATP: Una Asignatura Orientada a la Investigación en la Arquitectura

de los Procesadores Multinúcleo". In *Actas de las XXVI Jornadas de Paralelismo*. Córdoba, Spain. 2015, pp. 324–330.

- Julio Sahuquillo, Salvador Petit, Vicent Selfa and María Engracia Gómez. "A Research-Oriented Course on Advanced Multicore Architecture". In *Proceedings of the International Parallel and Distributed Processing Symposium Workshop (IPDPS)*, Hyderabad, India. 2015, pp. 760–765.

Bibliography

[1] Vish Viswanathan (Intel). *Disclosure of H/W prefetcher control on some Intel processors.* URL: https://software.intel.com/en-us/articles/disclosure-%20of-hw-prefetcher-control-on-some-intel-processors.

[2] ARM Holdings. *ARM DynamIQ Shared Unit.* URL: http://infocenter.arm.com/help/topic/com.arm.doc.100453_0300_00_en/dsu_trm_100453_0300_00_en.pdf.

[3] Jean-Loup Baer. *Microprocessor Architecture: From Simple Pipelines to Chip Multiprocessors.* Cambridge University Press, 2010. ISBN: 9780521769921.

[4] Peter Bergner, Brian Hall, Alon Shalev Housfater, Madhusudanan Kandasamy, Tulio Magno, Alex Mericas, Steve Munroe, Mauricio Oliveira, Bill Schmidt, Will Schmidt, Bernard King Smith, Julian Wang, Suresh Warrier, and David Wendt. *Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8.* IBM redbooks. 2015. ISBN: 9780738440927.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. "The Gem5

Simulator". In: *Computer Architecture News* 39.2 (2011), pp. 1–7. ISSN: 0163-5964.

[6] Shankar Balachandran Biswabandan Panda. "Expert Prefetch Prediction: An Expert Predicting the Usefulness of Hardware Prefetchers". In: *IEEE Computer Architecture Letters* 15.1 (2016), pp. 13–16. ISSN: 0360-0300.

[7] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. "An Evaluation of High-Level Mechanistic Core Models". In: *Transactions on Architecture and Code Optimization (TACO)* 11.3 (2014), 28:1–28:25. ISSN: 1544-3566.

[8] Francisco J. Cazorla, Alex Ramírez, Mateo Valero, Peter M.W. Knijnenburg, Rizos Sakellariou, and Enrique Fernández. "QoS for High-Performance SMT Processors in Embedded Systems". In: *IEEE Micro* 24.4 (2004), pp. 24–31. ISSN: 0272-1732.

[9] Jichuan Chang and Gurindar S. Sohi. "Cooperative Cache Partitioning for Chip Multiprocessors". In: *Proceedings of the 21st Annual International Conference on Supercomputing (ICS)*. Seattle, WA, USA, 2007, pp. 242–252.

[10] Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut Kademir, Anand Sivasubramaniam, Onur Mutlu, and Chita R. Das. "Application-aware prefetch prioritization in on-chip networks". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Minneapolis, MN, USA, 2012, pp. 441–442.

[11] C. K. Chow. "Determination of Cache's Capacity and Its Matching Storage Hierarchy". In: *IEEE Transactions on Computers* 25.2 (1976), pp. 157–164. ISSN: 0018-9340.

[12] Henry Cook, Miquel Moretó, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. "A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-Efficiency while Preserving Responsiveness". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. Tel-Aviv, Israel, 2013, pp. 308–319.

[13] Xianglei Dang, Xiaoyin Wang, Dong Tong, Zichao Xie, Lingda Li, and Keyi Wang. "An adaptive filtering mechanism for energy efficient data prefetching". In: *Proceedings of the 18th Asia and South Pacific Design Automation Conference (ASP-DAC)*. Yokohama, Japan, 2013, pp. 332–337.

[14] Reetuparna Das, Rachata Ausavarungnirun, Onur Mutlu, Akhilesh Kumar, and Mani Azimi. "Application-to-core mapping policies to reduce memory system interference in multi-core systems". In: *Proceedings of the 19th International Symposium on High Performance Computer Architecture (HPCA)*. Shenzhen, China, 2013, pp. 107–118.

[15] Kristof Du Bois, Stijn Eyerman, and Lieven Eeckhout. "Per-thread Cycle Accounting in Multicore Processors". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 9.4 (2013), 29:1–29:22. ISSN: 1544-3566.

[16] Joseph. C. Dunn. "A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters". In: *Journal of Cybernetics* 3.3 (1973), pp. 32–57.

[17] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. "Fairness via Source Throttling: A Configurable and High-performance Fairness Substrate for Multi-core Memory Systems". In: *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Pittsburgh, PA, USA, 2010, pp. 335–346.

[18] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. "Prefetch-aware shared resource management for multi-core systems". In: *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*. San Jose, CA, USA, 2011, pp. 141–152.

[19] Eiman Ebrahimi, Onur Mutlu, Chang Joo Lee, and Yale N. Patt. "Coordinated Control of Multiple Prefetchers in Multi-core Systems". In: *Proceedings of the 42nd International Symposium on Microarchitecture (MICRO)*. New York, NY, USA, 2009, pp. 316–326.

[20] Brian Everitt. *The Cambridge dictionary of statistics*. 2002. ISBN: 978-0-521-76699-9.

[21] Stijn Eyerman and Lieven Eeckhout. "Per-thread Cycle Accounting in SMT Processors". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Washington, DC, USA, 2009, pp. 133–144.

[22] Stijn Eyerman and Lieven Eeckhout. "Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance". In: *IEEE Computer Architecture Letters* 13.2 (2014), pp. 93–96. ISSN: 1556-6056.

[23] Stijn Eyerman and Lieven Eeckhout. "System-Level Performance Metrics for Multiprogram Workloads". In: *IEEE Micro* 28.3 (2008), pp. 42–53. ISSN: 0272-1732.

[24] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. "Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler". In: *Proceedings of the 29th International Parallel and Distributed Processing Symposium (IPDPS)*. Hyderabad, India, 2015, pp. 187–196.

[25] Antonio Flores, Juan L. Aragón, and Manuel E. Acacio. "Energy-Efficient Hardware Prefetching for CMPs Using Heterogeneous Interconnects". In: *Proceedings of the 18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*. Pisa, Italy, 2010, pp. 147–154.

[26] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. "Ginseng: Market-Driven LLC Allocation". In: *Proceedings of the Usenix Annual Technical Conference (USENIX ATC)*. Denver, CO, USA, 2016, pp. 295–308.

[27] Ron Gabor, Shlomo Weiss, and Avi Mendelson. "Fairness Enforcement in Switch on Event Multithreading". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 4.3 (2007). ISSN: 1544-3566.

[28] J. D. Gindele. "Buffer block prefetching method". In: *IBM Technical Disclosure Bulletin* 20.2 (1977), pp. 696–697.

[29] Saurabh Gupta and Huiyang Zhou. "Spatial Locality-Aware Cache Partitioning for Effective Cache Sharing". In: *Proceedings of the 44th In-*

ternational Conference on Parallel Processing (ICPP). Beijing, China, 2015, pp. 150–159.

[30] J. A. Hartigan and M. A. Wong. "Algorithm AS 136: A K-Means Clustering Algorithm". In: *Journal of the Royal Statistical Society* 28.1 (1979), pp. 100–108. ISSN: 00359254, 14679876.

[31] Allan Hartstein, Vijayalakshmi Srinivasan, Thomas R. Puzak, and Philip G. Emma. "On the Nature of Cache Miss Behavior: Is It $\sqrt{2}$?" In: *Journal of Instruction-Level Parallelism* 10 (2008). ISSN: 1942-9525.

[32] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. "Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family". In: *Proceedings of the 22nd International Symposium on High Performance Computer Architecture (HPCA)*. Barcelona, Spain, 2016, pp. 657–668.

[33] ARM Holdings. *ARM Cortex-A53 MPCore Processor. Technical Reference Manual.* URL: `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0500d/DDI0500D_cortex_a53_r0p2_trm.pdf`.

[34] Yatin Hoskote, Sriram R. Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. "A 5-GHz Mesh Interconnect for a Teraflops Processor". In: *IEEE Micro* 27.5 (2007), pp. 51–61. ISSN: 0272-1732.

[35] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar R. Iyer, and Srihari Makineni. "Communist, Utilitarian, and Capitalist Cache Policies on CMPs: Caches As a Shared Resource". In: *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Seattle, WA, USA, 2006, pp. 13–22.

[36] Intel Corporation. *Improving Real-Time Performance by Utilizing Cache Allocation Technology.* 31843-001US. 2015. URL: `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf`.

[37] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual.* URL: `https://software.intel.com/en-us/articles/intel-sdm#combined`.

[38] Intel Corporation. "Intel Xeon Processor E5-2600 Product Family Uncore Performance Monitoring Guide". In: (). URL: https://www.intel.com/content/dam/www/public/us/en/documents/design-guides/xeon-e5-2600-uncore-guide.pdf.

[39] Intel Corporation. *Intel Xeon Processor E5-2658 v3*. URL: http://ark.intel.com/es/products/81905/Intel-Xeon-Processor-E5-2658-v3-30M-Cache-2_20-GHz.

[40] Intel Corporation. *Processor Counter Monitor*. GitHub. URL: https://github.com/opcm/pcm.git.

[41] Intel Corporation. *User space software for Intel Resource Director Technology*. GitHub. URL: https://github.com/01org/intel-cmt-cat.

[42] Ravi Iyer. "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms". In: *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*. Malo, France, 2004, pp. 257–266.

[43] Ravishankar R. Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa R. Hsu, and Steve K. Reinhardt. "QoS Policies and Architecture for Cache/Memory in CMP Platforms". In: *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. San Diego, CA, USA, 2007, pp. 25–36.

[44] Bruce Jacob and David Wang. *Principles and practices of interconnection networks*. Morgan Kaufmann, 2007. ISBN: 9780122007514.

[45] *JEDEC Website*. URL: http://www.jedec.org/.

[46] Harshad Kasture and Daniel Sanchez. "Ubik: Efficient Cache Sharing with Strict QoS for Latency-critical Workloads". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, UT, USA, 2014, pp. 729–742.

[47] P.J. Kaufman. *The new commodity trading systems and methods*. Wiley, 1987, pp. 58–64. ISBN: 9780471878797.

[48] Samira Manabi Khan, Alaa R. Alameldeen, Chris Wilkerson, Onur Mutlu, and Daniel A. Jiménez. “Improving Cache Performance Using Read-Write Partitioning”. In: *Proceedings of the 20th International Symposium on High Performance Computer Architecture (HPCA)*. Orlando, FL, USA, 2014, pp. 452–463.

[49] Seongbeom Kim, Dhruba Chandra, and D. Solihin. “Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture”. In: *Proceedings of the 13rd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Antibes Juan-les-Pins, France, 2004, pp. 111–122.

[50] Junghoon Lee, Minjeong Shin, Hanjoon Kim, John Kim, and Jaehyuk Huh. “Exploiting Mutual Awareness between Prefetchers and On-chip Networks in Multi-cores”. In: *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Galveston Island, TX, USA, 2011, pp. 177–178.

[51] M. M. Lee, J. Kim, D. Abts, M. Marty, and J. W. Lee. “Probabilistic Distance-Based Arbitration: Providing Equality of Service for Many-Core CMPs”. In: *Proceedings of the 43rd International Symposium on Microarchitecture (MICRO)*. Atlanta, GA, USA, 2010, pp. 509–519.

[52] Minghua Li, Guancheng Chen, Qijun Wang, Yonghua Lin, Peter Hofstee, Per Stenstrom, and Dian Zhou. “PATer: A Hardware Prefetching Automatic Tuner on IBM POWER8 Processor”. In: *IEEE Computer Architecture Letters* 15.1 (2016), pp. 37–40. ISSN: 0360-0300.

[53] Fang Liu, Xiaowei Jiang, and Y. Solihin. “Understanding how off-chip memory bandwidth partitioning in Chip Multiprocessors affects system performance”. In: *Proceedings of the 16th International Conference on High-Performance Computer Architecture (HPCA)*. Bangalore, India, 2010, pp. 1–12.

[54] Fang Liu and Yan Solihin. “Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip-multiprocessors”. In: *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. San Jose, CA, USA, 2011, pp. 37–48.

[55] Lei Liu, Yong Li, Chen Ding, Hao Yang, and Chengyong Wu. “Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random?” In: *IEEE Transactions on Computers* 65.6 (2016), pp. 1921–1935. ISSN: 0018-9340.

[56] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. “Heracles: Improving Resource Efficiency at Scale”. In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, USA, 2015, pp. 450–462.

[57] R. Manikantan and R. Govindarajan. “Performance Oriented Prefetching Enhancements Using Commit Stalls”. In: *Journal of Instruction-Level Parallelism* 13 (2011). ISSN: 1942-9525.

[58] R Manikantan, Kaushik Rajan, and R Govindarajan. “Probabilistic Shared Cache Management (PriSM)”. In: *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*. Portland, OR, USA, 2012, pp. 428–439.

[59] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. “Evaluation Techniques for Storage Hierarchies”. In: *IBM Systems Journal* 9.2 (1970), pp. 78–117. ISSN: 0018-8670.

[60] Pierre Michaud. “Best-offset hardware prefetching”. In: *Proceedings of the 22nd International Symposium on High Performance Computer Architecture (HPCA)*. Barcelona, Spain, 2016, pp. 469–480.

[61] O. Mutlu and T. Moscibroda. “Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems”. In: *Proceedings of the 35th International Symposium on Computer Architecture (ISCA)*. Beijing, China, 2008, pp. 63–74.

[62] Onur Mutlu and Thomas Moscibroda. “Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors”. In: *Proceedings of the 40th International Symposium on Microarchitecture (MICRO)*. Chicago, IL, USA, 2007, pp. 146–160.

[63] *NAS Parallel Benchmark Suite*. URL: `http://www.nas.nasa.gov/publications/npb.html`.

[64] K.J. Nesbit, A.S. Dhodapkar, and J.E. Smith. "AC/DC: an adaptive data cache prefetcher". In: *Proceedings of the 12nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Antibes Juan-les-Pins, France, 2004, pp. 135–145.

[65] Kyle J. Nesbit and James E. Smith. "Data Cache Prefetching Using a Global History Buffer". In: *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA)*. Madrid, Spain, 2004, pp. 96–. ISBN: 0-7695-2053-7.

[66] J. Owen and M. Steinman. "Northbridge Architecture of AMD's Griffin Microprocessor Family". In: *IEEE Micro* 28.2 (2008), pp. 10–18. ISSN: 0272-1732.

[67] S. Palacharla and R. E. Kessler. "Evaluating Stream Buffers As a Secondary Cache Replacement". In: *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*. Chicago, IL, USA, 1994, pp. 24–33.

[68] Dongkook Park, Chrysostomos Nicopoulos, Jongman Kim, Narayanan Vijaykrishnan, and Chita R. Das. "Exploring Fault-Tolerant Network-on-Chip Architectures". In: *Proceeding of the International Conference on Dependable Systems and Networks (DSN)*. Philadelphia, PA, USA, 2006, pp. 93–104.

[69] Seth Pugsley, Zeshan Chishti, Chris Wilkerson, Troy Chuang, Robert Scott, Aamer Jaleel, Shih-Lien Lu, Kingsum Chow, and Rajeev Balasubramonian. "Sandbox Prefetching: Safe, Run-Time Evaluation of Aggressive Prefetchers". In: *Proceedings of the 20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Orlando, FL, USA, 2014.

[70] Moinuddin K. Qureshi and Yale N. Patt. "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches". In: *Proceedings of the 39th Annual International Symposium on Microarchitecture (MICRO)*. Orlando, FL, USA, 2006, pp. 423–432.

[71] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. "A Resource Allocation Model for QoS Management". In: *Proceedings of the 18th*

Real-Time Systems Symposium (RTSS). San Francisco, CA, USA, 1997, pp. 298–.

[72] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. "DRAMSim2: A Cycle Accurate Memory System Simulator". In: *Computer Architecture Letters* 10.1 (2011), pp. 16–19. ISSN: 1556-6056.

[73] Peter J. Rousseeuw. "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis". In: *Journal of Computational and Applied Mathematics* 20 (1987), pp. 53–65. ISSN: 0377-0427.

[74] Daniel Sanchez and Christos Kozyrakis. "The ZCache: Decoupling Ways and Associativity". In: *Proceedings of the 43rd Annual International Symposium on Microarchitecture (MICRO)*. Atlanta, GA, USA, 2010, pp. 187–198.

[75] Daniel Sanchez and Christos Kozyrakis. "Vantage: Scalable and Efficient Fine-grain Cache Partitioning". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*. San Jose, CA, USA, 2011, pp. 57–68.

[76] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores". In: *Proceedings of the 24th International Symposium on High Performance Computer Architecture (HPCA)*. Vienna, Austria, 2018, pp. 104–117.

[77] Alberto Scolari, Davide Basilio Bartolini, and Marco Domenico Santambrogio. "A Software Cache Partitioning System for Hash-Based Caches". In: *ACM Transactions on Architecture and Code Optimization (TACO)* 13.4 (2016), 57:1–57:24. ISSN: 1544-3566.

[78] Vicent Selfa, Julio Sahuquillo, Salvador Petit, and María Engracia Gómez. "A Hardware Approach to Fairly Balance the Inter-Thread Interference in Shared Caches". In: *IEEE Transactions on Parallel and Distributed Systems* 28.11 (2017), pp. 3021–3032. ISSN: 1045-9219.

[79] Akbar Sharifi, Emre Kultursay, Mahmut T. Kandemir, and Chita R. Das. "Addressing End-to-End Memory Access Latency in NoC-Based Multicores". In: *Proceedings of the 45th Annual International Sympo-*

sium on Microarchitecture (MICRO). Vancouver, Canada, 2012, pp. 294–304.

[80] Akbar Sharifi, Shekhar Srikantaiah, Mahmut T. Kandemir, and Mary Jane Irwin. "Courteous Cache Sharing: Being Nice to Others in Capacity Management". In: *Proceedings of the 49th Annual Design Automation Conference (DAC)*. San Francisco, CA, USA, 2012, pp. 678–687.

[81] Timothy Sherwood, Brad Calder, and Joel Emer. "Reducing Cache Misses Using Hardware and Software Page Placement". In: *Proceedings of the 13rd International Conference on Supercomputing (ICS)*. Rhodes, Greece, 1999, pp. 155–164.

[82] B. Sinharoy, J. A. Van Norstrand, R. J. Eickemeyer, H. Q. Le, J. Leenstra, D. Q. Nguyen, B. Konigsburg, K. Ward, M. D. Brown, J. E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J. W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K. M. Fernsler. "IBM POWER8 processor core microarchitecture". In: *IBM Journal of Research and Developement* 59.1 (2015), 2:1–2:21. ISSN: 0018-8646.

[83] Alan Jay Smith. "Cache Memories". In: *ACM Computer Surveys* 14.3 (1982), pp. 473–530. ISSN: 0360-0300.

[84] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. "Knights Landing: Second-Generation Intel Xeon Phi Product". In: *IEEE Micro* 36.2 (2016), pp. 34–46. ISSN: 0272-1732.

[85] S. Srinath, O. Mutlu, H. Kim, and Y. N. Patt. "Feedback Directed Prefetching: Improving the Performance and Bandwidth-Efficiency of Hardware Prefetchers". In: *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*. Scottsdale, AZ, USA, 2007, pp. 63–74.

[86] *Standard Performance Evaluation Corporation*. URL: `http://www.spec.org`.

[87] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. "The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-application Interference at Shared Caches and Main Memory". In: *Proceedings of the 48th Annual International Symposium on Microarchitecture (MICRO)*. Waikiki, HI, USA, 2015, pp. 62–75.

[88] G. E. Suh, L. Rudolph, and S. Devadas. "Dynamic Partitioning of Shared Cache Memory". In: *Journal of Supercomputing* 28.1 (2004), pp. 7–26. ISSN: 0920-8542.

[89] David Tam, Reza Azimi, Livio Soares, and Michael Stumm. "Managing shared L2 caches on multicore systems in software". In: *Proceedings of the Workshop on the Interaction between Operating Systems and Computer Architecture (WIOSCA)*. San Diego, CA, USA, 2007.

[90] Micron Technology. *Data sheet for the 4Gb DDR3 SDRAM device MT41 J512M8 - 64Meg x 8 x 8 banks*. URL: https://www.micron.com/~/media/documents/products/data-sheet/dram/ddr3/4gb_ddr3_sdram.pdf.

[91] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. "Multi2Sim: A Simulation Framework for CPU-GPU Computing". In: *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Minneapolis, MN, USA, 2012, pp. 335–344.

[92] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. "Fairness-aware Scheduling on single-ISA Heterogeneous Multi-cores". In: *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. Edinburgh, Scotland, UK, 2013, pp. 177–188.

[93] De-Li Wang, De-Yuan Gao, and Dang-Hui Wang. "Enhancing the performance and fairness of shared DRAM systems with sharing-aware scheduling (ICCET)". In: *Proceedings of the 2nd International Conference on Computer Engineering and Technology*. Vol. 6. Chengdu, China, 2010, pp. 709–713.

[94] Ruisheng Wang and Lizhong Chen. "Futility Scaling: High-Associativity Cache Partitioning". In: *Proceedings of the 47th Annual International Symposium on Microarchitecture (MICRO)*. Cambridge, United Kingdom, 2014, pp. 356–367.

[95] X. Wang, S. Chen, J. Setter, and J. F. Martínez. "SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support". In: *Proceedings of the International Symposium on High Performance Computer Architecture (HPCA)*. 2017, pp. 121–132.

[96] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. "SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection". In: *Proceedings of the 53rd Annual Design Automation Conference (DAC)*. Austin, TX, USA, 2016, 74:1–74:6.

[97] Wayne A Wong and Jean-Loup Baer. "The Impact of Timeliness for Hardware-based Prefetching from Main Memory". In: *Technical Report, Department of Computer Science and Engineering, University of Washington* (2002).

[98] Chenggang Wu, Jin Li, Di Xu, Pen-Chung Yew, Jianjun Li, and Zhenjiang Wang. "FPS: A Fair-Progress Process Scheduling Policy on Shared-Memory Multiprocessors". In: *IEEE Transactions on Parallel and Distributed Systems* 26.2 (2015), pp. 444–454. ISSN: 1045-9219.

[99] Di Xu, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Zhenjiang Wang. "Providing Fairness on Shared-Memory Multiprocessors Via Process Scheduling". In: *ACM SIGMETRICS Performance Evaluation Review* 40.1 (2012), pp. 295–306. ISSN: 0163-5999.

[100] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. "Mapping the Intel Last-Level Cache". In: *Cryptology ePrint Archive Report 2015/905* (2015). URL: `https://eprint.iacr.org/2015/905`.

[101] Y. Ye, R. West, Z. Cheng, and Y. Li. "COLORIS: A Dynamic Cache Partitioning System Using Page Coloring". In: *Proceedings of the 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*. Edmonton, Canada, 2014, pp. 381–392.

[102] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. "Towards Practical Page Coloring-based Multicore Cache Management". In: *Proceedings of the 4th European Conference on Computer Systems (EuroSys)*. Nuremberg, Germany, 2009, pp. 89–102.

[103] Haishan Zhu and Mattan Erez. "Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems". In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Atlanta, GA, USA, 2016, pp. 33–47.

[104] X. Zhuang and H. h. S. Lee. "Reducing Cache Pollution via Dynamic Data Prefetch Filtering". In: *IEEE Transactions on Computers* 56.1 (2007), pp. 18–31. ISSN: 0018-9340.