IMPROVING CACHE BEHAVIOR IN CMP ARCHITECTURES THROUGH CACHE PARTITIONING TECHNIQUES

Miquel Moretó Planas

Barcelona, 2009

A thesis submitted in fulfillment of
the requirements for the degree of
DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC
Doctor Europeus Mention

Department of Computer Architecture
Technical University of Catalonia

ACTA DE QUALIFICACIÓ DE LA TESI DOCTORAL

Reunit el tribunal integrat pels sota signants per jutjar la tesi doctoral:

Títol de la tesi: .....
Autor de la tesi: .....

Acorda atorgar la qualificació de:

- ☐ No apte
- ☐ Aprovat
- ☐ Notable
- ☐ Excel·lent
- ☐ Excel·lent Cum Laude

Barcelona, …… de/d'…..…….. de .....….

El President

.....
(nom i cognoms)

El Secretari

.....
(nom i cognoms)

El vocal

.....
(nom i cognoms)

El vocal

.....
(nom i cognoms)

El vocal

.....
(nom i cognoms)

IMPROVING CACHE BEHAVIOR IN CMP ARCHITECTURES THROUGH CACHE PARTITIONING TECHNIQUES

Miquel Moretó Planas

Barcelona, 2009

ADVISORS: **Francisco J. Cazorla Almeida**
Barcelona Supercomputing Center
Mateo Valero Cortés
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center

COLLABORATORS: **Alex Ramirez Bellido**
Universitat Politècnica de Catalunya
Barcelona Supercomputing Center
Rizos Sakellariou
University of Manchester

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

DOCTOR OF PHILOSOPHY / DOCTOR PER LA UPC
Doctor Europeus Mention

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya

A la meva família

Abstract

In the last few years, chip multiprocessors (CMP) have become widely used in academia and industry in order to increase aggregate system performance. CMPs reduce design costs and average power consumption by promoting design re-use and simpler processor cores, while at the same time increasing hardware resource utilization.

However, new architectures also have to face the challenge of making better use of shared resources. A key shared resource in CMP architectures is the cache hierarchy, as it is one of the resources that has the most impact on the final performance of the application. In CMP architectures, last levels of cache (LLC) shared on-chip have become popular, as they allow increased utilization of the cache (and consequently, aggregate performance) and, additionally, they simplify the coherence protocol.

Some applications present low re-use of their data and pollute caches with data streams (such as multimedia, communications or streaming applications), or have many compulsory misses that cannot be solved by assigning more cache space to the application. Traditional eviction policies, such as Least Recently Used (LRU), pseudo LRU, or random, are demand-driven, that is, they tend to give more space to the application that has more accesses and misses to the cache hierarchy. As a result, some threads suffer a severe degradation in performance when running together with threads of those types in a CMP architecture with a shared cache.

In this thesis, we analyze in detail cache-sharing effects in a CMP architecture, and we argue for the use of enhanced mechanisms that are aware of these effects. We propose architectural changes in order to allow CMP architectures to better manage shared caches. In addition, we describe our efforts to significantly improve the performance of individual applications running on a CMP, providing quality of service (QoS) to applications in these architectures. We also propose new mechanisms to balance parallel applications in CMP architectures and describe the challenges that future CMP architectures with hundreds of cores will have to face in the near future.

Acknowledgments

Aquest projecte suposa el final d'un llarg camí que vaig començar ara fa una mica més de quatre anys. Durant aquest temps, molta gent ha estat al meu costat ajudant-me a formar-me i a ser millor persona. Família, amics, companys i professors, a tots ells els hi agraeixo aquest esforç.

En primer lloc, vull destacar per sobre de tot el paper de la meva família. Els meus pares, la meva germana i la Isabel sempre han estat en tot moment un exemple d'amor i sacrifici envers la meva persona. Sense ells, això no hagués estat possible.

Molt especialment, volia mencionar als meus directors de tesi, Francisco J. Cazorla i Mateo Valero, que han tingut un paper crucial en la seva elaboració. Els seus consells i ajuda sempre que els necessitava m'han demostrat la seva gran vàlua com a persones i científics. També vull donar les gràcies a dues persones que han participat molt activament en la realització d'aquesta tesi, n'Alex Ramirez i en Rizos Sakellariou. Finalment, també voldria agrair l'ajuda de Jim Smith i Kyle Nesbit de la Universitat de Wisconsin per ajudar-me en l'elaboració del darrer capítol d'aquesta tesi.

M'agradaria donar les gràcies en especial als meus companys de llargs dinars i cafès: Oriol, Xevi, Alex i Felipe. Moltes discussions (sempre amigables) durant aquests anys que ens han ajudat a desconnectar i a fer-nos més duguedores les incerteses del doctorat.

També voldria destacar a la gent del departament d'Arquitectura de Computadors de la Universitat Politècnica de Catalunya i del Barcelona Supercomputing Center. Intentaré fer una llista amb tots ells, però segur que me n'oblido d'algú: Tana, Indu, Manoj, Shrikanth, Govind, Carlos V., Isaac G., Carmelo A., Montse F., Marc C., Beatriz, Kamil, Carlos B., Jose Carlos R., Eduard Q.,... No puc oblidar la gent d'administració del departament, la universitat i el BSC, ja que sense ells no hauria pogut sobreviure als complexos procediments per acabar sent doctor.

También quiero agradecer a Ramón Beivide, Carmen Martínez y Enrique Vallejo, del grupo de Arquitectura y Tecnología de Computadores de la Universidad de Cantabria,

por ayudarme en muchos momentos del doctorado y por distraerme con otros temas de investigación cuando tenía un rato libre.

I should not forget to thank the people at IBM T.J. Watson who, together with my advisors, helped and guided me during a very decisive part of this research. Special, but not exclusive, thanks go to the team with whom I worked most closely: Alper Buyuktosunoglu, Roberto Gioiosa, Chen-Yong Cher, Pradip Bose and Jaime Moreno. I also want to thank all my other colleagues in the research center, specially Shuguang, Rafael, M. Angeles, Salman and Daniele. Finally, I should also mention that I have made use of a picture of the IBM Power6 chip as the cover art for the print edition of this thesis.

I also want to thank the people at the University of Edinburgh, Mike O'Boyle, Tim Jones, Salman, Chronis, Georgios, Sofia and Nikolas, for an excellent summer which provided me with a broader view of this field, for their technical expertise, and for their friendship.

I finalment, els agraïments oficials: This work has been partially supported by the Ministry of Education and Science of Spain under contracts TIN2004-07739, TIN2007-60625 and grant AP-2005-3318, and by the Ministry of Universities, Research and Information Society (DURSI) of the Catalan Government and the European Social Fund (grant 2006-FI-00838). The author would also like to thank the support of the HiPEAC European Network of Excellence and the SARC European Project (Contract number 27648).

Contents

Chapter 1
Introduction

The evolution of microprocessor design in the last few decades has changed significantly, moving from simple in-order single core architectures to superscalar and vector architectures in order to better extract the maximum available instruction level parallelism (ILP). Executing several instructions from the same thread in parallel allows for a significant improvement in the performance of applications. However, there is only a limited amount of parallelism available in each thread, because of data and control dependences. Furthermore, due to power and chip latencies constraints, designing a high performance, single, monolithic processor has become very complex. These limitations have given rise to the use of thread level parallelism (TLP) as a common strategy for improving processor performance. Multithreaded processors allow for the execution of different threads at the same time, sharing some hardware resources. There are several flavors of multithreaded processors that exploit the TLP, such as chip multiprocessors (CMP) [45], coarse grain multithreading (CGMT) [2, 107], fine grain multithreading (FGMT) [44, 102], simultaneous multithreading (SMT) [98, 110], and combinations of them [51, 63, 101, 111].

To improve cost and power efficiency, the computer industry has adopted multithreaded processors. In particular, CMP architectures have become the most common design decision (combined sometimes with multithreaded cores). Firstly, CMPs reduce design costs and average power consumption by promoting design re-use and simpler processor cores. For example, it is less complex to design a chip with many small, simple cores than a chip with fewer, larger, monolithic cores. Furthermore, simpler cores have less power-hungry hardware structures. Secondly, CMPs reduce costs by improving hardware resource utilization. On a multicore chip, co-scheduled threads can share costly microarchitecture resources that would otherwise be underutilized (for example, off-chip bandwidth and power resources). Higher resource utilization improves aggregate performance and enables lower cost design alternatives (for example, smaller design area or less exotic battery

technology).

One of the hardware resources that impacts most on the performance of an application is the cache hierarchy. Caches store data recently used by the applications in order to take advantage of the temporal and spatial locality of the applications. Caches provide fast access to data, improving the performance of applications. Caches with low latencies have to be small, which prompts the design of a cache hierarchy organized in several levels of cache.

In CMPs, the cache hierarchy is normally organized in a first level (L1) of instruction and data caches private to each core. The last level cache (LLC) is normally shared among different cores in the processor (either L2, L3 or both). Shared caches increase resource utilization and system performance. Large caches improve performance and efficiency by increasing the probability that each application can access data from a closer level of the cache hierarchy. Furthermore, they allow an application to make use of the entire cache if needed.

A second advantage of having a shared cache in a CMP design has to do with the cache coherency. In parallel applications, different threads share the same data and keep a local copy of this data in their cache. With multiple processors, it is possible for one processor to change the data, leaving another processor's cache with outdated data. The cache coherency protocol monitors any changes made to data and ensures that all processor caches have the most recent data. When the parallel application executes on the same physical chip, the cache coherency circuitry can operate at the speed of on-chip communications, rather than having to use the much slower communication between chips, as is required with processors on separate chips [3]. These coherence protocols are simpler to design with a unified and shared level of cache on-chip.

Due to the advantages that multicore architectures offer, chip vendors use CMP architectures in current high performance, network, real-time and embedded systems. Several of these commercial processors have a level of the cache hierarchy shared by different cores. For example, the IBM Power5 [101] has a 10-way 1.875MB L2 cache shared by two cores, each one two-way SMT. The Sun UltraSPARC T2 has a 16-way 4MB L2 cache shared by 8 cores, each one up to 8-way FGMT [111]. Other processors like the Intel Core 2 family also share a level of the cache hierarchy among cores (the L2 cache) [34], with up to a 12MB 24-way L2 cache [51]. In contrast, the AMD K10 family has a private L2 cache per core and a shared L3 cache [6], with up to a 6MB 64-way L3 cache [5]. The same cache hierarchy design has been chosen for the Intel Xeon 7100 processor, with a

shared on-chip 16-way 16MB L3 cache [25].

As the long-term trend of increasing integration continues, the number of cores per chip is also predicted to increase with each successive technology generation. Some significant studies have shown that processors with hundreds of cores per chip will appear in the market in the next few years [50]. The *manycore era* has already begun [9].

Although this era provides many opportunities, it also presents many challenges. In particular, higher hardware resource sharing among concurrently executing threads can cause individual thread performance to become unpredictable and can lead to violations of the individual applications' performance requirements [23, 83]. Current resource management mechanisms and policies are no longer adequate for future multicore systems, as they are not aware of these resource-sharing effects.

In general, applications make very different uses of the cache hierarchy, depending on their data re-use and access patterns. Some multimedia, communications or streaming applications pollute caches with data streams, or have many compulsory misses that cannot be solved by assigning more cache space to them. When running multiple applications in a CMP architecture with a shared cache, undesired situations can occur where a subset of the applications monopolizes the shared cache, degrading the performance of the other ones. And even worse, the operating system (OS) has no way to enforce a Quality of Service (QoS) to applications.

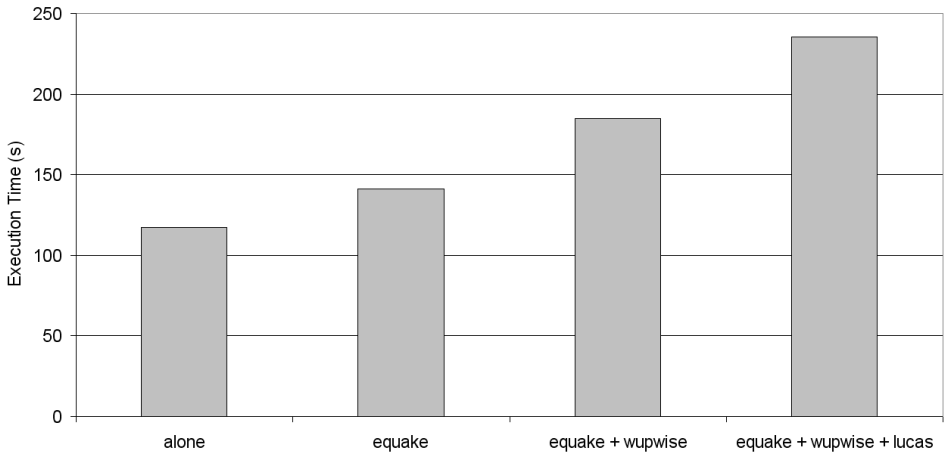


Figure 1.1: Total execution time of `swim` in different workloads running on an Intel Xeon Quad-Core processor with a shared L2 cache

To illustrate this phenomenon, Figure 1.1 shows the total execution time of the `swim` SPEC CPU 2000 benchmark [105] when it is executed in different workloads. For this

experiment, we use an Intel Xeon 2.5 GHz Quad-Core processor[1], which has four cores in the chip. The OS running in the machine is Linux 2.6.18. During the experiments, we move all the OS activity to the first core, leaving the other cores as isolated as possible from OS activity.

When `swim` runs alone in the system, it completes its execution in 117 seconds. Next, we execute `swim` with several workloads of two, three and four benchmarks (as shown by the x-axis in Figure 1.1). In these experiments, each benchmark is assigned to a different core. Consequently, the CPUs are not time shared between different applications since the number of running processes is equal or less than the number of virtual CPUs (cores) in the system. Figure 1.1 shows a variation of up to 2x in the execution time of `swim`, depending on the workload in which it runs. This means that the performance of a process running on a CMP with a shared cache may be different, depending on the cache necessities of the other processes running on the same chip at the same time. From the user point of view this is an undesirable situation, as the same application with the same input set is executed in a different amount of time depending on the processes it is co-scheduled with.

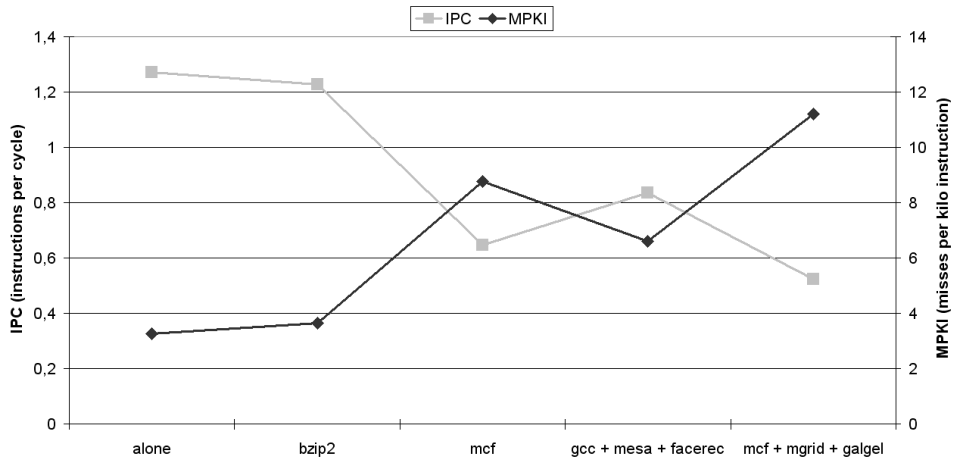


Figure 1.2: Performance and MPKI variability of `ammp` in different workloads running in a 4-core CMP environment with a shared L2 cache using LRU as eviction policy

Next, we performed a similar experiment using a cycle-accurate microarchitecture simulator (a complete description of the simulation environment is given in Chapter 2). This experiment models a four core architecture with private instruction and data caches and a shared L2 cache. We also use SPEC CPU 2000 benchmarks in this experiment.

[1]Though we believe the general trends drawn from Figure 1.1 apply to all current CMPs with a shared cache

Figure 1.2 shows the IPC (instructions per cycle) of `ammp` when mixed with different workloads in a CMP architecture using Least Recently Used (LRU) as eviction policy. The processor configuration remains constant in this experiments. We vary the number of active cores from one to four and label the x-axis with the applications that form the workload (together with `ammp`). Depending on the co-scheduled applications, the performance of `ammp` varies significantly (from 1.27 to 0.52 instructions per cycle) due to negative interferences with other applications. Note that these interferences do not only depend on the number of co-scheduled applications, as `mcf` degrades the performance of `ammp` more than `gcc`, `mesa` and `facerec` together. Figure 1.2 also shows the number of misses per thousand (kilo) instructions (MPKI). The number of misses of `ammp` increases when more cache hungry applications are competing for the shared cache, which matches the behavior of `ammp`'s IPC. Consequently, if we do not control the usage of the shared caches adequately, we can obtain noticeable performance variability.

When no direct control over shared resources is exercised (the last level cache in this case), it is possible that a particular thread allocates most of the shared resources, causing a degradation in other threads performance. As a consequence, high resource sharing and resource utilization can cause systems to become unstable and violate individual applications' requirements. If we want to provide a Quality of Service (QoS) to applications, we need to enhance the control over shared resources and enrich the collaboration between the OS and the architecture.

1.1 Thesis Objectives and Contributions

In this section we provide a brief description of the topic we deal with in this thesis. We present the problems we are trying to solve, the approach we take to solve them, and the contributions of our work.

The main goal of this thesis is to propose software and hardware mechanisms to improve cache sharing in CMP architectures. Given the importance of shared caches in CMP processors, we target an enhanced utilization of this shared resource by all the executing applications.

In order to reach this goal, we start by better understanding the cache sharing effects on a CMP architecture. Acquiring this insight into the architecture is fundamental to the task of designing better processors in the future. Furthermore, this knowledge is a key element in developing models that allow for accurate prediction of the performance of an

application in CMP architectures. These models should be very fast and useful, allowing designers to quickly explore the design space of a processor, to detect interesting design points and, consequently, to reduce simulation time.

Next, we propose hardware and software solutions to improve aggregate system performance in terms of metrics such us throughput or fairness. We also aim to provide a quality of service to applications running in a CMP architecture. The concept of QoS changes depending on the target scenario, which motivates the use of a flexible framework that can adapt to these different scenarios. In the case of parallel applications, we aim to reduce execution time by load balancing the different threads of the application.

Finally, we envision the resource allocation in the future manycore era with thousands of cores per chip. With the acquired experience in this thesis, we analyze the major challenges that future CMP architectures will have to face when dealing with shared resources, and propose a general framework to manage these resources.

1.1.1 Weighted Dynamic Cache Partitioning Algorithm

Previous work has suggested that the performance of shared caches can be improved by using *static* and *dynamic cache partitioning algorithms*. These mechanisms monitor the shared cache accesses and decide a partition in order to maximize throughput [29, 90, 99, 108] or fairness [58]. Basically, the dynamic proposals split the execution of workloads into intervals of fixed duration. Based on the data collected in the current and previous intervals, they predict a metric related to performance (for example, number of misses per application, miss rates, data re-use, etc.) for each possible cache partition at a *way granularity*. Then, they use the cache partition that optimizes this metric for the next interval (for example, the partition that minimizes the total number of cache misses). This process is repeated until the workload finishes executing.

These dynamic cache partitioning (DCP) algorithms mainly work with the number of misses caused by each thread, and they treat all misses equally. However, cache misses in out-of-order architectures cause different impacts on performance, depending on their distribution [55]. Clustered misses share their miss penalty as they can be served in parallel, while isolated misses have a greater impact on performance since the memory latency is not shared with other misses. We take this fact into account and propose a new DCP algorithm that considers misses differently depending on how clustered they are.

1.1.2 Obtention of IPC Estimations

Driving cache partition decisions with indirect indicators of performance such as misses, weighted misses or data re-use may lead to suboptimal cache partitions. For that reason, we propose using direct estimations of performance to decide between different configurations of the cache. To that end, we introduce a dedicated hardware in the architecture, OPACU, which monitors the cache accesses, their clustering level, and some statistics of the pipeline in order to obtain accurate predictions of the performance of an application at run-time, when running with different cache assignments.

1.1.3 Use of IPC Estimations to Guide Resource Assignments

One of the main contributions of this thesis is the use of direct estimations of performance when dynamically assigning cache resources to each application. This novel approach has the following advantages:

- Predicting the performance of applications for all possible configurations offers the possibility of using different metrics when deciding the optimal partition. Thus, we can use our methodology to maximize throughput, fairness, or to ensure a Quality of Service (QoS). The use of direct estimations of performance leads to a more flexible dynamic resource management than previous proposals.

- The use of direct estimations of performance makes it possible to build a run-time mechanism to dynamically partition shared resources with more accurate decisions. Our experiments consistently obtain performance benefits over previous proposals when optimizing different metrics.

- Using performance predictions, we allow the OS to run jobs at a certain percentage of their maximum speed, regardless of the workload in which these jobs are executed. With this novel approach, we ensure predictable performance for critical applications in CMP scenarios.

- Finally, performance predictions can be used to balance parallel application through cache allocation in a CMP scenario. The mechanism detects applications sensitive to cache allocation and reduces imbalance by assigning more cache space to the slowest threads. This mechanism helps to reduce the long and expensive optimization time of large-scale parallel applications.

1.1.4 Resource Management in Future Manycore Architectures

Current resource management mechanisms and policies are inadequate for future multicore systems, since the OS is not aware of the interaction between different concurrently running threads. These inadequacies can be met by an enriched hardware/software interface, which would allow software policies to explicitly manage microarchitecture resources. This new interface would allow improved system performance and would provide QoS to applications. We review some basic system design principles that are essential to build well-structured scalable systems. Next, we present our vision of future multicore system architecture and discuss how the envisioned system architecture can efficiently satisfy the diverse demands of future manycore systems.

1.2 Thesis Structure

Figure 1.3 shows an overview of this thesis, from the problem we observed, to the different solutions we proposed, leading to new problems or observations, which in turn opened up the possibility for new proposals.

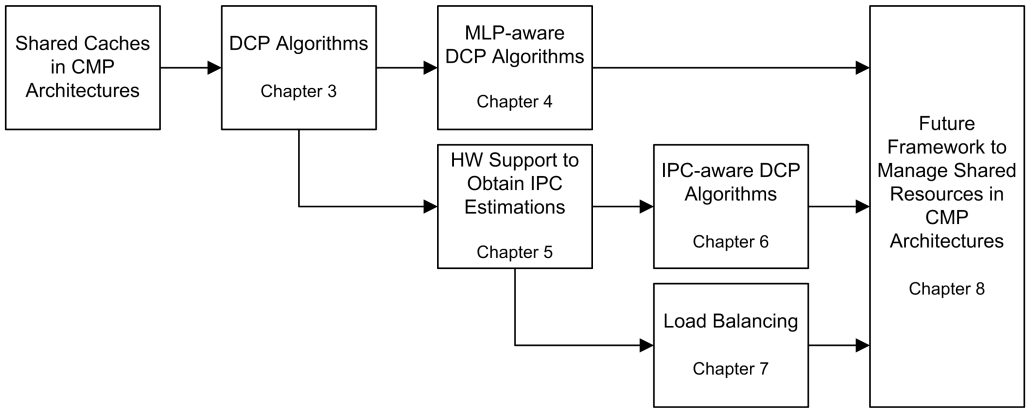


Figure 1.3: Thesis structure (DCP stands for Dynamic Cache Partitioning)

The structure of this dissertation is as follows:

- Chapter 1 presents the research field and problem matters, along with the objectives of this research. It also presents the contributions of this research and the structure of the thesis.
- Chapter 2 presents our simulation environment. The reference platform for this work is presented, as well as the benchmarks and tools used in this thesis.

- Chapter 3 describes the background to this thesis and previous cache partitioning schemes.
- Chapter 4 describes a new dynamic cache partitioning algorithm which considers the memory-level parallelism (MLP) of each access to the shared cache when deciding new cache partitions.
- Chapter 5 introduces a dedicated hardware that predicts the performance of an application when the cache space assigned to it changes. The accuracy of this mechanism is evaluated with extensive simulations.
- Chapter 6 shows that IPC predictions are the adequate metric to guide cache partition decisions. With these estimations, different performance metrics can be optimized, such as throughput, fairness or individual QoS.
- Chapter 7 evaluates the utility of using cache partitioning techniques to balance parallel applications.
- Chapter 8 introduces a generic framework for future manycore architectures. This framework has to deal with the challenge of sharing microarchitecture resources among many cores with a continuously increasing performance.
- Chapter 9 concludes this dissertation by commenting on the most important contributions of this thesis, providing a brief summary of future work, and listing the main publications related to this thesis.

Chapter 2

Platform, Tools and Benchmarks

2.1 Introduction

Computer system design is a time-consuming process and simulation is an essential tool to drive the design of new systems. Simulation tools have been widely used to verify, analyze and improve computer systems. Simulation is used at different levels of detail, from circuit to system level, depending on the particular target system that we wish to study. The trade-off between simulation speed and accuracy is always present in these studies. Ideally, we would like to have very accurate results with very low simulation time.

Functional simulators emulate the behavior of the target system, including the operating system (OS) and the different devices of the system (memory, network interfaces, disks, etc.). These simulators allow designers to verify the correctness of systems and to develop software before the system has been built, but the real performance of the system cannot be estimated with them. Some examples are SimOS [94], QEMU [16] or SimNow [15].

Specialized simulation aims to discover the behavior in isolation of a particular part of the processor, such as the branch predictor or the cache. *Microarchitecture simulators* model in detail the architecture of the processor and can estimate the performance of an application with different processor configurations. Simplescalar [10], SMTSim [110] and Turandot [79] are examples of this kind of simulators. However, these simulators normally do not model the interaction between the architecture and the OS and other system devices.

Full system simulators include the features of functional and microarchitecture simulators at the cost of simulation time. Some examples are Simics [69] and COTSon [7].

Simulating a single processor with these simulators is time-consuming, which makes it unaffordable to evaluate systems where thousands of processors are involved. To solve this problem, sampling techniques and extremely simple processor models have to be used [7].

A different approach consists of using *analytical models* to predict the performance of a processor, or of part of it. Analytical models are fast and, more importantly, give an insight into what is really happening in the processor. Some interesting approaches have been tried [55], but, until now, there is no complete solution.

Given that in this research we evaluate the performance of a CMP architecture with different workloads and configurations, the most appropriate decision is to choose a cycle-level microarchitecture simulator. There are different flavors of microarchitecture simulators: execution and trace-driven simulators. The execution-driven approach allows higher simulation accuracy, since, for example, the simulator fetches and simulates instructions from the wrong path after a branch has been mispredicted by the timing simulation. In contrast, trace-driven simulators simulate a trace recorded in a previous execution of the application. This approach leads to more efficient simulators with less accuracy than the execution-driven approach.

In our experiments we decided to use a *trace-driven* simulator. In order to benefit from the trace-driven simulator's reduced computational cost, without severely compromising the accuracy of the results obtained, the simulation tool was adapted accordingly. Thus, the simulator allows us to simulate the impact of wrong path instructions, using a separate basic block dictionary that contains the information of all static instructions.

2.2 MPSim Simulator

To evaluate the performance of the different mechanisms shown in this thesis, we use a trace-driven simulator derived from SMTSim [110] that supports CMP and SMT architectures. The simulator consists of our own trace driven front-end and an improved version of SMTSim's back-end. This simulator has been developed at UPC [1] and is called *MPSim* (Multiple Purpose Simulator).

In our baseline configuration, shown in Figure 2.1, the instruction fetch policy determines which of the available threads instructions are fetched from. Next, instructions are decoded and renamed in order to track data dependences. When an instruction is renamed, it is allocated an entry in the issue queues until all its operands are ready. Each

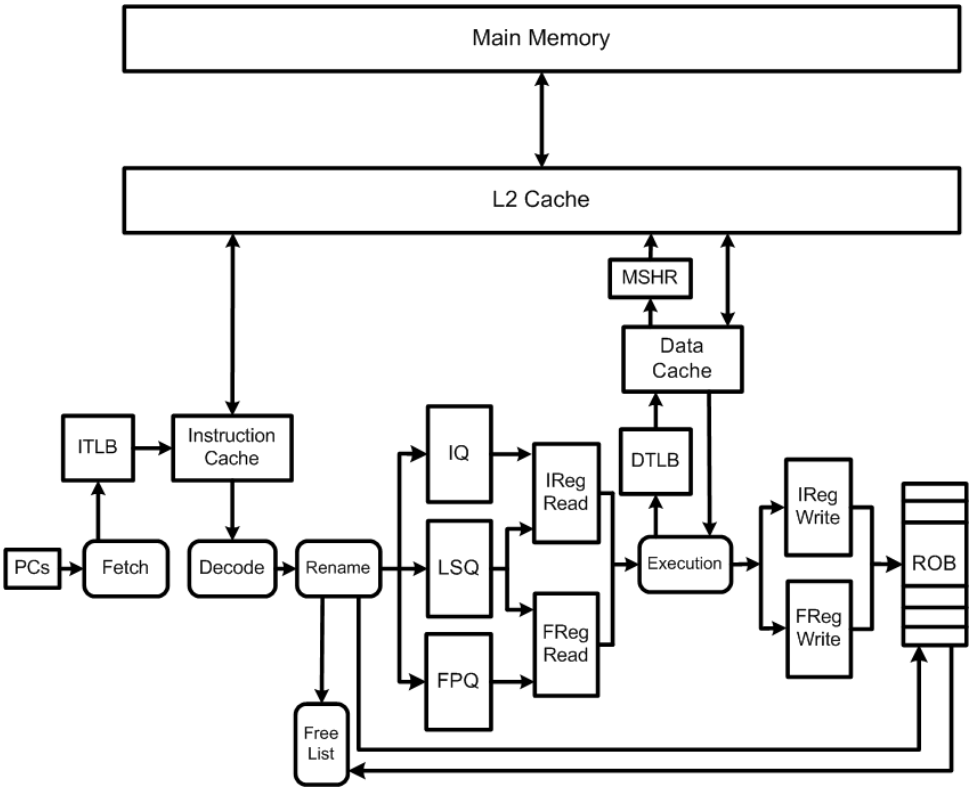


Figure 2.1: Blocks diagram of our baseline architecture

instruction also allocates one entry in the reorder buffer (ROB) and a physical register, if required. ROB entries are assigned in program order. When an instruction has all its operands ready, it is issued: it reads its operands, executes, writes its results, and finally commits. MPSim allows us to execute wrong path instructions by using a separate basic block dictionary that contains all static instructions.

The data and the instruction caches are accessed with physical addresses. The data cache uses write back as write hit policy and write allocate as write miss policy. Caches are tagged with the identifier of threads so that threads do not share data and/or instructions.

Figure 2.2: Pipeline stages in our baseline architecture

The pipeline of our baseline architecture is composed of nine stages as shown in Figure 2.2. In our experiments, the decode stage takes up to four cycles and, in this way, the

final number of stages in our pipeline is twelve.

The default processor setup we used is summarized in Table 2.1. In a CMP configuration, we have several copies of the default processor setup sharing the L2 cache, which is the shared last level cache (LLC) on-chip.

Table 2.1: MPSim baseline processor configuration

Processor Configuration	
Pipeline depth	12 stages
Fetch/Issue/Commit Width	8
Queues Entries	32 int, 32 fp, 32 ld/st
Execution Units	6 int, 3 fp, 4 ld/st
Physical Registers	256
(shared) ROB size	256 entries
Branch Prediction Configuration	
Branch Predictor	16K entries gshare
Branch Target Buffer	256-entry, 4-way associative
RAS	256 entries
Memory Configuration	
Icache Dcache	16 Kbytes, 4-way, 8-bank, 64-byte lines, 1 cycle access
L2 cache	1 Mbyte, 16-way, 16-bank, 64-byte lines, 15 cycle access
Main memory latency	300 cycles
TLB miss penalty	300 cycles

2.3 Benchmarks

In the experiments performed during this research, we used the SPEC CPU 2000 benchmark suite [105] to evaluate our proposals[1]. This benchmark suite is released by the Standard Performance Evaluation Corp. (SPEC), and is a worldwide standard for measuring and comparing computer performance across different hardware platforms.

SPEC CPU 2000 comprises two suites of benchmarks: SPEC CPU INT 2000 for compute-intensive integer performance and SPEC CPU FP 2000 for compute-intensive floating point performance. SPEC CPU 2000 benchmarks are selected from existing applications, representing high performance computing applications that stress the architecture of the processor. Benchmark source codes run in different platforms so that performance comparisons can be made between different systems.

[1]By the year 2005, when this thesis started, the most referenced benchmarks in general-purpose computer architecture papers came from this benchmark suite. Due to the analysis of the applications involved, we did not migrate to the next release at the end of 2006.

Table 2.2: SPEC CPU INT 2000 benchmarks description and simulation starting point using the SimPoint methodology [100]

Benchmark	Description	Input	Language	Fast forward (Millions of instructions)
164.gzip	Data compression utility	graphic	C	68.100
175.vpr	FPGA circuit placement and routing	place	C	2.100
176.gcc	C compiler	166.i	C	14.000
181.mcf	Minimum cost network flow solver	inp.in	C	43.500
186.crafty	Chess program	crafty.in	C	74.700
191.parser	Natural language processing	ref.in	C	83.100
252.eon	Ray tracing	cook	C++	57.600
253.perlbmk	Perl	splitmail.535	C	45.300
254.gap	Computational group theory	ref.in	C	79.800
255.vortex	Object Oriented Database	lendian1.raw	C	58.200
256.bzip2	Data compression utility	inp.program	C	13.500
300.twolf	Place and route simulator	ref	C	324.300

Each program is compiled with the *-O2 -non_shared* options using DEC Alpha AXP-21264 C/C++ compiler and executed using the reference input set. Fortran programs are compiled with the DIGITAL Fortran 90/Fortran 77 compilers. The fast forwards applied to each application, in order to obtain the traces, are shown in Tables 2.2 and 2.3. Next, we list the 26 benchmarks that are included in the SPEC CPU 2000 benchmark suite and that we use in this thesis:

- **SPEC CPU INT 2000**: gzip, vpr, gcc, mcf, crafty, parser, eon, perlbmk, gap, vortex, bzip2 and twolf.
- **SPEC CPU FP 2000**: wupwise, swim, mgrid, applu, mesa, galgel, art, equake, facerec, ammp, lucas, fma3d, sixtrack and apsi.

2.3.1 Simulation Time Reduction

During the research covered by this thesis a huge number of experiments were performed. Each of these experiments involved hundreds or thousands of simulations, each one comprising several hundred million simulated instructions. As a consequence, it was critical to reduce the computational cost constraints of these experiments.

Thus, reducing simulation time is an important issue that can significantly reduce a design budget. With the objective of reducing simulation time without losing accuracy,

Table 2.3: SPEC CPU FP 2000 benchmarks description and simulation starting point using the SimPoint methodology [100]

Benchmark	Description	Input	Language	Fast forward (Millions of instructions)
168.wupwise	Quantum chromodynamics	wupwise.in	Fortran77	263.100
171.swim	Shallow water modeling	swim.in	Fortran77	47.100
172.mgrid	Multi-grid solver in 3D potential field	mgrid.in	Fortran77	187.800
173.applu	Parabolic/elliptic partial differential equations	applu.in	Fortran77	10.200
177.mesa	3D Graphics library	frames100 + msea.in	C	294.600
178.galgel	Fluid dynamics: analysis of oscillatory instability	galgel.in	Fortran90	175.800
179.art	Neural network simulation; adaptive resonance theory	-scanfile c756hel.in -trainfile1 a10.img -trainfile2 hc.img -stride 2 -startx 110 -starty 200 -endx 160 -endy 240 -objects 10	C	13.200
183.equake	Finite element simulation; earthquake modeling	inp.in	C	27.000
187.facerec	Image processing	facerec.in	Fortran90	
188.ammp	Computer vision: recognizes faces	ammp.in	C	13.200
189.lucas	Computational chemistry	lucas2.in	Fortran90	30.000
191.fma3d	Finite element crash simulation	fma3d.in	Fortran90	10.500
200.sixtrack	Particle accelerator model	sixtrack.in	Fortran77	173.500
301.apsi	Solves problems regarding temperature, wind, velocity and distribution of pollutants	apsi.in	Fortran77	192.600

some interesting proposals have appeared in the last few years. First, *sampled simulation* consists in choosing the most representative segment of the entire program trace to simulate. The selection of this portion of the program is not easy and many efforts have been made to find a successful solution [22, 100, 114]. Another option is *statistical simulation*, which consists of creating a synthetic trace from program and architecture statistics that is simulated in a full simulator [37]. Finally, an interesting idea is to reduce the size of the inputs of a program in order to reduce the simulation time [59].

Sampled simulation has been implemented in different ways, basically selecting one or multiple portions of a program execution trace. Selecting these representative samples is an important issue [100, 114]. Random samples appear to be inadequate, while just choosing the beginning of a program could be incorrect due to initialization code. How-

ever, we know that a program execution consists of many different phases, where statistics such as cache or branch misses significantly change among them. Thus, a sampled trace should represent major program phases.

This idea drives the *SimPoint* methodology [100]. Sherwood et al. [100] explain how to detect a program's phases by using the Basic Block Vector (BBV) which counts how many times each basic block appears. Two phases are considered the same if Mannheim's distance between their BBVs is small. At the beginning, the execution of the program is split into a set of *intervals* of fixed size (10 million instructions). Using clustering algorithms, such as *random linear projection* or *k-means*, the samples are joined. The first algorithm is used to reduce the dimension of the BBV and, in that way, accelerate the k-means algorithm. This last algorithm is run for values of k between 1 and M (M is the maximum number of phases to use) and the intervals are grouped into phases. Using the *Bayesian Information Criterion* (BIC), which measures the goodness of fit of a clustering within a dataset, the smallest value of k with a minimum BIC score is chosen. SimPoint choses the representative of each phase that is closest to its centroid. Finally, these representatives are accurately simulated and the results are weighted by the size of each phase.

In the experiments performed in this thesis, we made use of sampled simulation techniques in order to reduce simulation time without losing accuracy. In particular, we collected traces of the most representative 300 million instruction segment of each program, following the SimPoint methodology [100], as this methodology is widely accepted in the literature.

2.3.2 Simulation Methodology

Working with several traces at a time involves an important decision, namely, to determine when a simulation finishes. A *simulation methodology* precisely defines when the measurements of a given workload execution are taken. In a single-threaded processor, the simulator runs the full trace until completion. However, it is not so easy in a multithreaded processor simulator to run a workload composed of several traces. Each benchmark in a workload can execute at a different speed due to the different features of each program, as well as the availability of the shared resources. Therefore, they do not have to necessarily complete execution at the same time.

Common simulation methodologies such as *first* (simulation ends when the first thread finished executing), *last* (simulation ends when all threads have finished executing), and

fixed instructions or cycles (simulation ends after a fixed number of executed instructions or cycles has been reached) cannot ensure that the trace of every benchmark is fully executed, and thus, it is not possible to assure that the measurements are representative of the whole program behavior.

In this thesis we use the FAME simulation methodology [112, 113]. It has been shown that this methodology provides more accurate measurements than previous methodologies when simulating multithreaded workloads. This evaluation methodology measures the performance of multithreaded processors by re-executing all the benchmarks in a multithreaded workload until all of them are fairly represented in the final IPC taken from the workload. The number of times that a benchmark is re-executed depends on the evolution of the IPC during its execution and a Maximum Allowable IPC Variance (MAIV) that is chosen by the user. In our simulations, a MAIV value of 5% is chosen. Tables 2.4(a) and (b) show the number of repetitions per benchmark with the desired MAIV value.

Table 2.4: Number of repetitions required for each SPEC CPU 2000 benchmark in our baseline configuration for a 5% MAIV value

(a) SPEC CPU INT 2000.

Benchmark	Repetitions
gzip	2
vpr	1
gcc	7
mcf	1
crafty	1
parser	5
eon	1
perlbmk	4
gap	8
vortex	1
bzip2	2
twolf	1

(b) SPEC CPU FP 2000.

Benchmark	Repetitions
wupwise	1
swim	2
mgrid	2
applu	1
mesa	1
galgel	7
art	1
equake	1
facerec	3
ammp	1
lucas	1
fma3d	1
sixtrack	1
apsi	6

2.4 Workload Classification

In order to better understand the behavior of different partitioning techniques, it is convenient to classify workloads in different groups so that results are consistent inside each

group. With that purpose we introduce the following two metrics.

Metric 1. The $w_{P\%}(B)$ metric measures the number of ways needed by a benchmark *B* to obtain at least a given percentage *P*% of its maximum IPC (when it uses all L2 ways).

The intuition behind this metric is to classify benchmarks depending on their cache utilization. Using $P = 90\%$ we can classify benchmarks into three groups: *Low utility* (L), *Small working set* or *saturated utility* (S), and *High utility* (H). In a 16- and 32-way associative cache, L benchmarks have $1 \leq w_{90\%} \leq \frac{K}{8}$ where K is the L2 associativity ($K = 16$ or $K = 32$). L benchmarks are not affected by L2 cache space because nearly all L2 accesses are misses. This situation can occur when the application has a working set that does not fit in the L2 cache or when it has low re-use of the data stored in the L2 cache. S benchmarks have $\frac{K}{8} < w_{90\%} \leq \frac{K}{2}$ and just need some ways to have maximum throughput as they fit in the L2 cache. Finally, H benchmarks have $w_{90\%} > \frac{K}{2}$ and always improve IPC as the number of ways given to them is increased. Clear representatives of these three groups are `applu` (L), `gzip` (S) and `ammp` (H) in Figure 2.3. Table 2.5 gives the values of $w_{90\%}$ for all SPEC CPU 2000 benchmarks.

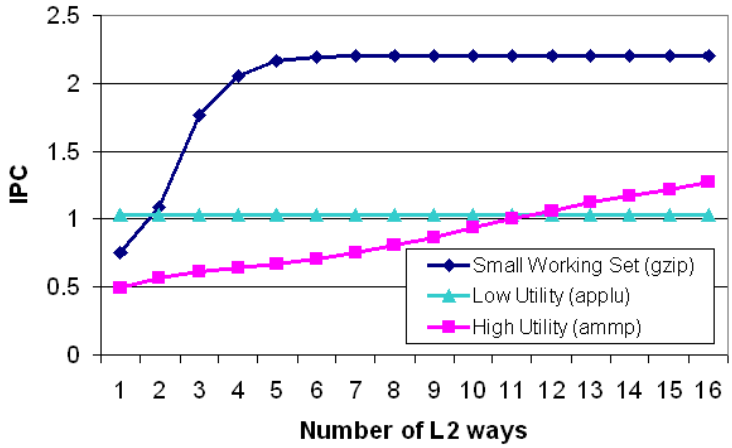


Figure 2.3: IPC curve as we vary the number of assigned ways to `applu` (L), `gzip` (S), and `ammp` (H) in a 1MB 16-way L2 cache

It is interesting to note that, on average, SPEC CPU 2000 benchmarks need 6.11 ways to attain 90% of their peak IPC (in our baseline configuration). This means that, ideally, 61.8% of the 16 ways in the L2 can be turned off with just a 10% IPC degradation. This result is a good motivation for dynamic power and cache partitioning mechanisms.

Next, we list the benchmarks that belong to each group.

- *L*: applu, bzip2, equake, gap, lucas, mcf, mesa, sixtrack, swim and wupwise.

Table 2.5: For all SPEC CPU 2000 benchmarks, we give the metrics $w_{90\%}$ and APTC needed to classify workloads together with their IPC for a 1MB 16-way L2 cache configuration

Bench	$w_{90\%}$	APTC	IPC	Bench	$w_{90\%}$	APTC	IPC	Bench	$w_{90\%}$	APTC	IPC
ammp	14	23.63	1.27	applu	1	16.83	1.03	apsi	10	21.14	2.17
art	10	46.04	0.52	bzip2	1	1.18	2.62	crafty	4	7.66	1.71
eon	3	7.09	2.31	equake	1	18.6	0.27	facerec	11	10.96	1.16
fma3d	9	15.1	0.11	galgel	15	18.9	1.14	gap	1	2.68	0.96
gcc	3	6.97	1.64	gzip	4	21.5	2.20	lucas	1	7.60	0.35
mcf	1	9.12	0.06	mesa	2	3.98	3.04	mgrid	11	9.52	0.71
parser	11	9.09	0.89	perl	5	3.82	2.68	sixtrack	1	1.34	2.02
swim	1	28.0	0.40	twolf	15	12.0	0.81	vortex	7	9.65	1.35
vpr	14	11.9	0.97	wupw	1	5.99	1.32				

- *S*: crafty, eon, gcc, gzip, perlbmk and vortex.
- *H*: ammp, apsi, art, facerec, fma3d, galgel, mgrid, parser, twolf and vpr.

Metric 2. The $w_{LRU}(th_i)$ metric measures the number of ways given by LRU to each thread th_i in a workload composed of N threads. This can be done simulating all benchmarks alone and using the frequency of L2 accesses for each thread [24]. We denote the number of L2 Accesses in a Period of one Thousand Cycles for thread i as $APTC_i$. Table 2.5 lists these values for each benchmark in the SPEC CPU 2000 suite.

$$w_{LRU}(th_i) = \frac{APTC_i}{\sum_{j=1}^{N} APTC_j} \cdot Associativity$$

For example, in a 16-way cache shared by four cores running four applications with $APTC_0 = 2$, $APTC_1 = 4$, $APTC_2 = 10$, and $APTC_3 = 16$, we have that LRU will assign to each thread the following number of ways: $w_{LRU}(th_0) = \frac{2}{32} \cdot 16 = 1$, $w_{LRU}(th_1) = 2$, $w_{LRU}(th_2) = 5$, and $w_{LRU}(th_3) = 8$.

Next, we use these two metrics to classify workloads with two or more benchmarks. This classification is the first contribution of the thesis and allows better understanding the behavior of cache partitioning techniques in a CMP architecture [76].

Case 1. When $w_{90\%}(th_i) \leq w_{LRU}(th_i)$ for all threads. In this situation LRU attains 90% of each benchmark performance. Thus, it is intuitive that in this situation there is very little room for improvement.

Case 2. When two threads A and B exist, such that $w_{90\%}(th_A) > w_{LRU}(th_A)$ and $w_{90\%}(th_B) < w_{LRU}(th_B)$. In this situation, LRU harms the performance of thread A,

because it gives more ways than necessary to thread B. Thus, in this situation, LRU is assigning some shared resources to a thread that does not need them, while the other thread could benefit from these resources.

Case 3. Finally, the third case happens when $w_{90\%}(th_i) > w_{LRU}(th_i)$ for all threads. In this situation, the L2 cache configuration is not big enough to assure that all benchmarks will have at least 90% of their peak performance. In [76] it was observed that pairings belonging to this group showed worse results when the value of $|w_{90\%}(th_1) - w_{90\%}(th_2)|$ grows. In this case, we have a thread that requires much less L2 cache space than the other to attain 90% of its peak IPC. LRU treats threads equally and manages to satisfy the less demanding thread necessities. In the case of previous partitioning algorithms [29, 58, 90, 99, 108], they assume that all misses are equally important for throughput and tend to give more space to the thread with higher L2 cache necessity, while harming the less demanding threads. This is a problem caused by these algorithm. In the following chapters we will show different partitioning policies that overcome this problem.

Table 2.6: Workloads belonging to each case for a 1MB 16-way and a 2MB 32-way shared L2 cache

	1MB 16-way			2MB 32-way		
#cores	Case 1	Case 2	Case 3	Case 1	Case 2	Case 3
2	155 (48%)	135 (41%)	35 (11%)	159 (49%)	146 (45%)	20 (6.2%)
4	624 (4%)	12785 (86%)	1541 (10%)	286 (1.9%)	12914 (86%)	1750 (12%)
6	306 (0.1%)	219790 (95%)	10134 (5%)	57 (0.02%)	212384 (92%)	17789 (7.7%)
8	19 (0%)	1538538 (98%)	23718 (2%)	1 (0%)	1496215 (96%)	66059 (4.2%)

Table 2.6 shows the total number of workloads that belong to each case for different configurations. We generate all possible combinations without repeating benchmarks. The order of benchmarks is not important. In the case of a 1MB 16-way L2 cache, we note that Case 2 becomes the dominant case as the number of cores increases. The same trend is observed for L2 caches with larger associativity. In Table 2.6 we can also see the total number of workloads that belong to each case as the number of cores increases for a 32-way 2MB L2 cache. Note that with different L2 cache configurations, the value of $w_{90\%}$ and $APTC_i$ will change for each benchmark. An important conclusion from Table 2.6 is that as we increase the number of cores, there are more combinations that belong to the second case, which is the one with more improvement possibilities.

To evaluate our proposals, we randomly generate 16 workloads belonging to each group for four different configurations[2]. We denote these configurations *2C* (2 cores and 1MB 16-way L2), *4C-1* (4 cores and 1MB 16-way L2), *4C-2* (4 cores and 2MB 32-way

[2]Thus, we composed a total of 48 workloads for each different configuration

L2) and *8C-2* (8 cores and 2MB 32-way L2). We also use a 2MB 32-way L2 cache as future CMP architectures will continue scaling L2 size and associativity. For example, the Intel Core 2 family has up to a 12MB 24-way shared L2 cache [51], while the Sun UltraSPARC T2 has a 16-way 4MB L2 cache shared by 8 cores [111].

Average improvements do consider the distribution of workloads among the three groups. We denote this mean *weighted mean*, as we assign a weight to the speed up of each case depending on the distribution of workloads from Table 2.6. For example, for the *2C* configuration, we compute the weighted mean improvement as $0.48 \cdot x_1 + 0.41 \cdot x_2 + 0.11 \cdot x_3$, where x_i is the average improvement in Case i.

2.4.1 Performance Metrics

As performance metrics, we use the IPC of an application (computed as the average number of committed instructions per cycle), and the *IPC throughput*, which corresponds to the sum of individual IPCs in a workload with N threads.

$$\text{IPC Throughput} = \sum_{i=1}^{N} IPC_i \tag{2.1}$$

We also use the harmonic mean of relative IPCs to measure fairness, which we denote *Hmean*. The relative IPC is defined as $\frac{IPC_{multithreaded}}{IPC_{alone}}$, where the $IPC_{multithreaded}$ is the IPC of a thread in a given workload in the multithreaded architecture, and the IPC_{alone} is the IPC of a thread when it runs in isolation in the system. The Hmean is calculated as shown in Equation 2.2.

$$\text{Hmean} = \frac{N}{\sum_{i=1}^{N} \frac{IPC_{i,alone}}{IPC_{i,multithreaded}}} \tag{2.2}$$

Other authors use the *weighted speed up* metric, defined in Equation 2.3

$$\text{Weighted Speed Up} = \sum_{i=1}^{N} \frac{IPC_{i,multithreaded}}{IPC_{i,alone}} \tag{2.3}$$

In this work, we use *Hmean* instead of weighted speed up because it has been shown to provide better fairness-throughput balance than weighted speed up [68].

Chapter 3
Dynamic Cache Partitioning Algorithms

3.1 Introduction

Some applications have many compulsory misses that cannot be solved by assigning more cache space to the application, or they present low re-use of their data and pollute caches with data streams, such as multimedia, communications or streaming applications. Traditional eviction policies such as Least Recently Used (LRU), pseudo LRU or random are demand-driven, that is, they tend to give more space to the application that has more accesses and misses to the cache hierarchy [24, 87]. As a consequence, some threads can suffer a severe degradation in performance when running with those type of threads in a CMP architecture with a shared cache.

Previous work has tried to solve this problem by using static and dynamic partitioning algorithms that monitor the L2 cache accesses and decide a partition for a fixed amount of cycles in order to maximize throughput [29, 90, 108] or fairness [58]. Basically, these dynamic proposals predict the number of misses per application for each possible cache partition. Then, they use the cache partition that leads to the minimum number of misses for the next interval. In this chapter, we describe in detail all the different parts involved in a cache partitioning algorithm.

Figure 3.1 shows the architectural changes required to support a dynamic cache partitioning (DCP) technique in a 2-core CMP with a shared L2 cache. In our baseline CMP processor setup, each core has a private L1 instruction and data caches, while the unified L2 cache is shared between the cores. The L2 cache partitioning is enforced by using a *modified replacement logic* (MRL). The *monitoring logic* (ML) tracks the execution of

each application and provides the necessary information to the *partitioning logic* (PL), which is in charge of deciding cache assignments.

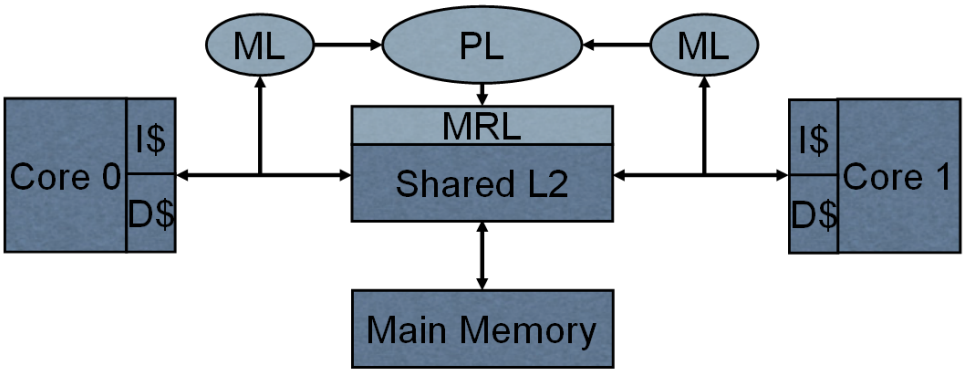


Figure 3.1: Main components of a cache partitioning framework for a 2-core CMP architecture with a shared L2 cache: monitoring logic (ML), modified replacement logic (MRL), and partitioning logic (PL)

The rest of this chapter is structured as follows. Section 3.2 describes the modifications needed in the replacement policy to allow partitioning a shared cache. Next, Section 3.3 explains the mechanisms that are necessary in order to obtain information of each running application. In Section 3.4 we discuss different partitioning algorithms that will be used as the baseline in subsequent chapters. Next, Section 3.5 describes different methods of implementing the described cache partitioning algorithms. Finally, Section 3.6 summarizes other partitioning algorithms which have already been proposed.

3.2 Modified Replacement Logic

Cache partitions at a way granularity can be implemented with *column caching* [29], which uses a bit mask to mark reserved ways (or columns) to each thread. This implementation makes use of a global per thread mask. For correctness, threads can read data from all ways, but can only evict cache lines from their owned ways. The evicted line will be the LRU line among its owned ways.

A second possible implementation consists of augmenting the LRU policy with counters that keep track of the number of lines in a set belonging to each thread [108]. Each thread has an assigned quota of owned lines per set, which implies that each thread owns one counter per cache set. On a cache miss, if the thread reaches its quota, the evicted line is the LRU line among its owned lines. If it does not reach its quota, the evicted line

is the LRU line among the lines of other threads. In the case of an L2 hit, the access is done as usual, which guarantees the correctness of the program execution.

3.3 Monitoring Logic

Mattson et al. [71], in their discussion of storage hierarchies and the Stack Distance Histogram (SDH), introduce the concept of stack distance in order to study the behavior of storage hierarchies. Common eviction policies such as LRU have the *stack property*. Basically, each set in a cache can be seen as an LRU stack, where lines are sorted by their last access cycle. In this way, the first line of the LRU stack is the Most Recently Used (MRU) line, while the last line is the LRU line. The position that a line has in the LRU stack when it is accessed again is defined as the *stack distance* of the access. As an example, we can see in Table 3.1(a) a stream of accesses to the same set with their corresponding stack distances and in Table 3.1(b) the evolution of the contents in the cache (hits are marked in bold).

Table 3.1: Stack distance computation. Cache hits are marked in bold

(a) Stream of accesses to a given cache set

# Reference	1	2	3	4	5	6	7	8
Cache Line	A	B	C	C	A	D	B	D
Stack Distance	-	-	-	1	3	-	4	2

(b) Cache contents evolution

MRU	-	A	B	**C**	C	A	D	B	D
	-	-	A	B	B	C	A	**D**	B
	-	-	-	A	**A**	B	C	A	A
LRU	-	-	-	-	-	-	**B**	C	C

For a K-way associative cache with LRU replacement algorithm, we need $K + 1$ counters to build a SDH, denoted $C_1, C_2, \ldots, C_K, C_{>K}$. On each cache access, only one of the counters is incremented. If it is a cache access to a line in the i^{th} position in the LRU stack of the set, C_i is incremented. If it is a cache miss, the line is not found in the LRU stack and, as a result, we increment the miss counter $C_{>K}$. SDHs can be obtained during execution by running the thread alone in the system [29] or by adding some hardware counters that profile this information [90, 108]. A characteristic of these histograms is that the number of cache misses for a smaller cache with the same number of sets can be easily computed. Some authors have used SDHs to improve the management of main memory and reduce the number of page faults [118]. For example, for a K'-way associative cache, where $K' < K$, the new number of misses can be computed as follows:

$$misses = C_{>K} + \sum_{i=K'+1}^{K} C_i \tag{3.1}$$

As an example, in Table 3.2 we show an SDH for a set with 4 ways. Here, we have 5 cache misses. However, if we reduce the number of ways to 2 (keeping the number of sets constant), we will experience 20 misses (5 + 5 + 10).

Table 3.2: Stack distance histogram example

Stack Distance	1	2	3	4	>4
# Accesses	60	20	10	5	5

Next, Figure 3.2 shows the SDHs of all the SPEC CPU 2000 benchmarks using a heat map. Each color represents a different number of accesses per thousand cycles (APTC): darker colors mean higher number of accesses with a given stack distance. Misses are represented in the column labeled with a stack distance >16, as we are using a 16-way 1MB L2 cache.

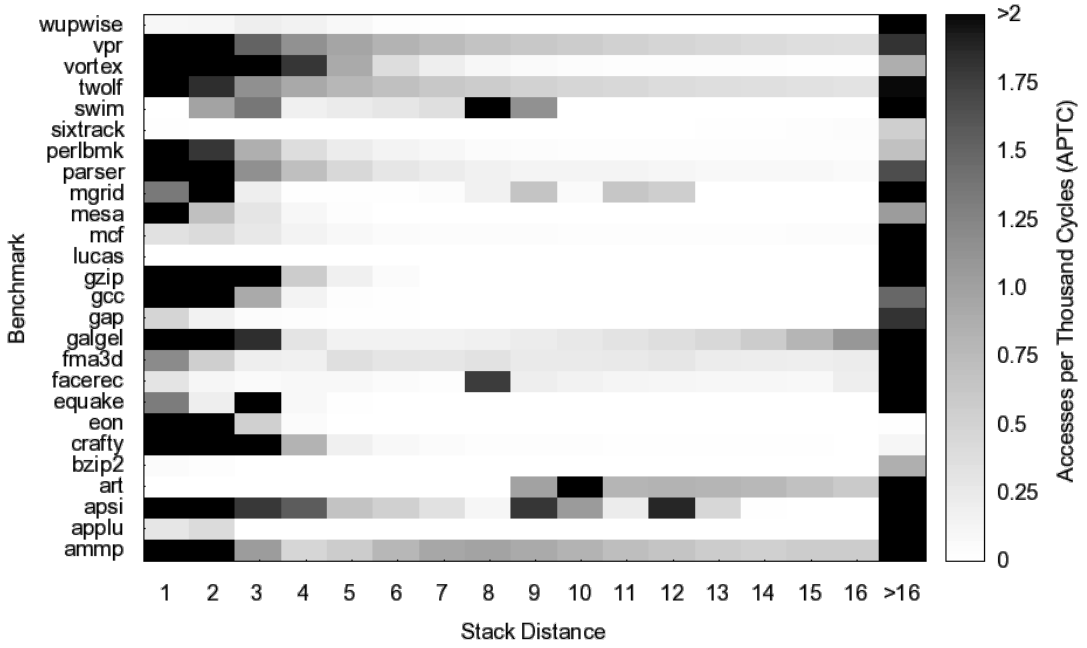


Figure 3.2: Stack distance histograms of all SPEC CPU 2000 benchmarks with a 16-way 1MB L2 cache. Darker colors correspond to more accesses per thousand cycles with a given stack distance

It is interesting to note that some benchmarks like `bzip2`, `lucas` or `wupwise` always miss in the L2 cache. Others like `ammp`, `galgel`, `twolf` or `vpr` have accesses with all possible stack distances (from 1 to 16). In the case of `art`, there is no access with stack distance between 1 and 8: the working set begins to fit in the cache when at least 9 ways are assigned to that benchmark. A different situation occurs with `mcf` and `fma3d`,

as these benchmarks have many accesses with all possible stack distances, but their very low IPC translates into a low number of accesses per thousand cycles. Finally, `eon` and and `crafty` perfectly fit in the L2 cache if they receive 3 and 4 ways, respectively. There is a clear correlation between the shape of the SDHs shown in Figure 3.2 and the values of $w_{90\%}$ shown in Table 2.5, which proves that this metric is adequate to represent the behavior of these benchmarks.

3.4 Partitioning Logic

Using the SDHs of N applications, we can derive the L2 cache partition that minimizes the total number of misses: this latter number corresponds to the sum of the number of misses of each thread for the given configuration. The optimal partition in the last period of time is a suitable candidate to become the future optimal partition. Partitions are decided periodically after a fixed amount of cycles. In this scenario, partitions are decided at a *way granularity*. This mechanism is used in order to minimize the total number of misses and try to maximize throughput. A first approach proposed a static partitioning of the L2 cache using profiling information [29]. Then, a dynamic approach estimated SDHs with information inside the cache [108]. Finally, Qureshi et al. [90] presented a suitable and scalable circuit to measure SDHs using sampling and obtained performance gains with just 0.2% extra space in the L2 cache. Throughout this manuscript, we will call this last policy *MinMisses*.

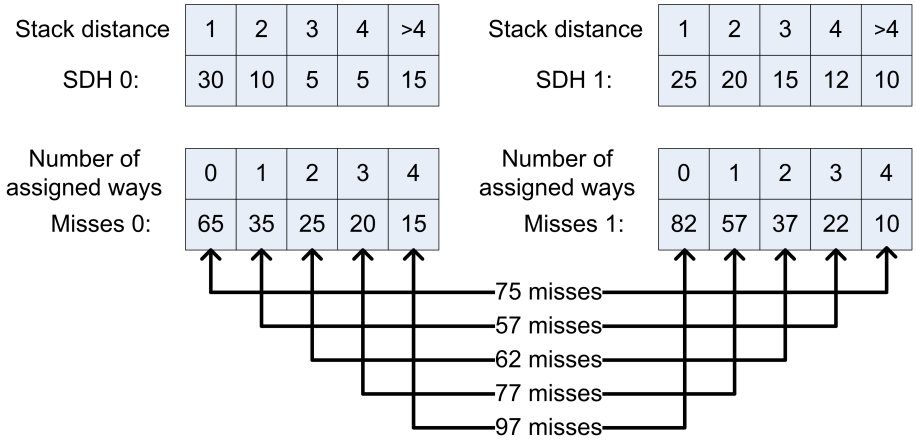


Figure 3.3: MinMisses dynamic cache partitioning example

Figure 3.3 shows a synthetic example of how *MinMisses* works. We represent the SDHs of two applications for a 4-way associativity cache. Thanks to Formula 3.1 we can

derive the number of misses of each application with a given number of assigned ways. Finally, we compute the total number of misses for all possible partitions and decide the optimal one, which consists of assigning 1 way to thread 0 and 3 ways to thread 1 (35 misses for thread 0 and 22 misses for thread 1).

Fair Partitioning. In some situations, *MinMisses* can lead to unfair partitions that assign nearly all the resources to one thread while harming the others [58]. For that reason, Kim et al. [58] propose considering fairness when deciding new partitions. In this way, instead of minimizing the total number of misses, they try to equalize the statistic $X_i = \frac{misses_{shared_i}}{misses_{alone_i}}$ of each thread i. They aim to force all threads to have the same increase in percentage of misses. Partitions are decided periodically using an iterative method. The thread with largest X_i receives a way from the thread with smallest X_i until all threads have a similar value of X_i. Throughout this manuscript, we will call this policy *Fair*.

In the example shown in Figure 3.3, the partition that equalizes the values of the statistic X_i for all threads consists of assigning 1 way to thread 0 and 3 ways to thread 1. In this situation, we have $X_0 = \frac{misses_{shared_0}}{misses_{alone_0}} = \frac{35}{15} = 2.33$ and $X_1 = \frac{22}{10} = 2.2$.

3.5 Cache Partitioning Decision

Cache partitions can be decided statically at the beginning of the execution of the application using some offline profiling, or dynamically during the execution of the application. The frequency of cache partitioning decisions directly impacts the performance improvement obtained by the cache partitioning algorithms. Dynamic mechanisms are more flexible and can adapt to different program phases or changes in the executing workload, but also require specialized hardware support in the architecture. In the following chapters we will discuss in detail what the optimal frequency of cache partition decisions for each proposal is.

Ideally, we would like to decide the optimal cache partition using the stored information in the SDHs in just a couple of CPU cycles. Different heuristics have already been proposed [90, 109] to reduce the time needed to decide the optimal partition for the next interval of time. Next, we describe these different heuristics that can be used to find the optimal partition for the algorithms presented in Section 3.4.

3.5.1 Generating All Possible Combinations

The simplest algorithm to find the optimal partition for a given cost function consists of making use of exhaustive search, which evaluates all possible partition and choosing the one with the best result. In our situation we have K ways to assign to N different cores so that $\sum_{i=0}^{N-1} k_i = K$, where k_i is the number of ways assigned to core i and $k_i \geq 0$ for any i. This problem has a closed formula that gives the number of combinations:

$$\#combinations = \binom{K+N-1}{K} = \frac{(K+N-1)!}{K! \cdot (N-1)!}$$

If we increase the values of K and N, this number explodes, as we have a factorial number in the numerator. For example, when $K = 32$ and $N = 4$, this number is $6,545$, while for $K = 32$ and $N = 8$, this number is $15,380,937$. Furthermore, we have to evaluate every combination, which has a non negligible cost. We will call this algorithm *EvalAll*. However, if we assume a minimum number of ways min_{ways} assigned to each core, then this formula changes as we have to assign just $K - N \cdot min_{ways}$ ways among different cores: $\#combinations = \binom{K+N-N \cdot min_{ways}-1}{K-N \cdot min_{ways}}$

3.5.2 Marginal Gains Algorithm

Suh et al. [109] make use of a greedy algorithm denoted *marginal gains* to obtain an approximation of the optimal partition at low cost. This algorithm is an extension of the one presented earlier by Stone et al. [106], and is described in Algorithm 1.

Algorithm 1: Marginal gains greedy algorithm [109]

Data: SDH of each application, $SDH_i[j], 0 \leq i \leq N, 0 \leq j \leq K$

Result: Final cache partition $(k_1, \ldots, k_N)$

begin

1. Initialize $k_1 = 0, \ldots, k_N = 0$.
2. **while** *less than K ways have been assigned* **do**
   - Find the process i with maximum value $SDH_i[k_i]$.
   - Assign one extra way to that process: $k_i = k_i + 1$.

end

end

This algorithm is shown to be optimal if the involved curves are monotonically decreasing convex curves. In the case of stack distance histograms, it is clear that as more

ways are assigned to a given application, less misses will be obtained (which means that these curves are monotonically decreasing). However, the assumption of convexity is not always true.

In order to overcome the problem of having non convex curves, the authors in [109] propose generating initial partitions randomly and invoking the previous algorithm. The problem with this solution is that we need to generate a number of random partitions that grows with the total number of combinations.

3.5.3 Look Ahead Algorithm

Algorithm 2: Look ahead greedy algorithm [90]

Data: SDH of each application, $SDH_i[j], 0 \leq i \leq N, 0 \leq j \leq K$

Result: Final cache partition $(k_1, \ldots, k_N)$

begin

1. Initialize $k_1 = 0, \ldots, k_N = 0$.
2. **while** *less than K ways have been assigned* **do**
   - **foreach** *application* i **do**
     - Find the value $b_{i,opt}$ that maximizes $MU_{i,k_i}^{b_i}$
   - **end**
   - Find the process i with maximum value $MU_{i,k_i}^{b_{i,opt}}$.
   - Assign $b_{i,opt} - k_i$ extra way to that process: $k_i = b_{i,opt}$.

end

end

Qureshi et al. [90] note that the basic problem with the *marginal gains* greedy algorithm presented in the previous section is that it only considers the marginal improvement of adding just one way to a thread. Thus, it fails to see potentially high gains after the first way if there is no gain from that first way. For that reason, they propose an algorithm that also takes into account the gains from far ahead to make better partitioning decisions. This algorithm, denoted *look ahead* greedy algorithm, considers the marginal utility for all possible number of extra ways that the application can receive, and is depicted in Algorithm 2. They define the marginal utility of increasing the number of ways from a to b to a thread i, denoted $MU_{i,a}^{b}$, as:

$$MU_{i,a}^{b} = \frac{U_{i,a}^{b}}{b-a} = \frac{\sum_{k=a}^{b} SDH_i[k]}{b-a}$$

3.5.4 Marginal Gains in Reverse Order Algorithm

Our first proposed algorithm is based on the observation that the number of misses of applications as more cache space is assigned to them normally follows a curve with a knee. When the working set fits in the cache, we obtain only compulsory misses. Thus, the first heuristic consists of detecting these knees. With this objective, we initially assign all the ways to each benchmark and begin to take away the ways that have less utility. Next, we present this algorithm, denoted *marginal gains in reverse order*, as it is essentially the same idea as Algorithm 1, but relieving ways instead of assigning them.

Algorithm 3: Marginal gains in reverse order greedy algorithm

Data: SDH of each application, $SDH_i[j], 0 \leq i \leq N, 0 \leq j \leq K$

Result: Final cache partition $(k_1, \ldots, k_N)$

begin

1. Initialize $k_1 = K, \ldots, k_N = K$.
2. **while** *less than K ways have been assigned* **do**
   Find the process i with minimum value $SDH_i[k_i]$.
   Assign one less way to that process: $k_i = k_i - 1$.

end

end

3.5.5 EvalAll Dynamic Programming Solution

Finally, we propose a new algorithm based on dynamic programming techniques [32]. This algorithm makes use of the fact that finding the optimal partition for just two cores is straightforward as we have to check only K possible partitions. Thus, if a given number of ways are assigned to two cores, the optimal partition will be independent of the partitions in the rest of cores. The idea consists of dividing the demanding threads into two groups and assigning a portion of the cache to each group (there are K possibilities per group). Then, each group is subdivided into two groups and this is iterated until we have to partition the cache between only two cores.

To illustrate Algorithm 4, we next show an example with a 4-way associative cache shared among four cores. Figure 3.4 shows the misses histograms for the four running applications. In the first iteration of the algorithm, we generate two new histograms of misses, $misses_0$ and $misses_1$, with the corresponding number of assigned ways in the optimal partition. For instance, when assigning 2 ways between the first two cores, both

Algorithm 4: *EvalAll* dynamic programming algorithm

Data: SDH of each application, $SDH_i[j]$, $0 \leq i \leq N, 0 \leq j \leq K$
Result: Final cache partition $(k_1, \ldots, k_N)$
begin
1. Build histogram of misses per application, $misses_i[j]$, $0 \leq i \leq N, 0 \leq j \leq K$.
2. **for** $(iter = 1; iter < N; iter = iter \cdot 2)$ **do**
for $(i = 0; i < N/iter; i++)$ **do**
Generate the optimal partition between misses histograms $misses_{2 \cdot i}$ and $misses_{2 \cdot i+1}$ with j assigned ways, $0 \leq j \leq K$.
Store the new misses histogram in $misses_{iter}[j]$ and the cache partition decision in $assigned\_ways_i[iter][j]$, $0 \leq j \leq K$.
end
end
3. The optimal number of misses is stored in $misses_0[K]$ and the cache partition can be reconstructed using vectors $assigned\_ways_i[iter][j]$, $0 \leq i \leq N, 0 \leq j \leq K$.
end

ways are assigned to core 0, with a total of 60 misses. If we assign 4 ways, the optimal partition consists of assigning 1 way to core 0 and 3 to core 1, with a total of 40 misses. In the second iteration of the algorithm, the optimal partition is obtained with a total of 100 misses. Two ways are assigned to each one of the groups of cores (0-1 and 2-3). Then, checking $assigned\_ways_0[0][2]$ and $assigned\_ways_1[0][2]$, we can see that the optimal partition is $(k_1, k_2, k_3, k_4) = (2, 0, 2, 0)$[1].

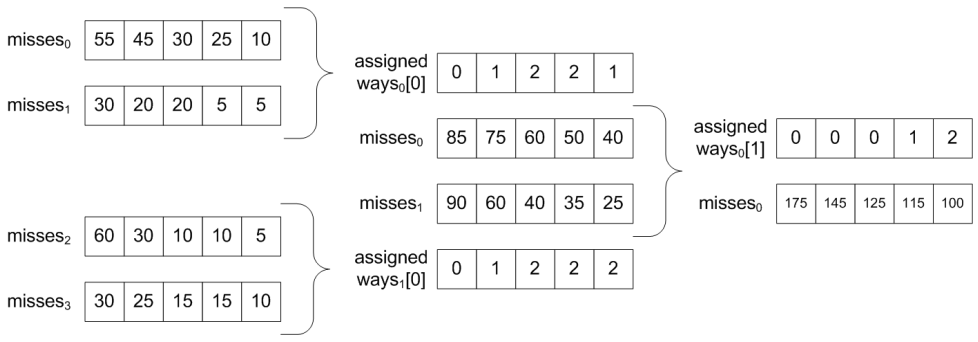


Figure 3.4: *EvalAll* dynamic programming algorithm (Algorithm 4) example with four cores sharing a 4-way associative cache

[1]In this example, we assume that the minimum number of ways assigned to a core is zero.

3.5.6 Overhead and Performance Comparison

In this section, we evaluate the computational cost of the algorithms presented in the previous sections as well as the final performance obtained.

In the case of the *EvalAll* algorithm that computes all possible partitions, it has to evaluate $\binom{K+N-1}{K}$ different combinations. This number explodes as the number of cores and the associativity grow. In the case of *marginal gains* (Algorithm 1), we repeat $K \cdot N$ evaluations of the number of misses. In the case of *look ahead* (Algorithm 2), in the worst case we have to assign one way in each iteration. In such a situation, in every iteration i we have to compare $N \cdot (K - i)$ *marginal utilities*, which imply two subtractions and a division by a small number between 1 and K. Thus, we have to evaluate a total number of combinations:

$$Total = \sum_{i=0}^{K-1} N \cdot (K - i) = N \cdot \sum_{j=1}^{K} j = \frac{K(K+1)}{2} \cdot N \approx \frac{K^2 \cdot N}{2}$$

Next, in the case of *marginal gains in reverse order* (Algorithm 3), we have to repeat $(N - 1) \cdot K$ times N evaluations, which gives a total of $(N - 1) \cdot K \cdot N \approx N^2 \cdot K$ evaluations.

Finally, the implementation of *EvalAll* with dynamic programming techniques (Algorithm 4) has to build the misses histograms at the beginning, which implies $K \cdot N$ operations. Then, we repeat in each iteration i, $N/2^i$ executions of the main loop. This means a total of $N/2+N/4+N/8+\ldots+2+1 = N-1$ repetitions. In the main loop, we have to generate the new histogram of misses. For each possible value of j ($0 \leq j \leq K$), we have to consider $j + 1$ candidate partitions: $\sum_{j=0}^{K} j + 1 = \frac{(K+1)(K+2)}{2} \approx \frac{K^2}{2}$. Finally, we need $N \cdot log_2 N$ to reconstruct the final cache partition. Putting it all together, we need $K \cdot N + (N - 1)\frac{K^2}{2} + N \cdot log_2 N$ to find the optimal partition. Table 3.3 summarizes the computational cost of the different algorithms analyzed so far.

Thus, we can see that the best heuristic in terms of overhead is *marginal gains* (Algorithm 1). Instead, *look ahead* algorithm (Algorithm 2) has an overhead for a given associativity which is higher than the cost of *marginal gains in reverse order* (Algorithm 3) when $N \leq \frac{K}{2}$. In fact, *look ahead* makes use of divisions when it decides the next cache assignment, which implies a significant computational cost that the other proposals do not have. Finally, the implementation of *EvalAll* using dynamic programming techniques (Algorithm 4) has slightly more overhead than the previous proposals, but reduces dra-

Table 3.3: Computational complexity of the different cache partitioning decision algorithms

Algorithm	**Complexity**
EvalAll	$\binom{K+N-1}{K}$
Marginal gains	$K \cdot N$
Look ahead	$\frac{K^2 \cdot N}{2}$
Marginal gains in reverse order	$N^2 \cdot K$
EvalAll with dynamic programming techniques	$K \cdot N + (N-1)\frac{K^2}{2} + N \cdot log_2 N$

matically the overhead of the first implementation of *EvalAll*.

Next, we compare the performance that these different algorithms attain when combined with *MinMisses*. Figure 3.5 shows the results for a 4-core CMP architecture with a shared 1MB L2 cache (configuration *4C-1*). All algorithms except *marginal gains* in reverse order show similar average results. In fact, less than 1% of difference is observed in the harmonic mean of the IPC throughput. Furthermore, in all situations the median speed up is 1.

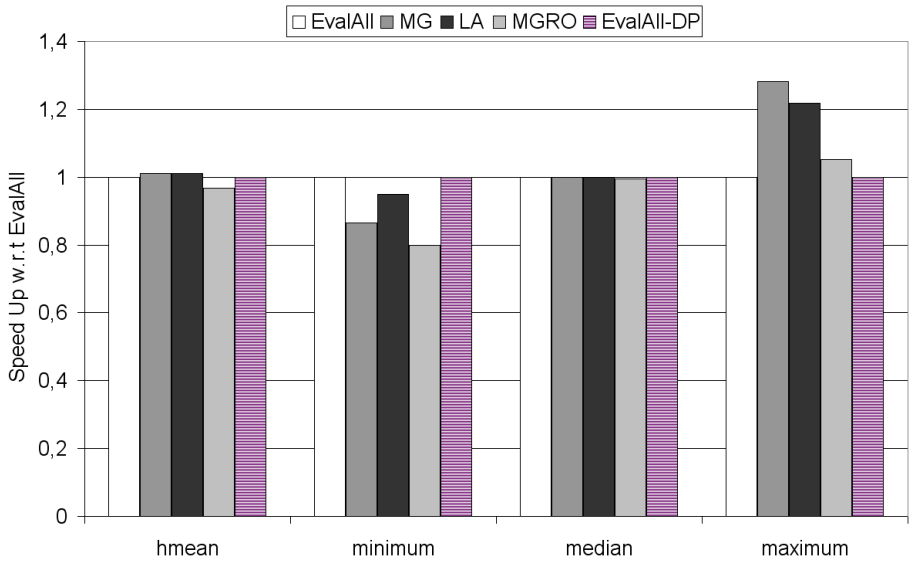


Figure 3.5: Performance comparison between the different algorithms: *EvalAll*, *marginal gains* (MG), *look ahead* (LA), *marginal gains in reverse order* (MGRO) and *EvalAll* with dynamic programming techniques (EvalAll-DP)

Marginal gains (Algorithm 1) normally shows similar performance to *EvalAll*, but when the executed benchmarks have non convex miss curves, then performance drops.

In these situations, *marginal gains* presents a 13.5% maximum performance degradation. However, this algorithm tends to distribute ways more equally to all the threads. In that way, some benchmarks that apparently do not need many ways (typically benchmarks belonging to group S) receive more ways than with *EvalAll*, which is translated into important speed ups (reaching a maximum of 28.2%).

In the case of *look ahead* algorithm (Algorithm 2), we obtain similar results to *marginal gains*. In fact, in a 32.4% of the combinations of benchmarks the decisions are exactly the same. The important difference between these two algorithms is that when benchmarks have non convex curves, performance losses are reduced, and, as a result, the maximum performance degradation is now 5.5%.

In the case of *marginal gains in reverse order* algorithm (Algorithm 3), we obtain slightly worse results. Problems with some benchmarks are detected (`apsi`, `art`, `galgel` and `mgrid`). Theses benchmarks tend to monopolize all the ways as they have many accesses with stack distance near to the L2 cache associativity. Consequently, we have an average 3% performance degradation, with a maximum value of 20.2%.

Finally, *EvalAll* and its implementation with dynamic programming techniques show the same performance, which is wht for this work we always use the implementation of *EvalAll* which makes use of dynamic programming techniques (Algorithm 4). In some cases, we will use other heuristics with similar performance and computational cost, such as the *marginal gains* and *look ahead* algorithms. In fact, *look ahead* and *EvalAll* with dynamic programming techniques have similar complexity, but the first one makes use of divisions to compute the marginal utility $MU_{i,a}^{b}$ while the latter only makes use of additions. The *marginal gains in reverse order* algorithm will not be used, as it shows worse results than the other algorithms.

3.6 Other Cache Partitioning Algorithms

Several papers propose different dynamic cache partitioning (DCP) algorithms in a multithreaded scenario. Table 3.4 summarizes these proposals with their most significant characteristics: if it is a dynamic or static mechanism, the target metric to optimize, who is in charge of deciding the partition, the cache partition decision algorithm and the modified replacement logic that they make use of.

Settle et al. [99] introduce a DCP similar to *MinMisses* that decides partitions depending on the average data re-use of each application. Rafique et al. [92] propose managing

Table 3.4: Different cache partitioning proposals

Paper	Partitioning	Objective	Decision	Algorithm	Replacement Policy
[29]	Static	Minimize Misses	Programmer	–	Column Caching
[108]	Dynamic	Minimize Misses	Architecture	Marginal gains	Augmented LRU
[90]	Dynamic	Maximize Utility	Architecture	Look ahead	Augmented LRU
[58]	Dynamic	Fairness	Architecture	Equalize X_1^i	Augmented LRU
[99]	Dynamic	Maximize re-use	Architecture	Re-use	Column Caching
[92]	Dyn./Static	Configurable	Operating System	Configurable	Augmented LRU

shared caches with a hardware cache quota enforcement mechanism and an interface between the architecture and the OS to let the latter decide quotas. Hsu et al. [48] evaluate different cache policies in a CMP scenario. They show that none of them is optimal among all benchmarks and that the best cache policy varies depending on the performance metric being used. Thus, they propose using a thread-aware cache resource allocation. In fact, their results reinforce one of the main concerns of this thesis: if we do not consider the impact of each cache miss in performance, we can decide suboptimal L2 partitions in terms of throughput or any other IPC-related metric.

Other proposals also partition shared caches with non-uniform cache access time (NUCA) with a different approach: they differentiate between a fast local or private cache and remote on-chip caches. These proposals use data migration and replication in different local caches in order to reduce access time [14, 26, 30, 115, 117]. Finally, other previous work tries to determine the behavior of SMT architectures when some shared resources among threads are statically and equally split. Raash et al. [91] present a study of the effects of partitioning the reorder buffer (ROB), issue queues and fetch bandwidth.

Chapter 4

MLP-aware Dynamic Cache Partitioning Algorithm

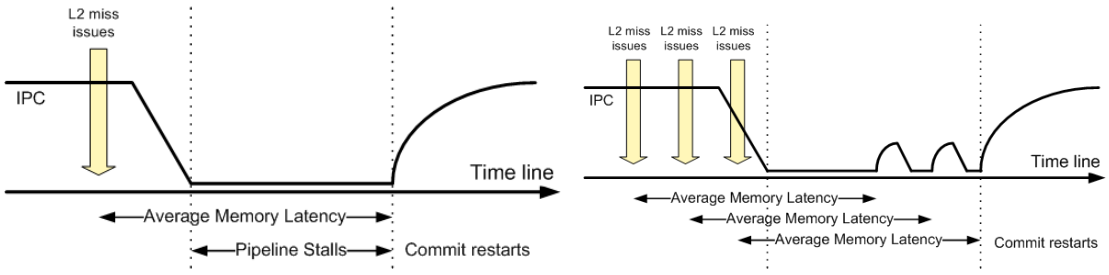
Dynamic partitioning of shared caches has been proposed to improve performance of traditional eviction policies in modern multithreaded architectures. These existing Dynamic Cache Partitioning (DCP) algorithms mostly work with the number of misses caused by each thread and treat all misses equally. However, cache misses have different effects on performance depending on their distribution. Clustered misses share their miss penalty because they can be served in parallel, while isolated misses have a greater impact on performance, as the memory latency is not shared with other misses.

Taking this fact into account, we propose a new DCP algorithm that considers misses differently, depending on their influence on performance. Our proposal results in improvements over traditional eviction policies of up to 63.9% (10.6% on average) and it also outperforms previous DCP proposals by up to 15.4% (4.1% on average) in a four-core architecture. Our proposal reaches the same performance as a 50% larger shared cache. Finally, we present a practical implementation of our proposal that requires less than 8KB storage.

4.1 Introduction

A common characteristic of previous cache partitioning proposals is that they treat all cache misses equally. However, in out-of-order architectures cache misses affect performance differently, depending on how clustered they are. An isolated last level cache (LLC) miss has approximately the same miss penalty as a cluster of misses, since they can be served in parallel if they all fit in the reorder buffer (ROB) [55], as shown in Figure 4.1.

We have represented an *ideal* IPC curve which is constant until an L2 miss occurs[1]. After some cycles, commit stops. When the cache line comes from main memory, commit ramps up to its steady state value. As a consequence, an isolated L2 miss has a higher impact on performance than a miss in a burst of misses, since the memory latency is shared by all clustered misses.



(a) Isolated L2 miss (b) Clustered L2 misses

Figure 4.1: Cache miss penalty of isolated and clustered L2 misses in an out-of-order architecture

To clarify this idea, Figure 4.2 shows the average miss penalty of an L2 miss for the whole SPEC CPU 2000 benchmark suite in our baseline configuration. Results are classified in groups low utility (L), small working set or saturated utility (S), and high utility (H). Note that this average miss penalty varies considerably, even inside each group of benchmarks, ranging from 30 cycles (`art`) to 294 cycles (`bzip2`). This figure reinforces the idea that the clustering level of L2 misses changes for different applications.

Based on this fact, we propose a new DCP algorithm that gives a cost to each L2 access according to its memory-level parallelism (MLP). We detect isolated and clustered misses and assign a higher cost to isolated misses. Then, our algorithm determines the partition that minimizes the total cost for all threads, which is used in the next interval. Our results show that differentiating between clustered and isolated L2 misses leads to cache partitions with higher performance than previous proposals. The main contributions of this chapter are the following:

1) A run-time mechanism to dynamically partition shared L2 caches in a CMP scenario that takes into account the MLP of each L2 access. We obtain improvements over LRU of up to 63.9% (10.6% on average) and over previous proposals by up to 15.4% (4.1% on average) in a four-core architecture. Our proposal reaches the same performance as a 50% larger shared cache.

[1]We assume a shared L2 cache as the LLC in a chip multiprocessor (CMP) architecture

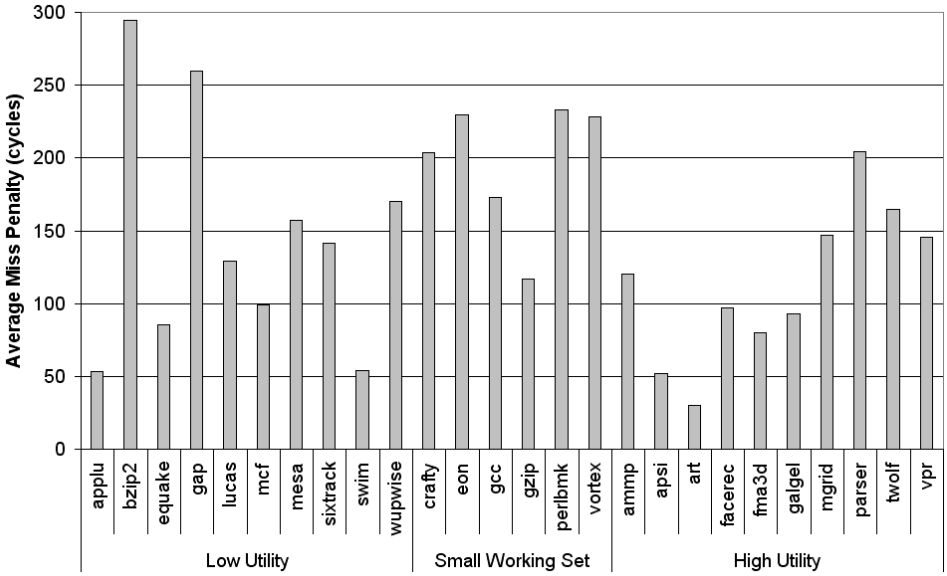


Figure 4.2: Average miss penalty of an L2 miss in a 1MB 16-way L2 cache for the whole SPEC CPU 2000 benchmark suite

2) We present a sampling technique that reduces the hardware cost in terms of storage to less than 1% of the total L2 cache size with an average throughput degradation of only 0.76% (compared to the throughput obtained without sampling). We also show that the scalable decision algorithms presented in Section 3.5 give near optimal partitions, 0.59% close to the optimal decision.

The rest of this chapter is structured as follows. Section 4.2 explains the new monitoring logic and the MLP-aware DCP algorithm. In Section 4.3 we discuss simulation results. Finally, Section 4.4 summarizes our main findings.

4.2 MLP-Aware Dynamic Cache Partitioning

Chapter 3 describes in detail all the architectural changes that are required to implement a dynamic cache partitioning algorithm. These algorithms make use of a basic property of common eviction policies such as LRU, the *stack property*, which makes it possible to build Stack Distance Histograms (SDHs). With these histograms, we can predict the number of misses of the same application in a partitioned L2 cache. In this chapter, we present a new monitoring logic that estimates the MLP of each application and then minimizes the total MLP of the applications instead of the number of misses. Qureshi et al. [89] also propose estimating the MLP of each miss in order to introduce a new eviction

policy for *private* caches in single-threaded architectures. This policy gives a weight to each L2 miss according to its MLP when the block is filled from memory. Eviction is decided using the LRU counters and this weight. This idea was proposed for a different scenario, where the focus was on single-threaded architectures.

Algorithm 5 shows the necessary steps to dynamically decide cache partitions according to the MLP of each L2 access. At the beginning of the execution, we decide an initial partition of the L2 cache. As we have no prior knowledge of the applications, we evenly distribute ways among cores. Hence, each core receives $\frac{K}{N}$ ways of the shared L2 cache, where K is the associativity of the cache and N the number of cores.

Algorithm 5: MLP-Aware dynamic cache partitioning algorithm.

Step 1: Establish an initial even partition for each core ;
Step 2: Run threads and collect data for the MLP-aware SDHs ;
Step 3: Decide new partition ;
Step 4: Update MLP-aware SDHs ;
Step 5: Go back to Step 2 ;

Next, a *measuring period* begins, where the total MLP cost of each application is measured for different cache assignments. The histogram of each thread containing the total MLP cost for each possible partition is denoted *MLP-aware SDH*. For a K-way associative cache, exactly $K+1$ registers are needed to store this histogram. For short periods, dynamic cache partitioning (DCP) algorithms react more quickly to phase changes. Our results show that, for different periods from 10^5 to 10^8 cycles, small performance variations are obtained, with a peak for a period of 5 million cycles.

At the end of each interval, MLP-aware SDHs are analyzed and a new partition is decided for the next interval. We assume that running threads will have a similar pattern of L2 accesses in the next measuring period. Thus, the optimal partition for the last period is chosen for the following period. Evaluating all possible cache partitions gives the optimal partition. This evaluation is done concurrently with a dedicated hardware, which sets the partition for each process in the next period. Having old values of partitions decisions does not impact correctness of the running applications and does not affect performance, as deciding new partitions typically takes few thousand cycles and is invoked once every 5 million cycles.

Since characteristics of applications dynamically change, MLP-aware SDHs should reflect these changes. However, we also wish to maintain some history of the past MLP-aware SDHs to make new decisions. Thus, after a new partition is decided, we multiply all

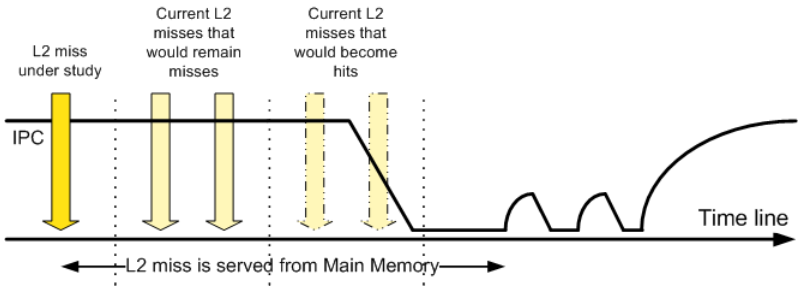
the values of the MLP-aware SDHs times $\rho \in [0, 1]$. Large values of ρ have larger reaction times to phase changes, while small values of ρ quickly adapt to phase changes but tend to forget the behavior of the application. Small performance variations are obtained for different values of ρ ranging from 0 to 1, with a peak for $\rho = 0.5$. Furthermore, this value is very convenient as we can use a shifter to update histograms. Next, a new period of measuring MLP-aware SDHs begins. The key contribution of this chapter is the new monitoring logic required to obtain MLP-aware SDHs that we explain in the following subsection.

4.2.1 MLP-Aware Stack Distance Histogram

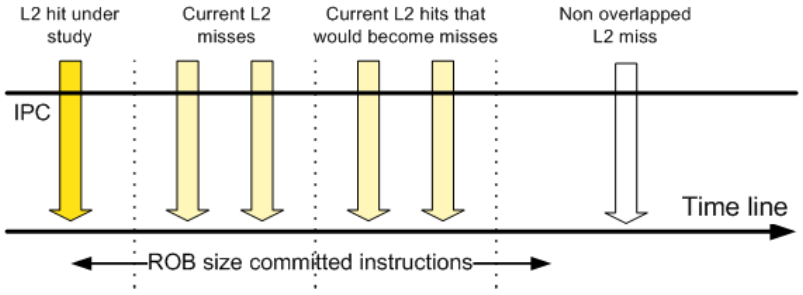
As previously stated, *MinMisses* assumes that all L2 accesses are equally important in terms of performance. However, cache misses have different effects on the performance of applications, even inside the same application [55, 89]. An isolated L2 data miss has a miss penalty that can be approximated by the average memory latency. In the case of a burst of L2 data misses which fit in the ROB, the miss penalty is shared among misses, since L2 misses can be served in parallel. In the case of L2 instruction misses, they are serialized as fetch stops. Thus, L2 instruction misses have a constant miss penalty and MLP.

In this section we present a new monitoring logic that assigns a cost to each L2 access according to its MLP. In [89] a similar idea was used to modify LRU eviction policy for single core and single-threaded architectures. In our case, we have a CMP scenario where the shared L2 cache has a number of reserved ways for each core. At the end of each period, we decide either to continue with the same partition or to change it. If we decide to modify the partition, a core i that had w_i reserved ways will receive $w'_i \neq w_i$ ways. If $w_i < w'_i$, the thread receives more ways and, as a consequence, some misses in the old configuration will become hits. Conversely, if $w_i > w'_i$, the thread receives less ways and some hits in the old configuration will become misses. Thus, we want to have an estimation of the performance effects when misses are converted into hits and vice versa. Throughout this chapter, we will call this impact on performance MLP_cost.

With regard to the **MLP_cost of L2 misses**, in order to compute the MLP_cost of an L2 miss with stack distance d_i, we consider the situation shown in Figure 4.3(a). If we force an L2 configuration that assigns exactly $w'_i = d_i$ ways to thread i with $w'_i > w_i$, some of the L2 misses of this thread will become hits, while other will remain being misses, depending on their stack distance. In order to track the stack distance and MLP_cost of



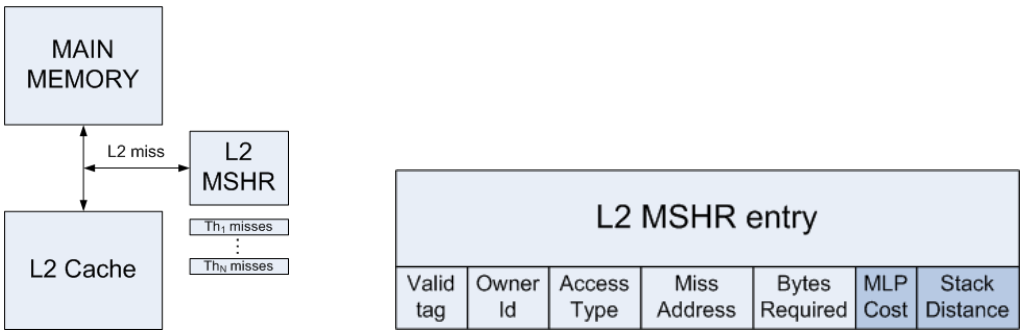
(a) MLP_cost of an L2 miss



(b) Estimated MLP_cost when an L2 hit becomes a miss

Figure 4.3: MLP_cost of L2 accesses in an out-of-order architecture

each L2 miss, we have modified the L2 Miss Status Holding Registers (MSHR) [61]. This structure is similar to an L2 miss buffer and is used to hold information about any load that has missed in the L2 cache. The modified L2 MSHR has one extra field that contains the MLP_cost of the miss, as can be seen in Figure 4.4(b). It is also necessary to store the stack distance of each access in the MSHR. Figure 4.4(a) illustrates the MSHR in the cache hierarchy.



(a) MSHR

(b) MSHR fields

Figure 4.4: Miss Status Holding Register (MSHR) description

When the L2 cache is accessed and an L2 miss is determined, we assign an MSHR

entry to the miss and wait until the data comes from main memory. We initialize the MLP_cost field to zero when the entry is assigned. We store the stack distance of the access together with the identifier of the owner core. In each cycle, we obtain N_{d_i}, the number of L2 accesses with stack distance greater or equal to d_i. We have a hardware counter that tracks this number for each possible number of d_i, which means a total of K counters[2]. If we have N_{d_i} L2 misses that are being served in parallel, the miss penalty is shared. Thus, we assign an equal share of $\frac{1}{N_{d_i}}$ to each miss. The value of the MLP_cost is updated until the data comes from main memory and fills the L2, at which moment the MSHR entry can be freed.

The number of adders required to update the MLP_cost of all entries is equal to the number of MSHR entries. However, this number can be reduced by sharing several adders between valid MSHR entries in a round robin fashion. Thus, if an MSHR entry updates its MLP_cost every 4 cycles, it has to add $\frac{4}{N_{d_i}}$. In this chapter, we assume that the MSHR contains only four adders for updating MLP_cost values, which has a negligible effect on the final MLP_cost [89].

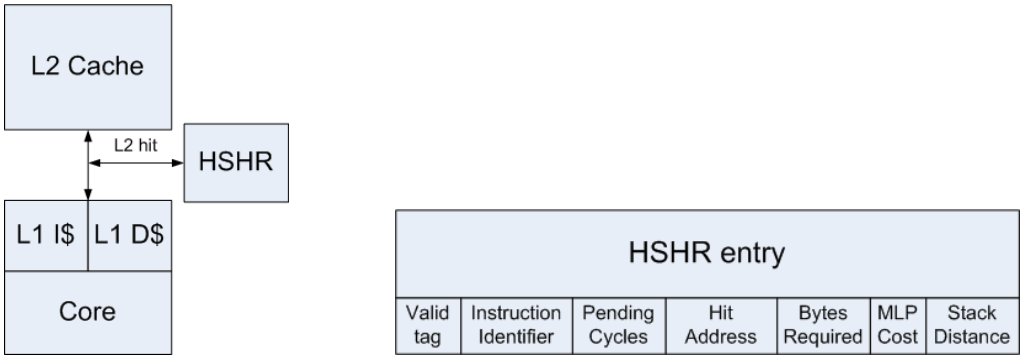
When dealing with th **MLP_cost of L2 hits**, we want to estimate the MLP_cost of an L2 hit with stack distance d_i when it becomes a miss. If we forced an L2 configuration that assigned exactly $w'_i = d_i$ ways to the thread i with $w'_i < w_i$, some of the L2 hits of this thread would become misses, while L2 misses would remain as misses (see Figure 4.3(b)). The hits that would become misses are the ones with stack distance greater or equal to d_i. Thus, we count the total number of accesses with stack distance greater or equal to d_i (including L2 hits and misses) to estimate the length of the cluster of L2 misses in this configuration.

Deciding on the moment to free the entry used by an L2 hit is more complex than in the case of the MSHR. As was said in [55], in a balanced architecture, L2 data misses can be served in parallel if they all fit in the ROB. Equivalently, we say that L2 data misses can be served in parallel if they are at ROB distance smaller than the ROB size. Thus, we should free the entry if the number of committed instructions since the access has reached the ROB size or if the number of cycles since the hit has reached the average latency to memory. The former condition is clear, since L2 misses can overlap only if their ROB distance is less than the ROB size. When the entry is freed, we have to add the number of pending cycles divided by the number of misses with stack distance greater or equal to d_i. The latter condition is also necessary, since it is possible that no L2 access is done for a

[2]In our CMP baseline, we assume a K-way associative cache

period of time. To obtain the average latency to memory, we add a specific hardware that counts and averages the number of cycles that a given entry is in the MSHR.

We use new dedicated hardware to obtain the MLP_cost of L2 hits. We denote this hardware Hit Status Holding Registers (HSHR) as it is similar to the MSHR. However, the HSHR is private for each core. In each entry, the HSHR needs an identifier of the ROB entry of the access, the address accessed by the L2 hit, the stack distance value and a field with the corresponding MLP_cost, as can be seen in Figure 4.5(b). Figure 4.5(a) shows the HSHR in the cache hierarchy.



(a) HSHR (b) HSHR fields

Figure 4.5: Hit Status Holding Register (HSHR) description

When the L2 cache is accessed and an L2 hit is determined, we assign an HSHR entry to the L2 hit. We initialize the fields of the entry as in the case of the MSHR. We have a stack distance d_i and we want to update the MLP_cost field in every cycle. With this objective, we need to know the number of active entries with stack distance greater or equal to d_i in the HSHR, which can be tracked with one hardware counter per core. We also need a ROB entry identifier for each L2 access. For each cycle, we obtain N_{d_i}, the number of L2 accesses with stack distance greater or equal to d_i as in the L2 MSHR case. We have a hardware counter that tracks this number for each possible number of d_i, which means a total of K counters[3].

In order to avoid array conflicts, we need the same number of entries in the HSHR as possible L2 accesses in flight. This number is equal to the L1 MSHR size. In our scenario, we have 32 L1 MSHR entries, which means a maximum of 32 in flight L2 accesses per core. However, we have checked that we have enough with 24 entries per core to ensure that we have an available slot 95% of the time in an architecture with a

[3]In our CMP baseline, we assume a K-way associative cache

Table 4.1: MLP_cost quantification

MLP_cost	Quantification	MLP_cost	Quantification
From 0 to 42 cycles	0	From 171 to 213 cycles	4
From 43 to 85 cycles	1	From 214 to 256 cycles	5
From 86 to 128 cycles	2	From 257 to 299 cycles	6
From 129 to 170 cycles	3	300 or more cycles	7

ROB of 256 entries. If there are no available slots, we simply assign the minimum weight to the L2 access as there are many L2 accesses in flight. The number of adders required to update the MLP_cost of all entries is equal to the number of HSHR entries. As we did with the MSHR, HSHR entries can share four adders with a negligible effect on the final MLP_cost estimation.

When we examine the **quantification of the MLP_cost**, then dealing with values of MLP_cost between 0 and the memory latency (or even greater) can represent a significant hardware cost. Instead, we decide to quantify this MLP_cost with an integer value between 0 and 7 as was done in [89]. For a memory latency of 300 cycles, we can see in Table 4.1 how to quantify the MLP_cost. We have split the interval $[0; 300]$ with 7 intervals of equal length.

Finally, when we have to update the corresponding MLP-aware SDH, we add the quantified MLP_cost value. Thus, isolated L2 misses will have a weight of 7, while two overlapped L2 misses will have a weight of 3 in the MLP-aware SDH. In contrast, *MinMisses* always adds a constant value (one) to its histograms.

4.2.2 Obtaining Stack Distance Histograms

Normally, L2 caches have two separate parts: one that stores data and one that stores address tags. Tags are used to indicate whether the access is a hit or a miss. Basically, our prediction mechanism needs to track every L2 access and store a separate copy of the L2 tags information in an *Auxiliary Tag Directory* (ATD), together with the LRU counters [90]. We need an ATD for each core that keeps track of the L2 accesses for any possible cache configuration. Independently of the number of ways assigned to each core, we store the tags and LRU counters of the last K accesses of the thread, where K is the L2 associativity. As we have explained in Chapter 3, an access with stack distance d_i corresponds to a cache miss in any configuration that assigns less than d_i ways to the thread. Thus, with this ATD we can determine whether an L2 access would be a miss or a hit in all possible cache configurations.

4.2.3 Putting It All Together

Figure 4.6 shows a sketch of the hardware implementation of our proposal. On an L2 access, the ATD is used to determine the stack distance d_i of such access. Depending on whether it is a miss or a hit, either the MSHR or the HSHR is used to compute the MLP_cost of the access. Using the quantification process, we obtain the final MLP_cost. This number estimates how performance is affected when the application has exactly $w'_i = d_i$ assigned ways. If $w'_i > w_i$, we are estimating the performance benefit of converting this L2 miss into a hit. In case $w'_i < w_i$, we are estimating the performance degradation of converting this L2 hit into a miss. Finally, using the stack distance, the MLP_cost and the core identifier, we can update the corresponding MLP-aware SDH.

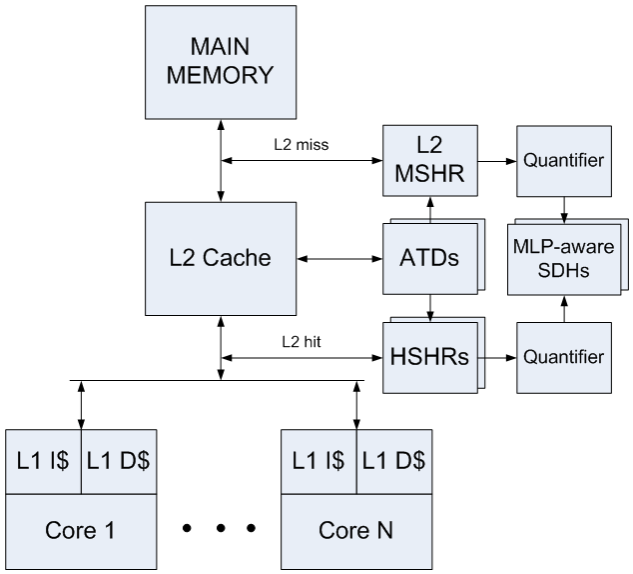


Figure 4.6: Hardware implementation of the MLP-aware monitoring logic

We have used two partitioning algorithms that optimize different target metrics. The first one, which we denote ***MLP-DCP*** (standing for MLP-aware Dynamic Cache Partitioning), decides the optimal partition according to the MLP_cost of each way. We define the *total* MLP_cost (TMLP) of a thread i that uses w_i ways as $TMLP(i, w_i) = MLP\_SDH_{i,>K} + \sum_{j=w_i}^{K} MLP\_SDH_{i,j}$. We denote the total MLP_cost of all accesses of thread i with stack distance j as $MLP\_SDH_{i,j}$. Thus, we have to minimize the sum of total MLP_costs for all cores:

$$\sum_{i=1}^{N} TMLP(i, w_i), \text{ where } \sum_{i=1}^{N} w_i = K$$

The second algorithm consists of assigning a weight to each total MLP_cost using the IPC of the application running in core i, IPC_i. In this situation, we are giving priority to threads with higher IPC. This point will give better results in throughput at the cost of being less fair. IPC_i is measured at run-time with a hardware counter per core. We denote this proposal *MLPIPC-DCP*, which consists of minimizing the following expression:

$$\sum_{i=1}^{N} IPC_i \cdot TMLP(i, w_i), \text{ where } \sum_{i=1}^{N} w_i = K$$

4.2.4 Case Study: `galgel` and `gzip`

We have seen that SDHs can give the optimal partition in terms of total L2 misses. However, minimizing the total number of L2 misses is not the goal of DCP algorithms. Maximizing throughput (or other metrics related to IPC) is the real objective of these policies. The underlying idea of *MinMisses* is that while minimizing total L2 misses, we are also increasing throughput. This idea is intuitive as performance is clearly related to the L2 miss rate. However, this heuristic can lead to suboptimal partitions in terms of throughput, as can be seen in the next case study.

Figure 4.7 shows the IPC curves of benchmarks `galgel` and `gzip` as we increase the L2 cache size in a way granularity (the size of each way is 64KB). This figure also shows the throughput for all possible 15 partitions. In this curve, we assign x ways to `gzip` and $16 - x$ to `galgel`. The optimal partition consists of assigning 6 ways to `gzip` and 10 ways to `galgel`, obtaining a total throughput of 3.091 instructions per cycle. However, *MinMisses* algorithm assigns 4 ways to `gzip` and 12 ways to `galgel` according to the SDHs values. Figure 4.7 also shows the total number of misses for each cache partition as well as the per thread number of misses.

In this situation, misses in `gzip` are more important in terms of performance than misses in `galgel`. Furthermore, `gzip`'s IPC is larger than the one from `galgel`. Consequently, *MinMisses* obtains a suboptimal partition in terms of IPC, with a throughput of 2.897 instructions per cycle, which is 6.3% smaller than the optimal one. In fact, `galgel` clusters of L2 misses are, on average, longer than the ones from `gzip`. In this way, *MLP-DCP* assigns one extra way to `gzip` and increases performance by 3%. If we use *MLPIPC-DCP*, we give more importance to `gzip` as it has a higher IPC and, as a result, we end up assigning another extra way to `gzip`, reaching the optimal partition and increasing throughput an extra 3%.

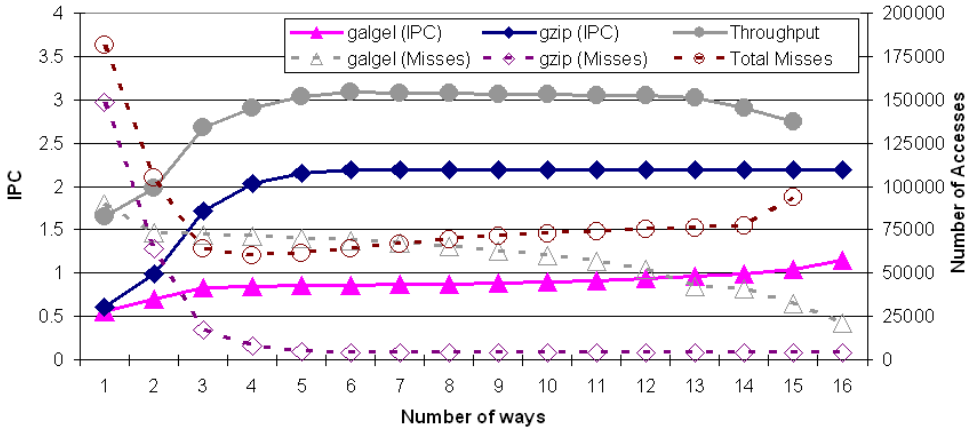


Figure 4.7: Misses and IPC curves for `galgel` and `gzip`

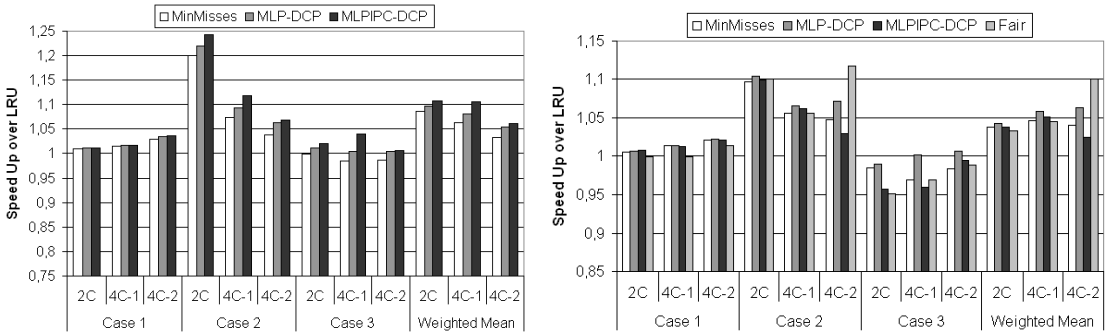
4.3 Evaluation Results

In this section, we evaluate the performance of MLP-aware DCP algorithms for two- and four-core architectures according to the experimental environment explained in Chapter 2.

4.3.1 Performance Results

The first experiment consists of comparing **throughput** results for different DCP algorithms, using LRU policy as the baseline. We simulate *MinMisses* and our two proposals with the 48 workloads that were selected in Section 2.4 for configurations *2C*, *4C-1* and *4C-2*. Figure 4.8(a) shows the average speed up over LRU for these mechanisms. *MLPIPC-DCP* systematically obtains the best average results, nearly doubling the performance benefits of *MinMisses* over LRU in the four-core configurations. In configuration *4C-1*, *MLPIPC-DCP* outperforms *MinMisses* by 4.1%. *MLP-DCP* always improves *MinMisses* but obtains worse results than *MLPIPC-DCP*.

All algorithms have similar results in Case 1. This is intuitive as in this situation there is little room for improvement. In Case 2, *MinMisses* obtains a relevant improvement over LRU in configuration *2C*. *MLP-DCP* and *MLPIPC-DCP* achieve an extra 2.5% and 5% improvement, respectively. In the other configurations, *MLP-DCP* and *MLPIPC-DCP* still outperform *MinMisses* by a 2.1% and 3.6%. In Case 3, *MinMisses* presents larger performance degradation as the asymmetry between the necessities of the two cores increases. Consequently, it has worse average throughput than LRU. Assigning an appropriate weight to each L2 access gives the possibility of obtaining better results than LRU using *MLP-DCP* and *MLPIPC-DCP*.



(a) Throughput speed up over LRU

(b) Fairness speed up over LRU

Figure 4.8: Average performance speed ups over LRU of the different MLP-aware DCP algorithms

Next, we have used the harmonic mean of relative IPCs [68] to measure **fairness**. The relative IPC is computed as $\frac{IPC_{shared}}{IPC_{alone}}$. Figure 4.8(b) shows the average speed up over LRU of the harmonic mean of relative IPCs. *Fair* stands for the policy explained in Section 3.4, the best policy in the literature optimizing fairness.. We can see that in all situations, *MLP-DCP* always improves on both *MinMisses* and LRU (except in Case 3 for two cores). It even obtains better results than *Fair* in configurations *2C* and *4C-1*. *MLPIPC-DCP* is a variant of the *MLP-DCP* algorithm optimized for throughput. As a result, it obtains worse results in fairness than *MLP-DCP*.

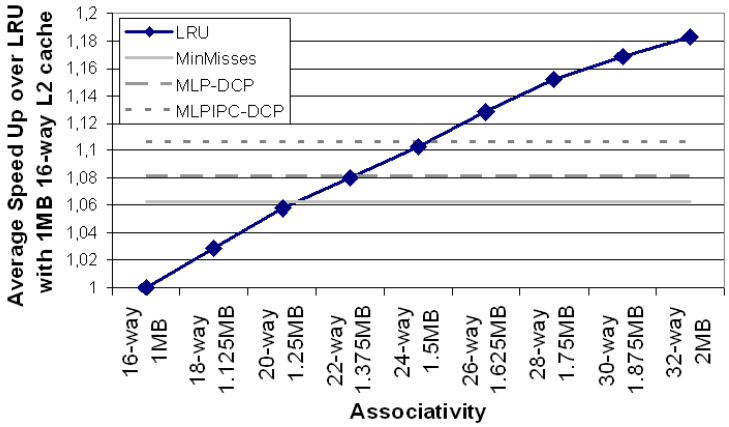


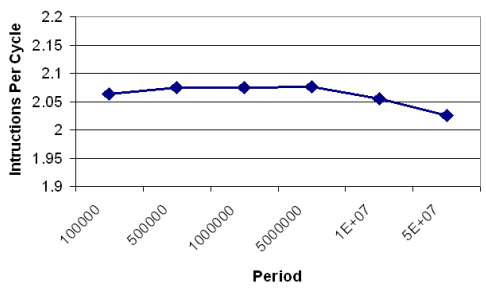
Figure 4.9: Average throughput speed up over LRU of the different MLP-aware DCP algorithms with a 1MB 16-way L2 cache

Regarding the **equivalent cache space**, DCP algorithms obtain the same performance as LRU eviction policy with a larger cache. Figure 4.9 shows the performance evolution when the L2 size increases from 1MB to 2MB with LRU as eviction policy. In

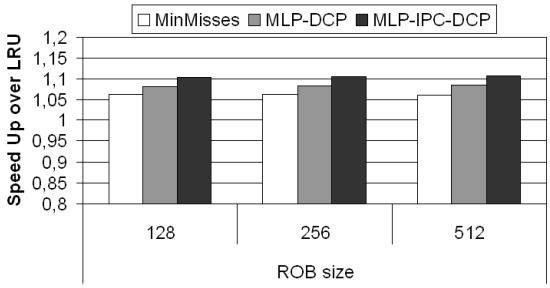
this experiment, the evaluated workloads correspond to those selected for configuration *4C-1*. Figure 4.9 also shows the average speed up over LRU of *MinMisses*, *MLP-DCP* and *MLPIPC-DCP* with a 1MB 16-way L2 cache. *MinMisses* has the same average performance as a 1.25MB 20-way L2 cache with LRU, which means that *MinMisses* provides the same performance as that obtained with a 25% larger shared cache. *MLP-DCP* matches the performance of a 37.5% larger cache. Finally, *MLPIPC-DCP* doubles the increase in size of *MinMisses*, matching the performance of an L2 cache that is 50% larger.

4.3.2 Design Parameters Analysis

Figure 4.10(a) shows the sensitivity of the *MLP-DCP* algorithm to the period of partition decisions. For shorter periods, the partitioning algorithm reacts more quickly to phase changes. Once again, small performance variations are obtained for different periods. However, we observe that for longer periods throughput tends to decrease, because wrong decisions are very costly over a longer period. As can be seen in Figure 4.10(a), peak performance is obtained with a period of 5 million cycles.



(a) Average throughput for different periods for the *MLP-DCP* algorithm with the *2C* configuration



(b) Average speed up over LRU for different ROB sizes with the *4C-1* configuration

Figure 4.10: Sensitivity analysis to different design parameters of the different MLP-aware DCP algorithms

Finally, we varied the size of the ROB from 128 to 512 entries to show the sensitivity of our proposals to this parameter of the architecture. Our mechanism is the only one which is aware of the ROB size: the higher the size of the ROB, the larger the size of the cluster of L2 misses. Other policies only work with the number of L2 misses, which will not change if we vary the size of the ROB. When the ROB size increases, clusters of misses can contain more misses and, as a consequence, our mechanism can differentiate

better between isolated and clustered misses. As we show in Figure 4.10(b), average improvements in the *4C-1* configuration are a little higher for a ROB with 512 entries, while *MinMisses* shows worse results. *MLPIPC-DCP* outperforms LRU and *MinMisses* by 10.4% and 4.3% respectively.

4.3.3 Hardware Cost

We used the hardware implementation of Figure 4.6 to estimate the hardware cost of our proposal. In this subsection, we focus on configuration *2C*. We assume a 40-bit physical address space. Each entry in the ATD needs 29 bits (1 valid bit + 24-bit tag + 4-bit for LRU counter). Each set has 16 ways, so we have an overhead of 58 bytes (B) for each set. As we have 1024 sets, we have a total cost of 58KB per core.

The hardware cost that corresponds to the extra fields of each entry in the L2 MSHR is 5 bits for the stack distance and 2 bytes for the MLP_cost. As we have 32 entries, we have a total of 84 bytes. Four adders are needed to update the MLP_cost of the active MSHR entries. HSHR entries need 1 valid bit, 8 bits to identify the ROB entry, 34 bits for the address, 5 bits for the stack distance and 2 bytes for the MLP_cost. In total we need 64 bits per entry. As we have 24 entries in each HSHR, we have a total of 192 bytes per core. Four adders per core are needed to update the MLP_cost of the active HSHR entries. Finally, we need 17 counters of 4 bytes for each MLP-Aware SDH, which supposes a total of 68 bytes per core. In addition to the storage bits, we also need an adder for incrementing MLP-aware SDHs and a shifter to halve the hit counters after each partitioning interval.

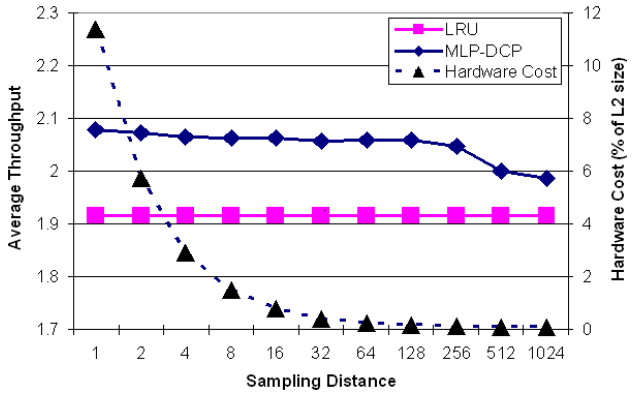


Figure 4.11: Throughput and hardware cost depending on d_s in a two-core CMP

The main contribution to hardware cost corresponds to the ATD. Instead of monitoring every cache set, we can decide to track accesses from a reduced number of sets with a

sampled ATD. This idea was also used by Qureshi et al. [90] with *MinMisses* in a CMP environment. Here, we use it in a different situation, to estimate MLP-aware SDHs with a sampled number of sets. We define the *sampling distance* d_s that gives the distance between tracked sets. For example, if $d_s = 1$, we are tracking all the sets. If $d_s = 2$, we track half of the sets, and so on. Sampling reduces the size of the ATD at the expense of less accuracy in MLP-aware SDHs predictions, since some accesses are not tracked. Figure 4.11 shows the throughput degradation in a two-core scenario as the d_s increases. This curve is measured on the left y-axis. We also show the storage overhead in percentage of the total L2 cache size, measured on the right y-axis. Thanks to the sampling technique, storage overhead drastically decreases. Thus, with a sampling distance of 16 we obtain average throughput degradations of 0.76% and a storage overhead of 0.77% of the L2 cache size, which is less than 8KB of storage. We think that this is an interesting design point.

4.3.4 Scalable Algorithms to Decide Cache Partitions

By evaluating all possible combinations, the algorithm can determine the optimal partition for the next period. However, this algorithm does not scale adequately when associativity and the number of applications sharing the cache is raised (see Section 3.5 for more details). Consequently, the time to decide new cache partitions does not adequately scale. Several heuristics have been proposed to reduce the number of cycles required to decide the new partition [90, 108], which can be used in our situation. These proposals bound the length of the decision period by $10,000$ cycles. This overhead is very low compared to 5 million cycles (less than 0.2%). We also evaluated the version of *EvalAll* that makes use of dynamic programming techniques [32].

Figure 4.12 shows the average speed up of *MLP-DCP* over LRU with the *4C-1* configuration with three different decision algorithms. Evaluating all possible partitions (denoted *EvalAll*) gives the highest speed up. *Marginal gains* assigns one way to a thread in each iteration [108]. The selected way is the one that gives the largest increase in MLP_cost. This process is repeated until all ways have been assigned. The number of operations (comparisons) is of order $K \cdot N$, where K is the associativity of the L2 cache and N the number of cores. With this heuristic, an average throughput degradation of 0.59% is obtained. The *look ahead* algorithm is similar to *marginal gains*. The basic difference between them is that *look ahead* considers the total MLP_cost for all possible number of blocks that the application can receive [90] and can assign more than one way

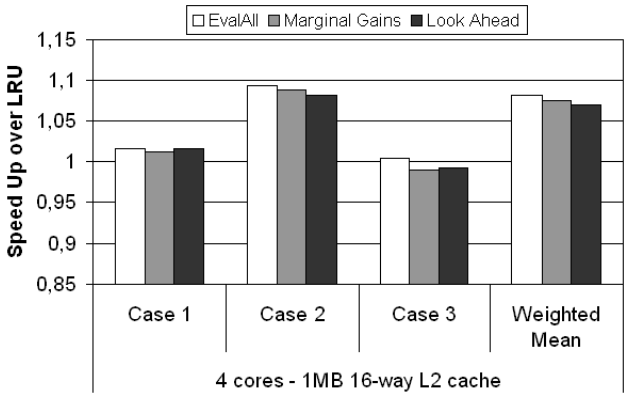


Figure 4.12: Average throughput speed up over LRU for different decision algorithms in the *4C-1* configuration

in each iteration. The number of operations (add-divide-compare) is of order $\frac{K^2 \cdot N}{2}$, where K is the associativity of the L2 cache and N the number of cores. With this heuristic, an average throughput degradation of 1.04% is obtained. These results are consistent with those obtained with the *MinMisses* algorithm discussed in Section 3.5.

4.4 Summary

In this chapter we propose new DCP algorithms that assign a cost to each L2 access according to its memory-level parallelism: isolated misses receive higher costs than clustered misses. Next, our algorithm decides the L2 cache partition that minimizes the total cost for all running threads.

We show that our proposal reaches high throughput for two- and four-core architectures. In all evaluated configurations, our proposal consistently outperforms both LRU and *MinMisses*, reaching a speed up of 63.9% (10.6% on average) and 15.4% (4.1% on average), respectively. With our proposals, we match the performance of a 50% larger cache. Next, we make use of a sampling technique to propose a practical implementation with a storage cost that supposes less than 1% of the total L2 cache size. Finally, we evaluate different scalable algorithms to determine cache partitions with nearly no performance degradation.

Chapter 5

Online Prediction of Applications Cache Utility

General purpose processors are designed to offer high average performance regardless of the particular application that is being run. As a result, performance and power inefficiencies appear for some programs. Reconfigurable hardware (cache hierarchy, branch predictor, execution units, bandwidth, etc.) has been proposed to overcome these inefficiencies by dynamically adapting the architecture to the application requirements. However, previous work normally used indirect measures, or performance heuristics to guide hardware reconfigurations, which often led to suboptimal decisions. This is the case of the mechanisms presented above.

In this chapter we propose a run-time mechanism which involves predicting the performance of an application running on an architecture with a reconfigurable L2 cache at a way granularity. In this scenario, the number of ways assigned to an application can be periodically changed. Thus, when an application runs with a given number of assigned ways, the hardware mechanism predicts the performance of the same application on all other possible L2 cache configurations. We obtain for different L2 cache sizes an average error of 3.11%, a maximum error of 16.4% and standard deviation of 3.7%. No profiling or collaboration with the operating system (OS) is needed in this mechanism. We also give a hardware implementation that makes it possible to reduce the hardware cost in terms of area to under 0.4% of the total L2 cache size, while maintaining high accuracy. This prediction mechanism can be used to reduce power consumption in single-threaded architectures and also to improve performance in multithreaded architectures that dynamically partition shared L2 caches.

5.1 Introduction

The problem of dynamically adapting resources to program requirements does not only apply to multithreaded architectures. Several mechanisms have been proposed in superscalar architectures to use reconfigurable hardware that adapts microarchitecture features to different program phases [4, 11, 12, 33, 56]. The common problem with all these self-tuning techniques is that decisions are based on indirect performance metrics or empirical heuristics. For instance, Bahar et al. [11] dynamically adjust the issue width and the number of execution units in an out-of-order architecture depending on the number of issued instructions. This solution reduces power consumption at the cost of a reduced performance. However, in some situations the power consumption reduction is not enough to compensate for the performance loss, resulting in energy losses in these situations.

In this chapter we focus on architectures with a dynamically reconfigurable cache hierarchy. In particular, we propose a mechanism that allows us to predict with high accuracy the Instruction Per Cycle (IPC) of the application as we vary the amount of cache devoted to it. The L2 cache size is changed by activating/deactivating some ways of a set associative cache. This mechanism combines *Stack Distance Histograms* (SDHs) [71] and an analytical model for predicting processor IPC introduced by Karkhanis et al. [55]. This model has to be adapted to the particular target scenario in order to increase the accuracy of the IPC predictions. On average, for all SPEC CPU 2000 benchmarks, our mechanism obtains an average error of 3.11%, with a maximum error of 16.4% (with `twolf` benchmark) and a standard deviation of 3.7%. The ability to predict IPC as we change the cache configuration can be applied in two different scenarios:

First, **cache sharing in multithreaded architectures**. A better sharing of the L2 cache among the running threads can be obtained using cache partitioning techniques. Previous work proposed static and dynamic cache partitioning algorithms in chip multiprocessors (CMP) and simultaneous multithreading (SMT) architectures in order to maximize throughput or fairness [29, 58, 90, 99, 109]. These proposals used indirect metrics of performance, such as the total number of misses, or data re-use, to predict the best cache partition. The mechanism proposed in this chapter provides direct estimations of performance for different cache configurations, which is the appropriate metric to maximize total throughput, fairness or other IPC-related metrics. Chapter 6 describes a framework that makes use of the IPC predictions obtained with this mechanism to partition a shared cache among different applications.

The second scenario is **power reduction**. A reduction in the power dissipated in the cache can be obtained by adjusting the hardware resources to the requirements of the applications. By having a direct estimation of the performance of the application, it is possible to obtain the desired trade-off between power consumption and performance. Previous work considered statically switching on or off L2 cache ways [4] or switching off lines after a number of cycles without being accessed [56]. However, these proposals cannot bound performance losses and rely on empirical heuristics. Giving the real contribution of each way to the final IPC allows us to bound the performance degradation while saving power.

In both scenarios, previous work relied on empirical heuristics and thresholds to make decisions. To our knowledge, we are the first to mix run-time measurements with analytical models to dynamically predict the actual impact on performance of such decisions. The main contributions of this chapter are:

1) A run-time mechanism to predict IPC for different cache configurations with high accuracy. This proposal can help to reconfigure L2 caches in CMP/SMT scenarios for dynamic cache partitioning and to reduce power consumption. The important difference between our mechanism and previous work in these areas is that, with our mechanism, new configurations are based on real estimations of IPC instead of indirect measures or ad hoc heuristics.

2) A modified version of the memory model in [55] that makes it possible to predict the cost on performance of an L2 miss with high accuracy.

3) A sampling technique to reduce hardware cost in terms of area under 0.4% of the total L2 cache size without significantly affecting the accuracy of the prediction mechanism (the average error raises to 4%).

The rest of this chapter is structured as follows. Section 5.2 introduces the methods used to predict IPC curves and Section 5.3 shows how to combine them to obtain IPC predictions. Next, Section 5.4 presents the simulation results. Section 5.5 deals with a practical implementation in hardware as well as a sampling technique to reduce its cost. Next, Section 5.6 reports related work and, finally, Section 5.7 summarizes the chapter.

5.2 Basis of IPC Curves Prediction

We define the IPC curve of an application as the evolution of the IPC values that the application obtains for different configurations of the L2 cache. These configurations

have a *way* granularity. Thus, in a K-way L2 cache, IPC curves have exactly K points (since we assume that at least one way is assigned to the application). Naturally, IPC curves monotonically increase as the IPC of an application increases, since more cache space is devoted to it (until it saturates to its maximum IPC). In order to accurately predict IPC curves, we combined two instruments: the stack distance histogram (SDH) [71], and an analytical model for superscalar processors performance [55]. In Chapter 3, the mechanism for obtaining the SDH of an application was explained in detail. Next, we describe the analytical model.

5.2.1 Superscalar Processors Analytical Modeling

Karkhanis et al. [55] propose a model that estimates the performance of an application running on a superscalar processor. This model computes an ideal Cycle Per Instruction (CPI) when no misses occur and adds CPI penalties for each type of possible hazard, including branch mispredictions, instruction cache misses and data cache misses. Some assumptions and simplifications are made in this model to make it manageable, but simulations prove that it has high accuracy. Compared to detailed simulation, errors are within 5.8% on average and within 13% in the worst case.

We make use of this analytical model to predict the performance of a superscalar processor as we vary the L2 cache size. In our scenario, we can assume that the ideal CPI is independent of the cache configuration, as it only depends on data dependencies of the particular application. We further assume that the branch miss penalty remains constant for different cache sizes. Thus, we are only interested in using the part of the model that concerns the cache hierarchy. The model considers L2 instruction and data misses separately.

With regard to **instruction misses**, initially, the processor issues instructions at the steady-state IPC. When an instruction L2 miss occurs, instructions in the issue queue and the front-end pipeline maintain issue rate for some cycles, but when the issue queue drains, the issue rate drops to zero following a linear descend [74]. After a miss delay, ΔI, instructions are delivered from main memory. Then, IPC ramps up to its steady-state value. Karkhanis et al. [55] show that lost cycles until steady-state IPC is attained compensate useful cycles until the issue queue drains. Thus, the penalty for an isolated instruction cache misses is approximately equal to the main memory latency. Furthermore, as instruction cache misses are serialized, each miss in a burst of consecutive instruction cache misses has the same penalty as an isolated one: ΔI cycles.

Turning now to **data misses**, the basic difference between instruction and data cache misses is that instruction fetch and issue continue after the data cache miss, and so several data misses can occur in parallel. After ΔD cycles, data is delivered from main memory. In [55] it is shown that the miss penalty for an isolated data L2 miss can be approximated by ΔD.

When we have a burst of n L2 cache data misses, they will overlap if they all fit in the reorder buffer (ROB) and are independent. In this situation, the miss penalty of ΔD cycles is shared among all misses. Then, given that M_D is the total number of L2 data misses and N_i the number of times that we have a burst of i misses that fit in the ROB, we can approximate the *average L2 data miss penalty* (avgDMP) with the following formula:

$$avgDMP = \frac{1}{M_D} \cdot \sum_{i=1}^{ROBsize} N_i \cdot \Delta D \tag{5.1}$$

5.3 Prediction of IPC Curves

In this section, we detail how to obtain a prediction of the IPC curve of an application. We call this methodology Online Prediction of Applications Cache Utility (OPACU). We use SDHs to compute the number of misses for each possible L2 cache size, and the memory model described in the previous section to determine the miss penalty of each L2 miss.

5.3.1 OPACU Methodology

In the baseline configuration, the L2 cache has a variable number of active ways. We start by assigning w ways to an application and measuring its IPC during C cycles. This value is denoted $IPC_{real,w}$, with $IPC_{real,w} = \frac{I}{C}$, where I is the number of committed instructions in this period. This IPC value is only valid for the particular number of ways that are being used (w).

Thanks to the SDH, we know whether an access would be a miss or a hit with a different number of ways $w' \in [1, K]$. Independently of the number of active ways, we store the LRU counters of the last K accesses of the thread and obtain the SDH for the whole K-way associativity L2 cache, as explained in Chapter 3. Next, using the analytical model explained in Section 5.2, we can estimate at run-time the number of times that a burst of i L2 data misses occurs with an L2 cache with w' ways. This number is denoted $N_i^{w'}$ for $1 \leq i \leq$ ROB size, and for any possible number of ways $w' \in [1, K]$. The total number of instruction and data L2 misses are denoted $M_I^{w'}$ and $M_D^{w'}$, respectively.

These values are obtained at run-time using the hardware explained in Section 5.5. Using $N_i^{w'}$, we can compute the average L2 data miss penalty when w' ways are being used, $avgDMP^{w'}$, with Formula 5.1. Thus, we can estimate the variation in cycles of the *total miss penalty* for the new configuration due to data and instruction L2 misses, $\Delta C_D^{w,w'}$ and $\Delta C_I^{w,w'}$ respectively.

$$\Delta C_D^{w,w'} = avgDMP^{w'} \cdot M_D^{w'} - avgDMP^{w} \cdot M_D^{w} \tag{5.2}$$

$$\Delta C_I^{w,w'} = (M_I^{w'} - M_I^{w}) \cdot \Delta I \tag{5.3}$$

The value $\Delta C^{w,w'} = \Delta C_I^{w,w'} + \Delta C_D^{w,w'}$ may be positive or negative depending on the values of w and w'. Thus, the IPC when using w' ways, denoted $IPC_{pred,w'}$, can be predicted with the following formula.

$$IPC_{pred,w'} = \frac{I}{C + \Delta C^{w,w'}} \tag{5.4}$$

To illustrate this technique, we chose the `vortex` benchmark from SPEC CPU 2000 suite. Table 5.1 shows the monitored values when using a cache configuration with 16 ways, with just 7 active ways. The measured IPC, $IPC_{real,7}$, is 1.52 instructions per cycle. Using Formula 5.4, the IPC for a configuration with 16 active ways, $IPC_{pred,16}$, can be predicted, obtaining a value of 1.637 instructions per cycle, which is very close to the real value of 1.647. The relative error of this prediction is 0.65%. In this example, we assume $\Delta D = \Delta I = 250$ cycles.

Table 5.1: IPC prediction when moving from 7 to 16 active ways for `vortex`

I	C	w	w'	$\Delta C^{w,w'}$	ΔD
272M	178M	7	16	-12.5M	250

5.3.2 Modified Memory Model

We have modified the memory model introduced by Karkhanis et al. [55] to increase the accuracy of OPACU IPC predictions. The original model assumed that two L2 data misses overlap if their ROB distance (in number of instructions) is less than the ROB size. One of the main assumptions of the model is that the ROB fills after an L2 data cache miss. However, this is not always true. To illustrate this point, we measured the average ROB occupancy after an L2 data miss is serviced from main memory and commits for a ROB with 256 entries. This value varies depending on the application and is always

less than the size of the ROB, as can be seen in Figure 5.1. Some benchmarks, such as `mcf` or `sixtrack`, have less than 50% of the ROB occupied when the load commits, while other benchmarks, such as `art` or `mesa`, have the ROB almost full. On average, 184 entries out of 256 are occupied when the load commits. Thus, L2 cache misses will overlap if their ROB distance is less than this measured value[1]. This approximation works better than using the ROB size as a constant value.

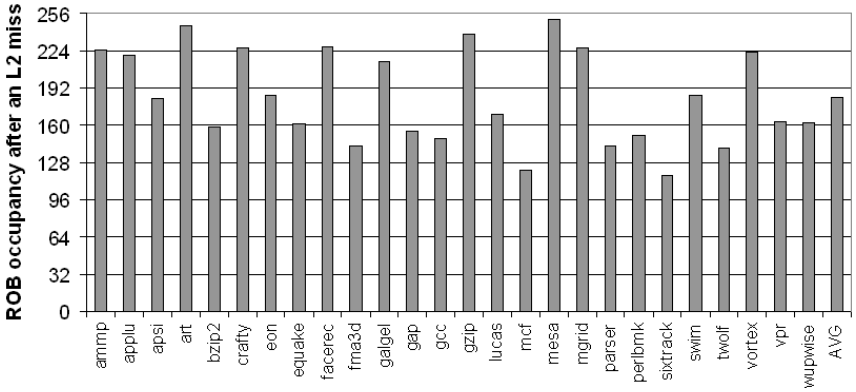


Figure 5.1: Average ROB occupancy after an L2 miss commits

The real problem is that the ROB is not always the bottleneck for performance. Sometimes issue queues are full with dependent instructions on the missing load, causing fetch and issue to stall. Thus, it is more representative to use the average ROB occupancy after an L2 miss to determine if two L2 misses overlap. This value is easily obtained at runtime with a hardware counter. This intuition is confirmed by the simulations described in Section 5.4. This improvement decreases the average and maximum errors from 3.34% and 20.9% (with the memory model by Karkhanis et al. [55]) to an average and maximum errors of 3.11% and 16.4%, respectively.

5.4 Evaluation Results

In this section we show the accuracy results of OPACU IPC prediction mechanism, as well as a sensitivity analysis to different processor parameters. In order to evaluate the accuracy of the IPC predictions obtained with OPACU, we evaluate that mechanism on a single core superscalar processor. The processor configuration is the same as the one explained in Chapter 2, but with just one core (see Table 2.1 for further details). Recall that the configuration has a unified 16-way 1MB L2 cache.

[1] We consider that this value remains constant when varying the L2 cache size.

We simulated all SPEC CPU 2000 benchmarks with all possible L2 cache sizes (from 1 way to 16 ways assigned to the application) until completion. At the end of each simulation, OPACU predicts the performance of the same application with all the other possible L2 cache sizes, as described in Section 5.3. For example, when running with 2 ways assigned to the application, OPACU predicts the performance for $1, 3, \ldots, 16$ assigned ways (the entire IPC curve for that particular application).

Subsequently, we can compare the measured IPC (in a previous simulation) with the prediction obtained with OPACU. For each IPC prediction, denoted $IPC_{pred,w'}$, of a real IPC, denoted $IPC_{real,w'}$, we measure the relative error as:

$$rel\_error = 100 \cdot \frac{|IPC_{pred,w'} - IPC_{real,w'}|}{IPC_{real,w'}}$$

Finally, we compute the arithmetic mean of all the relative errors. Recall that the arithmetic mean $\bar{x}$ of L numbers $x_1, \ldots, x_L$ is $\bar{x} = \frac{1}{L} \cdot \sum_{i=1}^{L} x_i$.

5.4.1 Accuracy Results

Figure 5.2 shows the average relative error for each SPEC CPU 2000 benchmark. Benchmarks belonging to the same group (L, S, H) are shown together. Overall, the average prediction error is 3.1%. It is important to note that average error is consistent across benchmarks of the same type. On average, L and S benchmarks have lower error than H benchmarks, reaching a maximum error of 16.4% in case of `twolf`.

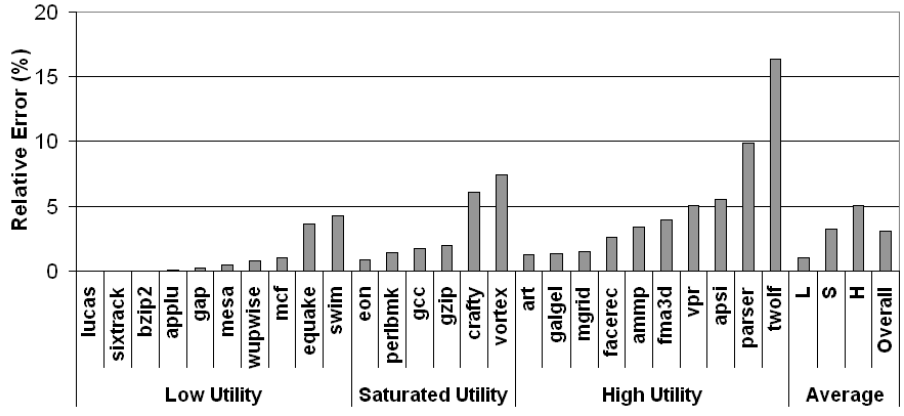
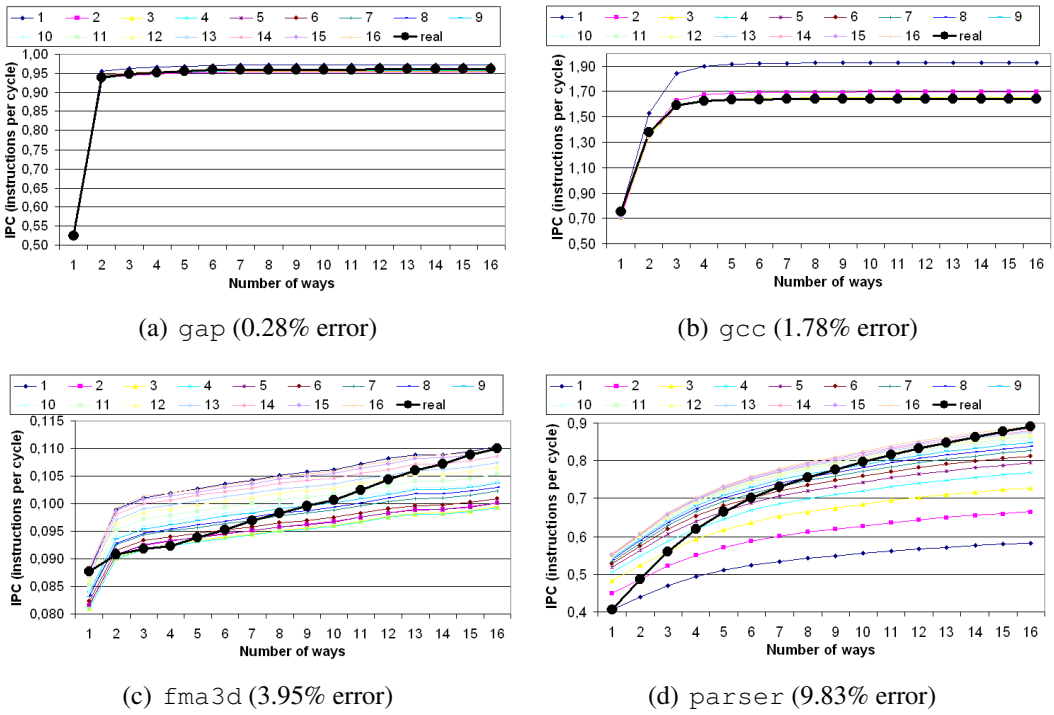


Figure 5.2: OPACU mean relative error for all SPEC CPU 2000 benchmarks

Figure 5.3 shows detailed IPC prediction results for four benchmarks representative of different error ranges. Each figure shows the IPC curves predicted when running with

a constant number of active ways (denoted i in the figure). It also shows the measured real IPC value (denoted *real*). The closer a predicted IPC curve to the real IPC curve is, the smaller relative errors are obtained.



(a) `gap` (0.28% error)

(b) `gcc` (1.78% error)

(c) `fma3d` (3.95% error)

(d) `parser` (9.83% error)

Figure 5.3: Real and predicted IPC curves for `gap`, `gcc`, `fma3d` and `parser`

In the case of `gap` (Figure 5.3(a)), the average prediction error is 0.28%. We can see that, in this situation, predictions are extremely accurate for any number of assigned cache ways. Figure 5.3(b) shows the predicted IPC curves for `gcc`, which has an average relative error of 1.78%. In this situation, predictions, when running with a low number of ways, overestimate IPC curves. Figure 5.3(c) shows the predicted IPC curves for `fma3d`, which has an average relative error of 3.95%. Here, predictions show small errors consistent for every value of active ways. In all cases, the shape of the curve remains close to the real one. Finally, Figure 5.3(d) shows predictions for `parser`, which has an average relative error of 9.83%. In this situation, predictions are inaccurate for all values of active ways because the impact of L2 misses is underestimed in IPC predictions. Note that IPC curves axis are scaled so that prediction errors are easier to see.

With regard to **global error per benchmark group**, it is significant that all benchmark groups (H, S and L) have similar relative errors. As a result, we see that H benchmarks present an average relative error of 5.1%, while S benchmarks present lower errors (3.3%

on average) and L benchmarks have negligible errors (1.1% on average). These results are intuitive, as benchmarks that are memory bound present more IPC variability and, as a consequence, predictions are less accurate.

Regarding **error per cache size**, Figure 5.4 shows the average IPC prediction error for a given number of assigned ways (denoted *Mean*). For instance, the average error for predicting performance with 2-16 ways when running with 1 way is 8%. This figure also depicts the average error in each of the three groups of benchmarks (H, S and L). When running with 1 assigned way, the average error in group L is reduced to 2%.

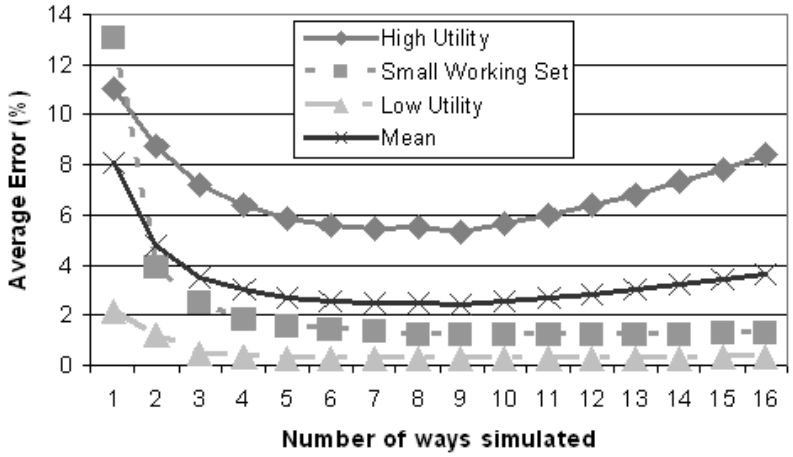


Figure 5.4: Average relative error for groups H, S and L when running with a given number of assigned ways

Our results show that the average error is higher when running with a very small or large number of assigned ways. In fact, if we have a number of assigned ways between 4 and 12, the average relative error is under 3%. When running with few assigned ways, the highest errors are obtained when predicting IPC for configurations with a large number of assigned ways. The same happens the other way round. This situation is intuitive, since we are trying to predict the IPC for caches up to 16 times larger or smaller. The figure shows a peak error for the smallest L2 cache configuration. In this particular case, we are predicting the performance of a highly associative cache based on results of a 64KB direct mapped L2 cache and 32K L1 caches. This unusual setup naturally leads to high prediction errors.

We observed that H benchmarks present less accuracy when they are executing with just one or two assigned ways. The same happens when S benchmarks are using just one way. However, this situation is unlikely to happen. On the one hand, in a low power

scenario that seeks to reduce power by switching off some cache ways without losing performance, as these benchmarks satisfy $w_{90\%} > 2$, then these benchmarks will have at least 3 active ways (they would use exactly $w_{90\%}$ active ways if the mechanism allowed only 10% performance degradation). On the other hand, in a CMP scenario with a partitioned cache, S and H benchmarks should always receive 2 or 3 ways to improve overall performance. To illustrate this point, we have done an experiment in a two core CMP architecture with configuration *2C* (see Section 2.4 for further details). Using a partitioning mechanism that minimizes the total number of L2 misses, such as *MinMisses* [90], S benchmarks receive more than one way in 92% of the decisions on average, while in the case of H benchmarks, this happens in 96% of all decisions. H benchmarks receive more than two ways in 91% of the decisions. Thus, in these situations, the mean IPC prediction error would be even lower than the reported 3.11%.

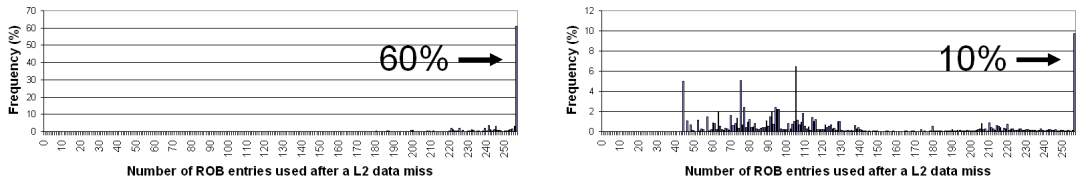
With regard to **outliers**, the highest errors among benchmarks are obtained for `parser` and `twolf`. These benchmarks do not satisfy the approximation that the miss penalty for an isolated data L2 miss is approximately ΔD. If there are instructions before the L2 miss in the pipeline, the processor can do some useful work while waiting for data to come from memory. This useful work is not easy to measure and even harder to predict for different sizes of L2. This observation was also made by Karkhanis et al. [55].

Another source of inaccuracy is the use of a constant value of ROB occupancy after a data L2 miss commits. It is clear that when the ROB size is the bottleneck, the ROB will nearly always fill when an L2 miss occurs. This is exactly the case for `art`, which is the H benchmark with the fewest errors. Figure 5.5(a) shows the histogram of `art`'s ROB occupancy after an L2 miss commits: 60% of the time the ROB is completely full and 96% of the the time there are more than 200 entries occupied in the ROB. However, in the case of `twolf`, the ROB occupancy after a data L2 miss commits varies considerably, and the mean value is less representative of overlaps (see Figure 5.5(b)). Thus, when the ROB is not the main bottleneck for performance, we obtain higher errors in predictions.

5.4.2 Sensitivity Analysis

In this section, we measure the sensitivity of OPACU methodology to several processor parameters. We perform six different studies varying a particular processor parameter while all the other parameters remain constant. The first two parameters are related to the cache hierarchy, while the last four are related to the analytical model.

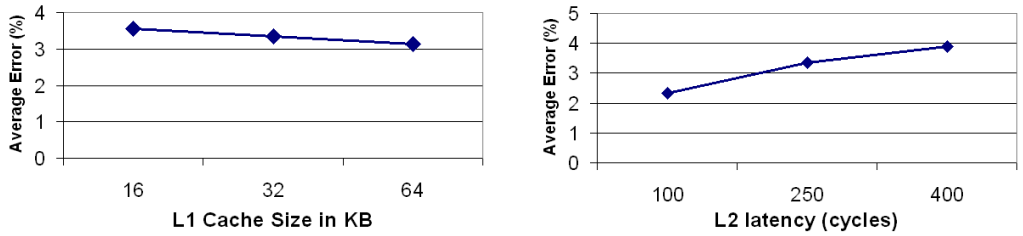
First, we examine **data and instruction L1 cache size**. We vary the L1 cache size



(a) `Art` (b) `Twolf`

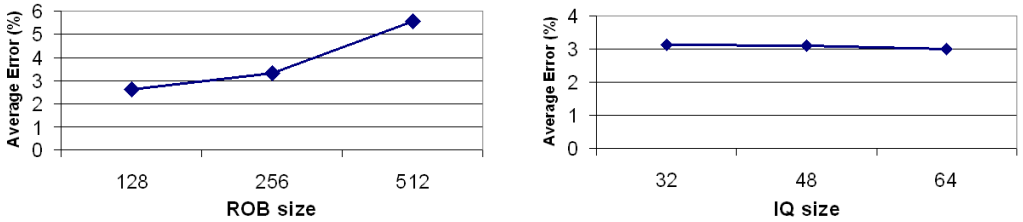
Figure 5.5: ROB occupancy after a data L2 miss commits for `art` and `twolf`

from 16KB to 64KB, while keeping the associativity constant. When the L1 cache size varies, the number of L2 accesses also varies. Larger L1 caches lead to less L2 accesses. A new hit to the L1 instruction or data cache would probably be a hit in L2 if the whole L2 cache is active. However, if just some ways of the L2 are active, this L1 hit could have been a miss in the L2 due to its low associativity. Thus, it is intuitive that the mean relative error will decrease with L1 cache size. Figure 5.6(a) depicts the evolution of the mean relative error depending on the L1 cache size, confirming this intuition.



(a) Average relative error depending on the L1 cache size

(b) Average relative error depending on the latency from L2 to memory



(c) Average relative error depending on the ROB size

(d) Average relative error depending on the issue queue size

Figure 5.6: Sensitivity analysis to different processor parameters. Only one parameter is changed in each experiment, remaining the rest of the processor parameters constant

Second, we measure the sensitivity to the **latency from L2 to memory**. In this experiment we vary the latency to memory from 100 cycles to 400 cycles, while keeping

other parameters constant. In this situation, the average relative error increases according to the latency from L2 to memory. When this latency increases, the contribution of L2 misses to the total CPI becomes more significant and, consequently, the error increases. Figure 5.6(b) shows the evolution of the mean relative error.

Next, we turn to **ROB size**. In this study, we vary the ROB size from 128 to 512 entries. Figure 5.6(c) shows that for smaller ROB sizes, the mean relative error decreases, while the opposite occurs for greater ROB sizes. In our model, we expect the ROB size to be the main bottleneck after a data L2 miss. When the ROB is larger, there is a greater chance that the issue queue becomes the bottleneck. Thus, the mean ROB occupancy after an L2 miss becomes less representative of overlaps. When the ROB is smaller, ROB occupancy after an L2 miss commits is concentrated around the full ROB size.

Fourthly, we examine **issue queues size**, and here, we vary the size of the issue queues from 32 to 64 entries. Although it does not appear directly in the analytical model, it is clear that smaller issue queues lead to more conflicts, becoming the main bottleneck instead of the ROB. Consequently, we have larger errors in IPC predictions. In Figure 5.6(d) we can see the evolution of the mean relative error.

With regard to the **interaction between branch mispredictions and cache misses**, we observed that in some benchmarks many wrong path instructions are executed (for example, in `twolf`, 40% of the fetched instructions are from the wrong path). Thus, we check the assumption that there is no interaction between L2 misses and branch mispredictions. We ran all SPEC CPU 2000 benchmarks with a perfect branch predictor and the mean relative errors result in approximately the same values. Hence, this hypothesis is confirmed to be correct in this environment.

Finally, we examine the **interaction between instruction and data cache misses**. We observed that in some benchmarks, the number of instruction misses in the L1 cache is high (for example, in the case of `crafty`). It is important to establish whether the interaction between instruction misses and data misses is accurately modeled or not. To test this assumption, we considered a configuration with a perfect L1 instruction cache, and we found that simulations results make no substantial differences in average errors. Thus, instruction and data misses can be treated separately.

5.5 Hardware Implementation

This section introduces a possible implementation of the OPACU mechanism to predict IPC values. There are two main hardware components: First, we need a set of counters to establish the number of committed instructions, the number of cycles and the average ROB occupancy after an L2 miss commits. Second, we need some hardware to track L2 accesses, as previous proposals have already shown [90, 109]. For each cache configuration[2] we use specific hardware for three tasks, namely: to determine at run-time if two L2 accesses overlap, to count the number of bursts of overlapped L2 misses, and to predict IPC values.

Concerning **Overlap Counters**, for each possible cache configuration, there is a counter $overlap_w$ that counts the number of bursts of overlapped L2 misses. When we have a miss, we need to know if this miss is overlapped with previous misses. According to our memory model, instruction L2 misses do not overlap. Thus, bursts are always of only one miss. In case of an instruction L2 miss with stack distance i, we can directly increase the counter $overlap_i$. In case of a data L2 miss, we need to know the average ROB occupancy after an L2 miss commits, in order to establish whether this particular L2 miss overlaps with previous ones or not. This information can be tracked with a hardware counter denoted *AROAL2M* (Average ROB Occupancy After an L2 Miss commits). This counter is updated every time a served L2 miss commits. This value can be obtained with some cycles of latency, as this latency is not crucial because the mean value should not vary too much in a period.

A **Countdown Counter** is required for the second task. When a new non-overlapped L2 miss with stack distance i appears, we set a *countdown counter* to the value of AROAL2M. This counter, called cdc_i, is different for each L2 cache size. Each time an instruction is committed, we decrease the counters, which saturate to zero. Thus, when a new L2 miss with stack distance i appears, if $cdc_i \neq 0$, then it overlaps with the previous L2 miss. Notice that we can count committed instructions that are actually prior to the miss. However, this is not a big problem as the number of prior instructions in the ROB when an L2 miss occurs is normally small [55]. In any case, if we want to check that the committed instruction is before the data L2 miss, we can add an instruction identifier that is assigned at fetch time and sorts instructions.

As to the third task, **Predicted Cycle Variation Counters** are used. We need a

[2]We work at way granularity. Hence, we have as many configurations as the L2 cache associativity.

counter for each cache configuration that stores the variation in total miss penalty due to L2 misses. This counter is denoted PCV_w for a cache configuration with w ways.

Two more components need describing. We need a hardware counter that gives the total number of instructions committed in a fixed period of cycles, denoted *Icounter*. The amount of cycles elapsed since we began tracking L2 accesses must be also stored in a counter denoted *Ccounter*.

The final component is the **Auxiliary Tag Directory (ATD)**. Normally, caches have two separate parts that store data and address tags to determine if the access is a hit. Basically, the prediction mechanism needs to track every L2 access, and store a separate copy of the L2 tags information in an ATD, coupled with the LRU counters. In a 16-way associative cache, 4 bits are needed to encode the stack distance. As was described in Section 3.3, an access with stack distance larger than the cache associativity corresponds to a cache miss. Thus, the ATD helps to determine whether an L2 access would be a miss or a hit in the 16 possible cache configurations.

Figure 5.7 sketches a diagram of the hardware implementation described in this section. The main contributor to the total hardware cost is the ATD, for which we propose to use a sampled version, as we did in the previous chapter.

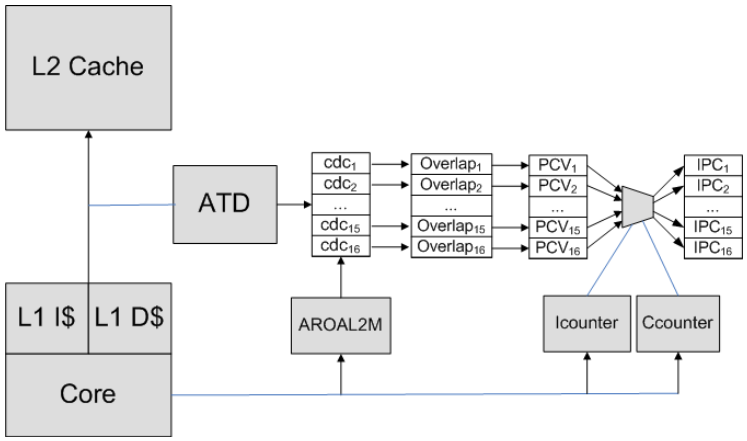


Figure 5.7: Hardware implementation of OPACU prediction mechanism

With regard to the **Sampled ATD**, instead of monitoring every cache set, we can decide to track accesses from a reduced number of sets. This idea was also used by Qureshi et al. [90] in a CMP environment to determine the cache partition that minimizes the total number of misses using a sampled number of sets. In our work, we use sampling in a different situation to predict IPC with a sampled number of sets. We define a *sampling distance* d_s that gives the distance between sampled sets. For example, if $d_s = 1$, we are

tracking all the sets in the cache. If $d_s = 2$, we track half of the sets, and so on. Sampling reduces the size of the ATD at the expense of less accuracy in IPC predictions. In this situation, some accesses are not tracked and, as a consequence, the information in the overlap counters is always less than real values.

Figure 5.8 shows the evolution with the sampling distance of the error curves (mean error and for each benchmark group). Errors are measured on the left y-axis. It is clear that, for L benchmarks, sampling makes no difference, since their IPC is nearly constant for any L2 cache size. In S and H benchmarks, however, accuracy degradation is more important and errors quickly become significant. With a sampling distance of 8 we obtain average errors around 9%. We think that this is an interesting point of design.

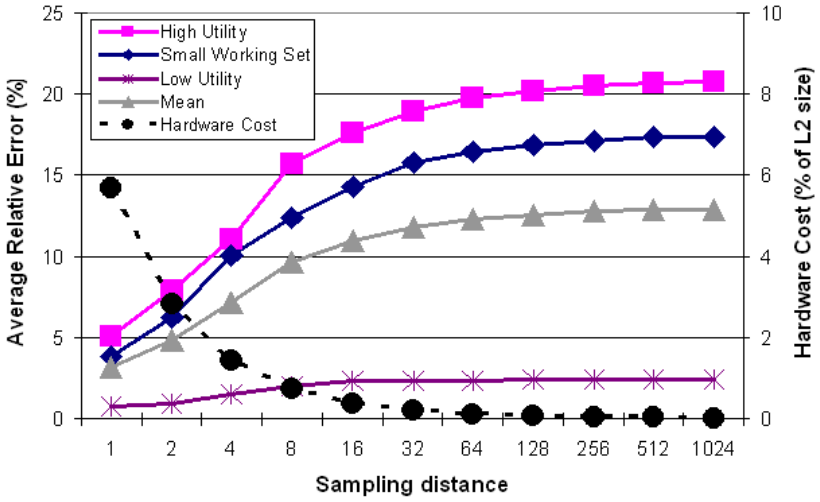


Figure 5.8: Average relative error and hardware cost depending on the sampling distance

Turning to the issue of **hardware cost**, we suppose a 40-bit physical address space. Each entry in the ATD needs 29 bits (1 valid bit + 24-bit tag + 4-bit for LRU counter). Each sampled set has 16 ways, so we have an overhead of 58 bytes for each set. We also need 16 cdc_i counters of 1 byte because the ROB has 256 entries. It is necessary an extra counter of 1 byte for AROAL2M. Next, we need 16 $overlap_i$ counters of 4 bytes, supposing a total of 64 bytes. For the predicted cycles variation stored in PCV_w, we need 16 registers of 4 bytes. Finally, we need two registers of 4 bytes for Icounter and Ccounter. In that way, the hardware overhead of these circuits can be seen in Figure 5.8 measured on the right y-axis.

Next, with regard to **improving accuracy**, we evaluated two different methods in order to further reduce prediction errors. First, as sampling leads to a number of untracked

accesses, we can scale the counters in the following way:

$$counter_{estimated} = scaling\_factor \cdot counter$$

However, it is not easy to establish such a scaling factor. It is clear that the number of misses overlapped by the first L2 miss depends on the size of available L2 cache. For example, if we have a smaller L2 cache, then we obtain more L2 misses and, as a consequence, bursts of overlapped L2 misses contain more L2 misses. Just to illustrate this point, in the case of `twolf`, the miss penalty for an L2 miss goes from 137 cycles to 192 cycles when we have 1 or 16 active ways. Thus, this scaling factor should depend on the number of ways that we predict, and, furthermore, this value depends on the application. As a result of these considerations, we were able to empirically establish the optimal scaling factor for each sampling distance. These values should be stored in hardware. Figure 5.9 shows the accuracy in IPC predictions as we increase the sampling distance.

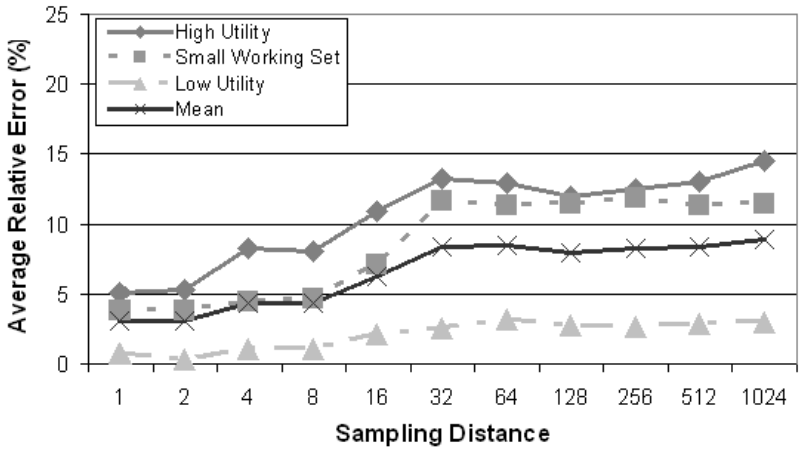


Figure 5.9: Average relative error with optimal scaling factor depending on the sampling distance

Another option is to combine the sampled ATD with information inside the L2 cache. If we have w ways active, we can use the LRU counters inside the L2 cache to detect any access with stack distance less or equal to w. This proposal requires that, on every L2 hit, the LRU counter of the accessed line can be read. Note that this information is readable, since it should be sent to the logic that determines which line to evict on an L2 miss. Thus, we only have to drive this information outside the L2 cache, which should not involve many changes in the cache design. With this information, we can check if the access overlaps with previous accesses. In contrast, the sampled ATD gives misses

information for larger caches configurations. In this situation, IPC estimations are more accurate, as can be seen in Figure 5.10, and we manage to obtain average errors of under 5%. An interesting point of design is obtained with a sampling distance of 16, where we obtain average errors of around 4%.

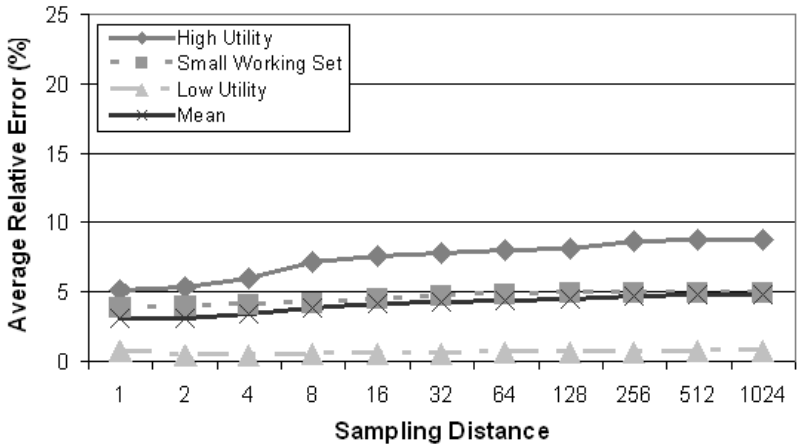


Figure 5.10: Average relative error using information inside the L2 cache and depending on the sampling distance

5.6 Related Work

Many papers focus on predicting the IPC of applications. Some papers seek to reduce the time complexity of design space explorations, while other studies focus on reducing simulation time by selecting a representative small trace of an application [37, 100, 114]. A different approach consists of sampling design space points to train artificial neural networks that predict performance in the whole design space with high accuracy [35, 57, 116]. However, our proposal lies in a different scenario, as it is a run-time mechanism. Another approach is to predict IPC at run-time [28]. The authors analyze IPC in a window of time together with information obtained at compile time, and predict the future value of IPC. The main difference is that the processor configuration remains constant in IPC predictions.

Regarding reconfigurable hardware for single-threaded processors, several proposals try to reduce power consumption without losing too much performance. Dhodapkar et al. [33] make use of *working set signatures* to represent program instruction working sets and to find out the minimal instruction cache size necessary in order to minimize

instruction misses (data misses are not treated). Albonesi [4] introduces *selective cache ways*. These caches just precharge lines in active ways. This study is expanded in a subsequent paper [12], where some algorithms are given to dynamically decide to switch on/off cache ways. However, they are unable to ensure a quality of service (QoS) as they are using indirect measures of IPC. For example, they report a maximum IPC reduction of 52%. Kaxiras et al. [56] propose using *cache decay* to dynamically switch off a cache line when it is highly probable that it will not be accessed again. This occurs after a constant number of cycles or other more agressive variants of this approach. Kobayashi et al. [60] present a run-time decision algorithm for activating or deactivating cache lines based on the number of accesses to the LRU and MRU active lines. Bahar et al. [11] dynamically modify the issue width and the number of execution units depending on the present IPC. When the IPC is under a given threshold, then issue width is decreased. Other authors make use of hints given by the compiler to dynamically adjust the issue width to the requirements of an application [54]. In this scenario, none of the proposals can estimate performance degradation, since they depend on empirical thresholds and heuristics related to indirect measures of performance. Our proposal gives the opportunity to bound these losses.

5.7 Summary

In this chapter we have presented a run-time mechanism that accurately predicts IPC as L2 cache size varies. We have shown average errors of 3.11% with predictions that accurately follow the shape of the real IPC curve. To obtain these results, we modified previous memory models to obtain higher accuracy in predictions. We have also discussed a practical implementation that has an extra cost in area between 5.68% of the L2 cache size (best accuracy) and under 0.4% (for a 4% error). Hardware cost is reduced using a sampling technique to the monitoring logic.

Our mechanism can be used to reduce power consumption in single-threaded architectures as it can be used to give the real contribution of each way to the final IPC and bound performance losses. A second possible application is to improve performance in multi-threaded architectures that dynamically partition shared L2 caches. OPACU gives direct estimations of performance for different cache configurations, instead of other indirect measures of performance that are currently used to maximize total throughput.

Chapter 6

FlexDCP: a QoS framework for CMP architectures

Current multicore architectures offer high throughput by increasing hardware resource utilization. As the number of cores in a chip multiprocessor (CMP) increases, providing Quality of Service (QoS) to applications in addition to throughput is becoming an important problem.

In this chapter we present FlexDCP, a framework that allows the Operating System (OS) to guarantee a QoS for each application running in a CMP. FlexDCP directly estimates the performance of applications for different cache configurations instead of using indirect measures of performance, such as the number of misses. These predictions are obtained using the OPACU methodology, explained above in Chapter 5. This information allows the OS to convert QoS requirements into resource assignments. Consequently, it offers more flexibility to the OS as it can optimize different QoS metrics, such as per-application performance, or global performance metrics, such as fairness, weighted speed up or throughput.

Our results show that FlexDCP is able to force applications in a workload to run at a certain percentage of their maximum performance in 94% of the cases considered, being on average 1.48% under the objective for remaining cases. When optimizing a global QoS metric, such as fairness, FlexDCP consistently outperforms traditional eviction policies such as LRU, pseudo LRU and previous dynamic cache partitioning (DCP) proposals for two-, four- and eight-core configurations. In an eight-core architecture FlexDCP obtains a fairness improvement of 10.1% over *Fair*, the best policy in the literature optimizing fairness.

6.1 Introduction

The current collaboration between the OS and multithreaded architectures is inherited from the traditional collaboration between the OS and multiprocessors: The OS perceives the different cores in a chip multiprocessor (CMP) [45] and the hardware contexts in a simultaneous multithreading architecture (SMT) [98, 110] as multiple, independent, virtual processors. Thus, the OS is not aware of the resource sharing problem and schedules threads onto what it regards as independent processing units. However, in multithreaded architectures the number of instructions executed by a thread depends on the activity of the co-scheduled threads. If no explicit control over shared resources is exercised, the performance of applications becomes unpredictable. Several studies [23, 81] have shown that in both SMTs and CMPs the performance of a task heavily depends on the workload[1] it is executed in. To deal with this performance variability problem, the OS should be able to exercise more control over how threads share the internal resources of the processor. More interaction is needed between the architecture and the OS to allow the latter to provide some *Quality of Service* (QoS) to applications [83].

General-purpose computing is moving off desktops onto diverse devices such as cell phones, digital entertainment centers, and data center servers. The interaction between the OS and the architecture must be flexible enough to cover different scenarios where the concept of QoS has different meanings. For instance, in a high throughput server scenario the target to maximize is system performance or *overall QoS* [58, 90, 92], which can be measured with metrics such as fairness, weighted speed up or throughput. In other scenarios, such as multimedia and real-time systems, per-application or *individual QoS* is required [43, 52, 82]. Finally, there are intermediate situations, such as soft real-time systems with *hybrid QoS* requirements, where some applications need an individual QoS and the remaining ones need a global QoS [43]. Hence, providing QoS to a wide range of scenarios is an important challenge for future multicore architectures.

Recently, Nesbit et al. [83] have proposed an abstract, generic framework for future many-core architectures that allows the OS to explicitly manage resource allocation. This framework incorporates features from previous QoS frameworks and provides a general approach to build new interfaces between the OS and the architecture. In Chapter 8, we will discuss in detail the specifications of such frameworks for future CMPs with hundreds of cores. Figure 6.1 shows the main components of this framework.

[1] A workload is a set of processes running simultaneously on the CMP.

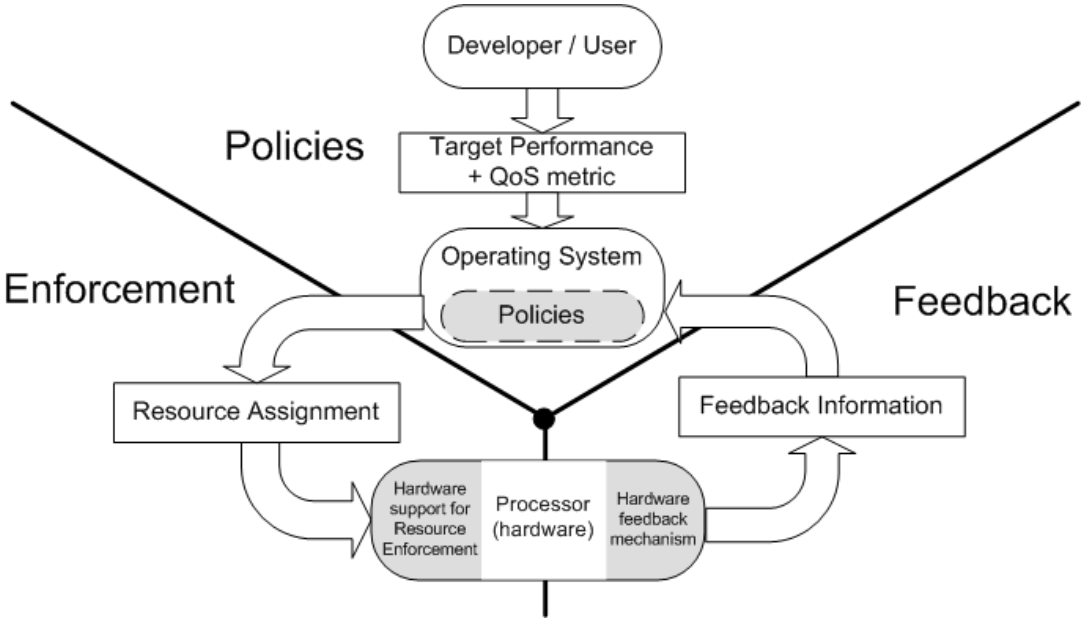


Figure 6.1: Generic framework to manage shared resources in a CMP architecture

1 **Policies**: they are implemented primarily in software. Policies should translate application and system objectives into resource assignments, thereby managing system resources.

2 **Enforcement mechanisms**: they securely multiplex, arbitrate, or otherwise distribute hardware resources in order to satisfy the resource assignments. Their main task is to make sure that each thread receives the amount of resources established by the policies.

3 **Feedback mechanisms**: they inform the software about the global resource usage, which can be used by the OS to find a new resource assignment to accomplish within the QoS requirements of the running applications.

A common characteristic of all previous OS/architecture interface proposals [43, 52, 82] is that they have not addressed one of the key points of this approach, namely: *converting a given QoS target by applications into a resource assignment* [83]. For example, let us assume that a given task has to be executed before a deadline *d*. None of the proposed interfaces provides the user with a method of converting this high-level QoS requirements into a resource assignment for this task in such a way that it meets the deadline *d*. The OS/architecture interfaces proposed to date assume that either the applications will be

able to specify a target usage of each shared resource, or that the OS will somehow know the way to convert performance targets into resource assignments. The former situation is not possible in most cases, since applications are normally architecture-independent. Hence, the developer cannot provide the exact amount of resources that an application will require to obtain a target performance. In the latter situation, the OS job scheduling has to be architecture independent to be portable between different architectures. With current OS/architecture interface proposals, the user establishes a given initial resource partition for the task. At the end of each time quantum, the OS checks whether the task can accomplish its QoS objective with this resource partition, increasing the amount of resources given to it if this is not the case. This iterative process can take a long time, and, as applications may change their behavior, this process has to be repeated frequently. As a consequence, applications are constantly executed in a sub-optimal resource partition.

In this chapter, we propose *FlexDCP*, a flexible framework that represents the first implementation of a complete QoS framework. On the one hand, we propose an effective feedback mechanism that makes it possible to translate QoS requirements from the user into a hardware resource assignment in a single step. FlexDCP is the first framework to do this. On the other hand, FlexDCP supports all kinds of QoS requirements in CMP architectures with a shared cache. FlexDCP can optimize any target metric related to IPC, leading to the best performance results for at least three different targets (ensuring an individual QoS level, fairness and throughput). This flexibility is not possible with previous proposals, which either focus on improving a particular metric, or cannot ensure a target individual QoS level.

In the FlexDCP framework, the architecture provides the OS with the performance of running applications under the current cache assignment, as current performance counters do. In addition, FlexDCP uses additional hardware that provides the OS with the performance that the running applications would have with all other possible cache size assignments. By reading this information, the OS can compute the performance degradation or improvement of each application when moving to another cache configuration. This allows the OS to translate QoS requirements into resource assignment, without profiling the application or forcing the OS to know the internal details of the architecture, making it totally architecture independent. Nor are application developers required to specify the exact amount of resources that their applications must use, which makes our solution closer to more realistic scenarios.

The main contributions of this chapter are the following:

1) **Flexibility**: We propose a feedback mechanism that predicts the performance of running applications under cache partitions which are different to the current one. FlexDCP can maximize overall QoS metrics, such as harmonic mean of relative IPCs [2], weighted speed up, or throughput, or ensure an individual QoS metric. Previous proposals do not offer this flexibility.

- *Individual QoS*: In contrast to previous work, FlexDCP allows jobs to run at a certain percentage of their maximum speed, regardless of the workload in which these jobs are executed. Our results, in a CMP scenario with a shared L2 cache, show that FlexDCP, working with the target IPC, successfully accomplishes its task in 94% of the cases considered, reaching an IPC that is 1.48% lower than the objective IPC in the remaining 6% of the cases.

- *Global QoS/Scalability*: Our results show that previous proposals based on indirect metrics of performance provide diminishing returns as the number of cores sharing the L2 cache increases. FlexDCP obtains sustained throughput and fairness improvements over LRU and previous proposals on the two-, four- and eight-core architecture setups used in this chapter. In the eight-core architecture, FlexDCP obtains a fairness improvement of 10.1% over *Fair*, the best policy in the literature on optimizing fairnes. When optimizing throughput, FlexDCP obtains improvements of 11% on average over MinMisses, the best policy in the literature on improving throughput.

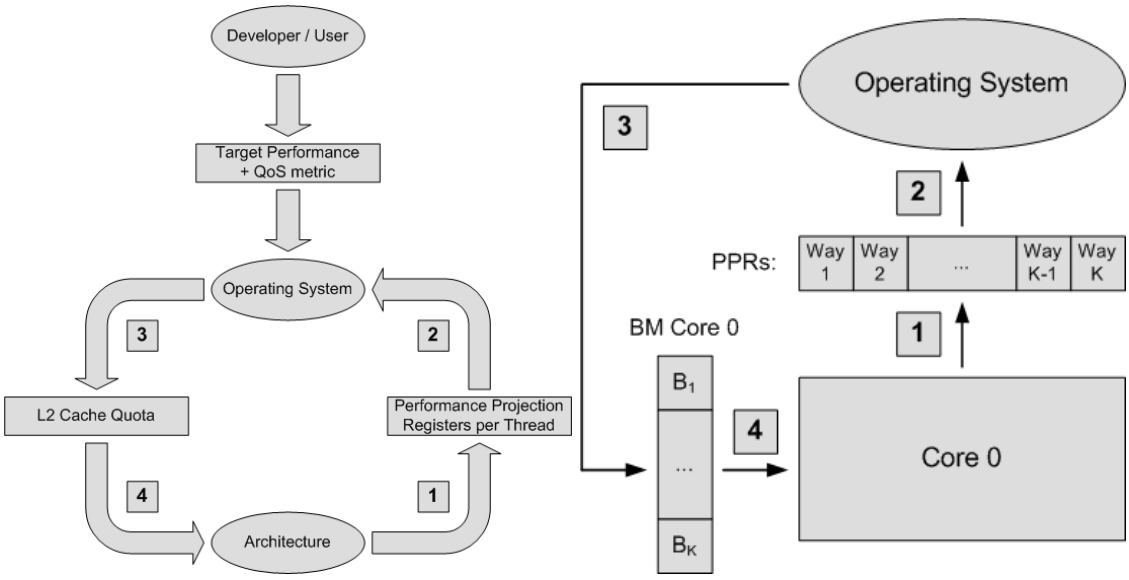
2) **Granularity Analysis**: In this chapter we show that the time granularity at which the resource assignment decisions are taken has a significant impact on performance. Wrong decisions are very costly with a coarse time granularity. At the same time, making resource assignment decisions too frequently also affects overall performance. In this chapter, we give a complete analysis of how to tune this decision period in order to obtain the highest performance.

The rest of this chapter is structured as follows. Section 6.2 presents our new framework that ensures both individual and global QoS. In Section 6.3 simulation results are discussed. Section 6.4 introduces the related work, and, finally, Section 6.5 summarizes our results.

[2]The relative IPC of a thread is the ratio of its IPC when it runs in a workload with respect to its IPC when it runs in isolation using all resources.

6.2 FlexDCP QoS Framework

FlexDCP is a framework that allows the OS to guarantee a QoS for each application in a CMP architecture with a shared last level cache (LLC) on-chip. FlexDCP provides the OS with the necessary information to convert user's QoS requirements into resource allocation. In particular, FlexDCP focuses on the shared caches as one of the main sources of interaction between threads in CMP architectures. The architecture provides the OS with the performance of running applications under the current L2 cache assignment[3], as current performance counters do, and the performance that the running applications would have with all other possible cache size assignments. With this information the OS can compute the performance degradation or improvement of each application when moving from the current cache allocation, *currentCA*, to a new cache allocation, *newCA*, simply by computing: $execution\_time\_reduction = \frac{IPC_{currentCA}}{IPC_{newCA}}$. Given that the IPC values are provided by the architecture and the OS works with the ratio between them, the OS does not need to know the internal details of the architecture, making it architecture independent.



(a) Framework overview (b) Interface between the OS and the architecture

Figure 6.2: FlexDCP: a QoS framework for CMP architectures with a shared LLC

Figure 6.2(a) describes the FlexDCP QoS framework. First, developers or users determine the target performance of the application and the QoS metric to optimize. In some

[3]We assume a shared L2 cache as the LLC in our baseline CMP architecture

scenarios, such as multimedia or real-time applications, different instances of the same application have approximately the same performance [49]. Thus, the user knows beforehand the full speed of the application and can provide it to our framework. The individual target performance determines the minimum performance that the application requires, while the QoS metric determines what to do with unassigned resources. If no individual target performance is specified, the QoS metric will guide resource assignments. For example, fairness or throughput might be optimized.

Next, the OS schedules applications according to their QoS necessities. In our experiments we assume that the workload has already been chosen by the OS. When different applications start executing in the CMP architecture, an initial partition of the shared L2 cache is decided. If the OS has no prior knowledge of the applications, resources are evenly partitioned among threads. Thus, it assigns $\frac{Associativity}{Number\ of\ Cores}$ ways of the shared L2 cache to each thread. If the OS already has prior knowledge of the applications, it can decide the initial partition based on that information.

Next, dedicated hardware estimates the performance of the application running in each core with all other possible cache allocations. We denote these performance estimations *Performance Projections*. This hardware mechanism is detailed in Chapter 5. Each thread stores its performance projections in a set of registers visible to the OS. We call these registers *Performance Projection Registers* (PPR). For a K-way associative cache, there are K 32-bit PPRs (Figure 6.2(b), Step 1).

After this estimation period, the OS analyzes the values of the PPRs of each core (Figure 6.2(b), Step 2). Using this information, the OS decides a new partition for the next period (Figure 6.2(b), Step 3). We assume that performance in the current measuring period is representative of the performance of the next period. Thus, the optimal partition for the last period will be chosen for the following period. FlexDCP assigns to each thread the required cache quota necessary to satisfy its individual QoS requirements[4]. Then, the remaining resources are assigned among all threads according to the overall QoS metric. The optimal partition is determined using *EvalAll* algorithm, explained in Section 3.5 for all the dynamic mechanisms studied in this chapter. We take into account this time overhead when reporting performance results.

Finally, cache partitions are implemented at a way granularity with *column caching* [29], which uses a mask that marks the cache ways (or columns) reserved for each thread. When a thread experiences an L2 miss, the evicted line is the LRU line among the lines owned

[4]We ensure at least one reserved way in the L2 cache for each application.

by that thread. When the OS decides the cache quota per thread, it writes the corresponding bit masks (BM) (Figure 6.2(b), Step 3). Each BM contains a bit per cache way and there is a BM per thread. If the i-th bit of the mask of a thread is set, the thread owns that way. Running threads can read from all cache lines and, consequently, correctness is ensured when updating bit masks [29]. Other authors have used more flexible implementations, such as *Augmented LRU* [108]. However, its hardware cost is considerable, as a counter per thread and set is needed. Thus, in this chapter we use column caching.

The following section argues for the use of direct estimations of performance as the adequate metric to decide L2 cache partitions. Next, Section 6.2.2 describes a case study that exemplifies that cache partitioning algorithms driven by the number of L2 misses may obtain suboptimal partitions in terms of performance. Finally, Section 6.2.3 discusses the adequate granularity of cache quota decisions.

6.2.1 Direct Vs Indirect Performance Metrics

A common characteristic of previous DCP proposals is that they decide new cache partitions using indirect indicators of performance, mainly the number of L2 misses [29, 58, 90, 99, 108]. However, the effect of L2 misses on performance varies depending on the application and even on the particular phase of the application.

To illustrate this point, Table 6.1 shows the variation in performance and the number of misses for some benchmarks from the SPEC CPU 2000 suite, as we vary the number of active ways, w. For this experiment, we simulate a single-threaded architecture with a 16-way associativity 1MB L2 cache (see Chapter 2 for more details). The remaining $16 - w$ ways are simply switched off. For example, observe that when we move from 7 to 16 active ways, `facerec` reduces its number of L2 misses by 40%. Analogously, `equake` reduces misses by 40% when it moves from 1 to 4 ways. However, the effect on performance is different: the IPC of `facerec` increases by 25% while the performance of `equake` only increases by 9%. In contrast, we observe similar variation in performance for `vpr` and `equake` and the reduction in misses is different (16% and 40%). This different impact on performance can also be observed in the case of `crafty`.

Translating IPC into resource assignment has been identified as a challenging problem [43, 83]. It has also been shown to be a key element in future multicore systems in order to improve the interaction between the OS and the architecture [83]. A solution to this problem consists of using the average miss penalty of L2 misses in the current L2 cache configuration and assuming that it is constant for other configurations [115].

Table 6.1: Variability of the impact on performance of L2 cache misses

Benchmark		Ways	Misses	IPC	Benchmark		Ways	Misses	IPC
crafty	config. 1	10	31K	1.689	equake	config. 1	1	10M	0.245
	config. 2	14	21K	1.707		config. 2	4	6M	0.266
	variation	+4	−32%	+1.1%		variation	+3	−40%	+8.6%
facerec	config. 1	7	2M	0.924	vpr	config. 1	15	714K	0.88
	config. 2	16	1.2M	1.16		config. 2	16	600K	0.966
	variation	+9	−40%	+25.5%		variation	+1	−16%	+9.7%

Figure 6.3 shows the average miss penalty of L2 data misses when we vary the number of active ways of the L2 cache from 1 to 16 for three different benchmarks. This miss penalty significantly varies among L2 cache configurations because the clustering level of the L2 misses changes for different cache sizes: an isolated L2 miss has approximately the same miss penalty as a cluster of L2 misses, if they all fit in the reorder buffer (ROB) and thus can be served in parallel [55].

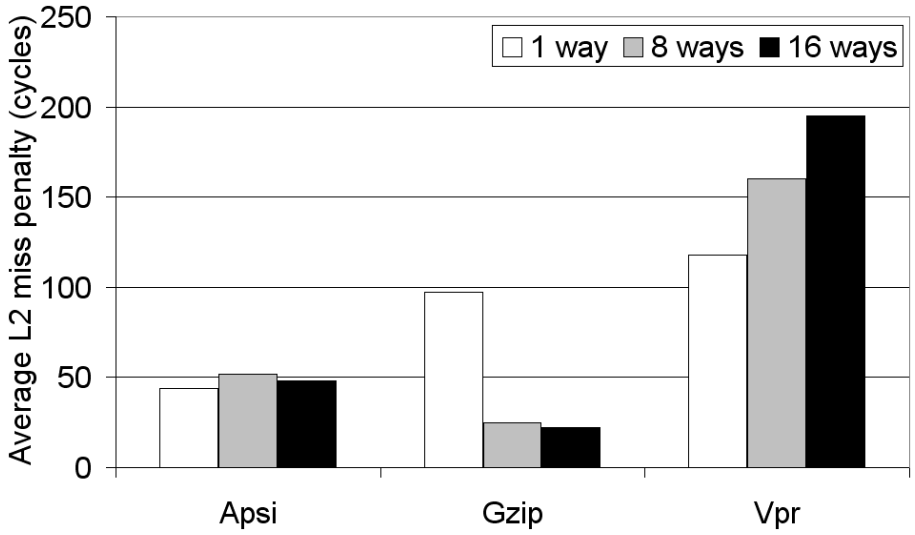


Figure 6.3: Average L2 miss penalty for `apsi`, `gzip` and `vpr` with three different L2 cache configurations

As an alternative, we propose a mechanism that estimates this miss penalty at run-time by using analytic models for superscalar processor performance. This mechanism is based on OPACU methodology [77] and allows us to predict IPC at run-time for different L2 cache configurations without running all these configurations. Chapter 5 describes OPACU methodology in detail.

6.2.2 Case Study: `swim` and `vpr`

We saw in Chapter 3 that SDHs can give optimal partition in terms of total L2 misses. However, minimizing the total number of L2 misses is not the goal of dynamic partitioning algorithms. Normally, other IPC-related metrics, such as fairness or throughput, are the *true target* of these policies. The underlying idea of *MinMisses* is that while we minimize total L2 misses, we are also increasing throughput. This idea is intuitive, since performance is clearly related to L2 miss rates. However, this heuristic can lead to sub-optimal partitions in terms of throughput or fairness, as can be seen in the case study discussed below.

Figure 6.4 shows the IPC curves of benchmarks `vpr` and `swim` as the L2 cache size is increased at a way granularity (measured on the left y-axis). We also show the number of misses of each application in the last interval of time for any possible cache configuration, measured on the right y-axis. We focus this study on a two-core CMP architecture with a 1MB 16-way L2 cache (see Chapter 2 for more details). The optimal partition when maximizing throughput consists of assigning 15 ways to `vpr` and 1 to `swim`[5], obtaining a total IPC throughput of 1.22 instructions per cycle. However, the *MinMisses* algorithm determines a new partition assigning 13 ways to `vpr` and 3 to `swim` according to the SDHs values.

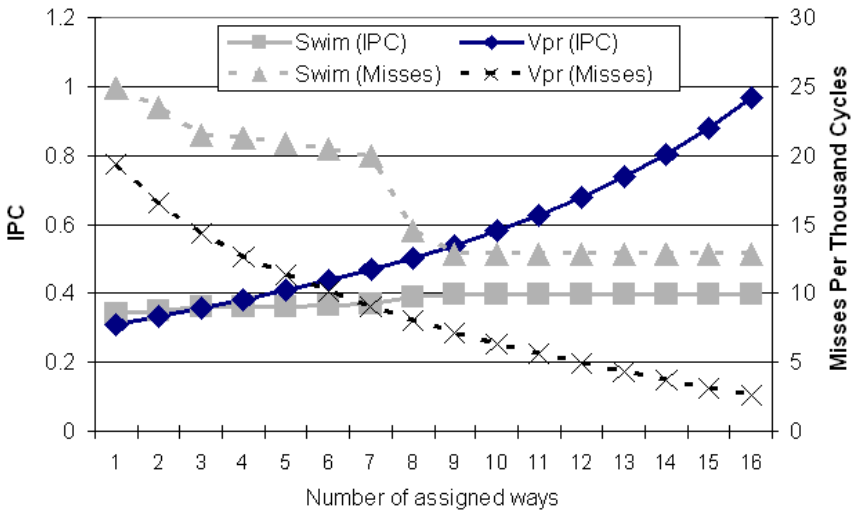


Figure 6.4: SDHs and IPC curves for `swim` and `vpr`

In this situation, misses in `swim` are less important in terms of performance than

[5]Remind that at least one way must be assigned to each thread

misses in `vpr`. Furthermore, `vpr` can reach a larger IPC than `swim`. As a consequence, *MinMisses* obtains a suboptimal partition in terms of IPC and its throughput is 1.1 instructions per cycle, which is 7.7% smaller than the optimal one. If we use the OPACU mechanism to predict IPC values, we can determine exactly what the optimal partition will be in terms of throughput. In this way, our approach assigns two extra ways to `vpr`, reaching the optimal partition.

6.2.3 Granularity of Cache Quota Decisions

The frequency of cache partitioning decisions directly impacts the performance improvement obtained by the mechanism. In this section we show three possible implementations, depending on the desired granularity of decisions. Figure 6.5 shows these three possible alternatives.

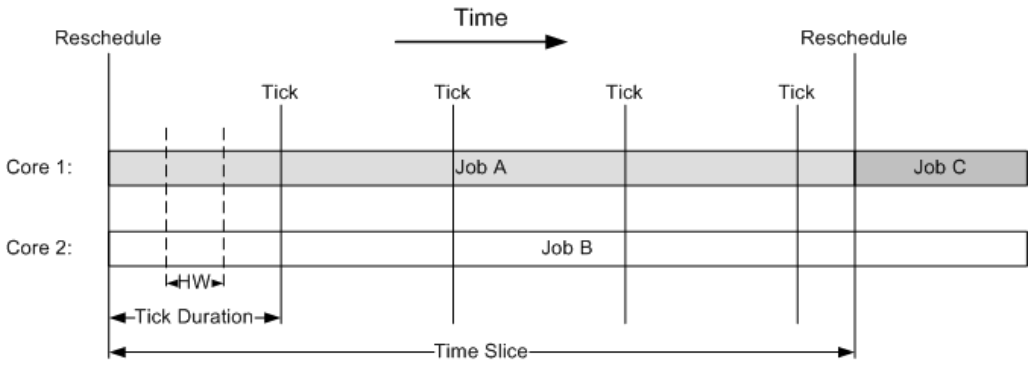


Figure 6.5: Partitioning granularities in a two-core architecture

1) Hardware granularity. The decision logic used to decide new partitions can be implemented in hardware [52]. With this solution, the OS specifies the desired target performance at the time slice boundary and the hardware decides new L2 cache partitions at a smaller time granularity. Consequently, this solution provides a quick response time to phase changes. If there is no time overhead in deciding new partitions, the hardware solution is the best one. However, as the number of cores and L2 associativity increase, the time overhead of making a new decision also increases. As a consequence, deciding new partitions too frequently can negate the performance benefits. Furthermore, it is difficult to implement a flexible decision algorithm with different metrics to optimize with complexity-effective hardware.

2) Interrupt granularity. The second option consists of programming a periodic interrupt to decide new L2 cache partitions [92]. With this option, the frequency of partition

decisions can be chosen, but we have to pay the overhead of interrupting the application and executing the interrupt handler. Thus, we have lower hardware complexity than the hardware solution and a higher time overhead as decisions are made in software. With this solution, the time overhead can become a problem as interrupt handlers must not take long: while the interrupt is running, other interrupts are inhibited and might be lost, which can be a critical problem. Instead of adding the decision algorithm inside the interruption handler, we propose using a microcode piece of code that is in charge of deciding the new partition. This solution assures that no other interrupts are lost. The idea is similar to *millicode* [46] or co-designed virtual machines [104], which have a *concealed memory* reserved in main memory at boot time and the conventional applications are never informed of its existence. The OS can program this concealed memory, which gives more flexibility to the framework. We modify the timer interrupt to invoke this microcode when an OS tick occurs. The code that resides in the concealed memory can take control of the hardware and decide new L2 cache partitions. Algorithm 6 shows the different actions performed when an OS tick occurs [20].

Algorithm 6: Timer interrupt pseudo code

1- Save architecture state.;
2- Call scheduler_ticks();
begin
    2.1- Update counters and statistics;
    2.2- Check if there are ready threads with higher priority ;
    2.3- Decrement quantum of time and check if the quantum has expired;
    2.4- Balance load between different task queues.;
end
3- ***Invoke microcode to decide L2 cache partition***;
4- Restore architecture state ;

3) OS quantum granularity. Finally, L2 cache partitions can be decided at time slice boundaries [108]. In this case, the OS already interrupts the application, reducing the time overhead of the solution. However, the frequency of decisions might be too coarse to adapt to phase changes. In Linux 2.6, the timer interrupt can be configured to different periods: 1ms, 3.33ms, 4ms (default) and 10ms. The time slice duration depends on the thread priority and is between 5ms and 800ms (100ms by default) [20]. At a frequency of 2GHz, this corresponds to a range from 2 to 20 million cycles for the timer interrupt (8 million by default), and a range from 10 to 1600 million cycles for the time slice (200 million by default). If deciding cache partitions in every OS tick becomes too

expensive, decisions can be made when the OS schedules threads. Thus, we have to make performance projections visible to the scheduler. We propose storing these projections in the *task struct*, which stores information of all processes alive in the system. With these performance projections, the scheduler can restore the values of the performance registers in a previous time slice. This idea prevents wrong L2 cache partition decisions when a new thread is scheduled. Finally, the microcode is invoked to decide the new cache partition.

The optimal granularity at which a policy decides new cache quotas depends on the application under consideration and also on the time overhead of making a new decision. Instead of deciding new cache partitions evaluating all possible combinations and choosing the one that optimizes a given metric (exhaustive search), we implement an algorithm that uses dynamic programming techniques to reduce the time overhead of deciding new partitions [32]. This algorithm is explained in detail in Section 3.5. Finding the optimal partition for just two cores is straightforward as we have to check just K partitions. Thus, we can compute the optimal number of misses when w ways are assigned to these two threads, $misses(w)$, in w steps. This function is independent of the other threads, allowing us to build this curve in parallel with the other thread pairings. Next, the same algorithm is repeated for the new histograms of misses. For each pairing, the complexity of the algorithm is $O(K^2/2)$, which is repeated $\frac{N}{2} + \frac{N}{4} + \ldots + 1 = N - 1$ times. This algorithm is described in detail in Section 3.5 (see Algorithm 4).

Using the PIN instrumentation framework [67], we measured the number of dynamic instructions executed by this algorithm when no individual QoS is specified (worst-case scenario). We evaluated different numbers of cores (2, 4 and 8) and associativities (16, 24 and 32). With a 16-way cache, the number of executed instructions is less than 5,000, while for a 32-way cache this number increases to less than 20,000 instructions. This algorithm only reads the PPRs and decides L2 cache quotas. Hence, no data cache miss is caused. The number of static instructions is between 50 and 100, causing little pollution in the instruction cache. In the case of current CMP architectures with up to 8 cores, the time overhead of this algorithm is less than 5,000 cycles (assuming an IPC of 1 instruction per cycle).

Next, Table 6.2 reports the performance improvements over LRU in a four-core architecture with a 1MB 16-way L2 cache when optimizing for throughput. The CMP configuration and workloads are detailed in Chapter 2. First, we give the ideal performance improvement when no time overhead is considered to make new cache decisions (see col-

Table 6.2: Performance improvement over LRU in a 4-core CMP with a time overhead of 5,000 cycles

Granularity	Decision period	Overhead	Maximum performance improvement	Real performance improvement
Hardware	100K	5%	10.95%	5.67%
	500K	1%	10.65%	9.54%
Interrupt	1M	0.5%	10.53%	9.98%
	5M	0.1%	10.15%	10.04%
Scheduler	50M	0.01%	8.51%	8.499%
	100M	0.005%	8.00%	7.99%

umn 4 in Table 6.2). Next, we evaluate two possible decision periods corresponding to the three possible implementations of our framework. We consider 100 and 500 thousand cycles for the hardware solution, 1 and 5 million cycles for the interrupt solution, and 50 and 100 million cycles when deciding partitions at the scheduler granularity.

On the one hand, wrong decisions are very costly, mainly with high granularities. On the other hand, making decisions too frequently also affects overall performance. On average, using the interrupt solution provides the highest improvement. For that reason, we propose using this granularity and deciding cache partitions every 5 million cycles. Even with a configuration with 8 cores and a shared 32-way associative cache, the overhead is under 0.4% with this granularity. In this chapter, we consider the time overhead of deciding new cache partitions when reporting results.

6.2.4 Scalability of FlexDCP

Currently there are processors with 32- and 64-way associative caches: Niagara T2 (32-way [111]), AMD Barcelona (32-way L3) or ARM920T (64-way [8]). Furthermore, in future manycore architectures, we do not expect to have dozens of cores directly sharing the same cache, due to limitations on bandwidth and the capacity of the cache. The cores will be clustered in reduced groups of cores sharing a cache. Hence, we believe that our mechanism will work well with manycore architectures.

As the number of cores increases, the time overhead of evaluating all possible partitions also grows. Previous work has proposed several heuristics to determine partition candidates with low time overhead and without losing performance [90, 108]. In this chapter we make us of the dynamic programming algorithm explained in Section 3.5, which drastically reduces the time overhead of evaluating all possible partitions. Further reducing this overhead is part of our future work.

6.3 Evaluation Results

In this chapter we focus on a CMP with two, four and eight cores. We make use of the simulation infrastructure described in Chapter 2, with configurations *2C* (2 cores and 1MB 16-way L2), *4C-1* (4 cores and 1MB 16-way L2), *4C-2* (4 cores and 2MB 32-way L2) and *8C-2* (8 cores and 2MB 32-way L2).

6.3.1 Ensuring an Individual Quality of Service

In scenarios such as real-time systems, we need to offer an *individual QoS* for each application. We use the relative IPC to measure individual QoS, which is computed as $\frac{IPC_{CMP}}{IPC_{alone}}$. Thanks to the flexibility of the FlexDCP framework, we are able to control the performance of an individual application when executed with other applications. Our framework allows the OS to run jobs at a certain percentage of their maximum speed, regardless of the workload in which these jobs are executed and without dedicating all shared resources to them. Thus, non-time-critical jobs can also make significant progress, without significantly compromising overall performance.

To evaluate the individual QoS results of our proposal, we generate workloads containing four SPEC CPU 2000 benchmarks. One of these benchmarks is a *High Priority Thread* (HPT). Our objective is to force the HPT to run at a given *target percentage* of its *full speed* (IPC when it runs alone in the architecture). This full speed is estimated with the run-time mechanism to predict performance, which cannot be estimated in previous proposals. The HPT runs together with other Low Priority Threads (LPTs) which makes it difficult to ensure performance isolation of the HPT.

For this experiment we use a worst-case scenario. We select 5 benchmarks as HPT (`ammp`, `art`, `mgrid`, `parser` and `vpr`) that require many ways to achieve their maximum IPC (large $w_{90\%}$ value). For benchmarks with low cache requirements, it is less challenging to attain the target IPC. We use the *4C-2* configuration, since as the number of LPTs is high it is more difficult to ensure the target IPC for the HPT. As LPTs we generate 3 groups of threads: we form the H group with cache hungry applications (`apsi+facerec+galgel`), the S group with applications with small working sets (`crafty+gzip+vortex`) and the L group with applications that do not benefit from more cache space (`equake+mesa+mcf`).

Figure 6.6 shows the relative IPC of the HPT for the different workloads and different target percentages. On the x-axis, the target percentage of the full speed of the HPT is

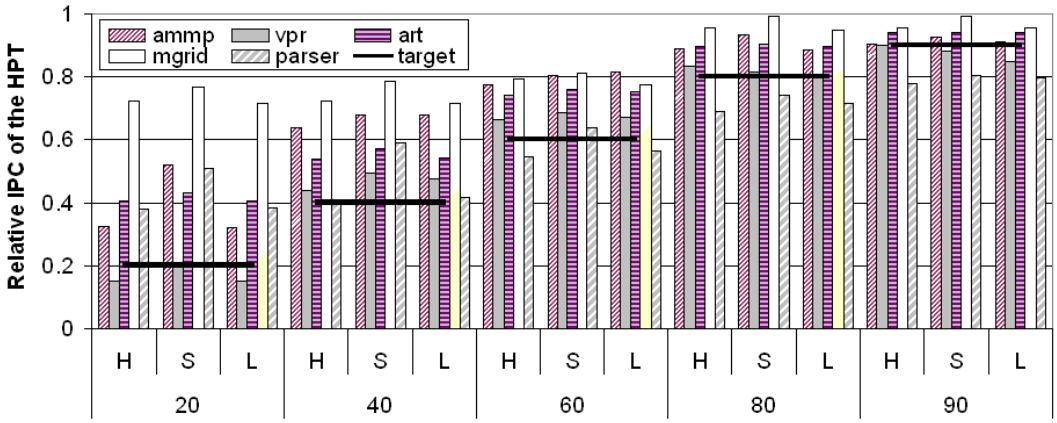


Figure 6.6: Predictable performance in the *4C-2* configuration

given, ranging from 20 to 90 percent. For each HPT and type of the LPTs, we give the achieved relative IPC for the HPT (measured on the y-axis). For example, the first bar corresponds to `ammp` when mixed with the H group `apsi+facerec+galgel`. The target relative IPC is 20% and we reach 32%. Note that the number of assigned ways is discrete and, as a consequence, we cannot always force an exact target IPC. Instead, our mechanism assigns to the HPT the minimum number of ways needed to be above the target IPC, which still ensures the target IPC for the HPT. Some benchmarks already exceed the target IPC with just one reserved way in the L2 cache. This is the case of `mgrid` that runs at 69% of its full speed with 1 way. Note that, over the entire range of different workloads and target percentages, the achieved IPC follows the trend of the target IPC.

If we consider that we accomplish the target IPC for the HPT when we obtain an IPC with a margin of error of 5%, then we have a success rate of 89.3%, `parser` being the only benchmark that does not succeed in some cases (the success rate is 47%, due to its poor prediction accuracy). We also observe that when the LPTs have high cache requirements (type H), it is more difficult for the HPT to obtain its target IPC than when LPTs are type L or S. When the target performance is not achieved, the HPT is, on average, 6.38% under the objective IPC.

Two main reasons explain why some HPTs are below their target performance. First, the OPACU methodology shows low accuracy in IPC predictions for some benchmarks like `parser`, which has an IPC prediction error of 9.8%. Thus, the predicted full speed IPC can differ from the real one. Our mechanism assigns to the HPT the minimum number

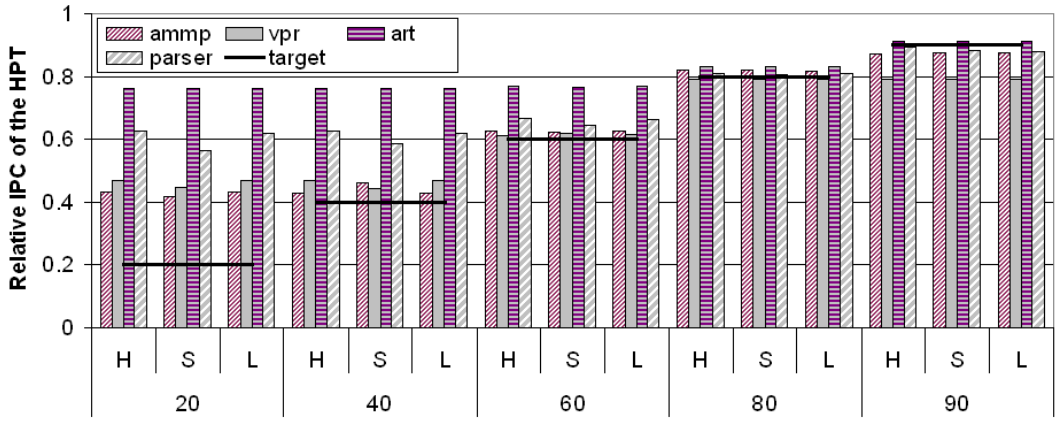


Figure 6.7: Predictable performance in the *4C-1* configuration when the target IPC is specified beforehand

of ways needed to reach the target IPC, which is computed as a given percentage of the predicted full speed IPC. If this prediction is inaccurate, the decision can be wrong. To solve this problem, the accuracy of the OPACU methodology must be improved. The second reason why some HPTs are below target is related to the shared bandwidth to access the different L2 cache banks. In some situations the number of ways assigned to the HPT is enough to reach the target IPC (the decision is correct), but bus contention is too high and performance drops. We could use a *bandwidth arbiter* to overcome this situation and reserve a fraction of bandwidth to the HPT [80, 82].

Figure 6.7 shows the results of FlexDCP when the full speed of the HPT is known beforehand. This performance could be given by the user or obtained with an improved version of the OPACU run-time mechanism. Also, it has been shown [49] that multimedia applications have approximately the same performance for different instances of the same application. Thus, in multimedia applications the user knows the full speed of the application beforehand, and can provide it to our framework.

We evaluate a four-core architecture with a 1MB 16-way L2 cache. As in the previous experiment, some benchmarks already exceed their target IPC with just one reserved way (`art` reaches 76% of its full speed with one way). Here, FlexDCP achieves the target IPC 94% of the time and the obtained IPC is much closer to the target IPC. For the remaining 6% of the time, our framework is 1.48% under the target IPC on average. We note that `vpr` and `ammp` cannot reach 90% of their maximum performance because they need more than 13 ways and we are assuming that each application has at least one reserved way of the L2 cache (that is, there are not enough resources in the architecture).

6.3.2 Ensuring a Global Quality of Service

In this section we evaluate the performance of FlexDCP when optimizing an overall QoS metric. We compare our proposal with the best state-of-the-art dynamic cache partitioning (DCP) mechanisms, namely: *MinMisses* [90], which is the best policy in the literature on improving throughput, and *Fair* [58], which is the best policy in the literature on improving fairness. *MinMisses* estimates the number of misses of each running application for all cache configurations and selects the L2 cache partition that minimizes the total number of misses. Instead of minimizing the total number of misses, *Fair* forces all threads to have the same increase in percentage of L2 misses, trying to equalize the statistic $X_i = \frac{misses_{shared_i}}{misses_{alone_i}}$ of each thread i. Section 3.4 describes these algorithms in detail.

With regard to **optimizing fairness**, by using predicted IPCs, we can decide to maximize any global QoS metric related to IPC. A relevant goal in some environments such as high performance servers is to have fairness among threads. Several metrics have been used to measure fairness, such as weighted speed up, or the harmonic mean of relative IPCs. In this chapter, we show results for the latter, since it has been shown to provide better fairness-throughput balance than weighted speed up [68]. In any case, our results for weighted speed up follow the same trends than for harmonic mean. We compute the relative IPC as $\frac{IPC_{CMP}}{IPC_{alone}}$. We denote our proposal *FlexDCP-MaxFair* as we maximize fairness. We compare our proposal with LRU, *MinMisses*, *Fair* and *FlexDCP-MaxIPC*.

To evaluate our proposals, we randomly generate 16 workloads belonging to each case for the four selected configurations[6] (48 workloads per configuration). Average improvements consider the distribution of workloads among the three groups. We denote this mean *weighted mean*, as we assign a weight to the speed up of each case depending on the distribution of workloads from Table 2.6. For example, for the *2C* configuration, we compute the weighted mean improvement as $0.48 \cdot x_1 + 0.41 \cdot x_2 + 0.11 \cdot x_3$, where x_i is the average improvement in Case i.

Figure 6.8 shows the average Hmean improvement of all policies over LRU for the four configurations. We observe that for all processor/cache setups, *FlexDCP-MaxFair* outperforms the other proposals on average. For the 4-core configurations, *FlexDCP-MaxFair* outperforms *Fair* by 3.5% and MinMisses by 6.5%. It is interesting to note that as the number of cores and cache size increase, the Hmean improvement of *FlexDCP-MaxFair* over previous proposals also increases, outperforming *Fair* by 10.1%, and *MinMisses* by 10.3% on average in the largest configuration (*8C-2*).

[6]Except Case 1 in configuration *8C-2*, as only one workload belongs to this group.

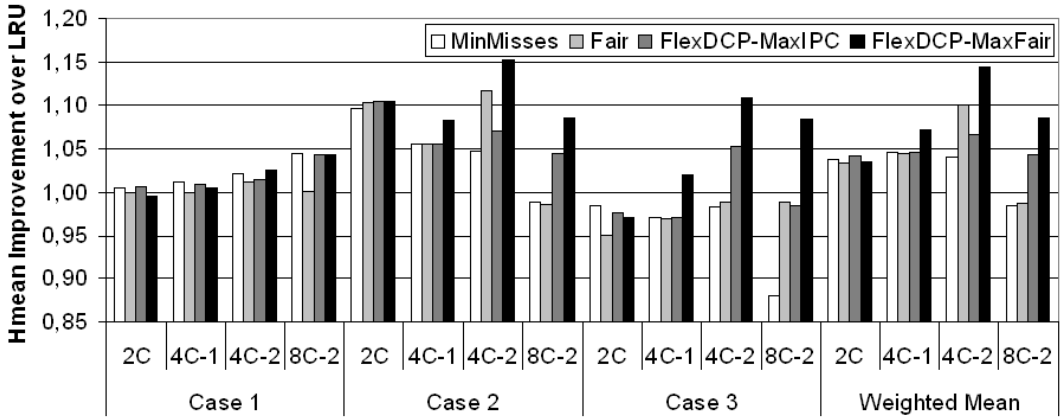


Figure 6.8: Hmean speed up over LRU when optimizing different QoS metrics

All algorithms have similar results in Case 1. This is intuitive, since in this situation there is little room for improvement as all threads fit in cache. In Case 2, *FlexDCP-MaxFair* improves previous approaches by between 8.2% and 15.2%. As the number of cores increases, *MinMisses* and *Fair* find it more difficult to find the optimal partition for fairness. In configuration *8C-2*, *FlexDCP-MaxFair* achieves an improvement of 10.8% over *Fair* and of 9.8% over MinMisses. In Case 3, *MinMisses* and *Fair* present performance degradations with respect to LRU because of the asymmetry between the cache requirements of applications [76]. As a result, *MinMisses* has worse average fairness than LRU (4.6% on average). The same happens with *Fair*, which has a performance 2.6% worse than LRU. By using IPC predictions, *FlexDCP-MaxFair*, in contrast, obtains better results than LRU, 8.6% in the 8-core configuration and 10.9% in the 4-core configurations.

Next, we analyze the results of FlexDCP when it maximizes **throughput**. We denote this proposal *FlexDCP-MaxIPC*, since the metric to optimize is throughput. We simulate *MinMisses*, *Fair* and *FlexDCP-MaxIPC* with the same 48 workloads that we selected for the fairness results. Figure 6.9 shows the average speed up over LRU for these mechanisms. *FlexDCP-MaxIPC* provides the best performance for all cache configurations. In 83.2% of the workloads, *FlexDCP-MaxIPC* outperforms the throughput obtained by *MinMisses*, which means that performance improvements are consistent among workloads.

Figure 6.9 shows that the performance benefits of *MinMisses* decrease with the increase in the number of cores and associativity, obtaining 2.4% less throughput than LRU in configuration *8C-2*. In the *2C* configuration it improves by 8.5% over LRU, while in

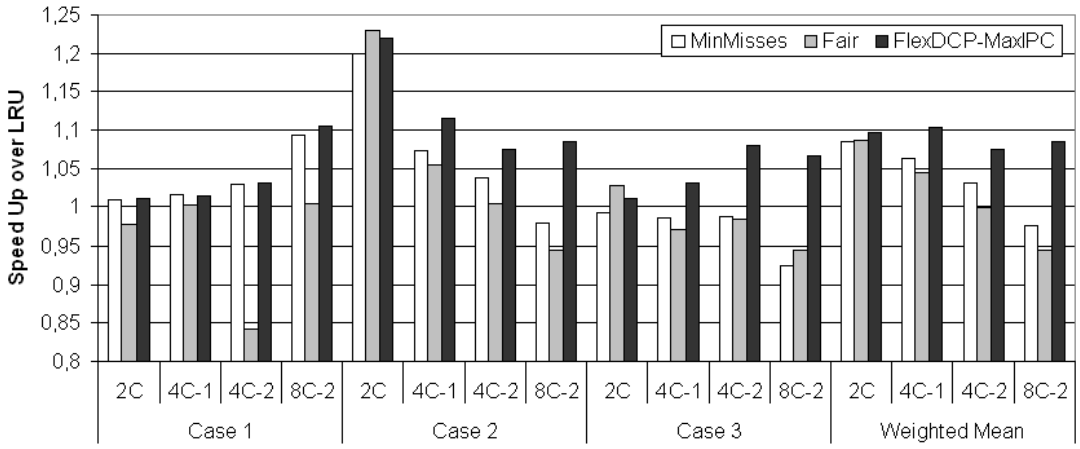


Figure 6.9: Throughput improvement of MinMisses, Fair and FlexDCP-MaxIPC over LRU

4C-1 and *4C-2* these benefits decrease to 6.2% and 3.1% respectively. *Fair* obtains even worse results than *MinMisses* in all configurations. In contrast, *FlexDCP-MaxIPC* has a more consistent throughput improvement over LRU: 9.7% (*2C*), 10.3% (*4C-1*), 7.5% (*4C-2*), and 8.4% (*8C-2*) respectively. Figure 6.8 shows that performance improvements over LRU and *MinMisses* are not at the expense of fairness, as we improve in both fairness and throughput.

We turn now to the issue of **interaction with real pseudo LRU implementations**. One of the key aspects of any QoS framework to be considered by industry is that it has to work with replacement policies implemented in real processors. For highly associative caches, implementing true LRU replacement becomes complex and implies a high hardware cost. As a consequence, current high performance processors implement other simpler replacement algorithms in the L2 cache with similar performance to LRU [47]. For example, the Sun UltraSPARC T2 [111] has a shared 4Mbyte 16-way associativity L2 cache with pseudo LRU replacement, which has a used-bit scheme to implement a Not Recently Used (NRU) replacement. The used bit is set to one each time a cache line is accessed or when initially fetched from memory. For a given cache set, if setting the used-bit causes all used bits to be set to one, the remaining used bits are cleared instead. On a miss, the L2 looks for the first line in that set with a used-bit set to zero, which is chosen as the evicted line.

We propose partitioning a shared L2 cache with the NRU replacement algorithm extending columnization, an idea which was previously proposed in partitioning caches with

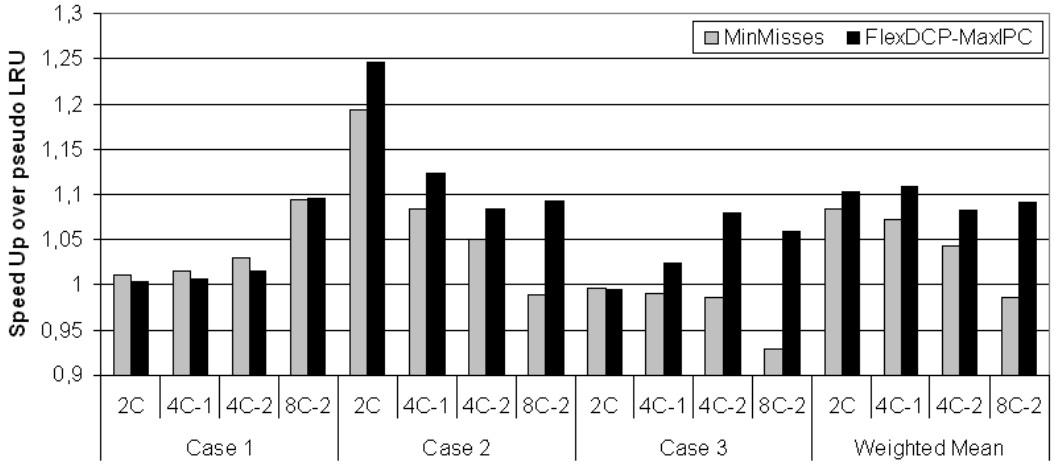


Figure 6.10: Average speed up over pseudo LRU when optimizing throughput

the LRU replacement algorithm [29]. We assign an initial used-bit mask (UBM) for each thread that sets the ways owned by other threads to one, and sets its owned ways to zero. Thus, threads can evict only lines from their owned ways. Extending columnization for the NRU replacement algorithm is mechanical, and we do not give all implementation details for the sake of simplicity. Exploring the interaction with other replacement algorithms is part of our future work.

Next, we show that our QoS framework is compatible with the NRU replacement algorithm. Figure 6.10 shows the average speed up over NRU when optimizing a global QoS metric such as throughput. We evaluate *MinMisses* and *FlexDCP-MaxIPC* with the workloads that were selected in Section 2.4. Neither the SDHs nor the PPRs will provide results as accurate as those obtained with LRU, although speed ups are nearly the same as those obtained with LRU. *MinMisses* presents diminishing returns as the number of core increases, with an average improvement of 4.6% over NRU. *FlexDCP-MaxIPC* has more consistent results with an average improvment of 9.6% over NRU. The results when optimizing fairness and individual QoS present the same trend as Figures 6.6, 6.7 and 6.8.

6.3.3 Putting it all together

FlexDCP is the only framework providing enough flexibility to provide individual or global QoS to applications. FlexDCP can maximize overall QoS metrics such as harmonic mean of relative IPCs, weighted speed up, or throughput, outperforming LRU and previous throughput- or fairness-oriented DCP algorithms. The performance results pre-

sented in this section prove that using performance projections to decide cache partitions is more adequate and leads to better performance than previous proposals guided by miss rates.

6.4 Comparison of Different QoS Frameworks

With regard to previous QoS frameworks, some efforts focus on ensuring QoS in multithreaded architectures. Cazorla et al. [23] introduce a mechanism to force predictable performance in SMT architectures. They manage to run time-critical jobs at a given percentage of their maximum IPC. To attain this goal, they need to control all shared resources of the SMT architecture, while we work with a CMP architecture.

Concerning CMP architectures, Rafique et al. [92] propose managing shared caches with a hardware cache quota enforcement mechanism, and an interface between the architecture and the OS to let the latter decide quotas. However, this proposal cannot guarantee individual QoS. Nesbit et al. [82] introduce Virtual Private Caches (VPC), which consist of an *arbiter* to control cache bandwidth and a *capacity manager* to control cache storage. They show how the arbiter makes it possible to meet QoS performance objectives, or fairness. However, the authors do not discuss the question of how resource assignments decisions are made. A similar framework is presented by Iyer et al. [52], where resource management policies are guided by thread priorities. Individual applications can specify their own QoS target (for example, IPC, miss rate, cache space) and the hardware dynamically adjusts cache partition sizes to meet their QoS targets. Guo et al. [43] present an extension of this work with an admission mechanism to accept jobs in a CMP architecture. However, the authors claim that IPC is not suitable for specifying a QoS target, because IPC is not easily convertible into resource allocation. In this chapter, FlexDCP successfully converts IPC into resource assignment.

Lee et al. [64] present METERG QoS system, which provides QoS in a soft real-time scenario. However, in this framework, the developer needs to run the application in the system beforehand, in order to guarantee a QoS in future executions. With our framework, no profiling information is needed to guarantee a QoS.

Table 6.3 compares the functionalities that previous proposals offer with those offered by FlexDCP. In this table, we use the following symbols: +++ (very high), ++ (high), + (medium), − (low, equivalent to LRU), ✔ (feature supported), × (feature not supported). FlexDCP is the first framework to cover all the necessary to convert performance and

Table 6.3: Functionalities offered by the different QoS frameworks

Framework	Performance/ resource translation	Provides individual QoS	Provides hybrid QoS	Provides global QoS	Fairness	Throughput
MinMisses [90]	×	×	×	✔	+	++
Fair [58]	×	×	×	✔	++	+
VPC [82]	×	✔	×	×	−	−
Guo et al. [43, 52]	×	✔	✔	×	−	−
FlexDCP	✔	✔	✔	✔	+++	+++

QoS requirements into resource assignments. Furthermore, the flexibility of the framework makes it possible to ensure that all concepts of QoS are covered, giving the best performance when optimizing global QoS metrics such as fairness or throughput.

6.5 Summary

In this chapter, we propose FlexDCP as a new framework which allows the OS to guarantee a QoS for each application running in a CMP architecture with a shared LLC. Instead of using indirect measures of performance, FlexDCP uses direct estimations of the performance of each thread for different cache configurations to decide cache quota assignments. These estimations enable our framework to control the performance of individual applications when executed in a workload, ensuring an individual QoS. In addition, this framework provides higher flexibility than previous proposals as it allows the OS to optimize either fairness, total throughput, or any other IPC-related metric.

Simulation results show that FlexDCP is able to force applications to run at a certain percentage of their maximum performance, which is required in real-time environments. We manage to reach the objective performance in 94% of the cases considered, being 1.48% under the objective for remaining cases. When optimizing for a global QoS metric such as fairness or throughput, FlexDCP obtains the best performance in all metrics. In an eight-core architecture, *FlexDCP-MaxFair* obtains an average 10.1% improvement over *Fair* in fairness, while *FlexDCP-MaxIPC* obtains an average 11.2% improvement over *MinMisses* in throughput. Finally, we showed that FlexDCP also works with pseudo LRU replacement algorithms currently implemented in processors such as the Sun UltraSPARC T2.

FlexDCP provides a platform that can also be used with parallel applications. In single process-multiple data applications, all the processes execute the same code on different data sets and use synchronization primitives to coordinate their work. Thus, the FlexDCP framework can be used to estimate the performance of each process between communication primitives. In the case of parallel applications in which threads concurrently work on

the same data, the parallel application can be seen as a whole accessing the shared cache. With that goal, bit masks should be assigned to processes instead of cores. In the next chapter, we evaluate a possible implementation of such an automatic balancing algorithm with parallel applications.

Chapter 7

Load Balancing Using Dynamic Cache Allocation

Supercomputers require an enormous budget to cover build and maintenance. In order to maximize the use of their resources, application developers spend time attempting to optimize the code of the parallel applications and to minimize execution time. Despite this effort, *load imbalance* still arises in many optimized applications, due to causes not controlled by the application developer, resulting in significant performance degradation and waste of CPU time. If the nodes of the supercomputer use chip multiprocessors, this problem may become even worse, as the interaction between different threads inside the chip may affect their performance in an unpredictable way.

Although there are many techniques that may be used to address load imbalance at run-time, in fact, these techniques may not be particularly effective when the cause of the imbalance is the performance sensitivity of the parallel threads when accessing a shared cache. To address this problem, we present a novel run-time mechanism, with minimal hardware, that automatically tries to balance parallel applications using dynamic cache allocation. The mechanism detects which applications are sensitive to cache allocation and reduces imbalance by assigning more cache space to the slowest threads. The efficiency of our proposed mechanism is demonstrated with both synthetic workloads and a real-world parallel application. In the former case, we reduce the execution time by up to 28.9%; in the latter case, our proposal reduces the imbalance of a non-optimized version of the application to the values obtained with a hand-tuned version of the same application, in which several man-years of effort have been devoted to manually balance the load, achieving an overall execution time reduction of 7.4%.

7.1 Introduction

In order to obtain good performance from parallel applications it is essential to guarantee that, during execution, the amount of time a processor is waiting for other processors' results is kept to a minimum. Achieving this essential condition requires good *load balancing* of the parallel threads. Expert programmers often spend a significant amount of time optimizing work and data distribution in parallel applications so that any potential sources of *load imbalance* are eliminated. However, no matter how good their work is, there might still be issues that cause load imbalance during the application's execution, which may not be easily addressable *a priori*.

According to the classification in [18], there are two main classes of load imbalance which result from causes that become apparent only during the application's execution: the first class is *intrinsic load imbalance*, and this type of imbalance is caused by characteristics which are intrinsic to the application, such as the input data. For example, sparse matrix computations depend heavily on the number of non-zero values in the matrix; the convergence time of iterative methods that approximate the solution of a problem (for example, partial differential equations) may change for different domains of the modeled space. It is very difficult (while not impossible) for the application programmer to balance the application *a priori* to deal with all possible input data sets. The second class is *extrinsic load imbalance*, caused by external (to the application) factors, that may slow down some processes but not others. A major source of load imbalance could be the operating system (OS) when performing services such as handling interrupts, reclaiming or assigning memory, etc. For example, the OS may decide to run another process (say a kernel daemon) in place of the process running on a CPU. Extrinsic load imbalance may also be caused by thread contention for processor's shared resources; this may be particularly true in the case of SMT architectures, where threads share and compete for most of the processor's resources [18]. Clearly, there is nothing that the application programmer could do *a priori* to prevent extrinsic load imbalance.

A standard way to address the aforementioned two types of load imbalance is to use dynamic load balancing mechanisms, which are triggered as necessary at run-time. We can distinguish between two main strategies used to perform dynamic load balancing: *work* and/or *data redistribution* and *resources redistribution* [18]. The former strategy includes run-time mechanisms which move some work or data (load) from processes whose execution lags behind fast-running threads [72, 96]. The latter strategy relates to

run-time mechanisms, which may dynamically assign more resources to slow-running threads. Such resources are mainly the number of CPUs, with more CPUs allocated to slow-running threads [31, 36]. Other authors make use of *hardware thread priorities* to change the instruction decode rate of each thread running on an SMT architecture. This type of software-controllable processor resource allocation makes it possible to balance parallel applications [18, 19], but memory-bound applications are insensitive to decoding priority [17] and many CMP processors have only one thread per core.

Adjusting the workload of each thread or adding more processors to slow-running threads may seem a natural solution to the problem. However, uniformly resorting to such solutions may forfeit the possible benefits stemming from other options, which may sometimes be potentially closer to the root of the problem. For instance, in the case of CMPs, most of them share some cache resources. It has been demonstrated many times in the literature that such a sharing may affect the performance of the threads that share the cache [27, 53, 90, 108]. A corollary of the body of work is that, in CMPs, the cache replacement policy implicitly determines the relative speed of each thread. Generalizing this observation, the hypothesis of this chapter is that we can dynamically partition the cache (shared by the parallel threads) in such a way that the impact of the cache replacement policy (and its direct impact on the relative speed of each thread) leads to *load balance*. Although there has been a significant amount of work on dynamic cache partitioning (DCP) this work has focused on issues such as fairness, throughput, or ensuring a minimum performance for an application [52, 58, 90, 108]. To the best of our knowledge, this chapter is the first work to describe a strategy for dynamic cache partitioning whose objective is to achieve load balance. Conversely, we are not aware of any dynamic load balancing strategies that are based on controlling the cache allocation.

To this end, this chapter takes advantage of the opportunity offered by shared caches in CMP architectures to propose a dynamic mechanism which reduces the load imbalance of parallel applications (whose parallel threads share a CMP's cache). Our mechanism detects in which situations cache allocation can be used to balance applications and, in these situations, it assigns more cache resources to processes computing longer. The iterative nature of many parallel applications facilitates this task because the behavior of an application in previous iterations can be used as a *learning phase* of the algorithm in the following iterations. Furthermore, we analyze the time granularity at which cache allocation decisions should be taken. We explore both application granularity at iteration-level (in the order of milliseconds), in which the balancing algorithm can be executed in

software, and a finer granularity that requires minimal hardware support. We conclude that the software solution provides better results as it has a global vision of the imbalance of the application.

When applied to synthetic workloads, our suggested mechanism can reduce execution time by up to 28.9%. When applied to a real-world parallel application, `wrf` [73], our mechanism helps reduce the long and expensive optimization time that expert programmers spend hand-tuning the application. In particular, our proposal reduces the load imbalance of a non-optimized version of `wrf` to values comparable to an optimized version of `wrf`, in which several man-years of effort have been devoted to balance the application; the overall reduction in execution time achieved is about 7.4%. This suggests that our balancing mechanism may represent an alternative approach to considerably reduce the development time invested in balancing parallel applications manually.

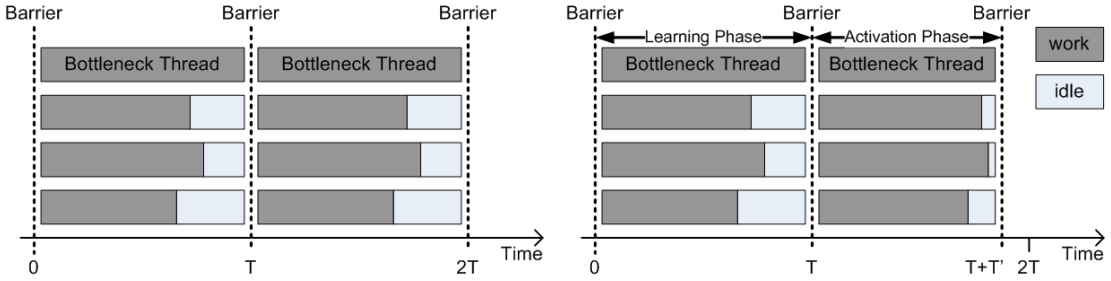
The rest of this chapter is structured as follows. Section 7.2 presents the motivation for this work. The dynamic mechanisms to balance applications are explained in detail in Section 7.3. Section 7.4 presents a theoretical analysis of the load imbalance problem. Section 7.5 describes the experimental environment, while, in Section 7.6, simulation results for synthetic workloads are discussed. Section 7.7 evaluates the mechanisms for a real-world parallel application. Finally, Section 7.8 presents the main findings of this chapter.

7.2 Motivation

In CMP architectures, other shared resources, such as cache space, can be redistributed to balance parallel applications. Caches are built up with equally-sized groups of cache lines, called *sets*. The size of each cache set is called the K-associativity of the cache. Different cache lines that collide into the same cache set can be distributed along the K different ways of the set. When a new request arrives to a set that is full, a cache eviction policy, like Least Recently Used (LRU), chooses the victim line to evict from the cache set. The LRU eviction policy is demand-driven and tends to give more cache space to threads accessing more frequently the cache hierarchy. Moreover, the OS and software cannot exercise any control over how threads share a cache when using LRU as eviction policy. In contrast, *way-partitioned* caches provide the opportunity to control cache allocation from the software level, which can lead to significant performance speed ups for sequential applications [58, 90, 108]. A way-partitioned cache prevents threads

from negative interference: a thread checks all cache lines in a set when it accesses the cache, but is allowed only to evict lines from a reduced number of ways [108]. Partitioned caches can also be used for ensuring a minimum performance to applications [52].

The use of cache allocation control mechanisms may not be a useful load balancing mechanism in all execution settings. In some cases, the execution of the parallel threads may be highly cache sensitive (and, hence, it would be feasible to balance the load dynamically through appropriate adjustment of cache resources), but, in other cases, the parallel threads may be cache insensitive (in which case, different ways to allocate cache resources may have little or no impact on their performance). Detecting whether a particular parallel workload is cache sensitive or not is something that cannot be assumed to be known; instead, some kind of on-the-fly assessment of the parallel workload, to *learn* whether it is cache sensitive or not, would be required. Assuming that the parallel workload follows some kind of an iterative pattern, where there is a repetition of a sequence consisting of a computation phase followed by a barrier synchronization, then the behavior of the application in the previous *iteration* can be used to make decisions for the next *iteration*[1].



(a) Execution without controlling the shared LLC

(b) Execution after assigning more cache space to the bottleneck thread

Figure 7.1: Synthetic example of a parallel application with 4 threads running in the same CMP

To illustrate this, consider the example shown in Figure 7.1(a), which shows the execution of a parallel application running with four threads. The application consists of a number of iterations, in each of which a computation phase is followed by barrier synchronization; two iterations are shown in the figure. The first thread in Figure 7.1(a) is the

[1]Iterative patterns are often encountered in many High-Performance Computing (HPC) Applications. For example, a partial differential equation (PDE) solver may consist of an outermost sequential loop inside which a parallel loop is executed. This specifies a number of iterations each of which completes with a call at a barrier at the joint point of the parallel loop that synchronizes all threads.

bottleneck (*bottleneck thread*), as other threads have to wait at the barrier until it finishes executing. Assuming that the threads run in a multithreaded processor sharing the last level cache (LLC), then ideally, assigning more cache resources to the first thread would reduce its execution time without excessively degrading the other threads' performance. Figure 7.1(b) shows the execution of the application after a new assignment of the cache resources has been applied to the second iteration; it is assumed that $T' < T$. If the parallel application consists of many more iterations, the new cache allocation can be used to reduce the overall execution time of the application. The idea is that, in one iteration, an appropriate detection mechanism learns the behavior of the application (*learning phase*) and activates a balancing algorithm from the next iteration of the application (*activation phase*).

Anecdotal evidence collected after many man-years of experience with several HPC applications running on the MareNostrum supercomputer [70] suggests that, usually, intrinsic load imbalance in HPC applications is due to two main reasons. First, all processes of the HPC application execute the same number of instructions but each of them has a different LLC behavior, since some processes experience more LLC misses than others. Second, all processes of the HPC application have a similar LLC behavior but the number of instructions each of them executes is different.

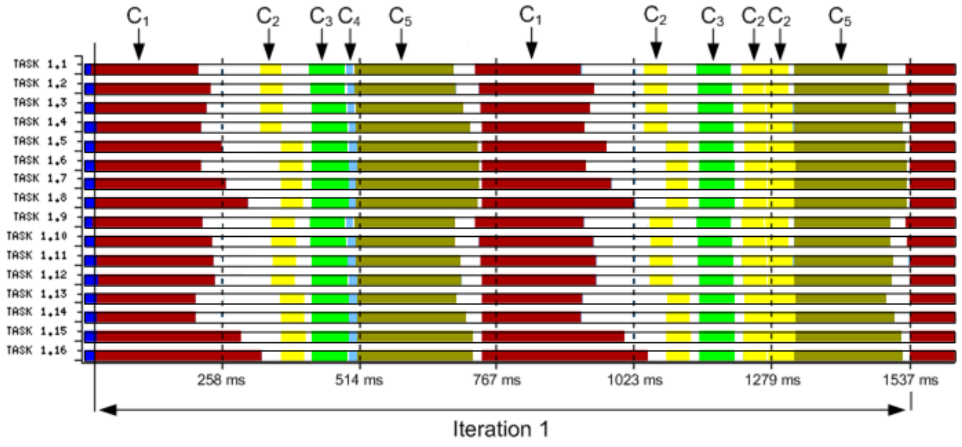


Figure 7.2: Execution of the `wrf` parallel application with 64 threads applications. Only the first 16 threads are shown for simplicity. The same behavior is observed in the other 48 MPI processes

A real-world example is given in Figure 7.2, which shows part of the execution of a parallel MPI application, `wrf` (which will be explained in detail in Section 7.7), when running on the MareNostrum supercomputer [70] with 64 MPI processes. Here we show

the execution of the first 16 threads for simplicity; the same behavior is observed in the other 48 MPI processes. Each application iteration takes about 1.5 seconds; one iteration is shown in Figure 7.2. The iteration is comprised of several *computation phases*, denoted C_i, each having a different color. At the end of each computation phase, a synchronization phase begins. Communication and waiting times are marked in white. The iteration begins with a long computation phase of about 0.3 seconds, denoted C_1. Note that many MPI processes have to wait for a long period of time until all threads finish this phase and reach the barrier. This computation phase is a clear representative of the second scenario of intrinsic imbalance mentioned above (where the number of instructions of each thread is different), as process 14 executes 40.1% less instructions than process 16. Next, there are three short phases (C_2, C_3 and C_4, during 0.2 seconds in total) and then, a long computation phase begins (C_5 between times 0.5 and 0.75 seconds). This phase is more balanced than the first one, but load imbalance is still present. In phases C_2, C_3 and C_4 the number of executed instructions is the same and the load imbalance is mainly due to different LLC behavior. Subsequently, in the same application iteration, the execution continues with computation phases C_1, C_2, C_3, C_2 and C_5. As will be shown in our experimental results section, by adjusting the cache resources allocated to each thread, and by using an appropriate learning mechanism in one iteration and applying it to the next iteration, we were able to achieve a performance improvement for the application, which is comparable to the performance obtained through manual tuning after several man-years of effort.

A key idea in this example is the notion of an *iteration*. Detecting iterations in a parallel application can be done with different approaches. Static offline profiling information can be used to inform the OS about the iteration borders. Some authors propose introducing checkpoints in the source code of the parallel application to identify possible unbalanced points in a parallel loop [21]. Other authors make use of run-time libraries that dynamically measure the percentage of load imbalance per process [31]. In fact, parallel applications alternate *computing phases* with *waiting phases* (when a process is waiting for synchronization). Thus, some authors consider the sum of a computing and a waiting phase as one iteration of the parallel application [19]. Other solutions based on analyzing performance counters are also useful to detect these parallel iterations [22]. Online detection of iterations is still an open research issue that is beyond the scope of this thesis.

7.3 Dynamic Load Balancing Through Cache Allocation

In this section, we describe in detail two different algorithms, which implement a mechanism that can dynamically change the cache allocation of a parallel application, as it runs, to reduce load imbalance. It is assumed that the parallel application consists of *iterations*, as described above. The key idea is that at the end of each iteration our mechanism invokes one of the two proposed algorithms to make a decision on whether to change the current cache allocation or not. The decision of the algorithms is based on an analysis of the behavior of the parallel threads in the preceding iteration; this decision is then applied to the immediately following iteration. The main difference of the two algorithms is their complexity. Thus, the first algorithm implements a simple heuristic that tries to give more cache space (one extra way in the cache) to slow threads by removing an equal amount of cache space from fast threads, while the second algorithm tries to (re-)allocate cache space for all threads at once, in a way that minimizes their overall execution time. The two algorithms are presented in the next section, below.

7.3.1 Iterative Method: Load Imbalance Minimization

The first algorithm proposed is based on monitoring the execution time of running threads at each iteration. At the end of the iteration, the degree of load imbalance is calculated and the algorithm responsible for reallocating ways of the cache is invoked if this imbalance is above threshold ϵ_1.

To measure the degree of load imbalance amongst parallel threads, two metrics have been suggested: the *relative load imbalance* [95] and the *imbalance percentage* (IP) [93]. The former is a ratio of the deviation of the execution time of the longest running thread from the average execution time of the threads, divided by the execution time of the longest running thread. The latter is a normalized version of the former with values between 0 and 100. High values indicate high load imbalance. We chose the latter because it makes understanding easier, especially when dealing with small numbers of threads. Thus, if we have a parallel application with N processes, $N \geq 2$, we define

$$IP = 100 \cdot \frac{MaxExecTime - AverageExecTime}{MaxExecTime} \cdot \frac{N}{N-1}.$$

Intuitively, it is useful to see the imbalance percentage (IP) as the average percentage of time that the parallel threads are waiting at the end of a parallel section for the slowest thread to finish [93].

Algorithm 7: An iterative method for load balancing using dynamic cache allocation: MinLoadImb

Data: Threshold values $0 \leq \epsilon_1, \epsilon_2 \leq 1$, execution time of all threads in the previous iteration, $ET_i^{previous}$, and current eviction policy (LRU or partitioned cache)
Result: Final cache allocation for the next iteration (LRU or new cache partition)
begin
  compute imbalance percentage, IP
  `//activation mechanism`
  if *(LRU is the current eviction policy)* **then**
    if $(IP/100 < \epsilon_1)$ **then** choose LRU and stop
    else choose a partitioned cache.
  end
  L $\leftarrow$ list of threads sorted by $ET_i^{previous}$
  while *number of threads in L* ≥ 2 **do**
    remove the slowest thread s from L
    find_fastest_thread:
      remove the fastest thread f from L
      if (thread f has no more than one assigned way)
        then if (L is not empty)
            then goto find_fastest_thread
            else stop
    if $\left(\frac{ET_s^{previous} - ET_f^{previous}}{ET_f^{previous}} > \epsilon_2\right)$
      then assign one way from thread f to thread s
  end
  stop
end

Once the balancing algorithm is activated, at the end of each iteration of the parallel application it tries to remove one way (from the cache allocation) of the fastest threads and assign it to the slowest threads. To do this, threads are ordered in terms of their execution time in the previous iteration, $ET_i^{previous}$. The fastest thread surrenders one way to the slowest thread, then the second fastest to the second slowest and so on. This is repeated until the ratio of the next slowest thread to the next fastest thread does not exceed the value of a given threshold ϵ_2, or there are no more threads to consider. During this process, the algorithm ensures that all threads retain at least one way from the cache allocation.

The intuition behind this proposal is that taking (cache) resources from fastest threads and allocating them to slowest threads may hopefully reduce the overall load imbalance. The proposed algorithm is denoted *MinLoadImb* and is described in Algorithm 7. Note that the cache partition is not changed during the execution of an iteration of the applica-

tion but only when the iteration completes. The algorithm uses as an input the execution time of each thread in the previous iteration $ET_i^{previous}$, which can be obtained using performance counters. The algorithm makes also use of two thresholds, ϵ_1 and ϵ_2. The value of ϵ_1 controls how often the algorithm will be invoked. High values indicate that the algorithm will be rarely invoked, only in cases of a high imbalance percentage. Conversely, the value of ϵ_2 controls how far the reallocation of ways from the fastest to the slowest threads would go. Again, high values indicate that only a small number of the fastest and slowest threads will be considered for reallocation of ways. After performing a sensitivity study we concluded that good performance is obtained when $\epsilon_1 = 0.075$ and $\epsilon_2 = 0.025$.

7.3.2 Single-step Method: Execution Time Minimization

The second algorithm proposed is denoted *MinExecTime*. Through the specialized hardware described in Chapter 5, *MinExecTime* can estimate the execution time of a given parallel thread with a different cache assignment (for example, when a specific number of ways are assigned to it). This information can be used to find an optimal cache partition. As before, this algorithm is also invoked at the end of each iteration of the parallel application.

Algorithm 8: A single-step method for load balancing using dynamic cache allocation: MinExecTime

Data: Threshold values $0 \leq \theta \leq 1$ and execution time of the previous iteration with the previous cache allocation, $ET_{previous}$
Result: Final cache allocation for the next iteration (LRU or new cache partition)
begin
  assign one way to each thread
  estimate ET_i for each thread with the current cache allocation
  while *(available ways exist)* **do**
    find the slowest thread s
    assign one extra way to thread s
    estimate ET_s
  end
  find the slowest thread s
  $ET_{MinExecTime} \leftarrow ET_s$
  `//activation mechanism`
  if $\left(\frac{ET_{previous} - ET_{MinExecTime}}{ET_{previous}} > \theta\right)$
    then apply the new cache partition found above
    else keep the previous cache allocation (LRU or previous cache partition)
  stop
end

To understand how to use the information provided by the specialized hardware we describe the problem to be solved next. We define $\phi(\bar{k}) = \max_i\{ET_i(k_i)\}$ the execution time of the application with the cache partition $\bar{k} = (k_1, \ldots, k_N)$, where $ET_i(k_i)$ is the execution time of thread i when k_i ways are assigned to it. Then, we are looking for the optimal cache partition $\bar{k}_{opt}$, which is the one that minimizes the execution time of the application, that is:

$$\phi(\bar{k}_{opt}) = \min_{\bar{k}=(k_1,\ldots,k_N)} \left\{ \phi(\bar{k}) \;\middle|\; \sum_{i=1}^{N} k_i = K \right\}$$

Checking all possible combinations for the values of k_i would be too expensive in time. However, we can avoid doing this by noting that $ET_i(k_i)$ is monotonically non-increasing and that $\phi(\bar{k})$ is determined by the execution time of the slowest thread. This means that we can find an optimal cache partition by following a procedure that: (i) starts with the assignment of one way to each thread; and, (ii) assigns the remaining ways, one-by-one, to the slowest thread (which thread is the slowest is recalculated after the assignment of each additional way according to the hardware estimates).

The algorithm is illustrated in Algorithm 8. Initially, one way is assigned to every thread. To minimize load imbalance, we need to minimize the execution time of the slowest thread. Clearly, it is not possible to do this if we do not assign more cache resources to this particular thread. Thus, the algorithm assigns, in each step, one extra way to the estimated slowest thread. Then, the algorithm obtains the optimal solution in $K - N$ iterations of the while loop, where K is the cache associativity and N is the number of threads. Finally, if the optimal cache partition is an improvement over LRU by more than a threshold value θ, this partition is applied to the next iteration of the application. The algorithm uses as an input the execution time of the application in the previous iteration $ET_{previous}$, which can be obtained with current performance counters. Also, with respect to the threshold θ, following sensitivity analysis, we chose to use in our experiments a value of $\theta = 0.05$.

A critical aspect of this algorithm is the use of specialized hardware to obtain performance estimations. We evaluated different approaches to obtain such estimations [77, 115] and concluded that having better accuracy in performance estimations leads to larger speed ups and less wrong activation decisions. The best results are obtained using OPACU [77], as it has an average 3.11% error over the whole SPEC CPU 2000 [105] benchmark suite. In Section 7.6 we show that, thanks to the high accuracy of this mechanism, we never

activate the cache partitioning mechanism in a situation where it would worsen performance.

Chapter 5 describes in detail the OPACU mechanism. Recall that in a CMP architecture with a shared L2 cache that is the last level cache on-chip, OPACU uses a sampled Auxiliary Tag Directory (ATD) to obtain the number of misses per L2 configuration as in [58, 90, 108]. The ATD has the same associativity and size as the tag directory of the shared L2 cache and uses the same replacement policy. It stores the behavior of memory accesses per thread in isolation. While the tag directory of the L2 cache is accessed by all threads, the ATD of a given thread is only accessed by the memory operations of that particular thread. In out-of-order architectures, different cache misses can occur concurrently if they all fit in the reorder buffer (ROB), which allows the Memory Level Parallelism (MLP) of the application to be exploited. A group of cache misses is denoted a *cluster* of misses. OPACU uses a reduced number of hardware counters to determine if L2 data misses are clustered or not. Three counters per core are needed to count the number of instructions, the number of cycles, and the average ROB usage after an L2 miss (*AROAL2M*). Three counters per cache assignment are also needed: last L2 miss identifier (cdc_i), number of clusters ($overlap_i$) and predicted total waiting cycles due to an L2 miss (PCV_i). When a load instruction accesses the L2, the sampled ATD is used to determine if it would be a miss in other possible cache assignments. Using the last L2 miss identifier, it can be determined whether a new cluster of misses begins or not. The number of clusters and lost cycles is updated accordingly. With this information, OPACU can predict the IPC of the process under different cache configurations. Figure 5.7 illustrates this mechanism.

7.3.3 Comparison of the Algorithms

Next, we list the main differences between the balancing algorithms discussed above.

First, **activation**. The activation decision represents a key feature of the load balancing mechanism. Wrong activations may lead to performance degradation as not all applications benefit from cache partition mechanisms: at one extreme, one application suffers a slowdown of up to 14% in our experimental setup when the balancing mechanisms are activated blindly. *MinLoadImb* estimates the load imbalance using the execution time of each thread in the previous iteration. However, this proposal is not aware of the effect that the assignment of one extra way to a thread will have on the performance of the other threads. As a consequence, *MinLoadImb* may suffer a performance degradation due to

wrong activation decisions. In contrast, *MinExecTime* is based on direct execution time estimations. In Section 7.6 we show that this mechanism has a high accuracy and that it never activates the dynamic cache partitioning mechanism in a situation where performance degradation would have been obtained.

The second difference concerns **convergence time**: the time needed to converge to the optimal solution is different for each mechanism. *MinLoadImb* assigns at most one extra way per thread every time it is called, while *MinExecTime* has no limit in new cache assignments. Thus, *MinExecTime* will respond more quickly to the imbalance of an application and reach the optimal partition sooner. Figure 7.3 illustrates this situation. In the first iteration, both mechanisms make use of the LRU eviction policy. In the next iteration, *MinExecTime* converges to the optimal partition, while *MinLoadImb* needs more iterations to reach the same optimal partition.

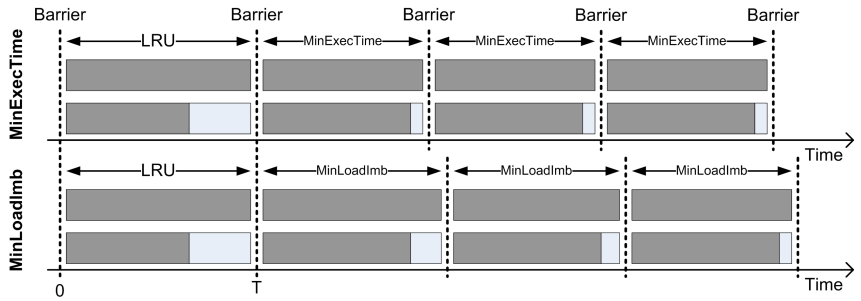


Figure 7.3: Convergence rate to the optimal cache partition solution for *MinLoadImb* and *MinExecTime*

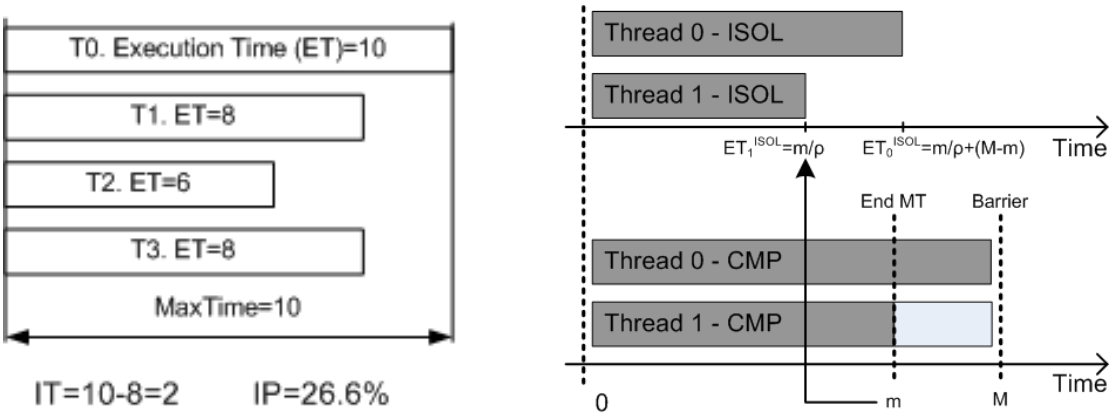
The third issue is **hardware cost**, and here, the hardware cost of thes mechanisms is significantly different. *MinLoadImb* decides cache partitions with nearly no hardware overhead, since it can read performance counters to obtain the execution time of each thread in the previous iteration. In contrast, *MinExecTime* requires special hardware to obtain performance estimations for all cache configurations. For a 4-core CMP with a shared 1MB 16-way L2 cache, OPACU needs less than 1KB of total storage per core (including a sampled ATD [90] and all the required hardware counters [77], as explained in Section 5.5). Some authors have embedded the monitoring logic inside the L2 cache, devoting some sets to monitor each thread [53]. Using a similar approach, the hardware cost of OPACU would be reduced to 204 bytes per core with a 16-way L2 cache.

7.4 Analysis of the Load Imbalance Problem

In this section we present an analytical model of the maximum execution time reduction that can be obtained with load balancing algorithms. This model allows us to better understand the balancing algorithms presented in the previous section and gives the necessary insight to explain the results presented in Sections 7.6 and 7.7.

Consider a CMP architecture with N cores and a shared last level of cache (LLC) in which we run N threads. We denote ET_i^{ISOL} the execution time of a thread i in isolation in the CMP, and ET_i^{CMP}, its execution time when running simultaneously with other threads in a workload (sharing the cache). Throughout this chapter, we use load imbalance and total execution time as performance metrics. As we have explained in the previous section, we use two different metrics to measure imbalance: Imbalance Time (IT) and Imbalance Percentage (IP) [93], both explained next.

We define $M = \max_{i=1,...,N} ET_i^{CMP}$ the execution time of the slowest thread in the workload, $m = \min_{i=1,...,N} ET_i^{CMP}$ the execution time of the fastest thread in the workload, and $Avg = \frac{1}{N}\sum_{i=1}^{N} ET_i^{CMP}$ the average execution time of all threads. Then, we have $IT = M - Avg$ and $IP = \frac{M-Avg}{M} \cdot \frac{N}{N-1}(\%)$. Figure 7.4(a) shows an example of IT and IP computation for an application with four threads. Here, we have $M = 10$, $m = 6$ and $Avg = 8$. As a result, $IT = 2$ and $IP = 26.6\%$.



(a) IT and IP computation

(b) Execution time in a two-core scenario

Figure 7.4: Load imbalance computation and definitions

The execution time of the entire workload equals the execution of the slowest thread in the workload $ET_{workload}^{CMP} = M = \max_{i=1,...,N} ET_i^{CMP}$. In contrast, the minimum time required to execute a workload equals $ET_{slowest}^{ISOL} = \max_{i=1,...,N} ET_i^{ISOL}$, where *slowest*

denotes the slowest thread in the workload. Ideally, the workload is executed so that the slowest thread is not delayed with respect to its execution time in isolation. In addition, the other threads are executed using the resources not used by the slowest thread. We assume that between synchronization points threads do not share the same address space as is the case for example in MPI applications. That is, a thread cannot *prefetch* data for another thread, and hence the minimum time to execute a thread is bound by $ET^{ISOL}_{slowest}$.

Hence, the maximum speedup, SU_{max}, in the execution time of the workload occurs when we reduce this time down to the shortest possible execution time ($ET^{ISOL}_{slowest}$):

$$SU_{max} \leq \frac{\max_{i=1,\ldots,N} ET^{CMP}_i}{\max_{i=1,\ldots,N} ET^{ISOL}_i} = \frac{ET^{CMP}_{workload}}{ET^{ISOL}_{slowest}} \tag{7.1}$$

Let us assume that all threads in the workload suffer the same slowdown because of sharing the LLC. This is a common situation in SPMD applications, since all processes carry out similar work. We can define a multiplicative factor $\rho \geq 1$ such that $ET^{CMP}_i = \rho \cdot ET^{ISOL}_i$. This parameter models the slowdown due to contention in shared resources (LLC in our case). Using this parameter in Equation 7.1 leads to the bound $SU_{max} \leq \rho$. However, this bound does not take into account that contention depends on the number of running threads. Let us define $\rho(n)$ as the slowdown due to inter-thread contention when n threads are active. Hence, $\rho(n)$ is an increasing function with the number of threads (the more active threads, the higher the contention). And $\rho(1) = 1$, as there is only one active thread. We define $\rho = \rho(N)$, the slowdown when all N threads are active. Now, let us define the average slowdown of thread i due to contention, $\bar{\rho}_i$ such that $ET^{CMP}_i = \bar{\rho}_i \cdot ET^{ISOL}_i$. With this definition, and given that we assume that all threads have similar behavior, the slowest thread when running all threads simultaneously in the CMP is the same slowest thread when running in isolation. Thus, we have $SU_{max} \leq \frac{\bar{\rho}_{slowest} \cdot ET^{ISOL}_{slowest}}{ET^{ISOL}_{slowest}}$ and $SU_{max} \leq \bar{\rho}_{slowest}$ as a consequence.

To illustrate this formula, let us focus on a two-core architecture where the slow thread executes alone in the architecture when the fast thread finishes executing (see Figure 7.4(b)). In this case, $\rho(1) = 1$, $\rho(2) = \rho$ and $ET^{ISOL}_{slowest} = \frac{m}{\rho} + (M - m)$[2]. On the other hand, we have that the imbalance percentage $IP = \frac{M - Avg}{M} \cdot \frac{N}{N-1} = \frac{M - \frac{M+m}{2}}{M} \cdot 2 = \frac{M-m}{M} = 1 - \frac{m}{M}$. Putting it all together, we have that $\bar{\rho}_{slowest} = \frac{ET^{CMP}_{slowest}}{ET^{ISOL}_{slowest}} = \frac{M}{m/\rho + (M-m)} = \frac{\rho}{m/M + \rho(1-m/M)} = \frac{\rho}{1+(\rho-1)IP}$.

Next, assume that we use a load balancing algorithm that tries to balance the applica-

[2]recall $M = \max_{i=1,\ldots,N} ET^{CMP}_i$ and $m = \min_{i=1,\ldots,N} ET^{CMP}_i$

tion (not restricted to balancing through cache allocation). In order to do this, the algorithm gives some shared hardware resources to the slowest thread, taking those resources from the remaining threads. In this scenario, we can refine the speed up bound assuming that the balancing algorithm improves the performance of the slowest thread i by a factor $\Delta_i(n)$ when running with n other threads. As a consequence of applying the balancing algorithm, the performance of the other threads is reduced (as they receive less shared resources). We have assumed that the system is fair, which implies that $\prod_{i=1}^{N} \Delta_i(n) = 1$. This assumption means that threads cannot obtain a speed up for free (this would happen if this product is larger than one) or that performance losses will be obtained (if the product is less than one). This is a limitation of our model (we are not modeling step function behavior), but the results obtained in Sections 7.6 and 7.7 show that the estimation accuracy is high.

In an instant of time t with n running threads, the total slowdown after the balancing algorithm is triggered is $\frac{\rho(n)}{\Delta_i(n)}$. As we have already said, the execution time of a thread in isolation is a lower bound of the execution time when running in the CMP: $ET_i^{CMP} \geq ET_i^{ISOL}$. Thus, we have that $\frac{\rho(n)}{\Delta_i(n)} \cdot ET_i^{ISOL} \geq ET_i^{ISOL}$, and so, $\Delta_i(n) \leq \rho(n)$. Next, in a balanced application all threads finish at the same time T, implying that $T = \frac{\rho(N) ET_i^{ISOL}}{\Delta_i(N)}$ for all $i \in \{1, \ldots, N\}$. If threads are perfectly balanced and finish at the same time T, we have $\bar{\rho}_i = \rho$ as there are always N active threads. Thus, we have:

$$T^N = \frac{\rho \cdot ET_1^{ISOL}}{\Delta_1(N)} \cdot \ldots \cdot \frac{\rho \cdot ET_N^{ISOL}}{\Delta_N(N)} = \prod_{i=1}^{N} \frac{\rho ET_i^{ISOL}}{\Delta_i(N)} \tag{7.2}$$

$$= \rho^N \cdot \frac{\prod_{i=1}^{N} ET_i^{ISOL}}{\prod_{i=1}^{N} \Delta_i} = \rho^N \cdot \prod_{i=1}^{N} ET_i^{ISOL} \tag{7.3}$$

Thus, the execution time when all threads are balanced is $T = \rho \cdot \sqrt[N]{\prod_{i=1}^{N} ET_i^{ISOL}}$, the geometric mean of individual execution time in isolation multiplied by ρ. Note that in our model, T can be reached only when $T \geq ET_{slowest}^{ISOL}$. Consequently, a new bound for the maximum speed up is obtained in Equation 7.5.

$$SU_{max} = \frac{ET_{workload}^{CMP}}{\max\left(ET_{slowest}^{ISOL}, T\right)} = \min\left(\bar{\rho}_{slowest}\ ;\ \frac{ET_{workload}^{CMP}}{T}\right) \tag{7.4}$$

$$= \min\left(\bar{\rho}_{slowest}\ ;\ \frac{ET_{workload}^{CMP}}{\rho \cdot \sqrt[N]{\prod_{i=1}^{N} ET_i^{ISOL}}}\right) \tag{7.5}$$

In a two-core architecture and using that $ET^{CMP}_{workload} = M$ and the execution time in isolation described in Figure 7.4(b), this bound translates into:

$$SU_{max} = \min\left(\frac{\rho}{1+(\rho-1)IP}\ ;\ \frac{1}{\sqrt{(1-IP)\cdot(1+(\rho-1)\cdot IP)}}\right) \tag{7.6}$$

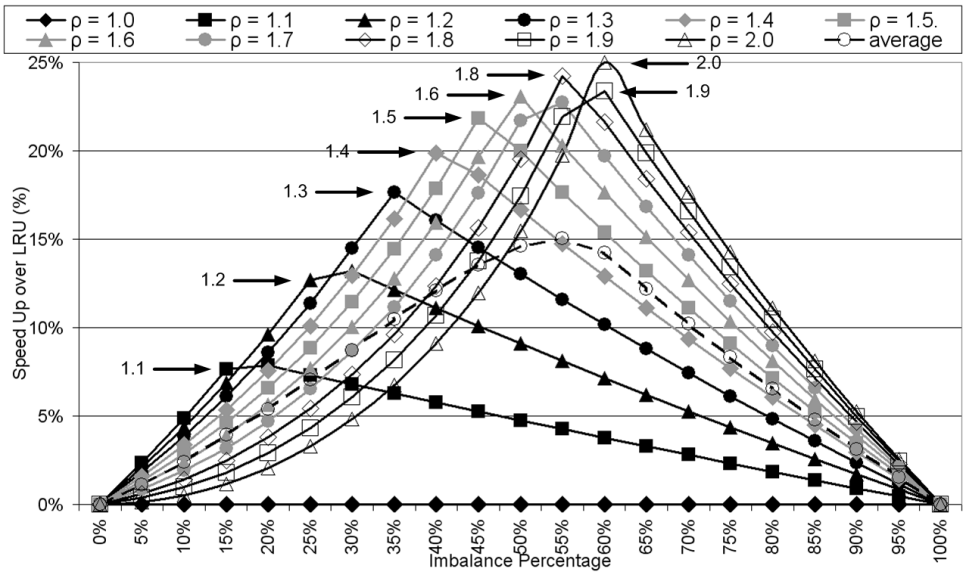


Figure 7.5: Maximum achievable speed up based on a multiplicative Δ (Formula 7.6)

Figure 7.5 shows the evolution of SU_{max} for different values of ρ and IP in a two-core architecture. Representing this figure for more threads is possible using Formula 7.5, and the same conclusions can be stated, but the intuition of metrics such as the imbalance percentage is lost. Using this model in a CMP with a shared LLC, we can conclude the following:

• Small values of ρ correspond to applications that are not much affected by the shared LLC. Consequently, execution time cannot be reduced by more than 5-10%, even for large imbalance percentages.

• When ρ becomes larger, the bound also becomes higher for large imbalance percentages. A maximum speed up of 25% is obtained with $\rho = 2.0$ and $IP = 60\%$. Note that for imbalance percentages close to 0% and 100% the bound is nearly 0%, which means that no execution time reduction can be obtained.

• If threads are perfectly balanced, there is no opportunity for performance speed ups.

• Analogously, if threads are totally unbalanced, the slowest thread will spend most

of the time running alone, and thus no performance improvements are possible.

• In intermediate situations, and depending on the value of ρ, we will have more opportunities for improvement. On average, the maximum speed up is between 5% and 15% for imbalance percentages between 20% and 80%.

In Sections 7.6 and 7.7 we evaluate the accuracy of this model through an extensive set of experiments with synthetic workloads and a real-world parallel application running on a supercomputer.

7.5 Experimental Environment

First, with regard to the **simulation configuration**, in this chapter we focus on a CMP architecture with two and four cores. We make use of the MPSim [1] simulator to model CMP architectures, as explained in Section 2.2, with a two-level cache hierarchy in which each core has a private data and instruction L1 caches and the unified L2 cache is shared among cores.

With regard to **workloads**, in order to evaluate our dynamic load balancing algorithms and model, presented in Section 7.3 and 7.4 respectively, we use both synthetic workloads and a real-world parallel application in production at our supercomputing center. Synthetic workloads allow us to evaluate our proposals in a wide range of scenarios, while the real-world HPC application completes the study and demonstrates the robustness of our proposals.

As pointed out in Section 7.2, there are two main situations of intrinsic load imbalance, namely: when all threads execute the same number of instructions but have different cache behavior, and when all threads have a similar cache behavior but the number of executed instructions is different. The objective of the synthetic workloads is to mimic these situations and to show that our proposals reduce load imbalance in both situations and that the model can be used to predict their results. We compose 2-thread workloads from SPEC CPU 2000 [105] benchmarks. From each benchmark we collect traces of the most representative 300 million instruction segment of each program, following the SimPoint methodology [100]. We artificially add a barrier at the beginning and end of the execution of these benchmarks to mimic a parallel HPC application. In our simulation methodology, each thread introduces a different skew to virtual addresses, which avoids different cores hitting the same cache set. Furthermore, each copy of the same program is forwarded a different number of instructions to avoid each copy being at exactly the same

point of execution.

These synthetic workloads are designed to have different cache necessities depending on their *building* benchmarks. To that end, we have used the classification explained in Section 2.4 of SPEC CPU 2000 benchmarks into three groups: *Low utility* (L), *Small working set* or *saturated utility* (S) and *High utility* (H).

Finally, we evaluate our model and load balancing algorithms (Section 7.3) with real traces from a parallel application running on an actual supercomputer: `wrf`. We used two versions of this application: (i) an *optimized* version, currently in production at our supercomputing center, to which several man-years of effort have been devoted to its (manual) optimization; and (ii) a *non-optimized* version of the same application to which less optimization techniques have been applied. Thus, we have two versions of the same application, `wrf`, with different imbalance percentages. Section 7.7 presents our evaluation results, including a full description of how traces of `wrf` were obtained, which was a complex and time-consuming task.

7.6 Performance Characterization with Synthetic Workloads

In this section we analyze the accuracy of the analytic model presented in Section 7.4 and the behavior of the load balancing algorithms, presented in Section 7.3, using synthetic workloads. We simulate a CMP architecture with two cores for the sake of clarity, although all the conclusions of this section apply to CMPs with N cores. In the experiments in this section, we run two benchmarks simultaneously. When the fastest benchmark ends, the slowest keeps executing until it finishes its execution. This is what we call an *iteration*. In each execution we run ten iterations. It is only at the end of the first iteration that the load balancing algorithms may be activated (since the first iteration is used purely as a learning phase), thus, we report the average execution time of the final nine iterations with respect to the results of the first iteration.

As pointed out in Section 7.2, there are two situations of (intrinsic) load imbalance that can be solved with our proposal. Section 7.6.1 evaluates the situation in which all threads execute a similar number of instructions but have different L2 cache behavior. Next, Section 7.6.2 evaluates the situation where all threads have a similar L2 cache behavior but the number of instructions is different.

7.6.1 Load Imbalance due to Different L2 Cache Behavior

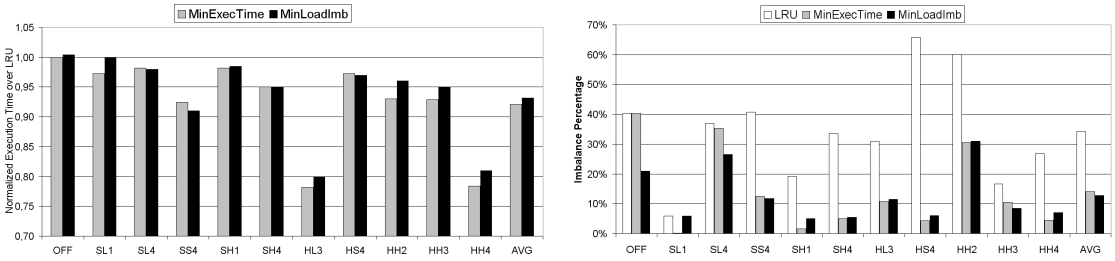
To mimic the situation where all threads execute the same number of instructions but have different L2 cache behavior, we randomly generate 36 pairs of benchmarks with different cache requirements. We choose the workloads listed in Table 7.1. We denote *XY* the pairings of benchmarks where *X* and *Y* are the slowest and fastest thread, respectively.

Table 7.1: Workloads of benchmarks with different L2 cache behavior. Four workloads per group are chosen

Group	Slow Th	Fast Th	IP (%)	Group	Slow Th	Fast Th	IP (%)	Group	Slow Th	Fast Th	IP (%)
LL1	sixtrack	bzip2	44.8	LS1	applu	gzip	48.3	LH1	lucas	art	29.9
LL2	mcf	wupwise	95.2	LS2	sixtrack	eon	33.9	LH2	swim	vpr	9.19
LL3	lucas	gap	63.5	LS3	mcf	vortex	95.3	LH3	mcf	facerec	93.6
LL4	equake	swim	30.8	LS4	gap	perlbmk	62.4	LH4	wupwise	apsi	35.7
SL1	perlbmk	mesa	5.95	SS1	crafty	eon	23.3	SH1	crafty	apsi	19.3
SL2	vortex	wupwise	2.11	SS2	vortex	crafty	14.8	SH2	gcc	apsi	20.6
SL3	gcc	sixtrack	8.78	SS3	gcc	perlbmk	35.1	SH3	vortex	ammp	16.2
SL4	crafty	bzip2	37.0	SS4	vortex	gzip	40.8	SH4	vortex	apsi	33.6
HL1	galgel	bzip2	60.8	HS1	twolf	crafty	54.0	HH1	galgel	ammp	6.42
HL2	mgrid	gap	28.5	HS2	ammp	gcc	30.4	HH2	parser	apsi	60.1
HL3	parser	applu	30.9	HS3	gap	mgrid	28.5	HH3	twolf	art	16.6
HL4	fma3d	swim	73.8	HS4	parser	perlbmk	65.8	HH4	vpr	facerec	26.8

Figure 7.6(a) shows the total execution time for both load balancing algorithms: *MinLoadImb* and *MinExecTime*. All results are normalized to the execution time of the first iteration, in which the LRU replacement policy is used. All workloads in which *MinExecTime* decides to make use of LRU as eviction policy are represented in the first set of bars, denoted *OFF* (as the cache partitioning algorithm is not activated) with their average results. Clearly, the average normalized execution time with respect to LRU is equal to 1 in case of *MinExecTime*. *MinLoadImb* is sometimes triggered in these workloads, without leading to any improvement of the execution time with respect to LRU. In fact, the imbalance percentage is reduced from 40% to 20% but this is not translated into any reduction in the execution time. This result suggests that the activation mechanism of *MinExecTime* is more accurate than the one of *MinLoadImb*.

All algorithms present similar results in groups LL, LS and LH as the shared L2 has little impact on the performance of the bottleneck thread ($\bar{\rho}_{slowest} \approx 1$). Consequently, *MinExecTime* detects that there is no room for improvement and keeps using LRU as eviction policy (the results are represented in bar *OFF*). In contrast, *MinLoadImb* is triggered



(a) Execution time normalized with respect to LRU

(b) Imbalance percentage

Figure 7.6: Execution time and imbalance percentage for pairings with different L2 cache behavior

in all these workloads as the imbalance percentage is greater than 7.5% in all of them.

When LRU is not harming the performance of the slowest thread, there is no room for improvement as $\bar{\rho}_{slowest} \approx 1$. The *MinExecTime* activation mechanism detects these situations and does not try to balance the application. Instead, *MinLoadImb* is triggered in 89.6% of the situations without significantly reducing the execution time. In general, *MinLoadImb* is triggered more often than *MinExecTime*, even in situations where significant performance degradations are obtained (9.6% in SH3 and 8.4% in HS2[3]). When LRU is clearly biased toward the fastest thread (SS4, HL3, HH2, HH3 and HH4), the reduction in execution time is more significant, ranging between 4.1% and 21.6% (HH4). In intermediate situations, the bottleneck thread is less affected by the fast thread and the performance speed ups are smaller (between 1.5% and 4.5%).

MinLoadImb converges more slowly to the optimal solution than *MinExecTime*, and as a result it normally returns worse results. However, as the number of total iterations of the application increases, this difference in performance decreases as both algorithms eventually reach an optimal partition. In some pairings, *MinLoadImb* gets stuck in a local minimum because the threshold ϵ_2 may be too big for the situation, while in other situations it suffers a ping-pong effect between the optimal and a suboptimal partition (because ϵ_2 may be too small for the situation). These problems do not occur with *MinExecTime*, since it estimates the execution time for each cache configuration and finds the optimal partition in one step.

On average, *MinLoadImb* and *MinExecTime* outperform LRU by 6.9% and 7.9%, respectively. Figure 7.6(b) shows the imbalance percentage for both algorithms with different workloads. Our algorithms reduce the imbalance percentage from an overall average

[3]Workloads SH3 and HS2 are not on Figures 7.6(a) and 7.6(b) as they are represented in the bar *OFF*

of 34.3% to 12.7% (*MinLoadImb*) and 14.1% (*MinExecTime*). In general, *MinLoadImb* succeeds in reducing the imbalance percentage more than *MinExecTime*, since this metric (that is, load imbalance minimization) is guiding the *MinLoadImb* algorithm.

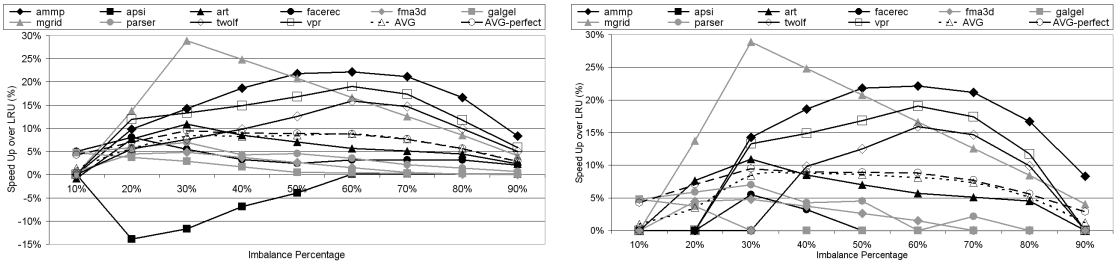
7.6.2 Load Imbalance due to a Different Instruction Count

In the next experiment, we run two instances of the *same* benchmark and we artificially generate load imbalance by reducing the number of executed instructions of the second thread. To generate an $I\%$ of imbalance, we appropriately reduce the number of instructions of the second trace. This experiment mimics the situation where all threads have the same cache behavior but different instruction count. As in the previous section, we repeat each iteration ten times and report the average execution time of the final nine iterations with respect to the results of the first iteration.

Given that both threads have the same cache behavior, LRU devotes a roughly equal portion of the cache to each thread. As a consequence, LL and SS pairings will leave no opportunity for execution time reduction. When the bottleneck thread belongs to group L, it will not be affected by the other executing threads. The same situation will happen if the bottleneck thread belongs to group S, as the cache requirements of each thread are fulfilled. Thus, for these benchmarks we cannot obtain significant improvements in execution time, even for high values of imbalance percentage because $\bar{\rho}_{slowest} \approx 1$. In these situations, *MinExecTime* is able to detect that it cannot obtain any performance gain, and it keeps LRU as replacement policy, avoiding any performance degradation. In contrast, *MinLoadImb* is triggered in 95% of the situations with a negligible impact on the execution time. For HH pairings, the situation is different, since an execution time reduction is possible through cache partitioning. Next, we study the pairs consisting of two instances of each of the 10 H benchmarks in more detail.

The speed-up that both proposed algorithms achieve over LRU with the 10 pairs of the H benchmarks and for an imbalance percentage varying from 10% to 90% is shown in Figures 7.7(a) and 7.7(b). The line denoted *AVG* indicates the average results with each algorithm among all benchmarks for a given imbalance percentage. The line denoted *AVG-perfect* indicates the average results of a partitioning algorithm with perfect knowledge that always makes the correct decision.

Studying the behavior of *MinLoadImb* in Figure 7.7(a), we observe that all pairs except `apsi` obtain significant performance speed ups as we vary the imbalance percentage of the application. In the case of `apsi` there is a performance degradation of 13.9% with



(a) Speed up over LRU with *MinLoadImb*

(b) Speed up over LRU with *MinExecTime*

Figure 7.7: Execution time reduction for HH pairings of SPEC CPU 2000 benchmarks with an imbalance due to a different number of executed instructions

an imbalance percentage of 20%. This is a consequence of the nature of *MinLoadImb* and its activation mechanism. In contrast, *MinExecTime* never degrades performance with respect to LRU. This is corroborated by the results in Figure 7.7(b). In fact, the activation mechanism of *MinExecTime* is conservative and may lose opportunities for performance improvements with small imbalance percentages. However, this is the cost that has to be paid in order to never lose performance with respect to LRU. Thus, on average, the speed up achieved by *MinExecTime* is 5.8%. In contrast, the *MinLoadImb* activation mechanism is much more aggressive and is triggered more often, reaching an average 6.3% improvement. As a consequence, in some situations it loses performance with respect to LRU (up to 13.9% of degradation in the case of `apsi`). Finally, it is noted that a perfect prediction mechanism, which always tries to balance an application when there are opportunities to do so, would obtain an average speed up of 7.1%. The gap is not very high, but closing the gap between the proposed mechanisms and a perfect predictor is part of our future work.

Another observation from both figures is that the improvement is, in general, less when the imbalance percentage is very high. This can easily be explained, because when the imbalance percentage is, say, 90%, it means that most of the time one thread is executing on its own, hence there are only limited opportunities for dynamic cache partitioning. Also, despite the differences of the two algorithms, the behavior of each benchmark is largely determined by its characteristics. Thus, the maximum speed up is obtained with `mgrid` with an imbalance percentage of 30%, reaching a speed up of 28.9%. The behavior of `ammp`, `twolf` and `vpr` corresponds to benchmarks that are very sensitive to cache allocation, reaching speed ups between 15% and 25%. In fact, their behavior matches the theoretical speed ups when ρ is between 1.6 and 1.9, which is very close to the measured

ρ in these workloads (1.80, 1.78 and 1.92, respectively). Other benchmarks, such as `art`, `facerec`, `fma3d`, `galgel`, `mgrid` or `parser`, are less sensitive to the cache allocation and exhibit speed ups between 5% and 10%. These results correspond to smaller values of ρ in the model (between 1.1 and 1.3), which also matches the measured value of ρ (1.36, 1.19, 1.16, 1.31, 1.35 and 1.22, respectively). The obtained results are close to the maximum speed ups predicted by the model explained in Section 7.4, which shows the robustness of the analytical model. Small differences are mainly due to the asymmetry in performance/cache space figures.

Table 7.2 shows the accuracy of the activation mechanism for *MinLoadImb* and *MinExecTime*. We compare their results with the correct decision that would be taken by an ideal activation mechanism with perfect knowledge. When LRU obtains better performance than a partitioned cache, the correct decision should be to keep LRU as eviction policy (denoted *No* in the 2nd row of Table 7.2). Conversely, if the partitioned cache obtains better performance, the correct decision should be to activate the load balancing algorithm (denoted *Yes* in the 1st row of Table 7.2). *MinExecTime* is never triggered when there is no opportunity for execution time reduction. This was the main design goal of this mechanism, because this situation may lead to heavy execution time degradation of the application. On the other hand, in 53.3% of the cases in which it is possible to benefit from cache partitioning to balance the workload, *MinExectime* is triggered. The lost opportunities are 24.4% of the total cases. As a future work we plan to develop a more aggressive prediction mechanism to decrease this loss of opportunities. In contrast, *MinLoadImb* is much more aggressive than *MinExecTime* and is triggered is 94.4% of the situations, leading to better average results, but suffering significant performance degradations in some situations.

Table 7.2: Accuracy of the activation mechanism with HH pairings

		MinLoadImb decision		*MinExecTime* decision	
		Yes	No	Yes	No
Correct decision (perfect knowledge)	Yes (77.7%)	73.3%	4.4%	53.3%	24.4%
	No (22.3%)	21.1%	1.2%	0%	22.3%

7.6.3 Granularity Analysis of the Load Balancing Mechanism

So far, we have assumed that the cache allocation is changed at the boundary of a computation phase (that is, at the end of each iteration). This is a coarse enough granularity to

implement this solution in software at the OS or runtime level. The cache partition found out at the end of one iteration is maintained throughout the whole iteration that follows (and its computation phases).

As an alternative to the previous approach, we can use a dynamic algorithm that changes the cache partition during the execution of a computation phase (inside each iteration). This mechanism continuously monitors the load imbalance (or predicted execution time) and adapts the cache partition to IPC variations inside the same computation phase. An algorithm working at this granularity is invoked periodically and executed in a dedicated hardware, such as in [90]. In our baseline configuration, cache partitions are decided every 5 million cycles[4]. As cache partitions are modified during the execution of a phase, we denote our algorithms *Dynamic-MinLoadImb* and *Dynamic-MinExecTime*.

In the case of ***Dynamic-MinLoadImb***, at the end of the first iteration of the application, the mechanism decides to activate the load balancing algorithm if the load imbalance of the application is above a threshold ϵ_1 (implemented at software level). During the next iteration new cache partitions are decided at intervals of 5 million cycles.

This hardware estimates the execution time of each thread with the current cache partition using the current IPC of the application and stores the result in the *Execution Time Table* (ETT). This structure sorts all threads according to their execution time. Once we have filled the ETT with all the values, we use Algorithm 7 to partition the cache. The fastest threads give one extra way to the slowest ones if their difference in execution time is greater than ϵ_2 and the slowest thread has more than one way. No extra ways will be assigned to a thread that has already finished executing. The main difference with the static algorithm is that here we use the predicted remaining execution time instead of the execution time in the last interval. In fact, using the IPC of the application in the last iteration (instead of the current IPC), we will obtain the same cache partition as with *MinLoadImb*.

In the case of ***Dynamic-MinExecTime***, the hardware implementation of this proposal is similar to the previous mechanism, since it also requires the ETT. The basic difference is that a dedicated hardware [77] provides the IPC of the application with a different cache assignment. First, the execution time of each thread with only one way is estimated and sorted in the ETT. Then, one way is assigned to the slowest thread and its new execution time is estimated and inserted again in the ETT. This process is repeated $K - N$ times

[4]The frequency of cache partition decision has been chosen after evaluating the mechanism for a wide range of values

until all ways have been assigned (see Algorithm 8).

Next, we account for the total extra storage needed for the versions of the load balancing algorithms implemented in hardware explained in this section. In the case of *Dynamic-MinLoadImb*, for an 8-core CMP, we need less than 20 bytes of total storage per thread. In the case of *Dynamic-MinExecTime*, we need extra storage for the IPC predictions. Thus, for an 8-core CMP with a 16-way L2 cache, we need less than 80 bytes of total storage per thread.

We evaluated the dynamic approaches with the same workloads as in the previous sections. These mechanisms show similar performance to the static ones. It is clear that the dynamic mechanisms that change the cache partition during the execution of a computation phase will react to phase changes and converge more quickly to the optimal partition. However, there are problems with benchmarks that exhibit large IPC variations during their execution. This point may seem counter-intuitive, but optimal decisions at one point of a computation phase may not be the optimal partition for the entire computation phase. This is the case of `apsi` in pair HH2 with `parser`: In the first 54.6% of the time, `apsi` has an IPC of 0.94 instructions per cycle. In the remaining 45.4% of the time it has an average IPC of 3.59 instructions per cycle. Thus, dynamic algorithms try to assign as much cache space as possible to `apsi` because it is assumed to be the bottleneck of the parallel application. However, this is a wrong decision as later it automatically catches up `parser`, which is the real bottleneck in this pairing. When benchmarks behave in a similar way during the entire execution, the dynamic mechanisms obtain better results.

On average, the performance benefits obtained with static balancing algorithms are slightly better than the dynamic mechanisms (between 2.3% and 2.7% in execution time and between 2.5% and 3% in imbalance percentage). For that reason, we conclude that using a static mechanism is more suitable to balance parallel applications.

7.6.4 Conclusions

In general, the largest reduction in execution time is obtained when a thread with more cache accesses and less cache utility is executed with a thread with more cache utility and less cache accesses. In this situation, an eviction policy such as LRU cannot restrict the cache space assigned to the thread with less cache requirements, harming the performance of the other thread. Next, we list the main findings of this section.

1) When threads are not cache sensitive, adjusting the cache space devoted to each thread has almost no impact on performance, as $\bar{\rho}_{slowest} \approx 1$.

2) When threads have different cache behavior and the bottleneck thread benefits little from additional cache space, there is little room for improvement, as $\bar{\rho}_{slowest} \approx 1$.

3) When threads have different cache behavior and the bottleneck thread is more sensitive to cache space, we can manage to improve total execution time by reducing the load imbalance.

4) When threads have similar cache behavior, performance speed ups are obtained when all threads are classified as high utility.

5) Cache partition decisions should be made at the beginning of an iteration of the application, since the load balancing algorithms provide better results when they have a global vision of the imbalance of the application.

Finally, we have shown that the model previously introduced is robust among a wide variety of synthetic workloads. In the next section, we show that the same conclusions apply to a real-world parallel application.

7.7 Performance Evaluation with a Parallel HPC Application

7.7.1 Extracting a Representative Trace from a Parallel HPC Application

In this section, we evaluate our balancing algorithms with real traces from a parallel HPC application running on an actual supercomputer: `wrf`. The Weather Research and Forecasting (`wrf`) model [73] is a mesoscale numerical weather prediction system designed to serve both operational forecasting and atmospheric research needs. In this experiment, we use the non hydrostatic mesoscale model dynamical core.

Simulating all threads of the parallel application implies a significant amount of simulation time, since these applications usually run for days or weeks on a supercomputer. We use an automatic mechanism to choose the most representative computation regions to be traced and simulated with a cycle-accurate simulator [40]. The simulation methodology starts with a `paraver` [62] trace file generated with OMPItrace package [84]. This trace file consists of a complete timestamped sequence of events of the whole execution of a parallel application. We used an automatic methodology to extract the internal structure of the trace [22], which allows us to select the most meaningful part of the trace file. This methodology uses non-linear filtering and spectral analysis techniques to determine the internal structure of the trace and detect periodicity of applications. The methodology

makes it possible to cut the original parallel trace and to generate a new trace between 10 and 150 times shorter [22], but still in the order of minutes of real execution and not affordable for a cycle-accurate simulator.

Next, we use a clustering algorithm to determine the most representative computation bursts inside an iteration of the new trace. In [41] the authors use a density-based clustering algorithm applied to the hardware counters offered by modern processors to automatically detect the representative sections of a parallel application. This algorithm obtains R representative sections of each computation phase, where R is given by the user. As we model a CMP with four cores, we select four representatives of each computation phase with the same load imbalance as the entire phase.

We used these reduced trace files to feed up the cycle-accurate architecture simulator described in Section 7.5. We simulate all threads sharing the L2 cache in a CMP architecture. When a thread finishes executing, it waits until all other threads have also finished. Finally, the obtained speed ups in the computation phases execution time are passed to `dimemas` [39] (a high-level MPI application simulator) to estimate the total execution time of the parallel application.

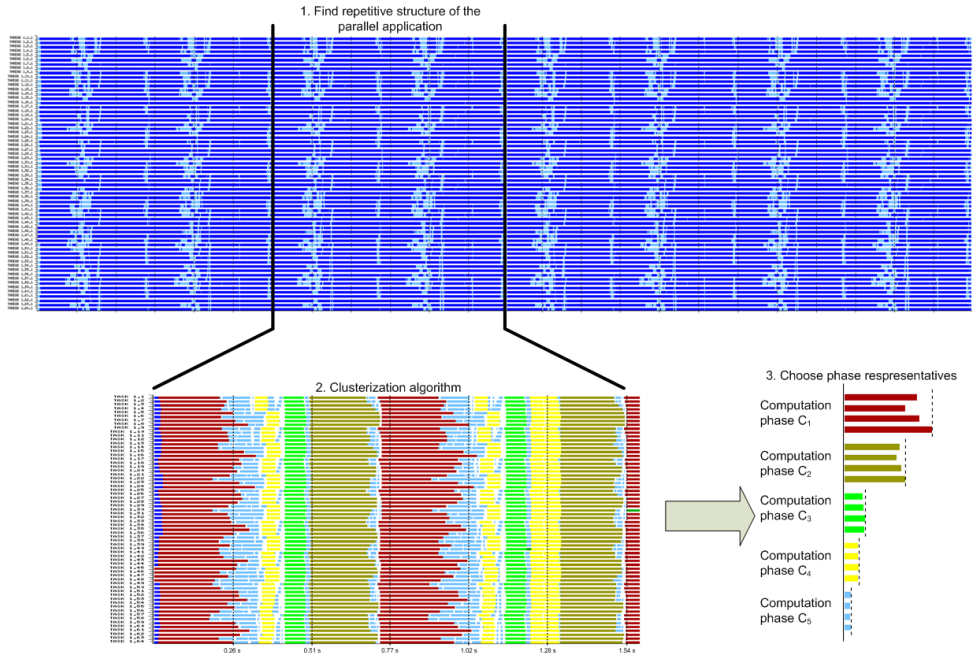


Figure 7.8: Experimental methodology to obtain representative traces of parallel applications. Example with `wrf` with 64 MPI processes. Four representatives are chosen per computation phase

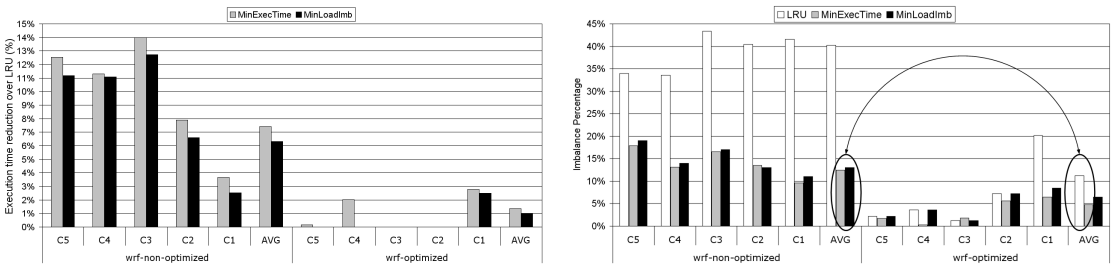
Figure 7.8 exemplifies this methodology for a real application running on a super-

computer. We execute `wrf` with 64 MPI processes. On the top of the figure, we show the beginning of the original trace. First, we determine the periodicity of the application and cut the meaningful part of the trace. This process reduces an original trace from 5.06GB to 38.7MB. Next, the clustering algorithm detects five representative computations phases. For example, computation phase C_1 is executed twice during one iteration of the application. Finally, we choose 4 representative processes of each computation phase with the same load imbalance as the original phase. Note that selecting one representative for each phase reduces the simulation time by a factor of 2.5x in this application (we simulate only 40% of the total time of an iteration), while choosing 4 representatives out of 64 gives a 16x speed up.

The simulation methodology described above significantly reduces the time needed to obtain an estimation of execution time. In the case of a real application such as `wrf` running with 64 threads, this methodology reduces simulation time by three orders of magnitude (5300x speed up). Using the selected traces on a single processing machine, we would need approximately half a day to estimate the speed up of a configuration, which would make it unaffordable to simulate the whole application (we would need more than 100 days of simulation time).

7.7.2 Case Study with a Real HPC Application: `wrf`

Using the methodology described in the previous section, we extracted a trace from an execution of `wrf` with 64 MPI processes and obtained 4 representatives for the 5 computation phases that compose the application.



(a) Execution time reduction over LRU

(b) Imbalance Percentage

Figure 7.9: Imbalance metrics when using LRU and the balancing algorithms with `wrf` (CMP architecture with 4 cores and a shared 1MB 16 ways L2 cache)

In this evaluation, we use two versions of the same application: `wrf-non-optimized` and `wrf-optimized`. The former is a version of the application that has not been op-

timized for our supercomputer infrastructure. Consequently, it suffers from the load imbalance problem: the average imbalance percentage is 41.6%. After a long optimization process, requiring several man-years of effort, the application has been heavily tuned to solve this problem and reduce the imbalance percentage to 11.2%; we call this version `wrf-optimized`.

Figure 7.9(a) shows the reduction in execution time obtained with *MinLoadImb* and *MinExecTime* with the two versions of the parallel application. In the case of `wrf-non-optimized`, the execution time with *MinExecTime* is consistently reduced, reaching 14% speed up in computation phase C_3 and 7.4% on average (average results take into consideration the weight of each phase in the application). The same behavior is observed with *MinLoadImb*, although the performance speed ups are not as good because the mechanism converges to the optimal partition more slowly. The obtained reduction in execution time with our algorithms matches the estimation of the analytical model in Section 7.4: the measured values of ρ are between 1.03 and 1.21, the imbalance is between 33.6% and 43.4%, and the expected improvements are in the range $[1.5\% \; ; \; 13.2\%]$.

MinExecTime also improves the performance of `wrf-optimized`, where a significant effort from the application writers was devoted to balance it, by 1.4%. The mechanism is not activated in phases C_2 and C_3 (LRU is maintained for these phases). For *MinLoadImb*, the balancing mechanism is only activated in phase C_1, obtaining an average 1.0% reduction in execution time.

Figure 7.9(b) shows the improvements in imbalance percentage obtained with our mechanisms. In the case of `wrf-non-optimized`, the imbalance percentage is consistently reduced, from an average of 41.6% to 12.4% (*MinExecTime*) and to 13.0% for *MinLoadImb*. Computation phase C_3 was the most unbalanced phase when using LRU, which explains the large speed ups in execution time that were obtained. In the case of `wrf-optimized`, the original 11.2% imbalance percentage is reduced to 5.1% for *MinExecTime* and to 6.5% for *MinLoadImb*. The most important conclusion of Figure 7.9(b) is that our balancing algorithms reduce the imbalance of `wrf-non-optimized` to the same values of `wrf-optimized`. That is, in both cases the imbalance percentage is similar, which means that, in balancing the non-optimized code, we reach the same performance with the optimized code, without having to spend several man-years of effort in order to change the code of the application.

7.8 Summary

In this chapter we present a model that estimates the maximum reduction in execution time which a load balancing algorithm can obtain when balancing a parallel HPC application. Thanks to this model, we know in which situations execution time can be reduced, which gives us the necessary to develop two load balancing algorithms for parallel applications, both of which make use of a dynamic cache allocation mechanism to balance the application.

Our balancing algorithms have a learning phase in which the mechanism monitors the usage of the shared last level cache in a CMP, determining whether it is useful to trigger cache partitioning or keep using LRU. When triggered, our algorithms assign more cache space to the slowest threads of the application. We suggest that these assignments should be done at the end of each computation phase, since finer granularities may have problems with workloads that exhibit large IPC variations during their execution. The proposed model and load balancing algorithms were validated through an extensive set of experiments with synthetic workloads. Our balancing algorithms reduce the execution time with synthetic workloads by up to 28.9%.

We also applied the proposed algorithms to a real-world parallel application. The balancing algorithms reduce the imbalance of `wrf-non-optimized` to 12%, the same value as in `wrf-optimized` in which the optimization phase required several man-years of effort. Overall we obtained a 7.4% execution time reduction. This is encouraging, because it indicates that our dynamic load balancing algorithms may represent a one-time effort that may considerably reduce the development time invested in balancing HPC applications.

Chapter 8
Multicore Resource Management in the Manycore Era

General-purpose computing is moving onto diverse devices such as cell phones, digital entertainment centers, and data center servers. At the same time, conventional monolithic processor designs have pushed technology to its fundamental limits, and consequently, the systems community has shifted its focus to distributed multicore architectures. Multicore architectures are more efficient than large monolithic processors, but they present a number of new challenges. At the same time, over the last three decades, general-purpose system architecture has become overly specialized for single-threaded personal computers.

Contemporary general-purpose system architecture, as exemplified by desktop/laptop computers, is too slight a foundation upon which to build scalable multicore systems capable of satisfying the diverse demands of future systems. Therefore, in this chapter, we present our vision of future multicore system architectures. We propose to enrich the current interaction between system software and architecture, allowing the application and system software to explicitly manage a multicore system's resources in order to satisfy system and application specific objectives. Finally, we would like to mention that this chapter has been developed with the collaboration of Kyle J. Nesbit and James E. Smith from the University of Wisconsin-Madison, United States.

8.1 Introduction

Continuing the long-term trend of increasing integration, the number of cores per chip is projected to increase with each successive new technology generation. These chips yield increasingly powerful systems with reduced cost and improved efficiency. At the same time, general-purpose computing is moving off desktops and onto new devices such as

cell phones, digital entertainment centers, and data-center servers [66]. These computers must have the key features of today's general-purpose systems (high performance and programmability) while satisfying stricter cost, power, and real-time performance constraints.

An important aspect of chip multiprocessors (CMP) is improved hardware resource utilization. On a CMP, concurrently executing threads can share costly microarchitecture resources that would otherwise be underutilized, such as off-chip bandwidth. Higher resource utilization improves aggregate performance and enables lower-cost design alternatives, such as smaller die area or less exotic battery technology. However, increased resource sharing presents a number of new design challenges. In particular, greater hardware resource sharing among concurrently executing threads can cause individual thread performance to become unpredictable and might lead to violations of the individual applications' performance requirements [23, 75, 82].

Traditionally, operating systems (OS) are responsible for managing shared hardware resources, processor(s), memory, and I/O. This works well in systems where processors are independent entities, each with its own microarchitecture resources. However, in CMPs, processors are concurrently executing threads that compete with each other for fine-grain microarchitecture resources. Hence, conventional operating system policies do not have adequate control over hardware resource management. To make matters worse, the operating system's software policies and the hardware policies in the independently developed microarchitecture might conflict. Consequently, this compromises operating system policies directed at overall system priorities and real-time performance objectives.

In the context of future applications, this poses a serious system design problem, making current resource management mechanisms and policies no longer adequate for future multicore systems. Policies, mechanisms, and the interfaces between them must change to fit the multicore era.

In this chapter, we outline our vision for resource management in future multicore systems, which involves enriched interaction between system software and hardware. Our goal is for the application and system software to coordinately manage *all* the shared hardware resources in a multicore system. For example, a developer or end user will specify an application's Quality of Service (QoS) objectives, and the developer or system software stack will translate these objectives into hardware resource assignments. Because QoS objectives are often application specific, the envisioned multicore architecture provides an efficient and general interface that can satisfy QoS objectives over a range of

applications. By enriching the interaction between hardware and software, the envisioned resource management framework facilitates a more efficient, better performing platform design.

Designing general-purpose systems requires a clear separation of policies and mechanisms [65]. Policies provide solutions; for flexibility, policies should be implemented in software. Mechanisms provide the primitives for constructing policies. Because primitives are universal, system designers can implement mechanisms in both hardware and software. In general, mechanisms that interact directly with fine-grain hardware resources should be implemented in hardware; to reduce hardware cost, mechanisms that manage coarse-grain resources should be implemented in software.

8.2 Virtual Private Machines

In a traditional multiprogrammed system, the operating system assigns each application a portion of the physical resources (for example, physical memory and processor time slices). From the application's perspective, each application has its own private machine with a corresponding amount of physical memory and processing capabilities. With CMPs containing shared microarchitecture level resources, however, an application's machine is no longer private, and consequently, resource usage by other independent applications can affect its resources.

As a result of these considerations, we make use of the virtual private machine (VPM) framework as a means of handling resource management in systems based on CMPs [82]. This framework was introduced by Nesbit et al. [82] to manage the cache hierarchy in CMP architectures. VPMs are similar in principle to classical virtual machines. However, classical virtual machines virtualize a system's functionality [88] (ignoring implementation features), while VPMs virtualize a system's performance and power characteristics, which are implementation specific. A VPM consists of a complete set of virtual hardware resources, both spatial (physical) resources and temporal resources (time slices). These include the shared microarchitecture-level resources. By definition, a VPM has the same performance and power characteristics as a real machine with an equivalent set of hardware resources.

The VPM abstraction provides the conceptual interface between policies and mechanisms. VPM policies, implemented primarily in software, translate application and system objectives into VPM resource assignments, thereby managing system resources.

Then, VPM mechanisms securely multiplex, arbitrate, or distribute hardware resources to satisfy the VPM assignments.

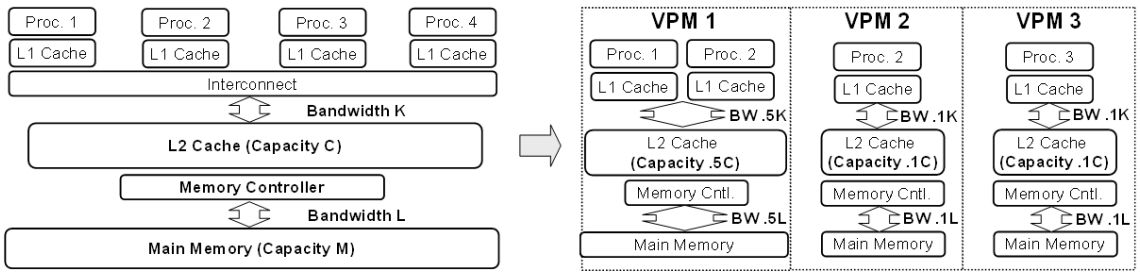


Figure 8.1: Virtual private machine (VPM) spatial component. The policy has distributed the CMP's resources among three VPMs. After assigning VPM 1 50 percent of the shared resources and VPMs 2 and 3 each 10 percent, it leaves 30 percent of the cache and memory resources unallocated for excess service

8.2.1 Spatial Component

A VPM's spatial component specifies the fractions of the system's physical resources that are dedicated to the VPM during the time(s) that it executes a thread. For example, consider a baseline system containing four processors, each with a private L1 cache. The processors share an L2 cache, main memory, and supporting interconnection structures. Figure 8.1 shows that the policy has distributed these resources among three VPMs. VPM 1 contains two processors, and VPMs 2 and 3 each contain a single processor. The policy assigns VPM 1 a significant fraction (50 percent) of the shared resources to support a demanding multithreaded multimedia application and assigns the other two VPMs only 10 percent of the resources. These assignments leave 30 percent of the cache and memory resources unallocated; these resources are called excess service. Excess service policies distribute excess service to improve overall resource utilization and optimize secondary performance objectives.

In our example, we focus on shared memory hierarchy resources, but VPM concepts also apply to internal processor resources, such as issue queue entries, and instruction decode and execution bandwidth [23]. Furthermore, we can apply the same concepts to architected resources such as memory capacity and I/O.

As Figure 8.1 illustrates, a VPM's spatial component might contain multiple processors. Multiprocessor VPMs are a natural extension of gang scheduling and support hierarchical resource management [42, 85]. For example, schedulers and resource management policies running within a multiprocessor VPM can schedule and manage the

VPM's processors and resources as if the VPM were a real multiprocessor machine. That is, multiprocessor VPMs can support recursive virtualization [88].

8.2.2 Temporal Component

A VPM's temporal component is based on the well-established concept of ideal proportional sharing [86]. It specifies the fraction of processor time (processor time slices) that a VPM's spatial resources are dedicated to the VPM (see Figure 8.2). As with spatial VPM resources, a VPM's temporal component naturally lends itself to recursive virtualization and hierarchical resources management, and excess temporal service might exist.

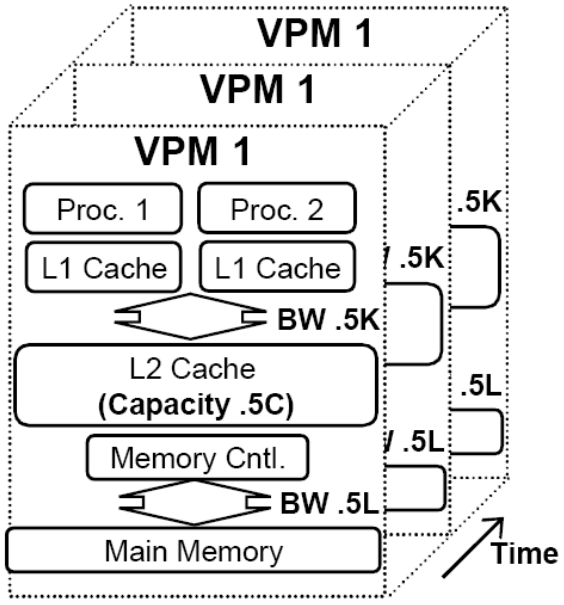


Figure 8.2: VPMs consist of a spatial component and a temporal component. The temporal component specifies the fraction of processor time that a VPM's spatial resources are dedicated to the VPM

8.2.3 Minimum and Maximum VPMs

To satisfy objectives, policies might assign applications minimum or maximum VPMs, or both, depending on the objectives. Mechanisms ensure an application is offered a VPM that is greater than, or equal to, the application's assigned minimum VPM. Informally, the mechanisms offer an application at least the resources of its assigned minimum VPM. When combined with the assumption that an application will only perform better if it is offered additional resources (performance monotonicity [64]), ensuring a minimum VPM assignment leads to desirable performance isolation; that is, the running application

performs at least as well as it would if it were executing on a real machine with a configuration equivalent to the application's assigned VPM. This performance level is assured, regardless of the other applications in the system.

Mechanisms can also ensure that an application receives no more than its maximum VPM resources. Policies can use maximum VPM assignments to control applications' power consumption, which is based on the assumption that an application's power consumption is a monotonically increasing function of its resource usage (power monotonicity). For maximum VPM assignments to improve power savings significantly, they should be supported by mechanisms that power down unused resources. Furthermore, because temperature and transistor wear-out strongly depend on power consumption, policies can use maximum VPMs to control temperature and lifetime reliability [97]. Lastly, application developers can use maximum VPMs to test whether a minimum VPM configuration satisfies an application's real-time performance requirements [64].

8.3 Policies

VPM assignments satisfy real-time performance, aggregate performance, power, temperature, and lifetime reliability objectives. Overall, the VPM policy design space is enormous and is a fertile area for future research. Here, we begin with a high-level overview of the policy design space and then discuss the basic types of policies and how they interact in our envisioned system architecture.

In general, we envision two basic types of policies. Application-level policies satisfy an application's QoS objectives by translating the QoS objectives into a VPM configuration as well as scheduling and managing VPM resources assigned to the application. System policies satisfy system objectives by controlling the distribution of VPM resources among applications. System policies control resource distribution by granting and rejecting requests for VPM resources and revoking VPM resources when the system is overloaded.

The policy architecture's main feature is its extensibility: we clearly separate policies from mechanisms [65], and policies are easy to replace or modify on a per system and per application basis [38]. For example, in Figure 8.3, a system is running five applications, each within its own VPM (not shown). The first application (on the left) is a real-time recognition application. To satisfy its real-time requirements, the application is running with an application-specific VPM that the application's developer computed offline. The

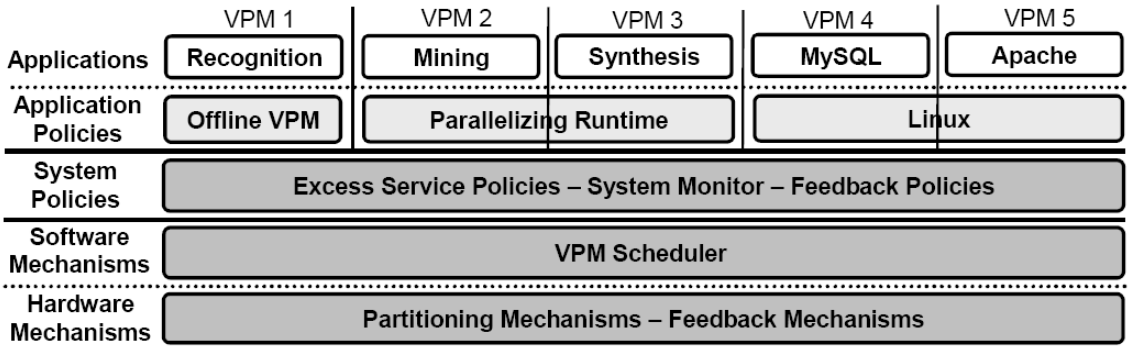


Figure 8.3: The VPM system architecture consists of application-level policies, system policies, software mechanisms, and hardware mechanisms. The extensible policy architecture lets policy builders modify policies on a per system and per application basis

second and third applications (mining and synthesis) are written in a domain-specific, concurrent programming language. The language includes a run-time that computes VPM configurations online and optimizes applications' concurrency in coordination with the applications' VPM resources. The mining and synthesis applications are isolated from each other, but they share the library code which implements the language's run-time. The last two applications (MySQL and Apache) are standard Linux applications which are oblivious to the underlying system's VPM support. The applications are running on a virtual version of Linux, which in turn is running on a thin software layer that monitors the applications' workload characteristics and roughly computes VPM configurations, using heuristics and ad hoc techniques. Most importantly, application-level policies allow application developers and run-time libraries to customize a system's behavior to satisfy a range of applications' requirements.

8.3.1 Application-level Policies

In general, there are two logical steps for determining VPM assignments: modeling and translation. We describe the steps as separate phases, although in practice they may be combined. As we described earlier, application policies compute VPM configurations either online (automated) or offline (manually). Online policies are based primarily on run-time analysis (for example, by using performance counters), while offline policies require an application developer to perform most of the program analysis. Online/offline hybrid policies are also feasible.

In addition, application policies can use standardized application-level abstraction layers which provide developers with abstract machine models. Such models are often imple-

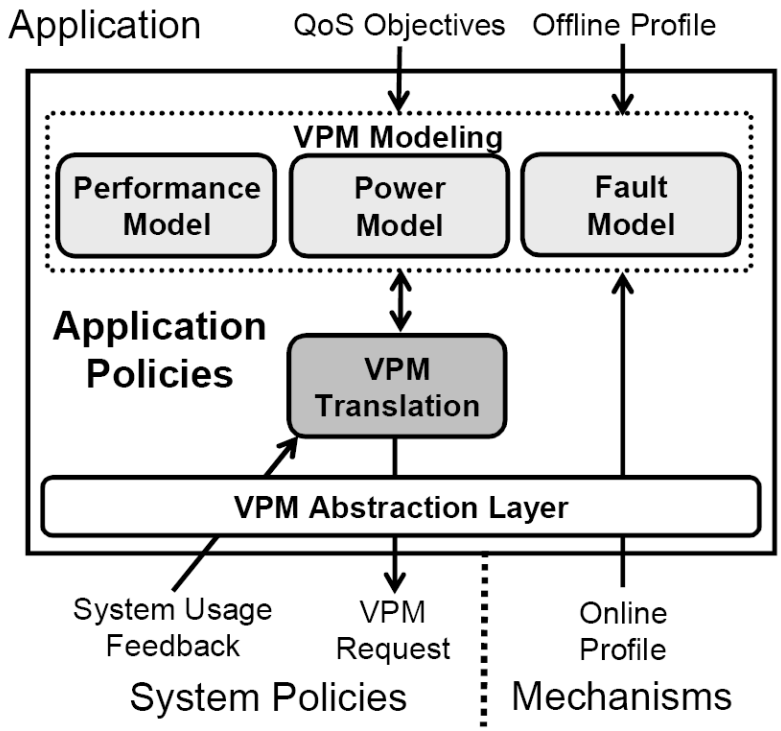


Figure 8.4: Application policies compute VPM configurations in two logical steps: VPM modeling and translation. Standardized application-level abstractions can be used to abstract away irrelevant implementation specific VPM details

mentation independent and have performance characteristics that are easier for developers to reason about (see Figure 8.4).

With respect to *VPM modeling*, the VPM modeling step maps application QoS objectives, such as minimum performance and maximum power, to VPM configurations. The sophistication of VPM modeling techniques spans a fairly wide range. At one end are simple, general-purpose analytical models that use generic online profiling information to roughly predict an application's performance and power when running on different VPM configurations. At the other end are models specialized to a specific application (or a domain of applications) through offline profiling and characterization. Such offline models can precisely capture an application's performance and power on different VPM configurations.

Application- or domain-specific VPM modeling can provide more precise predictions, but might require more developer effort (for example, to determine suitable VPM configurations offline). Applications with critical QoS objectives (such as real-time applications) will generally require more precise VPM modeling.

Turning now to *VPM translation*, the VPM translation step uses the VPM models to find VPM configurations which satisfy an application's QoS objectives. Multiple VPM configurations can satisfy the same objective. For example, multiple minimum VPM configurations can satisfy a single real-time performance objective; that is, one suitable VPM configuration might have a larger spatial component and smaller temporal component, while another suitable VPM might have a larger temporal component and smaller spatial component. Or there might be different combinations of resources within the same spatial component.

Furthermore, applications might have multiple objectives that a policy can use to prune the number of suitable VPM configurations. For example, an application policy can search for a VPM configuration that satisfies a real-time performance requirement and minimizes power consumption. Moreover, a policy can assign an application a maximum VPM to bound the application's power consumption. In many practical situations, finding the optimal VPM configuration is NP-hard. Consequently, online translation generally must use approximate and heuristic-based optimization techniques. For applications with critical QoS objectives, the application developer can do a portion of the VPM translation offline. Moreover, the developer can combine the VPM modeling and translation steps into a single step. For example, an application developer might use maximum VPM assignments and trial and error to compute a minimum VPM configuration that satisfies their application's real-time performance objective [64].

Once the policy has found a suitable VPM configuration, it initiates a request to the system's policies for the VPM resources. If the VPM resources are available, the system policies securely bind the VPM resources to the application [38]. When a system is heavily loaded, the system policies might reject an application's VPM request or revoke a previous VPM resource binding.

An application policy must implement a procedure for handling VPM rejections and revocations. When a rejection or revocation occurs, an application policy can find another suitable VPM configuration or reconfigure the application to reduce the application's resource requirements. For example, a real-time media player can downgrade its video quality or simply return an insufficient resource error message and exit.

To help with global (systemwide) resource optimization, VPM system policies can provide feedback to the applications' policies. The feedback informs the applications of global resource usage. An application's policies can use the system feedback information to further prune the suitable VPM configurations and find a VPM that is amenable

to systemwide resource constraints. In addition, an application's policies can use VPM modeling and online profiling information to dynamically respond to changing workload characteristics.

A third consideration is the *VPM abstraction layer*. In our discussion so far, we have assumed a relatively high-level VPM abstraction (for example, VPMs that consist of shares of cache bandwidth, cache storage, and memory system bandwidth). However, real hardware resources are more complex. For example, a physical cache implementation consists of banks, sets, and ways. VPM mechanisms do not abstract hardware resources; that is, the VPM mechanisms convey implementation specific details to the application policies. Exposing implementation details creates an interface that is more efficient to implement and grants more latitude to implementers of higher-level abstractions [38].

However, exposing implementation details to applications can make applications implementation dependent. Moreover, many application policies do not need low-level implementation details to satisfy their QoS objectives. To improve application compatibility, many application policies can use standardized VPM abstraction layers. A VPM abstraction layer provides a high-level VPM that abstracts away a system's implementation-specific details while capturing the system's first-order performance and power characteristics. At run-time, the abstraction layer efficiently translates high-level VPM configurations into low-level VPM assignments that the system's mechanisms support.

8.3.2 System Policies

System policies control the distribution of VPM resources by dealing with applications' VPM requests. System policies have three basic components: a system monitor, feedback policies, and excess service policies, as shown in Figure 8.5.

The *system monitor* tracks the load on the system's resources and detects system overload. A system is overloaded if it does not have enough resources to satisfy its applications' VPM assignments. In general, it is best to detect overload as early as possible, such as when an application initiates a VPM request that causes the system to become overloaded. To detect system overload, the system must monitor any shared resource that can become overloaded, such as cache storage, execution bandwidth, cooling capacity, and power delivery. We can detect that a VPM request overloads a system's physical resources with an admission control test [43]. For example, the system monitor can compare the set of dedicated VPM resources with the capacity of the system's physical resources.

Detecting that a VPM request overloads a system's less tangible resource constraints

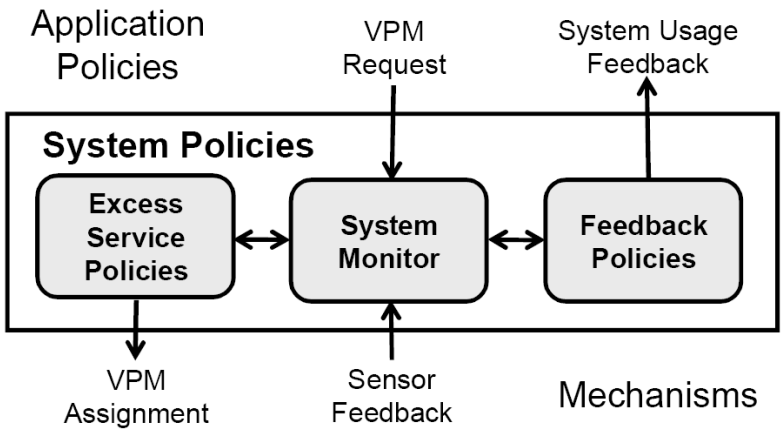


Figure 8.5: System policies deal with VPM requests and monitor application resource usage to ensure that the system does not become overloaded

(such as a power-delivery system, cooling capacity, or transistor lifetime reliability) is more difficult. The system monitor must use VPM models to translate between less tangible resource constraints and VPM configurations. The VPM models should be accurate enough to detect most overload conditions at the time an application initiates a VPM request; however, they do not have to be completely accurate. The system monitor's VPM models are used to supplement conventional hardware sensors, such as voltage and temperature sensors. Hardware sensors can detect anomalous overload conditions (such as overheating) in real-time and prevent catastrophic failures. If a sensor detects an overloaded resource, the system policies have accepted more VPM requests than the system's physical resources can satisfy. In this case, the system policies must revoke some applications' assigned VPM resources.

With regard to *feedback policies*, these determine which applications' VPM resources should be rejected or revoked when the system is overloaded. For example, when a system is overloaded, a feedback policy might reject any incoming VPM request or revoke the VPM resources of the application with the lowest user-assigned priority. Feedback policies also inform applications of systemwide resource utilization and the causes of rejected or revoked VPM requests (for example, the system might have insufficient power resources available).

As we described earlier, applications' policies can use the feedback to compute a suitable VPM configuration that is amenable to systemwide resource usage. The feedback policies should also provide applications with up-to-date systemwide resource usage information regardless of VPM rejections and revocations. In this way, the feedback poli-

cies can direct the global optimization of a system's resource usage.

The third element is *excess service policies*. After application and system policies have settled on a suitable set of VPM assignments, there might be excess service available. Excess service is service that is unassigned or unused by the application to which it is assigned. Excess service policies distribute excess service to optimize system objectives. For example, these policies can distribute service to improve response time or throughput averaged over all applications, to conserve power or transistor lifetime, or a combination of such objectives. To optimize power or transistor lifetime objectives, excess service policies prevent applications from consuming the excess service. That is, these policies assign tasks maximum VPMs, thus causing the excess resources to be powered off. Excess service policies must also ensure that distributing excess service does not violate applications' maximum VPM assignments.

In most cases, excess service policies transparently adjust applications' VPM assignments. For example, they can transparently increase a minimum VPM assignment or decrease a maximum VPM assignment without violating the application policy's VPM assignments. For some resources, the policies must notify an application's policies when excess resources are added (for example, they must notify an application's thread scheduler when processors are added to a VPM).

For some resources, excess service becomes available and must be distributed at a fine time granularity, such as with SDRAM memory system bandwidth. In such cases, a portion of the excess service policy must be implemented in hardware. However, a policy's hardware portion should be simple, parameterized, and general; that is, it should work with multiple software excess service policies.

8.4 Mechanisms

There are three basic types of mechanisms needed to support the VPM framework: a VPM scheduler, partitioning mechanisms, and feedback mechanisms, as shown in Figure 8.6. The first two types securely multiplex, arbitrate, or distribute hardware resources in order to satisfy VPM assignments. The third type provides feedback to application and system policies. Because mechanisms are universal, system builders can implement mechanisms in both hardware and software. Generally, the VPM scheduler is implemented in software (in a microkernel [38] or a virtual machine monitor [103]), while the partitioning mechanisms and feedback mechanisms are primarily implemented in hardware. Although a

basic set of VPM mechanisms are available [23, 75, 82], many research opportunities remain to develop more efficient and robust VPM mechanisms.

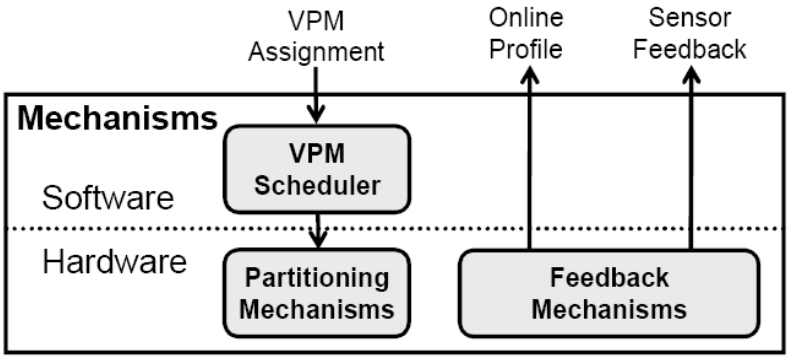


Figure 8.6: VPM mechanisms are implemented in hardware and software. The mechanisms satisfy VPM resource assignments and provide feedback regarding individual application and systemwide resource usage

8.4.1 VPM Scheduler

The VPM scheduler satisfies applications' temporal VPM assignments by time-slicing hardware threads. The VPM scheduler is a proportional-fair scheduler [13], but it must also ensure that co-scheduled applications' spatial resource assignments do not conflict (that is, it must ensure that the set of co-scheduled threads' spatial resource assignments match the physical resources available, and do not oversubscribe any microarchitecture resources). VPM scheduling in its full generality, satisfying proportional fairness without spatial conflicts, is an open research problem.

When the VPM scheduler context switches an application onto a processor (or a group of processors), the scheduler communicates the application's spatial VPM assignment to the hardware partitioning mechanisms through privileged control registers. Once the control registers are configured, the VPM resources are securely bound to the application. Secure binding decouples the authorization from resource usage [38] (that is, once a resource is securely bound to an application, the application's policies can schedule and manage its VPM resources without re-authorization). In this way, VPMs can efficiently support hierarchical scheduling [42].

The VPM scheduler we describe here is a first-level scheduler (or root scheduler), while application-level schedulers are second-level schedulers. Hierarchical scheduling is useful for satisfying different classes of QoS requirements [42]. Furthermore, precise

application-level schedulers will play an important role in future parallel programming models.

8.4.2 Partitioning Mechanisms

To satisfy the spatial component of VPM assignments, each shared microarchitecture resource must be under the control of a partitioning mechanism that can enforce minimum and maximum resource assignments. As we described earlier, the resource assignments are stored in privileged control registers that the VPM scheduler configures. In general, each shared microarchitecture resource is one of three basic types of resources: a memory storage, buffer, or bandwidth resource. Each type of resource has a basic type of partitioning mechanism. For example, thread-aware replacement algorithms partition storage resources (main memory and cache storage [75, 78, 82, 108]), upstream flow control mechanisms partition buffer resources (issue queue and miss status handling registers [23]), and fair-queuing and traffic-shaping arbiters partition bandwidth resources (execution and memory ports [81]). These basic techniques, combined with the proper control logic, can sufficiently enforce minimum and maximum resource assignments.

For maximum VPM assignments to be useful, the partitioning mechanisms must be accompanied by mechanisms to power down resources during periods of inactivity. An example would be mechanisms that clock-gate unused pipeline stages and transition inactive memory storage resources into a low-power state. As we mentioned earlier, by controlling resources' power consumption, policies can control other important characteristics such as die temperature and transistor wear out [88].

8.4.3 Feedback Mechanisms

Mechanisms also provide application and system policies with feedback regarding physical resource capacity and usage. Feedback mechanisms communicate to system policies the capacity of the system's resources and the available VPM partitioning mechanisms. They also provide application policies with information regarding individual applications' resource usage and performance.

Application resource usage information should be independent of the system architecture and the application's VPM assignments. For example, a mechanism that measures a stack distance histogram can predict cache storage behavior for many different cache sizes as we have shown in the previous chapters. Other mechanisms such as OPACU

would make it possible to obtain performance projections of the application with other resource assignments.

Lastly, feedback mechanisms provide information regarding overall resource utilization. For example, a system's mechanisms should provide system policies with die temperature and power consumption information.

8.5 Summary

Overall, the presented framework provides a solid foundation for future architecture research, but many challenges remain in evaluating this framework. First, the framework targets system-level metrics that occur over time granularities which preclude the use of detailed simulation. Second, many of the applications we are interested in (such as smart phone or cloud computer applications) are unavailable or unknown. To address these problems, we plan to develop most of the framework with analytical models and reduce the simulation time as much as possible using well-known techniques like sampled or statistical simulation.

Chapter 9
Conclusions

This chapter summarizes the main contributions of this thesis and presents an analysis of the results shown in each chapter. This chapter also presents future lines of work opened up by this thesis.

9.1 Goals, Contributions and Main Conclusions

In the last few years, chip multiprocessors (CMP) have become widely used in academia and industry in order to increase the system aggregated performance. CMPs increase hardware resource utilization, while reducing design costs and average power consumption by exploiting design re-use and simpler processor cores.

However, these new multithreaded architectures also have to face the challenge of making a better use of shared resources. A key shared resource in CMP architectures is the cache hierarchy, since this is one of the resources that has the most impact on the final performance of the application. In CMP architectures, shared last levels of cache (LLC) have become popular as they make it possible to increase the utilization of the cache (and consequently, aggregate performance) and to simplify the coherence protocol.

As we have seen in this thesis, applications make very different use of the cache hierarchy, depending on their data re-use and access patterns. When running multiple applications in a CMP with a shared cache, undesired situations can occur where a subset of the applications monopolizes the shared cache, degrading the performance of the others. And even worse, the Operating System (OS) has no way of enforcing a Quality of Service (QoS) to applications.

In this thesis, we propose software and hardware mechanisms whose aim is to improve cache sharing in CMP architectures. We make use of a holistic approach, coordinating targets of software and hardware, in order to improve system aggregate performance and provide QoS to applications. We make use of explicit resource allocation techniques to

control the shared cache in a CMP architecture, with resource allocation targets driven by hardware and software mechanisms.

The main contributions of this thesis are the following.

- We have characterized different single- and multithreaded applications and classified workloads using a systematic method, in order to better understand and explain the cache sharing effects on a CMP architecture. We have made a special effort to study previous cache partitioning techniques for CMP architectures, in order to acquire the insights necessary to propose improved mechanisms.

- In CMP architectures with out-of-order processors, cache misses can be served in parallel and share the miss penalty to access main memory. We take this fact into account in order to propose new cache partitioning algorithms which are guided by the memory-level parallelism (MLP) of each application. With these algorithms, system performance is improved (in terms of throughput and fairness) without significantly increasing the hardware required by previous proposals.

- Driving cache partition decisions with indirect indicators of performance such as misses, MLP, or data re-use, may lead to sub-optimal cache partitions. Ideally, the appropriate metric to drive cache partitions should be the target metric to optimize, which is normally related to IPC. Thus, we have developed a hardware mechanism, namely OPACU, that is able to obtain accurate predictions at run-time of the performance of an application when running with different cache assignments.

- Using performance predictions, we have introduced a new framework to manage shared caches in CMP architectures, namely FlexDCP, which allows the OS to optimize different IPC-related target metrics, such as throughput or fairness, and provide QoS to applications. FlexDCP allows for an enhanced coordination between the hardware and the software layers, which in turn leads to improved system performance and flexibility.

- Next, we have made use of performance estimations in order to reduce the load imbalance problem in parallel applications. We built a run-time mechanism that detects parallel applications sensitive to cache allocation and, in these situations, reduces the load imbalance by assigning more cache space to the slowest threads. This mechanism helps to reduce the long optimization time in terms of man-years of effort devoted to large-scale parallel applications.

- Finally, we have stated the main characteristics that future multicore processors with thousands of cores should have. An enhanced coordination between the software and hardware layers has been proposed to better manage the shared resources in these architectures.

9.2 Future Work

The different techniques presented in this dissertation can be further enhanced or extended. Moreover, this thesis opens up several new topics which we want to explore further. Among others, we would like to highlight the following:

- *Low power cache designs*. A reduction in the power dissipated in the cache can be obtained by adjusting the active hardware resources to the requirements of the applications. A direct estimation of the performance of the application (using IPC predictions obtained with OPACU mechanism) makes it possible to obtain the desired trade-off between power consumption and performance.

- *Parallel applications with shared data*. Dynamic cache partitioning (DCP) techniques can also be used in the case of parallel applications in which threads concurrently work on the same data. The parallel application can be seen as a whole accessing the shared cache. With that goal, bit masks and monitoring logic should be assigned to processes instead of to cores.

- *Enhanced feedback mechanisms between the architecture and the OS*. CMP architectures with shared caches introduce complexities when accounting for CPU utilization. This is due to the fact that the progress made by an application during an interval of time depends to a great extent on the activity of the other applications which it is co-scheduled with. An inaccurate measurement of the CPU utilization affects several key aspects of the system, such as the application scheduling, or the charging mechanism in data centers. CPU accounting should be aware of how different co-scheduled applications are sharing the cache, which can be achieved by using the monitoring logic described in this thesis.

- *Predictable performance for hard real-time applications*. The increasing demand for new functionalities in current and future hard real-time embedded systems like automotive, avionics and space industries is driving an increase in the performance

required in embedded processors. Multicore processors represent a good design solution for such systems, due to their high performance, low cost and power consumption characteristics. However, hard real-time embedded systems require time analyzability, and current multicore processors are less analyzable than single-core processors, due to the interferences between different tasks when accessing shared hardware resources. Cache partitioning techniques can be used to allow CMPs to be analyzed.

Some of these topics are already being developed. We hope to deal with the remaining topics in the near future.

9.3 Publications

Below we list the publications which our research to date has produced. First, we list accepted publications related to this thesis. Then, we enumerate submitted publications for which we are still waiting for an answer and, finally, we list other publications on other topics that we have done in the past few years.

9.3.1 Accepted Publications

- M. Moretó, F. J. Cazorla, A. Ramirez, R. Sakellariou and M. Valero. *FlexDCP: a QoS framework for CMP architectures*. In ACM SIGOPS Operating System Review, Special Issue on the Interaction among the OS, Compilers, and Multicore Processors, April 2009.

- K. J. Nesbit, M. Moretó, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. *Multicore Resource Management*. In IEEE Micro, Special Issue on Interaction of Computer Architecture and Operating System in the Manycore Era, vol. 38, no. 3, May/June 2008.

- M. Moretó, F. J. Cazorla, A. Ramirez and M. Valero. *Dynamic Cache Partitioning based on the MLP of Cache Misses*. In Transactions on High Performance Embedded Architectures and Compilers. vol. 3, no. 1, March 2008.

- M. Moretó, F. J. Cazorla, A. Ramirez and M. Valero. *MLP-aware dynamic cache partitioning*. In International Conference on High Performance Embedded Architectures and Compilers (HiPEAC). Goteborg, Sweden, January 2008.

- M. Moretó, F. J. Cazorla, A. Ramirez and M. Valero. *MLP-aware dynamic cache partitioning*. In International Conference on Parallel Architectures and Compilation Techniques (PACT), Poster Abstracts. Brasov, Romania, September 2007.

- M. Moretó, F. J. Cazorla, A. Ramirez and M. Valero. *Online Prediction of Throughput for Different Cache Sizes*. In XVIII Jornadas de Paralelismo. Zaragoza, Spain, September 2007.

- M. Moretó, F. J. Cazorla, A. Ramirez and M. Valero. *Online Prediction of Applications Cache Utility*. In International Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS). Samos, Greece, July 2007.

- M. Moretó, F. J. Cazorla, A. Ramirez and M. Valero. *Explaining Dynamic Cache Partitioning Speed Ups*. In IEEE Computer Architecture Letters, vol. 6, no. 1, March 2007.

- M. Moretó, F. J. Cazorla, A. Ramirez and M. Valero. *Reducing Simulation Time*. In Advanced Computer Architecture and Compilation for Embedded Systems (ACACES), Poster Abstracts. L'Aquila, Italy, July 2006, pp. 233-236. Academic Press, ISBN 90 382 0981 9.

9.3.2 Submitted Articles for Publication

- M. Moretó, F. J. Cazorla, R. Sakellariou and M. Valero. *Load Balancing Through Cache Allocation*. Submitted to Computing Frontiers 2010.

- J. González, M. Casas, M. Moretó, J. Gimenez, A. Ramirez, J. Labarta and M. Valero, *Simulating Whole Supercomputer Applications*. Submitted to ICS 2010.

9.3.3 Other Publications

- K. Kędzierski, M. Moretó, F. J. Cazorla and M. Valero. *Adapting Cache Partitioning Algorithms to the pseudo-LRU Replacement Policy*. To appear in IPDPS 2010.

- J. Cámara, M. Moretó, E. Vallejo, R. Beivide, C. Martínez, J. Miguel-Alonso and J. Navaridas. *Twisted Torus Topologies for Enhanced Interconnection Networks*. In IEEE Transactions on Parallel and Distributed Systems (TPDS). To appear in 2010.

- C. Luque, M. Moretó, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu and M. Valero. *ITCA: Inter-Task Conflict-Aware CPU Accounting for CMPs*. In PACT 2009. Raleigh, USA, September 2009.

- K. Kędzierski, M. Moretó, *Bringing Cache Partitioning in CMPs to Reality*. In PACT 2009. Poster Abstracts. Raleigh, USA, September 2009.

- A. Ramirez, M. Alvarez, F. Cabarcas, M. Moretó, A. Rico, and C. Villavieja. *Experiencia en el desarrollo colaborativo de documentos usando Wiki*. Jornadas Docentes del Departamento de Arquitectura de Computadores (JoDoDAC). Barcelona, Spain, February 2009.

- C. Luque, M. Moretó, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu and M. Valero. *CPU accounting in CMP Processors*. In IEEE Computer Architecture Letters. Volume 8, Issue 1, January 2009.

- C. Martínez, M. Moretó, R. Beivide, E. Gabidulin and E. Stafford. *Modeling Toroidal Networks with the Gaussian Integers*. In IEEE Transactions on Computers, vol. 57, no. 8, August 2008.

- P. A. Castillo, J. J. Merelo, M. Moretó, F. J. Cazorla, M. Valero, A. M. Mora, J. L. J. Laredo, and S.A. McKee. *Evolutionary system for prediction and optimization of hardware architecture performance*. In IEEE Congress on Evolutionary Computation (CEC). Hong Kong, June 2008.

- P. A. Castillo, A. M. Mora, J. J. Merelo, J. L. J. Laredo, M. Moretó, F. J. Cazorla, M. Valero, and S.A. McKee. *Architecture performance prediction using evolutionary artificial neural networks*. In European Workshop on Hardware Optimization Techniques (EVOHot). Napoli, Italy. March 2008.

- J. Cámara, M. Moretó, E. Vallejo, R. Beivide, C. Martínez, J. Miguel-Alonso and J. Navaridas. *Mixed-radix Twisted Torus Interconnection Networks*. In IEEE International Parallel and Distributed Processing Symposium (IPDPS). Long beach, USA, March 2007.

- C. Martínez, M. Moretó, R. Beivide and E. Gabidulin. *A Generalization of Perfect Lee Codes over Gaussian Integers*. In IEEE International Symposium on Information Theory. Seattle, USA, July 2006.

- C. Martínez, E. Vallejo, R. Beivide, C. Izu and M. Moretó. *Dense Gaussian Networks: Suitable Topologies for On-Chip Multiprocessors*. In International Journal of Parallel Programming, Vol. 33, No. 3, June 2006.

- C. Martínez, E. Vallejo, M. Moretó, R. Beivide and M. Valero, *Hierarchical Topologies for Large-scale Two-level Networks*. In XVI Jornadas de Paralelismo. Granada, Spain, September 2005

- M. Moretó, C. Martínez, R. Beivide, E. Vallejo and M. Valero. *Hierarchical Gaussian Topologies*. In ACACES, Poster Abstracts. L'Aquila, Italy, July 2005, pp. 211-214. Academic Press, ISBN 90 382 0802 2.

Bibliography

[1] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero. The MPSim Simulation Tool. Technical Report UPC-DAC-RR-2009-7, January 2009.

[2] A. Agarwal, B. H. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. Technical report, Cambridge, MA, USA, 1991.

[3] P. Aitken, T. Anderson, S. Apiki, A. Bailey, A. McNaughton, A. W. Morales, L. O'Brien, J. Whitney, and A. Zeichick. Surviving and thriving in a multi-core world. *AMD Developer Central White Paper*, 2006.

[4] D. H. Albonesi. Selective cache ways: on-demand cache resource allocation. In *MICRO*, pages 248–259, 1999.

[5] AMD. Family 10h amd phenom ii processor product data sheet. *AMD White Paper*, February 2009.

[6] AMD. Software optimization guide for amd family 10h processors. *AMD White Paper*, May 2009.

[7] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[8] ARM920T. Technical Reference Manual. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0151c/ARM920T_TRM1_S.pdf.

[9] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[10] T. Austin, E. Larson, and D. Ernst. Simplescalar: an infrastructure for computer system modeling. *Computer*, 35(2):59–67, February 2002.

[11] R. I. Bahar and S. Manne. Power and energy reduction via pipeline balancing. In *ISCA*, pages 218–229, 2001.

[12] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *MICRO*, pages 245–257, 2000.

[13] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *STOC*, pages 345–354, 1993.

[14] B. M. Beckmann, M. R. Marty, and D. A. Wood. ASR: Adaptive selective replication for CMP caches. In *MICRO*, pages 443–454, 2006.

[15] R. Bedichek. SimNow: Fast platform simulation purely in software. *Hot Chips Symposium*, August 2004.

[16] F. Bellard. QEMU, a fast and portable dynamic translator. *USENIX 2005 Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[17] C. Boneti, F. J. Cazorla, R. Gioiosa, C.-Y. Cher, A. Buyuktosunoglu, and M. Valero. Software-Controlled Priority Characterization of POWER5 Processor. In *ISCA*, pages 415–426, 2008.

[18] C. Boneti, R. Gioiosa, F. J. Cazorla, J. Corbalan, J. Labarta, and M. Valero. Balancing HPC Applications Through Smart Allocation of Resources in Multithreaded Processors. In *IPDPS*, pages 1–12, 2008.

[19] C. Boneti, R. Gioiosa, F. J. Cazorla, and M. Valero. A dynamic scheduler for balancing HPC applications. In *SC*, pages 1–12, 2008.

[20] D. P. Bovet and M. Cesati. *Understanding Linux kernel*. O'Reilly, 3rd edition, 2005.

[21] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *PACT*, pages 240–249, 2008.

[22] M. Casas, R. M. Badia, and J. Labarta. Automatic structure extraction from MPI applications tracefiles. In *Euro-Par*, pages 3–12, 2007.

[23] F. J. Cazorla, P. M. W. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero. Predictable performance in SMT processors: Synergy between the OS and SMTs. *IEEE Transactions on Computers*, 55(7):785–799, 2006.

[24] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *HPCA*, pages 340–351, 2005.

[25] J. Chang, M. Huang, J. Shoemaker, J. Benoit, S.-L. Chen, W. Chen, S. Chiu, R. Ganesan, G. Leong, V. Lukka, S. Rusu, and D. Srivastava. The 65-nm 16-mb shared on-die l3 cache for the dual-core intel xeon processor 7100 series. *IEEE Journal of Solid-State Circuits*, 42(4):846–852, April 2007.

[26] J. Chang and G. S. Sohi. Cooperative caching for chip multiprocessors. In *ISCA*, pages 264–276, 2006.

[27] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *ICS*, pages 242–252, 2007.

[28] S. Chheda, O. Unsal, I. Koren, C. M. Krishna, and C. A. Moritz. Combining compiler and runtime IPC predictions to reduce energy in next generation architectures. In *CF*, pages 240–254, 2004.

[29] D. Chiou, P. Jain, S. Devadas, and L. Rudolph. Dynamic cache partitioning via columnization. In *Design Automation Conference*, 2000.

[30] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. In *ISCA*, pages 357–368, 2005.

[31] J. Corbalan, A. Duran, and J. Labarta. Dynamic load balancing of MPI+OpenMP applications. In *ICPP*, pages 195–202, 2004.

[32] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 2001.

[33] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, pages 233–244, 2002.

[34] J. Doweck. Inside intel core microarchitecture and smart memory access. an in-depth look at intel innovations for accelerating execution of memory-related instructions. *Intel White Paper*, 2006.

[35] C. Dubach, T. Jones, and M. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *MICRO*, pages 262–271, 2007.

[36] A. Duran, M. Gonzàlez, and J. Corbalán. Automatic thread distribution for nested parallelism in openmp. In *ICS*, pages 121–130, 2005.

[37] L. Eeckhout, S. Nussbaum, J. E. Smith, and K. D. Bosschere. Statistical simulation: Adding efficiency to the computer designer's toolbox. *IEEE Micro*, 23(5):26–38, 2003.

[38] D. R. Engler, F. M. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[39] S. Girona and J. Labarta. Sensitivity of performance prediction of message passing programs. *Journal of Supercomputing*, 17(3):291–298, 2000.

[40] J. Gonzalez, M. Casas, M. Moretó, J. Gimenez, A. Ramirez, J. Labarta, and M. Valero. Simulating Whole Supercomputer Applications. Technical Report UPC-DAC-RR-CAP-2009-29, June 2009.

[41] J. Gonzalez, J. Gimenez, and J. Labarta. Automatic detection of parallel applications computation phases. In *IPDPS*, pages 1–11, 2009.

[42] P. Goyal, X. Guo, and H. M. Vin. A hierarchial CPU scheduler for multimedia operating systems. In *OSDI*, pages 107–121, 1996.

[43] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *MICRO*, pages 343–355, 2007.

[44] R. H. Halstead, Jr. and T. Fujita. Masa: a multithreaded processor architecture for parallel symbolic computing. In *ISCA*, pages 443–451, 1988.

[45] L. Hammond, B. A. Nayfeh, and K. Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[46] L. C. Heller and M. S. Farrell. Millicode in an ibm zseries processor. *IBM J. Res. Dev.*, 48(3-4):425–434, 2004.

[47] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[48] L. R. Hsu, S. K. Reinhardt, R. Iyer, and S. Makineni. Communist, utilitarian, and capitalist cache policies on CMPs: caches as a shared resource. In *PACT*, pages 13–22, 2006.

[49] C. J. Hughes, P. Kaul, S. V. Adve, R. Jain, C. Park, and J. Srinivasan. Variability in the execution of multimedia applications and implications for architecture. In *ISCA*, pages 254–265, 2001.

[50] Intel. Intel research advances "Era Of Tera". *Intel News Release*, February 2007.

[51] Intel. Introducing the 45nm next-generation intel core microarchitecture. *Intel White Paper*, 2007.

[52] R. R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. R. Hsu, and S. K. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *SIGMETRICS*, pages 25–36, 2007.

[53] A. Jaleel, W. Hasenplaugh, M. K. Qureshi, J. Sebot, S. C. Steely, and J. Emer. Adaptive insertion policies for managing shared caches on CMPs. In *PACT*, page 208Ű219, 2008.

[54] T. M. Jones, M. F. P. O'Boyle, J. Abella, and A. Gonzalez. Software directed issue queue power reduction. In *HPCA*, pages 144–153, 2005.

[55] T. S. Karkhanis and J. E. Smith. A first-order superscalar processor model. In *ISCA*, pages 338–349, 2004.

[56] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power. In *ISCA*, pages 240–251, 2001.

[57] S. Khan, P. Xekalakis, J. Cavazos, and M. Cintra. Using predictivemodeling for cross-program design space exploration in multicore systems. In *PACT*, pages 327–338, 2007.

[58] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *PACT*, pages 111–122, 2004.

[59] A. J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecture Letters*, 1(1):7–10, 2002.

[60] H. Kobayashi, I. Kotera, and H. Takizawa. Locality analysis to control dynamically way-adaptable caches. *Comput. Archit. News*, 33(3):25–32, 2005.

[61] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *ISCA*, pages 81–88, 1981.

[62] J. Labarta, S. Girona, V. Pillet, T. Cortes, , and L. Gregoris. Dip: A parallel program development environment. In *Euro-Par*, pages 665–674, 1996.

[63] H. Q. Le, W. J. Starke, J. S. Fields, F. P. O'Connell, D. Q. Nguyen, B. J. Ronchetti, W. M. Sauer, E. M. Schwarz, and M. T. Vaden. IBM POWER6 microarchitecture. *IBM J. Res. Dev.*, 51(6):639–662, 2007.

[64] J. W. Lee and K. Asanovic. Meterg: Measurement-based end-to-end performance estimation technique in QoS-capable multiprocessors. In *RTAS*, pages 135–147, 2006.

[65] R. Levin, E. Cohen, W. Corwin, F. Pollack, and W. Wulf. Policy/mechanism separation in hydra. In *SOSP*, pages 132–140, 1975.

[66] S. Lohr and M. Helft. Google gets ready to rumble with microsoft. *New York Times*, 16 December 2007.

[67] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[68] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In *ISPASS*, pages 164–171, 2001.

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

[70] MareNostrum Supercomputer. http://www.bsc.es.

[71] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[72] Metis. Family of multilevel partitioning algorithms. http://www.cs.umn.edu/metis.

[73] J. Michalakes, J. Dudhia, D. Gill, T. Henderson, J. Klemp, W. Skamarock, and W. Wang. The Weather Reseach and Forecast Model: Software Architecture and Performance. In *ECMWF*, 2004.

[74] P. Michaud, A. Seznec, and S. Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *PACT*, pages 2–10, 1999.

[75] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. FlexDCP: a QoS framework for CMP architectures. *ACM SIGOPS OSR*, 43(2):86–96, 2009.

[76] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Explaining dynamic cache partitioning speed ups. *IEEE Computer Architecture Letters*, 6(1), 2007.

[77] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Online prediction of applications cache utility. In *IC-SAMOS*, pages 169–177, 2007.

[78] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. MLP-aware dynamic cache partitioning. *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, January 2008.

[79] M. Moudgill, J.-D. Wellman, and J. H. Moreno. Environment for powerpc microarchitecture exploration. *IEEE Micro*, 19(3):15–25, 1999.

[80] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *MICRO*, pages 146–160, 2007.

[81] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In *MICRO*, pages 208–222, 2006.

[82] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. *ISCA*, pages 57–68, 2007.

[83] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. A Framework for Managing Multicore Resources. *IEEE Micro*, 23(5):26–38, 2008.

[84] OMPItrace tool. Instrumentation of combined OpenMP and MPI applications. http://www.bsc.es/media/1382.pdf.

[85] J. K. Ousterhout. Scheduling techniques for concurrent systems. In *ICDCS*, pages 22–30, 1982.

[86] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: The single-node case. *IEEE/ACM Transactions on Networking*, 1:344–357, 1993.

[87] P. Petoumenos, G. Keramidas, H. Zeffer, S. Kaxiras, and E. Hagersten. Modeling cache sharing on chip multiprocessor architectures. In *IISWC*, pages 160–171, 2006.

[88] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[89] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *ISCA*, pages 167–178, 2006.

[90] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *MICRO*, 2006.

[91] S. E. Raasch and S. K. Reinhardt. The impact of resource partitioning on SMT processors. In *PACT*, pages 15–25, 2003.

[92] N. Rafique, W.-T. Lim, and M. Thottethodi. Architectural support for operating system-driven CMP cache management. In *PACT*, pages 2–12, 2006.

[93] L. D. Rose, B. Homer, and D. Johnson. Detecting application load imbalance on high end massively parallel systems. In *Euro-Par*, pages 150–159, 2007.

[94] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta. Complete computer system simulation: The SimOS approach. *IEEE Parallel and Distributed Technology*, 3:34–43, 1995.

[95] R. Sakellariou and J. R. Gurd. Compile-time minimisation of load imbalance in loop nests. In *ICS*, pages 277–284, 1997.

[96] K. Schloegel, G. Karypis, and V. Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *Euro-Par*, pages 296–310, 2000.

[97] A. S. Sedra and K. C. Smith. *Microelectronic Circuits*. Oxford University Press, 4th edition, 1998.

[98] M. J. Serrano, R. Wood, and M. Nemirovsky. A study on multistreamed superscalar processors. Technical Report 93-05, University of California Santa Barbara, 1993.

[99] A. Settle, D. Connors, E. Gibert, and A. Gonzalez. A dynamically reconfigurable cache for multithreaded processors. *Journal of Embedded Computing*, 1(3-4), 2005.

[100] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder. Discovering and exploiting program phases. *IEEE Micro*, 2003.

[101] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM J. Res. Dev.*, 49(4/5):505–521, 2005.

[102] B. J. Smith. Architecture and applications of the hep multiprocessor computer system. *SPIE Real Time Signal Processing IV*, pages 241–248, 1981.

[103] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann, June 2005.

[104] J. E. Smith and R. Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[105] Standard Performance Evaluation Corporation. SPEC CPU 2000 benchmark suite. http://www.spec.org.

[106] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, 1992.

[107] S. Storino, A. Aipperspach, J. Borkenhagen, R. Eickemeyer, S. Kunkel, S. Levenstein, and G. Uhlmann. A commercial multithreaded risc processor. pages 234–235, 442, February 1998.

[108] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *HPCA*, pages 117–128, 2002.

[109] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[110] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *ISCA*, pages 392–403, 1995.

[111] UltraSPARC T2 Supplement to the UltraSPARC Architecture 2007. http://opensparc-t2.sunsource.net/specs/UST2-UASuppl-current-draft-HP-EXT.pdf.

[112] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. A novel evaluation methodology to obtain fair measurements in multithreaded architectures. In *MoBS*, 2006.

[113] J. Vera, F. J. Cazorla, A. Pajuelo, O. J. Santana, E. Fernandez, and M. Valero. FAME: Fairly measuring multithreaded architectures. In *PACT*, pages 305–316, 2007.

[114] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. *ISCA*, pages 84–95, 2003.

[115] T. Y. Yeh and G. Reinman. Fast and fair: data-stream quality of service. In *CASES*, pages 237–248, 2005.

[116] E. Ypek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz. Efficiently exploring architectural design spaces via predictive modeling. In *ASPLOS*, pages 195–206, 2006.

[117] M. Zhang and K. Asanovic. Victim replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *ISCA*, pages 336–345, 2005.

[118] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS*, pages 177–188, 2004.

List of Figures

List of Tables

Glossary

2C Configuration with 2 cores and 1MB 16-way L2 cache. 21, 48, 89

4C-1 Configuration with 4 cores and 1MB 16-way L2 cache. 21, 48, 89

4C-2 Configuration with 4 cores and 2MB 32-way L2 cache. 21, 89

8C-2 Configuration with 8 cores and 2MB 32-way L2 cache. 22, 89

AROAL2M Average ROB occupancy after an L2 miss commits. 68

ATD Auxiliary tag directory. 45, 69

avgDMP Average L2 data miss penalty. 59

BBV Basic block vector. 17

BIC Bayesian information criterion. 17

BM Bit mask. 82

CGMT Coarse grain multithreading. 1

CMP Chip multiprocessor. i, 1, 38, 56, 75, 76, 132, 147

CPI Cycles per instruction. 58

DCP Dynamic cache partitioning. 6, 23, 35, 37, 40, 75, 92, 101, 149

ETT Execution time table. 123

FGMT Fine grain multithreading. 1

FlexDCP Flexible dynamic cache partitioning framework. 78, 148

H High utility. 19, 38, 62, 89, 117

HPT High priority thread. 89

HSHR Hit status holding registers. 44

ILP Instruction-level parallelism. 1

IP Imbalance percentage. 112

IPC Instructions per cycle. 5, 18, 38, 56, 78

IT Imbalance time. 112

L Low utility. 19, 38, 62, 89, 117

LLC Last level on-chip cache. i, 2, 14, 37, 80, 104, 147

LPT Low priority thread. 89

LRU Least recently used. i, 5, 20, 23, 59, 79, 102

MAIV Maximum Allowable IPC Variance. 18

MLP Memory-level parallelism. 9, 38, 40, 148

MPKI Misses per thousand (kilo) instruction. 5

MRU Most recently used. 25, 73

MSHR Miss status holding registers. 42

NRU Not recently used. 94

OPACU Online prediction of applications cache utility. 59, 75, 83, 109, 144, 148

OS Operating system. 3, 11, 36, 55, 75, 132, 147

PPR Performance projection registers. 81, 95

QoS Quality of service. i, 3, 5, 7, 73, 75, 76, 132, 147

ROB Reorder buffer. 13, 36, 37, 59, 83, 110

S Saturated utility. 19, 38, 62, 89, 117

SDH Stack distance histogram. 25, 39, 56, 58, 95

SMT Simultaneous multithreading. 1, 56, 76

SPEC Standard Performance Evaluation Corp.. 14, 56, 82

TLP Thread-level parallelism. 1

TMLP Total memory-level parallelism cost. 46

UBM Used-bit mask. 95

VPC Virtual private caches. 96

VPM Virtual private machines. 133