UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Planificación consciente de la contención y gestión de recursos en arquitecturas multicore emergentes

Contention-aware scheduling and resource management for emerging multicore architectures

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Adrián García García

Directores

Manuel Prieto Matías
Juan Carlos Sáez Alcaide

Madrid

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

PLANIFICACIÓN CONSCIENTE DE LA CONTENCIÓN Y GESTIÓN DE RECURSOS EN ARQUITECTURAS MULTICORE EMERGENTES

-

CONTENTION-AWARE SCHEDULING AND RESOURCE MANAGEMENT FOR EMERGING MULTICORE ARCHITECTURES

ADRIÁN GARCÍA GARCÍA

DIRECTORES:

MANUEL PRIETO MATÍAS

JUAN CARLOS SÁEZ ALCAIDE

Universidad Complutense de Madrid

Facultad de Informática

Tesis doctoral

Planificación consciente de la contención y gestión de recursos en arquitecturas multicore emergentes

-

Contention-aware scheduling and resource management for emerging multicore architectures

Memoria para optar al grado de doctor.

presentada por:

Adrián García García

directores:

Manuel Prieto Matías

Juan Carlos Sáez Alcaide

Planificación consciente de la contención y gestión de recursos en plataformas multicore emergentes

Memoria presentada por Adrián García García para optar al grado de Doctor por la Universidad Complutense de Madrid, realizada bajo la dirección de Juan Carlos Sáez Alcaide y Manuel Prieto Matías.

Contention-aware scheduling and resource management for emergent multicore architectures

Dissertation submitted by Adrián García García to the Complutense University of Madrid in partial fulfillment of the requirements for the degree of doctor of philosophy, work supervised by Juan Carlos Sáez Alcaide and Manuel Prieto Matías.

Madrid, 20 de Julio de 2020.

Agradecimientos

En primer lugar, me gustaría agredecer el cariño y apoyo de mi familia y amigos, que en los momentos de incertidumbre me hicieron perseverar. En especial, a Fernando, Rosa y Paula que me han apoyado en todo momento y permitido tener todo lo necesario para terminar este proyecto y hacer llegar esta tesis a buen puerto.

Además, agradezco a mis directores, Juan Carlos Sáez y Manuel Prieto, por creer en mi desde un primer momento y por su interminable ayuda a lo largo de estos años sin la que hubiera sido imposible realizar este trabajo.

Por último, agradezco al grupo de investigación ArTeCs por los recursos técnicos facilitados para el desarrollo de esta investigación y a la Universidad de Complutense de Madrid por la financiación proporcionada a través de su programa de becas predoctorales. Este trabajo ha sido apoyado directa o indirectamente por la EU (FEDER), el MINECO y la Comunidad de Madrid CM bajo los proyectos y/o contratos TIN 2015-65277-R, RTI2018-093684-B-I00 y S2018/TCS-4423.

Contents

List of Figures

List of Tables

Abstract

Chip multicore processors (CMPs) currently constitute the architecture of choice for most general-purpose computing systems, and they will likely continue to be dominant in the near future. Advances in technology have enabled to pack an increasing number of cores and bigger caches on the same chip. Nevertheless, contention on shared resources on CMPs –present since the advent of these architectures– still poses a big challenge. Cores in a CMP typically share a last-level cache (LLC) and other memory-related resources with the remaining cores, such as a DRAM controller and an interconnection network. This causes that co-running applications may intensively compete with each other for these shared resources, leading to substantial and uneven performance degradation.

Previous research has demonstrated that the system software can be extremely useful in mitigating these problems. In this thesis, OS-level strategies have been devised, and implemented in the Linux kernel, to effectively deal with contention and improve system-wide fairness. One of the major proposals of this thesis is a contention-aware scheduler for asymmetric multicore processors (AMPs). These architectures combine high-performance big cores with low-power small cores on the same platform, all of them exposing a common ISA (Instruction Set Architecture). While many scheduling techniques where proposed in the last decade to mitigate the negative impact of contention, none of them was suitable for AMP systems, where an application may experience substantial performance degradation (relative to an execution in isolation) due to the combined competition for the utilization of big cores and shared resources. Our scheduling proposal –referred to as CAMPS–, accurately tracks the progress of each application as it runs on different core types during the workload execution and under varying levels of shared-resource contention. CAMPS fairly distributes big-core and small-core cycles among threads based on their observed progress.

This thesis also explores how to effectively partition the last-level cache (LLC) on symmetric CMPs. After years of research on cache-partitioning strategies, the necessary hardware support for the wide adoption of these strategies is now available on many commercial processors. Specifically, the thesis focuses on the design of cache-clustering or partition-sharing techniques, which potentially allow several applications to share the same cache partition. Motivated by the inherent complexity in the design of this type of techniques, we built the PBBCache simulator, which constitutes an open-source tool for rapid prototyping and evaluation of cache-clustering policies. PBBCache is also equipped with a parallel algorithm to efficiently determine the optimal solution for various optimization objectives. Via extensive experimentation with this simulator, we were able to guide the design of LFOC, an OS-level cache-partitioning scheme that strives to deliver fairness while providing acceptable system throughput.

Chapter 1

Introduction

In the midst of the two thousands, the Dennard scaling rule [52], which assured constant in-die power density in spite of transistor shrinkage, came to an end. All together, the capacity to integrate a higher amount of transistors in a given size was increasingly different to Moore's law predictions for every new generation; and this gap is expected to be even wider in the future as the industry gets closer to the physical limitations of CMOS technology. This alarming trend stands in stark contrast to the growing needs for computing power of emerging application domains.

Besides, there are huge technical challenges associated with downscaling the manufacturing process that have also contributed to increasing the fabrication costs substantially. A remarkable aspect is the slower rate in which new microarchitectures are being produced. The *Tick-Tock* model – promoted by manufacturers like Intel – no longer stands still; every 18 months a new family of chips was supposed to come out with either a new microarchitecture design or a shrinkage of the fabrication process. However, this cyclical process was halted by the challenge of getting past the 14 nm barrier and, instead, newer generations have been iterating over the Skylake architecture by introducing countless optimizations and creating a plethora of new *lakes*.

The main technical innovations in the semiconductor industry – such as the usage of strained silicon [145], materials with high dielectric constants (*high-k*) in transistor gates [31] or the FinFET/multigate transistors [47] – have enabled to move past the 90 nm nodes of 2003 to the current 10 nm, 7 nm [74, 30] or even 5nm production chips by TSMC [76]. Traditionally, this size referred to gate length, but currently is just a commercial name that does not reflect the real transistor geometry [100, 49, 75]. Figure 1.1 shows the manufacturing process roadmap of multiple top tier manufacturers and as it can be appreciated, Samsung remains close to TSMC. Even though Intel has been lagging behind and stuck in between 14nm and 10 nm on recent years, their scheduled leap for 7nm is said to be a contender for the other companies' 5 nm. The manufacturing process will likely keep improving in the near future thanks to technological innovations, but the rate will progressively get slower while the transistor gate size gets closer and closer to the atomic scale. At this size, undesirable quantum effects like electron tunnelling take over, where

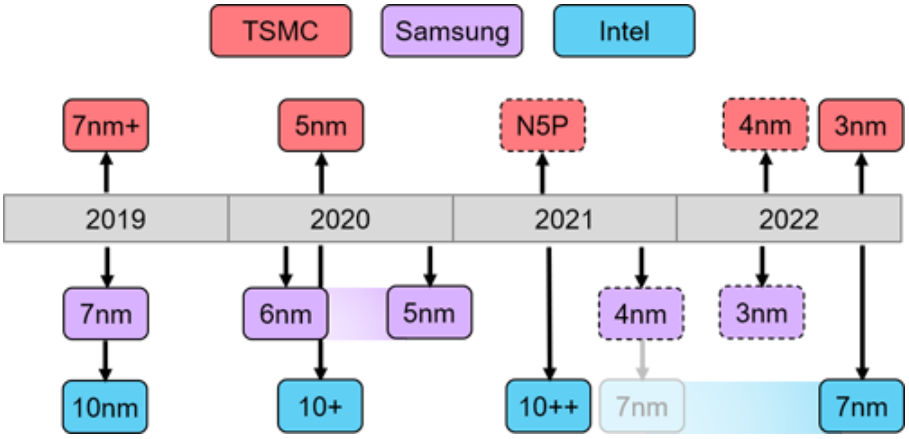


Figure 1.1: Manufacturing process roadmap of different foundries as portrayed in [76].

electrons moving through thin materials (around 1 nm thick) might literally *teleport* from one place to another.

Even if at slower pace than before, the number of components per unit area has not stopped rising. Hence, energy consumption has become the main limiting factor of microprocessor technological evolution. To cope with this challenge –commonly referred to as *power wall* [74]– two major trends have emerged in microprocessor manufacturing. First, the improved iterations of microprocessor technology allowed the integration of an increasing higher number of cores per chip with a limited frequency that avoided otherwise excessive temperatures. Second, a broad spectrum of heterogeneous architectures have been designed to efficiently cater to specific application domains, where a combination of cores of different types or accelerators on the same platform for a general or specific usage may bring substantial benefits [80]. Despite the progressively slower technological breakthroughs, multicore processors –aka Chip Multi-Processors (CMPs)– still constitute a basic building block of general-purpose architectures.

1.1. Multicore architectures

Processor design has changed substantially during the last decade in response to energy efficiency issues. The aggressive microprocessor designs of the early two thousands carelessly pushed for improving performance at the expense of increasing clock frequencies and power consumption, such as the Intel *NetBurst* or *Prescott* architectures [84], which featured execution pipelines of up to 31 stages. In order to keep thermal power dissipation under control, the clock frequencies have settled in the 3 to 5 GHz range and the pipeline depth remains in between 14 to 19 stages in general purpose multicores [60, 79]. Furthermore, out-of-order execution has become a standard in performance-oriented chip designs, even in mobile processors such as the ARM Cortex A76.

As soon as increasing the processor frequency stopped being feasible, packing multiple processing cores on the same chip became the next most used technique to

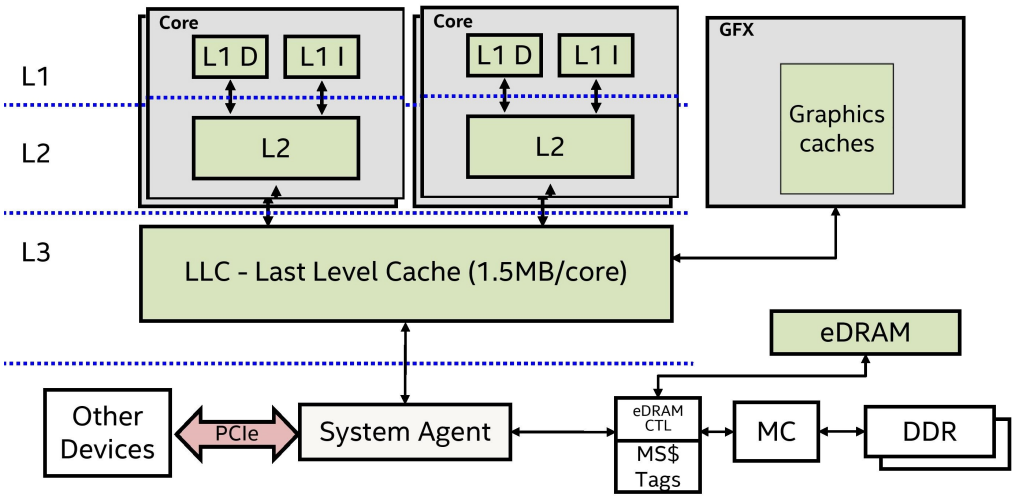
improve performance. However, this did not solve the issue of excessive energy consumption. In fact, saving power does not come from having a higher core count, but from using simpler and more energy-efficient core designs [81]. Moreover, adding cores brings additional challenges and potential sources of inefficiencies to deal with. This has been one of the design aspects that gathered more attention in the recent years and has had a substantial impact on both hardware and system software [113], specially since heterogeneous designs were shown to deliver higher energy efficiency.

A question that naturally arises is up until when this trend of adding cores in every other generation can be upheld. Mid-range CPUs usually integrate 4 to 8 cores, but the core counts in the server market segment stand significantly higher. *Chiplet* based designs, which consist in integrating multiple dies on the same physical package, constitutes an additional way to increase the number of cores as an alternative to technology scaling. This idea is nothing but an evolution of the *Multi-Chip-Module* (MCM) technology that was already used during the eighties, especially in mainframes [65, 109]. For instance, recent AMD EPYC processor families[178, 34] leverage chiplet-based layouts. Specifically, the second generation, codenamed as *Rome*, employs a scalable *System on Chip* (SoC) design based on 8 core-dies, which can be interconnected in a multi-chip package of up to 8 CPU dies using AMD's *Infinity fabric*, which is an evolution of the former HyperTransport [178] technology for multichip connections. Moreover, Intel has also opted for a chiplet design in multiple of their products, such as the Cascade Lake server processors, in which they combine 2 dies on the same package to reach up to 56 cores or 112 threads.

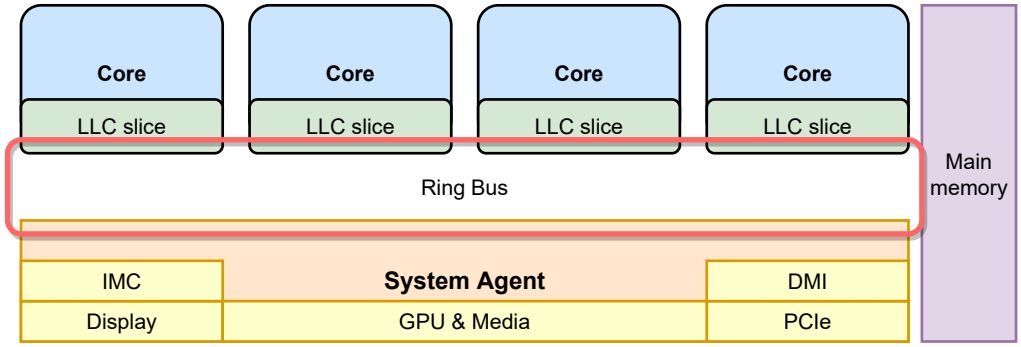
An important aspect associated with the rising number of cores is how to efficiently manage the interaction with the well-known *memory wall* problem. This issue has been aggravated due to the fact that there are more cores that simultaneously demand instructions and data from memory. This performance-limiting factor –alongside the increased energy demands, latency and memory bandwidth– constitute the main limitations to continue scaling the number of cores in this kind of architectures [85, 59].

Another related issue is the fact that having more cores available increases the contention present in different parts of the memory hierarchy; Figure 1.2 shows an example of a modern CPU memory hierarchy from a Skylake processor. Which shows how cache memory is partitioned in different levels and some of them are shared among cores. Intel follows a non-uniform cache architecture (NUCA) where the LLC is divided into multiple slices interconnected via a bi-directional ring bus. In Figure 1.2 we can as well take a closer look at the system agent (a.k.a. *uncore*) that acts as a centralized peripheral device integration unit, it manages various shared resources and I/O components such as the *Peripheral Component Interconnect Express* (PCIe) or *Direct Media Interface* (DMI). Additionally, one of its roles is to serve as on-die memory controller, enabling direct memory access (DMA) and maintaining cache coherence when there are simultaneous requests to the same cache line from different cores.

Taking all of this into account, cores in a CMP are not completely independent processors but instead typically share a last-level cache (LLC) and other memory-



(a) Different levels of the memory hierarchy



(b) Detailed view of the banked distribution of LLC slices among cores

Figure 1.2: On the top, Figure 1.2a illustrates the different levels of the memory hierarchy of an Intel Skylake processor [158]. Below, Figure 1.2b shows how the LLC is split among cores in various slices that are connected through a ring bus. The system agent or *uncore* acts as an interface through an interconnection network to different shared resources.

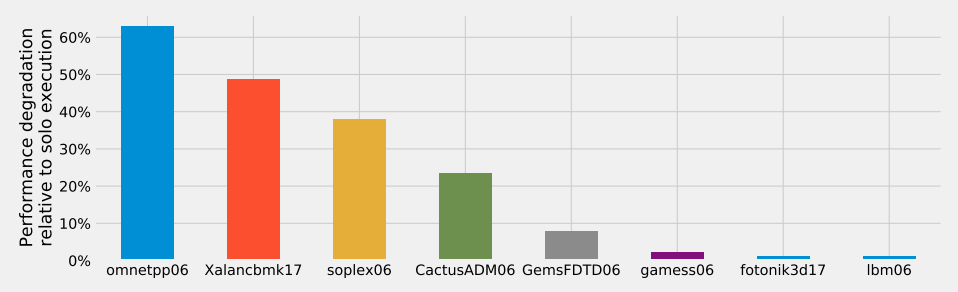


Figure 1.3: Relative performance degradation of different SPEC CPU benchmarks, when running simultaneously on an Intel Skylake multicore processor

related resources with the remaining cores, such as a DRAM controller and a memory bus or interconnection network [205, 54]. The applications that run simultaneously on different cores naturally compete for the usage of shared resources, which may degrade their performance unevenly, limiting global throughput and fairness on the platform [171].

In this thesis, we have mainly looked at the contention problem in the LLC. Notably, the amount of space in the LLC allotted by the hardware to a specific application is not proportional to its priority or to its potential performance benefit, but instead it is tightly related to its rate of demand [153]. Despite the advances in technology, which have made it possible to pack bigger LLCs on the same chip, cache contention stills has a great impact on performance, causing large disparities in the degradation suffered by various applications co-running on a CMP system. To illustrate this fact, Figure 1.3 reports the relative performance degradation that 8 single-threaded applications from SPEC CPU experience when running together on a server system featuring an Intel Xeon Gold 6138 ("Skylake") processor. As is evident, the per-application performance degradation –reported w.r.t. the execution of the corresponding application alone on the CMP system– widely differs across programs; while some of them, such as `fotonik3d` or `lbm` achieve nearly the same performance as in the solo execution, other programs, such as `omnetpp` slow down its execution by a factor of up to 1.63x. Note also that in this experiment, only 8 cores out of the 20 available on the platform were used. Therefore, substantial performance degradation may still become apparent when the workload does not even fully utilize all processing cores.

Shared-resource contention may introduce a number of undesirable effects on the system. For example, contention may cause an application's completion time to differ significantly across runs, depending on its co-runners in the workload [205, 63]. In addition, equal-priority applications may not experience the same performance degradation when running together relative to the performance observed when each application runs alone on the CMP [138, 54]. These issues make priority-based scheduling policies ineffective [54], reduce performance predictability [197] and may lead to unfair billings in commercial cloud-like computing services [63], where users are charged for CPU hours. Notably, unfairness also leads to uneven progress of the various threads in HPC multithreaded applications [182, 167], which may seriously limit scalability.

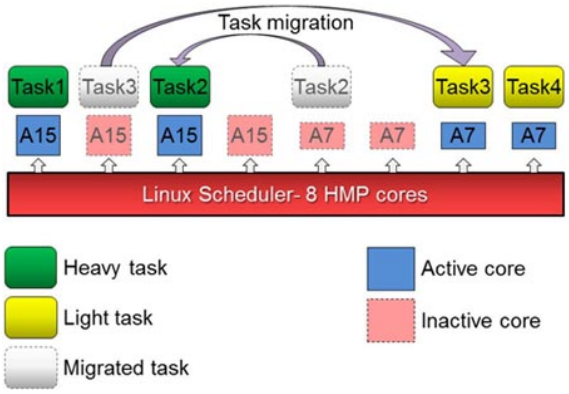


Figure 1.4: Linux *HMP* patch

Multiple OS-level techniques have been proven effective in mitigating the negative effects of shared-resource contention, such as co-scheduling "compatible" processes in different cores that share an LLC [203, 128] or using non-work-conserving techniques [161, 192], which rely on disabling some cores temporarily. Partitioning the shared LLC (i.e. dividing the available cache space among applications) also constitutes a very effective technique to deal with shared-resource contention [153, 86, 195, 137, 101]. After years of research showing the potential of cache-partitioning [133, 205], Intel [142], Cavium [187] and AMD [12] have finally opted to add hardware support for cache-partitioning in some of their latest multicore processors. This support enables the system software to distribute the cache space more effectively among applications. In the case of Intel, the corresponding hardware extensions are commercially known as Cache Allocation Technology (CAT), and provide an interface for LLC way-partitioning. In this thesis we devised means to efficiently exploit these hardware extensions from the operating system to automatically deliver its benefits to unmodified applications. In doing so we paid special attention to improving the degree of fairness on the platform.

1.2. Heterogeneity and Asymmetric multicore processors

Apart from the integration of an increasing number of cores per chip, coupling different core types on the same platform for diverse and specialized use has settled in as one of the strategies to knock down the different walls that microprocessor technology faces. In fact, heterogeneous architectures can deliver more energy efficiency than conventional multicores in some specific application domains [81].

The degree of diversity segregates heterogeneous architectures into various classes, each one becoming a unique point in the design space. One point of this spectrum is to augment conventional processors with special-purpose units and accelerators [48]. This is the case of systems that add application-specific processing components such as GPUs [144], FPGAs [48, 39] or Tensor Processor Units (TPUs) [99]. These architectures have been leading High-Performance Computing (HPC) over the last

few years thanks to their paramount efficiency in terms of performance per watt. Although substantial programming effort is usually required to tap into their full potential [120, 146], they are key elements for the success of emerging applications domains such as deep learning, cryptocurrency mining, or high definition image and video processing. Indeed, their widespread usage by large communities has also driven major advances in compiler technology and the development of new programming models, interfaces, and libraries that are promoting their adoption even further.

At the other end of the design spectrum we find asymmetric single-ISA multicore processors (AMPs) [114]. These designs integrate a mix of complex high-performance big cores and power-efficient small cores, that offer higher energy efficiency than conventional multicore processors. While an application can be designed so as to specifically exploit the features of the different core types in dedicated fashion, the shared ISA and general-purpose nature of these cores allows the execution of asymmetry-agnostic (unmodified) software. This versatility, coupled with the outstanding energy efficiency benefits of AMP designs [114], has drawn the attention of major hardware players, giving rise to products for different market segments [19, 43, 103, 17, 53].

In fact, current asymmetric multicore processors have become the bread and butter of commercial mobile devices, where the number of products that integrate this kind of processor architecture clearly outnumbers those based on symmetric designs, especially on high end handheld devices. The Samsung SoC Exynos 7 Octa or the Qualcom Snapdragon 855, present in smartphones like the Google Pixel 4 or the Samsung Galaxy S10, are recent examples of asymmetric platforms that feature ARM big.LITTLE processors [18], one of the most commercially widespread AMP architectures. Among the reasons they might be so popular are the great energy efficiency they provide alongside the simplicity for the OS to manage a common workload in mobile devices. In this context, high-performance big cores must preferably be used to accelerate foreground tasks, that the user is interacting with, and demand a high CPU usage. On the contrary, there are plenty of background or I/O tasks that can make do with running on energy-efficient little cores.

The *HMP* patch for the Linux kernel[154], which is an extension of CFS for big.LITTLE platforms commonly used in Android devices, introduces a series of changes to the scheduler to make this kind of thread assignments transparently to the user (see Fig. 1.4). As we demonstrate in this thesis, this patch, which has not been officially incorporated into the main branch of the Linux kernel, causes substantial problems when using non-mobile workloads, thus making an unlikely candidate for widespread adoption in desktop and server OSs.

In this thesis, we focused on how to fairly and efficiently schedule general-purpose and high-performance workloads (consisting of unmodified applications) on AMP systems. In this scenario, AMP processors introduce a number of critical challenges to the system software [134], which has been traditionally designed from the ground up to deal with identical cores. After several years of research on AMP systems, it is crystal clear that the OS scheduler plays a key role in automatically delivering

the benefits of these systems to unmodified applications [69].

In particular, to optimize system throughput, the scheduler has to dedicate big cores to running those applications that use them effectively, because they obtain a higher performance improvement or *speedup* relative to running on small cores [114]. Note that some applications fail to use high-performance big cores efficiently, due to incurring substantial pipeline stall cycles caused by frequent branch misprediction or numerous long-latency cache misses [111]. Additional throughput gains can be obtained on AMPs by using big cores to accelerate scalability bottlenecks present in multithreaded programs [165, 164, 98, 129, 96]. A critical challenge to effectively drive thread-to-core mappings on AMPs systems is to equip the scheduler with a mechanism enabling it to accurately approximate the relative benefit that every thread in the workload derives from running on different core types over time [164, 167].

Despite these challenges, the interest in asymmetric platforms even outside the mobile arena, is still substantial today. A clear example is the recent release of the Intel Lakefield SoC (System on Chip)[1], that combines 1 high-performance "Sunny cove" core with 4 energy-efficient "Tremont" cores on the same SoC platform, as shown in Fig. 1.5. Even more recently, Apple has opted for asymmetric-multicores for some of its latest desktop computers, which feature the Apple M1 SoC [17].

The importance of hardware specialization is growing day by day, specially in the field of systems on chip where there are components specifically designed to solve tasks with outstanding performance per watt. Despite the energy efficiency benefits of AMP designs, effectively dealing with the different performance delivered by heterogeneous cores to the various applications, still constitutes a significant challenge to the different layers of the system software, ranging from the operating system [69, 120] to the runtime system [45, 44]. Even though the interest on AMPs is quite high and there are many examples of new asymmetric platforms coming out recently; there is still many gaps to fill in terms of asymmetry-aware software, specially at the operating system level.

1.3. Thesis contributions

In this thesis we propose different OS-level techniques to improve fairness on symmetric and asymmetric multicore systems via contention-aware scheduling and resource management. We focused on solutions at the OS level so as to transparently address the effects of shared resource contention and, hence, to automatically deliver the benefits of current multicore architectures to unmodified applications.

The first major contribution of this thesis is the design and implementation of CAMPS, a contention-aware scheduler for off-the-self asymmetric

[1]Note also that 10 nm Intel Alder Lake [117] desktop processors feature asymmetric configurations with higher core counts than Lakefield including 8 Golden Cove big cores alongside 8 Gracemont little cores.

Figure 1.5: Block diagram of Intel Lakefield SoC

multicore processors (AMPs). While the vast majority of scheduling proposals for AMPs have strived to optimize throughput or reduce energy consumption only, our CAMPS scheduler was designed to deliver fairness while ensuring acceptable system throughput.Moreover, CAMPS exposes a configurable parameter enabling the user to trade fairness for throughput when needed. Our scheduling proposal constitutes the first fairness-oriented scheduler for AMPs that caters to both performance asymmetry (i.e., the fact that various cores on the platform deliver different performance) and to the relative performance degradation that an application may experience due the contention that may arise when sharing the system with other programs.

One of the main challenges in designing our contention-aware scheduler for AMPs was to equip it with an online mechanism for predicting the performance degradation (slowdown) that a thread in the workload experiences as it runs on the various cores of an AMP, relative to a hypothetical execution in isolation. While previous scheduling schemes relied on slowdown-prediction techniques that considered performance asymmetry only, ours is the first one that also factors in the degradation due to shared-resource contention as well. To guide the design of our slowdown prediction model, we conducted a comprehensive experimental analysis using current x86 and ARM asymmetric multicore platforms. Based on the main insights of our study, we devised a novel runtime mechanism that enables the scheduler to approximate a thread's current slowdown by gathering various runtime metrics via performance monitoring counters (PMCs), and by comparing that information with the thread's past history collected in low contention scenarios. This mechanism relies on the gathering of a reduced set of fixed PMCs metrics that can be easily measured at runtime on commercial AMP hardware, thus making our CAMPS scheduler portable across architectures.

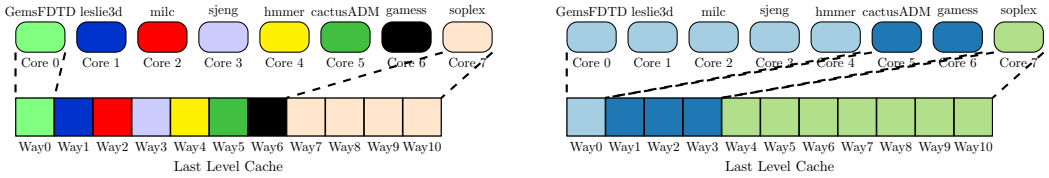
We implemented CAMPS in the Linux kernel, on top of the Completely Fair Scheduler (CFS), which is largely asymmetry agnostic. As we demonstrate in this thesis,

the completion time of an application under the stock Linux scheduler may vary substantially across multiple runs of the same workload on an AMP. As a result, CFS and the *HMP* (Heterogeneous Multi-Processing) scheduler [154] constitute unfair scheduling schemes for asymmetric multicores, especially when compute-intensive applications are present in the workload. Notably, CAMPS delivers more consistent performance from run to run and higher degree of fairness for a wider spectrum of workloads. We also performed an extensive experimental comparison with previously proposed asymmetry-aware schemes [111, 164, 167]. Our experimental analysis –using OS-level implementations and real asymmetric hardware– reveals that CAMPS improves fairness by up to 11% compared to a state-of-the-art fairness-aware scheduler [167], and at the same time improves throughput by up to 17%.

While for AMPs we exploited scheduling as a means to deliver fairness, for symmetric CMPs we focused on the design of resource-management techniques that leverage the hardware extensions for cache partitioning currently available on commercial high-performance symmetric multicores from Intel or AMD. Despite the fact that a large body of research exists on fairness-oriented cache-partitioning techniques, most of the previous partitioning approaches were evaluated via simulation or implemented as user-space resource-management strategies. Our goal was instead to build a lightweight partitioning mechanism ready for adoption in a real operating system.

Specifically, in this thesis we explored the potential of cache-clustering (aka. partition sharing) strategies to improve fairness on multicore systems. As depicted in Fig. 1.6, cache-clustering (aka partition-sharing) constitutes a generalization of strict cache partitioning, where, instead of assigning applications to separate cache partitions (as in Fig. 1.6a), each partition can be shared by a group of applications or *cluster* (as in Fig. 1.6b). We observed that on current high-performance multicore systems, which typically support the creation of a limited number of coarse-grained cache partitions (i.e., in the order of megabytes) cache-clustering proves more effective than strict cache-partitioning as the number of applications increases. This is due to the finer grained distribution of the cache space that naturally results from sharing cache ways between applications (in this thesis we leverage way-partitioning). Unfortunately, building efficient and effective cache-clustering strategies is substantially more challenging than designing strict cache partitioning strategies. While partitioning the cache optimally for a certain optimization objective is an NP-hard problem [133], determining the optimal cache-clustering solution adds a new level of complexity, as a decision must be made on how to best group applications into clusters, and how to optimally distribute cache space across clusters.

To enable rapid prototyping and evaluation of cache-partitioning and cache-clustering policies, we built the PBBCache [68] simulator, which constitutes the second key contribution of this thesis. This open-source parallel simulator relies on offline performance data (e.g., instructions per cycle, memory bandwidth consumption, etc.) to approximate the degree of throughput, fairness or other metrics for a workload under a particular partitioning approach.



(a) Strict cache-partitioning (b) Cache-clustering or partition-sharing

Figure 1.6: Strict cache partitioning vs. cache-clustering

The simulator is equipped with a slowdown-prediction model enabling to determine the performance degradation that an application suffers due to cache-sharing and memory-bandwidth contention. To approximate bandwidth contention for a certain distribution of cache space across applications in a workload, we extended the probabilistic model proposed in [136] with information on how sensitive an application is to a reduction in its effective bandwidth consumption at runtime. To increase programming productivity, PBBCache has been implemented in Python.The simulator allows researchers (1) to guide the design process of their algorithms and, more importantly, (2) to easily discard unpromising approaches without having to go through the tedious development process in the system software.

A powerful feature of the PBBCache simulator is its ability to determine the solution for the optimal strict cache partitioning and optimal cache-clustering problems for different optimization objectives. To efficiently determine the optimal cache space distribution (strict cache partitioning) the simulator leverages a novel parallel branch-and-bound (B&B) algorithm. Notably, this strategy has been specifically designed for the optimization problems that arise in the context of cache partitioning, and enables to effectively distribute the computation across cores on one or multiple computing nodes. To the best of our knowledge, our proposal is the first parallel approach to solve the optimal cache-partitioning problem by factoring in both cache-sharing and memory-bandwidth contention. To evaluate the effectiveness of PBBCache we implemented existing partitioning policies [153, 58, 195] on top of it, and compared the results it provides with the actual figures observed on real hardware equipped with Intel-CAT enabled processors. Moreover, to assess the performance and scalability of the parallel B&B algorithm we conducted experiments using single-node and multi-node machine configurations.

Via extensive analysis of the solution provided by PBBCache for the optimal cache-clustering problem, we were able to guide the design of LFOC, a novel OS-level cache-clustering strategy that strives to deliver fairness. **The design an implementation of LFOC constitutes the third major contribution of this thesis**. To mimic the behavior of the optimal cache-clustering solution for fairness, LFOC effectively identifies streaming aggressor programs and cache-sensitive applications at runtime, and then assigns these kinds of programs to separate cache partitions. The size of the various cache partitions is determined dynamically by leveraging a lightweight algorithm that factors in to the varying degree of cache sensitivity across applications.

We implemented LFOC in the Linux kernel by using a monitoring plugin of the PMCTrack tool [160]. In this thesis, PMCTrack's functionality was also augmented

to enable direct access to privileged hardware cache-partitioning facilities (e.g. Intel CAT) via a custom kernel-level API. We evaluated the effectiveness of LFOC on a real system featuring an Intel Skylake processor, where we compared its effectiveness to that of two previously proposed policies –Dunn [171] and KPart[58]–, which optimize fairness and throughput, respectively. Our analysis reveals that LFOC is able to deliver higher throughput and fairness than every analysed scheme for the vast majority of the workload scenarios considered.

To conclude this section, it is worth highlighting that the main contributions and results of this thesis (CAMPS, PBBCache and LFOC) have been summarized and discussed in articles published in the Journal of Computational Science [71] and IEEE Transactions on Computers [69] –two high impact factor journals ranked in the first quartile of the Journal Citation Reports –, and in the International Conference on Parallel Processing [70], a relevant conference on parallelism and computer architecture, ranked as Class 2 in the GII-GRIN-SCIE (GGS) Conference Rating.

1.4. Thesis structure

The remainder of this thesis is organized as follows:

- Chapter 2 provides some background in the most relevant fields of research in this thesis, shared resource contention and scheduling on Asymmetric Multi-core Processors.
- Chapter 3 introduces the different experimental platforms used in this thesis.
- Chapter 4 focuses on the CAMPS scheduler, a contention-aware fair scheduler for AMPs that primarily targets long-running compute-intensive workloads.
- Chapter 5 illustrates the inner workings of PBBCache, a parallel simulator that makes it possible to quickly compare the effectiveness of different cache-partitioning policies with the optimal solution for different optimization objectives.
- Chapter 6 presents the LFOC algorithm, a cluster-based cache partitioning strategy that seeks to reduce contention and deliver fairness while maintaining system throughput.
- Chapter 7 outlines the main conclusions of this thesis an discusses possible avenues of future work.

Chapter 2

Background

In this chapter we discuss the state of the art in shared-resource contention and asymmetry-aware scheduling.

2.1. Shared-resource contention

Multicore processors constitute the main architecture of choice for modern computing systems in different market segments and they will likely remain as such in the foreseeable future. Despite their potential, the contention that naturally appears when multiple applications compete for the use of shared resources among cores may bring performance and scalability bottlenecks on resources such as the last-level cache, memory bus, interconnection networks, DRAM controllers and prefetchers. This might have a negative impact on key system performance metrics such as throughput, energy efficiency and fairness.

Indubitably, the last-level cache represents a key component to consider when trying to tackle shared-resource contention. In this regard, replacement policies play a crucial role. A widely-used policy in modern systems is Least Recently Used (LRU) policy, which seeks to provide temporal locality by preserving the most recently accessed information on cache memory. This generally works well when a single thread runs on the system, but in the context of multithreaded workloads, performance variability arises; since all misses are served equally, cache allocation ends up depending on each thread's memory access rate [92]. Besides, those threads with higher access rates may not be the ones that experience a higher benefit from using more cache space.

Recent multicore processors typically integrate a tiled layout of the last level cache, which is shared by all cores on the chip but physically distributed among cores. The various LLC banks are interconnected with a shared bus or a network on chip (NOC) to the DRAM controller; this stands out as another crucial place where contention appears, since they represent the last chance to prevent costly off-chip memory accesses. In [110] the authors evaluate through simulation the performance penalty

that arises when two applications run simultaneously while competing for memory bus usage. They observed slowdowns up to 60% relative to the solo executions. Furthermore, there was a huge slowdown variability from run to run depending on the co-running applications.

Another major component of the memory hierarchy related to contention is the DRAM controller, which is in charge of serving memory requests. Notably, most controllers were designed with the goal of maximizing overall data throughput in mind [157]; and they ignore the contention present when multiple applications compete for its use. In [138], the authors prove that when multiple threads run on a system, different memory system performances could be obtained from run to run: while one thread is prioritized by the DRAM controller, others starve for memory bandwidth. Despite the fact that NUMA architectures (*Non-Uniform Memory Access*) have significantly improved the issue of memory bandwidth contention, several works have highlighted that shared-resource contention was still present on these systems [204, 27]. Likewise, these architectures bring several challenges that must be tackled, like the extra overhead of inter-processor migrations [119] and a requirement for complex chip-specific memory controllers [106].

Plenty of techniques have been proposed to mitigate the effects of shared-resource contention, including both *software techniques* and *hardware techniques* In the early two thousands, there was already a significant amount of work done in this field that strived mostly for improving system throughput. It fell mainly into two categories [205]: those that make cache microarchitecture optimizations [177, 20, 105, 122, 153, 176, 156, 184, 95, 190]– and *DRAM memory controller scheduling* [35, 157, 51, 180, 87, 186, 138, 139, 26, 125].

No matter what technique is used to manage shared-resource contention, even if it is applied at different parts of the memory hierarchy, a common requirement is the ability to measure how contention affects the performance of specific applications. In other words, the ability to measure the slowdown that a specific application suffers when sharing the system with others. Furthermore, it is worth noting that every application's performance suffers from contention in an uneven way, which also varies when they go through different program phases. Hence, predicting how each application is affected by contention at runtime is fundamental in realizing which applications need to receive a special treatment.

There have been plenty of techniques with the goal of measuring contention [37, 201, 188, 200], the majority focused on the contention generated in the LLC. One of the most relevant mechanisms used in predicting shared-resource contention have been Stack Distance Profiles (SDP), used as early as the 1970s [130], and Miss Rate Curves (MRC), which were initially used for prediction in [41].

Furthermore, the interference produced when more than two threads share memory resources substantially increases the complexity of predicting application-specific slowdowns. Multiple proposals have strived to address shared-cache interference [41, 189, 56, 42] by means of complex statistical models that may incur in significant overheads. On the other hand, there have been much simpler proposals, like [137], that actually manage to model the combined memory usage of multiple applications

that share a specific cache size. In doing so, it makes it possible to assess the impact of assigning certain applications to cache partitions of a given size. This model relies on creating a combined miss curve by taking into account the memory access rates of the threads at specific cache sizes.

Previous research has highlighted the usefulness of cache-partitioning (i.e. splitting the cache space between applications) in mitigating the negative effects of shared-resource contention [175]. In [20] they proved that offering configurable cache partitions in the LLC could provide an average 43% reduction in memory hierarchy energy consumption in addition to improved performance. . After some years of promising results [153, 54], major hardware manufacturers finally added the widely demanded support for cache partitioning with Intel CAT [143] and AMD Quality of Service extensions [12]. On these platforms, which support a limited number of coarse-grained cache partitions (e.g., 11 in the experimental platform used in this thesis), cache-clustering constitutes a more flexible alternative than strict cache-partitioning [70, 71]. Instead of establishing partitions for specific applications, cache clusters might be shared by a group or *cluster* of applications.

As far as we know, there were two early adopters of the Intel CAT technologies that used cache partitioning via way-partitioning of the LLC. One of them was Heracles [123], their strategy uses multiple software and hardware techniques to enforce CPU, memory, and network isolation. The main goal is to assure that latency-sensitive tasks fulfil their latency deadlines, while providing as much throughput as possible for best-effort jobs. Heracles uses cache-partitioning by searching for a right-sized allocation that eliminates latency violations and enforces SLOs (Service Level Objectives).

The other proposal that also used the Intel CAT technology was Ginseng [66]. They focused on making cloud providers able to optimize client satisfaction and improve hardware usage. Their strategy is a market-driven auction system that maximizes the aggregated benefit of the guests in terms of the economic value they attribute to the desired allocation. To guarantee achieving the client's performance goals, they apply per-application cache partitioning in the LLC to isolate applications and prevent them from suffering the negative effects of shared-resource contention. Even though if these approaches come from different research fields, they have proven themselves useful in minimizing the negative effects of shared-resource contention.

Conversely to cache-partitioning and bandwidth allocation, another approach for tackling shared resource contention is thread level scheduling [21, 108, 94, 181, 203, 132]. The advent of multicore processors and increasing core counts opened a new field for these techniques to shine and propelled tons of new research. Since multicores became widespread, schedulers evolved from deciding how to share single processing units (time-sharing) to handling multiple cores and establishing thread assignments (space-sharing). Different research proposals have targeted goals like enforcing fairness, thread priorities or real-time deadlines. But up until this thesis started, there was no other proposal that dealt with shared-resource contention and tackled multiple contention-inducing factors at the OS-level in real hardware.

2.2. Asymmetry-aware scheduling

A large body of work has advocated the benefits of AMPs over symmetric CMPs [114, 112, 83]. Despite these benefits, AMPs give rise to a number of challenges to the system software [155, 134, 163]. How to efficiently and fairly schedule a set of unmodified applications on these architectures constitutes an important problem; this has been one of the main problems we strived to address in this thesis.

Most previous proposals on scheduling for AMPs (as we do in this thesis) target workloads consisting of long-running compute-intensive applications [24, 172, 111, 183, 182, 164, 167]. Recent research on this domain has highlighted that to optimize fairness, system throughput or energy efficiency the scheduler must consider the *speedup factor* (SF) of the various threads when making decisions [111, 183, 164, 167, 159]. A thread's SF is defined as $\frac{IPS_{big}}{IPS_{small}}$, where IPS_{big} and IPS_{small} are the thread's instructions per second (IPS) ratios achieved on big and small cores respectively when running alone on the system.

In the remainder of this section we first describe the techniques proposed to determine the speedup factor at runtime. Next, we cover scheduling proposals that seek to optimize throughput, and then move on to outline strategies designed to improve fairness. Finally, we recap previous work that seeks to optimize other goals beyond fairness and throughput optimization or target non compute-intensive workloads.

2.2.1. Determining the speedup factor

On commercial processors, the system software can leverage hardware performance monitoring counters (PMCs) to determine a thread's speedup factor online. Overall, two different techniques have been explored to do so. The first one is to measure the SF directly [114, 24], which requires running each thread on big and small cores to track the IPC on both core types. Previous work has shown that this approach, also known as *IPC sampling*, is subject to inaccuracies that naturally come from using IPC values from different program phases to approximate the SF [172, 167]. The second approach relies on *predicting a thread's SF* using its runtime properties collected on the *current* core type using PMCs [111, 164, 152, 167]. This approach removes the need from potentially costly migrations required to measure the performance of a thread on the various core types, and it has been shown to provide more accurate SF estimates than IPS sampling. Unfortunately, SF prediction requires building an estimation model specifically tailored to the AMP platform in question. In chapter 4 we elaborate on the drawbacks associated with the reliance on platform-specific estimation models.

Other researchers have proposed the inclusion of new hardware monitoring facilities on the processor to provide the process scheduler with accurate SF estimates [183, 182]. Performance Impact Estimation (PIE) [183] constitutes an example of this kind of hardware support; unfortunately PIE poses certain shortcomings that complicate its integration on real hardware [152]. Despite the practical limitations of PIE, we strongly believe that devising hardware extensions for accurate

SF estimation on AMPs is a promising research avenue that could bring important benefits.

2.2.2. Throughput optimization

To maximize throughput in multi-application scenarios, previous research has demonstrated that the scheduler must follow the High-SPeedup (*HSP*) approach, namely it must preferentially run on big cores those applications that derive a higher big-to-small SF or speedup. The main difference between the available variants of the HSP approach [114, 24, 168, 164, 111, 152] lies in the mechanism employed to obtain threads' speedup factors online. For example, scheduling proposals presented in [114, 24] leverage IPC sampling, whereas those proposed in [168, 164, 111] rely on estimation models.

Recent research has highlighted that making scheduling decisions based on per-thread SFs only may lead to serious throughput degradation when multithreaded programs are included in the workload [165, 164]. This stems from the fact that the SF does not approximate the overall benefit that a multithreaded application as a whole derives from using the big cores in an AMP [16, 83]. Catering to application-wide speedups, as we do in this thesis, is the key to optimizing throughput in these workload scenarios. Previous research [164, 166] has devised analytical formulas to approximate the speedup for several types of multithreaded applications based on the runnable thread count (a proxy for the amount of thread-level parallelism in the applications), the SF of the application threads and the number of big cores in the AMP. We turned to these formulas to approximate the speedup for multithreaded applications in the implementation of our OS-level scheduling proposal.

Other researchers have proposed specific support to accelerate multithreaded programs on AMPs [16, 164, 97, 98, 129]. These proposals make use of big cores as acceleratators for different types of scalability bottlenecks in parallel applications by employing software [16, 164] or hardware-aided approaches [97, 98, 129]. Our OS-level scheduling proposal is largely orthogonal to these approaches.

2.2.3. Delivering fairness

To enforce fairness, equal-priority applications must experience a similar performance degradation or *slowdown* when running together relative to their solo execution. Notably, optimizing fairness usually comes at the expense of degrading throughput and energy efficiency substantially, as these three aspects constitute largely conflicting optimization goals on AMPs [159].

The first fairness-aware scheduler for AMPs was an asymmetry-aware Round-Robin (RR) scheme that simply fair-shares big cores among applications by triggering periodic thread migrations [24]. Fair-sharing big cores has proven to provide better performance and more repeatable completion times across runs on AMPs [165, 120] than default schedulers in general-purpose OSes, which are largely asymmetry

agnostic. For this reason, RR has been widely used as a baseline for comparison [24, 165, 104]. Unfortunately, RR constitutes a suboptimal fairness solution [167], since it does not consider per-thread big-to-small speedups when distributing big-core cycles.

The asymmetry-aware completely fair scheduler (ACFS[167]) was considered the state-of-the-art OS-level fairness-aware scheduling scheme for AMPs. In [167] the authors experimentally demonstrated that ACFS clearly outperforms previous fairness-aware schedulers, such as RR [24], Equal-Progress [182], and A-DWRR [120], for a wide range of workloads running on real AMP hardware. To optimize fairness, ACFS leverages per-thread SF values to continuously track the relative progress that each thread in the workload makes on the AMP, and enforces fairness by evening out the slowdown observed across applications. The main limitation of ACFS [167] (also present in earlier schemes [182, 120]) is the fact that the scheduler does not take shared-resource contention effects into consideration. As our experiments in Chapter 4 reveal, failing to cater to these effects leads the scheduler to exhibit unfair behavior when multiple memory-intensive programs are included in the workload. Our proposed scheduling algorithm effectively improves fairness in this scenario.

2.2.4. Other optimization goals and workload types

Specialized schedulers have been also proposed to properly deal with non compute-intensive workloads on AMPs, such as those including latency-sensitive applications [149, 77], programs with irregular non-scalable parallelism [96] or multimedia applications [107]. In [77] they propose a scheduling framework to manage interactive services on multicore servers by exploiting both software and hardware heterogeneity. They emulate Dynamic Voltage and Frequency Scaling (DVFS) and AMP configurations by using duty-cycling threads. In a nutshell, they leverage DVFS and the high energy efficiency of AMP configurations to ensure that service providers fulfil their tail latency targets while obtaining significant savings in energy consumption. Following a slightly different approach that is solely based on AMPs, Octopus-Man [149] uses a feedback-control mechanism to assign latency-critical threads to the least power-consuming core possible on the asymmetric platform. In this way, they seek to minimize quality of service violations while improving the energy efficiency of warehouse-scale computers. It is worth noting that they evaluated their approach on real asymmetric hardware by using the exclusive Intel Quick IA prototype [43]. Aside from that, their main difference is that Octopus-Man uses a single core type configuration at the same time, while in [77] they mix core types and assign short requests to slow cores.

Other researchers have devised ways to reduce energy and power consumption on AMPs [199, 150, 135, 115]. Mogul et al. [135] proposed using small cores in asymmetric multicores to execute system calls. They modified an operating system to switch the execution to a less powerful core when a thread invokes a system call. The effectiveness of this scheme relies on the observation that system calls and OS code in general use big high-performance cores inefficiently. In a similar vein,

Kumar and Fedorova [115] proposed binding the control domain *dom*0 of the Xen hypervisor to slow cores. These approaches are orthogonal to our proposal.

The PRIM [199] and EEF-Driven [159] scheduling strategies strive to optimize the system's energy efficiency for compute-intensive workloads. Specifically, PRIM is a rule-set-guided scheduling algorithm that performs thread-to-core mappings based on the values of different performance metrics (such as the IPC or the LLC miss rate) gathered with hardware counters. At high level PRIM works as follows, when a thread is created, it is mapped to a random core type in the system in order to preserve load balance. Every so often, the scheduler randomly selects a certain number of thread pairs consisting of a thread running on a big core (T_B) and another thread running on a small core (T_S). For each randomly-selected pair, the scheduler estimates whether swapping T_B with T_S would result in energy savings by means of a set of platform-specific rules. If that is the case, the scheduler swaps both threads. The main limitation of PRIM is the fact that platform-specific rules do not quantify the actual energy savings resulting from a thread swap, but instead indicate whether a specific thread swap would be beneficial or not in terms of energy consumption. This leads PRIM to making suboptimal thread-to-core mappings [159]. This shortcoming is addressed by the EEF-Driven [159] scheduler, which relies on a thread's energy-efficiency factor (EEF), a novel metric that indicates the relative energy efficiency that comes from assigning a thread to a big core, relative to a small one. An application's Energy-Efficiency Factor (EEF) is defined as follows: $\frac{SF}{EPI_{big}}$, where EPI_{big} denotes the Energy per instruction consumed on a big core. The EEF-Driven scheduler devotes big cores to running preferentially those threads with a high EEF value. This enables to improve throughput (by up to 20%) and reduce the energy-delay product (EDP) (by up to 15%) relative to PRIM.

For embedded workloads further energy-related optimizations are possible, since threads may exhibit more predictable execution patterns. Petrucci et al. [150] proposed a global optimization scheme that targets embedded thread sets with periodic characteristics. Their user-level scheduling proposal is able to determine energy-efficient thread assignments by leveraging an ILP (Integer Linear Programming) model. This scheduler enforces thread-to-core mappings by imposing thread affinities via system calls.

Chapter 3

Experimental setup

In this chapter we present the specifications of the different platforms used to gather experimental results. Moreover, we detail the linux-based software stack on which this thesis has been built.

3.1. Hardware

3.1.1. Asymmetric Multicore Platforms

For the experimental evaluation on real hardware of asymmetry-aware schedulers, we used three asymmetric platforms: the ARM Juno development board [19], which integrates a big.LITTLE ARM processor (64 bits); the Odroid XU-4 board [78], equipped with another big.LITTLE ARM processor (32 bits); and the Intel QuickIA prototype[1] [43], which integrates a high-performance Xeon processor, alongside a low-energy Atom processor. Table 3.1 summarizes the specifications of these systems. Figure 3.1 depicts the the memory hierarchy as well as the number of cores of each kind in the AMP configurations explored.

Table 3.1: Features of the AMP systems of this thesis

System	2B-4S-Juno		4B-4S-Odroid		2B-2S-QuickIA	
Core types	Cortex A57	Cortex A53	Cortex A15	Cortex A7	Xeon E5450	Atom N330
Core count	2	4	4	4	$4^{(*)}$	2
Frequency	1.10GHz	850 MHz	2.0 Ghz	1.4 Ghz	1.2 Ghz	1.6 Ghz
Pipeline	Out of order	In order	Out of order	In order	Out of order	In order
Cache L2	2MB/16-ways	1MB/16-ways	2MB/16-ways	512KB/8-ways	6MB/16-ways	512KB/8-ways
DRAM	8GB DDR3 @ 800MHz		2GB DDR3 @ 933MHz		16GB DDR2 @ 677MHz	

(*) For our experiments in Chapter 4.4, two high-performance cores were disabled to achieve a comparable topology to the ARM systems.

[1]This prototype was donated to our research group by Intel Labs (Hillsboro, OR, USA).

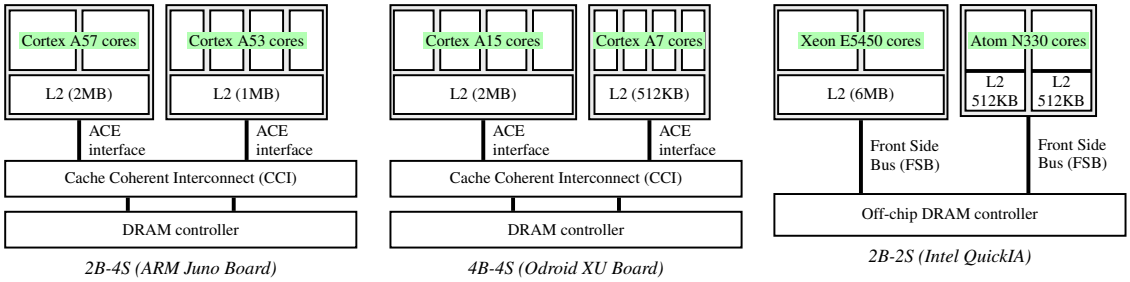


Figure 3.1: Experimental configuration of the asymmetric platforms

Table 3.2: Features of the various platforms used equipped with Intel RDT

Platform Name	**Broadwell-EP**	**Skylake**
Proc. Model	Xeon E5-2620 v4[90]	Xeon Gold 6138[91]
Proc. Frequency	2.1GHz	2.0 GHz
Core count	8	20
LLC (L3) cache	20MB/20-way	35MB/11-way
Main Memory	32GB@2133 MHz	96GB@2666 MHz
CAT enabled	Yes	Yes

3.1.2. Symmetric multicore platforms

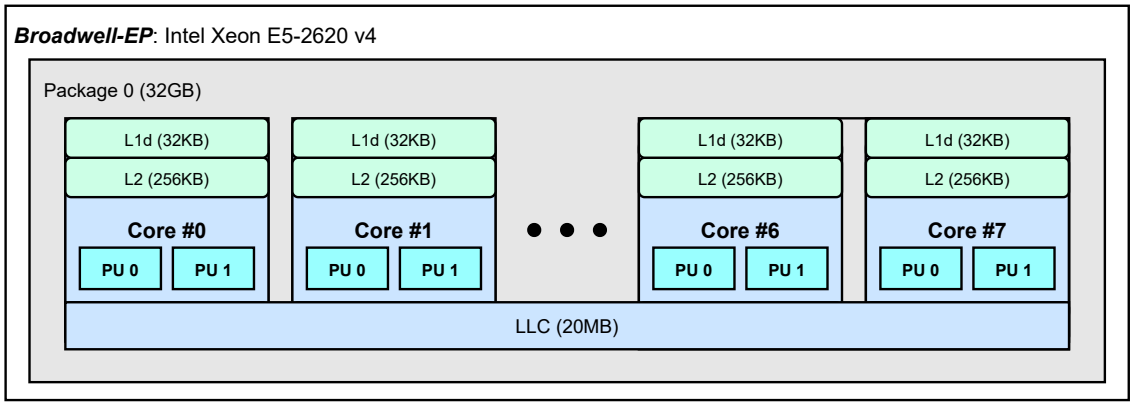
To carry out simulations with our PBBCache tool as well as the evaluation of various cache partitioning policies we used different platforms. Table 3.2 summarizes the features of the two hardware platforms that support hardware extensions for cache partitioning. The topology of these platforms is summarized in Figure 3.2. They are referred to as *Skylake* and *Broadwell-EP* (Section5.6.2). *Broadwell-EP* integrates a Xeon E5-2620 v4 processor and *Skylake* incorporates a Xeon Gold 6138 processor. They include distinct memory hierarchy organizations (e.g. the former uses an inclusive LLC whereas the later does not) and different cache partition granularities (i.e. the smallest partition we can create on *Skylake* – 2.5 megabytes – is 2.5 times bigger than on *Broadwell-EP*).

For the evaluation of the accuracy and scalability of the PBBCache simulator we used two additional platforms (Section 5.6.3), whose features are summarized in Table 3.3. The first platform, referred to as *Haswell*, is a dual-socket server equipped with two Intel Xeon E5-2695 v3 processors. The second one is a cluster consisting of four 16-core nodes, each node integrates two Intel Xeon E5-2650 processors (referred to as *SandyBridge-EP* in Table 3.3).

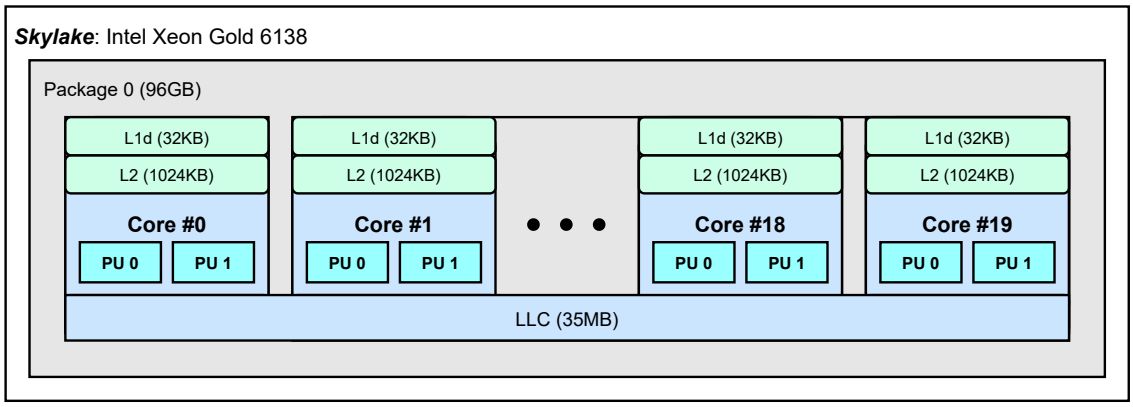
3.2. Software

3.2.1. Scheduling Framework for AMPs

To deliver the potential benefits of AMPs to unmodified applications on Linux or most general-purpose operating systems, substantial modifications in the OS



(a) *Broadwell-EP*.



(b) *Skylake*.

Figure 3.2: Architecture layout of the hardware platforms *Broadwell-EP* and *Skylake* presented in Table 3.2.

Table 3.3: Features of the various platforms used for the assessment of the simulator's accuracy and scalability.

Platform Name	**Haswell**	**SandyBridge-EP**
Proc. Model	2 x Xeon E5-2695 v3[89]	2 x Xeon E5-2650[88]
Proc. Frequency	2.3 GHz	2.0GHz
Core count	28	16
LLC (L3) cache	35MB/20-way	20MB/20-way
Main Memory	64GB@1600 MHz	64GB@1600 MHz
CAT enabled	No	No

scheduler are usually necessary. On top of all the development complications that might arise at kernel level, leading to countless reboots, this is on itself quite a challenging task. So as to implement contention-aware scheduling algorithms on Linux, we took advantage of previous work from our research group and used the scheduling framework for AMPs developed in [169, 151]. Originally, this framework was designed to work on the OpenSolaris process scheduler but, after Oracle took over Sun Microsystems in 2007, the support for this open source operating system was discontinued. The framework port to Linux entailed a significant amount of effort and it ended up taking 15000 lines of code.

Within the Linux scheduler, multiple scheduling classes exist, each one implementing a different policy. This design allows to simultaneously manage different processes with specific scheduling policies. All scheduling classes must react to common events in a thread's life cycle, such as picking a new task to run or assigning a thread to a specific core. Besides, they include different per-CPU runqueues so active threads can be accounted for and scheduled based on the metrics and priority that each algorithm considers.

The AMP scheduling framework [169, 151] makes it possible to implement and evaluate different scheduling strategies in a realistic scenario, by using a general-purpose operating system on top of real asymmetric hardware. The main component of this framework is the operating system process scheduler and a fork of the default scheduling class (CFS), which was used as a foundation to build an AMP-aware scheduling class.

In its early version, the scheduling framework's code directly gathered information from hardware counters to guide some scheduling policies with specific performance metrics. Unfortunately, the interaction with hardware counters varies from one architecture to another – and even differ across processor models –, giving rise to the necessity for an architecture-independent mechanism that enabled the interaction with the monitoring hardware in various experimental platforms. Further down the road, the requirement for this functionality crystallized in the development of PMCTrack [162], which allowed the operating system and the user to access performance monitoring information through a architecture independent API.

3.2.2. PMCTrack

PMCTrack is an open source performance monitoring tool that directly gathers information from hardware counters [162]. As mentioned above, it was originally developed to facilitate the implementation of scheduling algorithms in the Linux kernel that base their decisions by gathering online information from PMCs (*Performance Monitoring Counters*). Afterwards, different components were added to allow monitoring applications at runtime from userspace and exploit other hardware monitoring facilities beyond PMCs.

Previous work [160, 162] dissect the inner workings of PMCTrack and dive deeply into the advantages of using it over other performance monitoring tools. Notably,

PMCTrack supports monitoring of both sequential and parallel applications and, more importantly, has the ability to calculate high-level metrics and perform event multiplexing.

The `pmctrack` command line tool is the more direct way of interacting with PMCTrack from userspace. This command includes three basic use cases:

- *Time-Based Sampling* (TBS): this mode allows the user to periodically gather information from PMCs and virtual counters for a specific application within a certain sampling interval.
- *Event-Based Sampling* (EBS): this mode makes it possible to gather information from PMCs and virtual counters whenever a certain performance counter reaches a certain threshold, specified by the user.
- *Time-Based system-wide monitoring mode*: this mode is a variant of TBS that gathers information for each CPU (core), instead of a specific application.

To illustrate the behavior of `pmctrack` tool, we analyse a sample command that monitors the SPEC CPU 2017 application `roms` using the TBS mode:

```
$ pmctrack -c instr,cycles ./roms17
[Event-to-counter mappings]
pmc1=instr
pmc2=cycles
[Event counts]
nsample    pid      event          pmc1          pmc2
      1  8554       tick    2653440734    1255443685
      2  8554       tick    2572409764    1286580986
      3  8554       tick    2367460876    1234683457
      4  8554       tick    2479662345    1234603218
      5  8554       tick    2235759874    1268959764
      6  8554       tick    2486452456    1267962456
      7  8554       tick    2545742652    1267809345
      8  8554       tick    2584730986    1280471675
      ...
```

This command provides the user with the number of retired instructions and cycles per second. The command line argument `-c` is used to specify a set of hardware events to monitor. As it is shown in the example, event mnemonics are accepted like in any other monitoring tools [93, 46, 147], but the events can be expressed as well in their respective hexadecimal architectural codes. The last argument contains the full command that launches the application to be monitored – for instance, `./roms17`. Note also that the sampling period can be specified with the argument `-T`, by default it is set to 1 second. The command output starts with an event-to-counter mapping section that shows which physical counter corresponds to each requested event. Afterwards, the *Event counts* section contains a table that reflects in each row the sampler number, the process ID and each counter's value.

3.2.3. Scheduling mode and monitoring modules

The core of PMCTrack is implemented in a kernel module that is in charge of gathering performance information from the available hardware counters on the platform

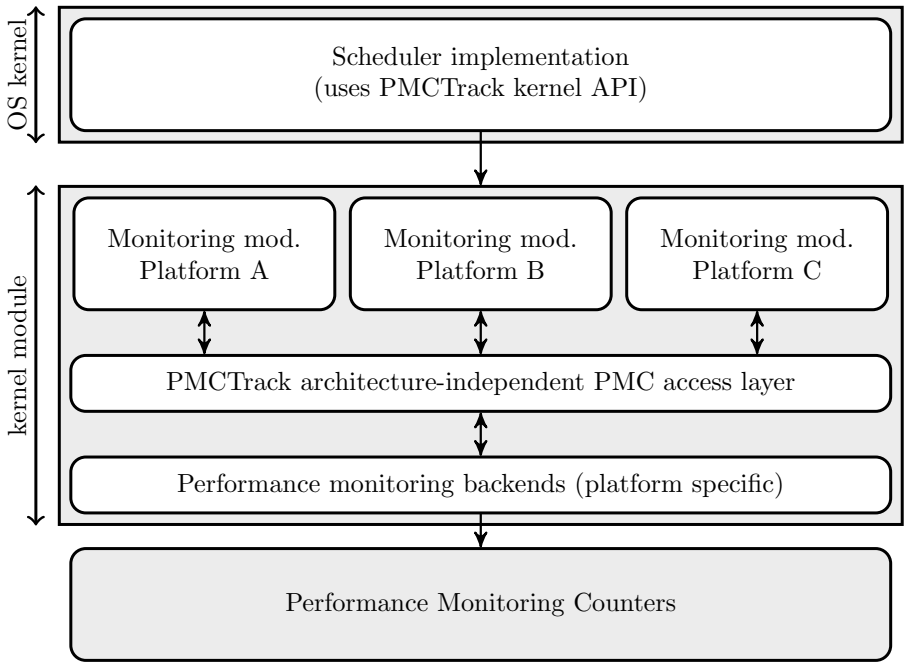


Figure 3.3: PMCTrack monitoring module architecture

at the user's request. In order to be able to independently gather information for each thread on the system, the kernel module must be aware of different events that occur in a thread's life cycle, such as context switches or blocking events. Because the Linux kernel does not offer an API to capture these events from loadable kernel modules in every architecture, this support has to be necessarily added via a simple and portable kernel patch for easy adoption across different kernel versions. For instance, our first experiments on this thesis were performed using the Linux version was the v3.10 (Chapter 4.4) and we have constantly updated to newer versions and experimented with up to v4.9.104 (Chapter 6.4). We have also upgraded PMCTrack for more recent kernels, the latest being v5.4.35.

The PMCTrack kernel API not only allows the PMCTrack kernel module to receive the necessary notifications or callbacks to manage the monitoring hardware, but also implements a set of functions to enable any scheduling class of the Linux kernel to access performance monitoring data in an architecture-independent manner. PMCTrack's scheduler mode enables kernel-level scheduling algorithms to collect per-thread performance metrics. The scheduler mode for a specific thread can be easily activated by enabling a special flag inside the thread's descriptor.

Figure 3.3 depicts the PMCTrack monitoring module architecture. The tool's functionality can be easily extended via plugins called monitoring modules. They are implemented in a separate .C file that and loaded as a kernel module. They interact with the PMCTrack kernel module via its API, as depicted in Figure 3.3. These monitoring modules make visible to the OS and the user any kind of HW monitoring information provided by modern processors, even if it is not readily available through regular performance counters. Some examples of information that can be gathered are the energy consumption or cache space that an specific application

uses, which makes this functionality specially useful for the prototyping and development of new algorithms that take advantage of this information for resource management and scheduling. In order to provide complex performance metrics to the different components of PMCTrack, a software abstraction known as virtual counters is used; unlike regular hardware counters from the *Performance Monitoring Unit* (PMU), they are able to account for compound metrics to the operating system or the user. These counters may also expose complex metrics to the user like IPC or LLCPKI, and expose information from other sources than hardware counters. Like the one provided by recent cache partitioning and monitoring functionalities bundled in Intel Resources Director Technology suite.

3.2.4. Intel Resource Director Technology

The Intel Resource Director Technology (Intel RDT) is a set of technologies available in modern Intel processors that provides support for monitoring shared-resource usage and performing dynamic resource allocations. Even though the different technologies existed earlier separately, Intel unified them to be a collection of related hardware extensions, commercially referred to as Intel RDT. Specifically, the various technologies that make up Intel RDT are as follows:

- **Intel Cache Monitoring Technology (CMT)** offers information about how much cache space each application is consuming.
- **Intel Memory Bandwidth Monitoring (MBM)** gives access to information directly measured from the memory controller and accounts for the memory bandwidth that each applications consumes.
- **Intel Cache Allocation Technology (CAT)** provides access to the way-based cache-partitioning utilities present in modern Intel processors.
- **Intel Memory Bandwidth Allocation (MBA)** enables to restrict the memory bandwidth consumption of a specific application by throttling memory requests.

3.2.4.1. Shared-resource monitoring technologies

The Intel RDT technology enables the operating system or virtual machine monitor (VMM) or hypervisor to determine the memory bandwidth consumption or cache usage of a specific level of the memory hierarchy (usually the LLC), even when multiple applications run on the system simultaneously. Second generation Intel Xeon scalable processors were one of the first processor family that fully supported these technologies.

At a high level, Intel RDT assigns a given identifier(ID) to each application or virtual machine process; that is known as *Resource Monitoring ID* (RMID). The hardware that supports Intel RDT monitors the LLC space used by each RMID,

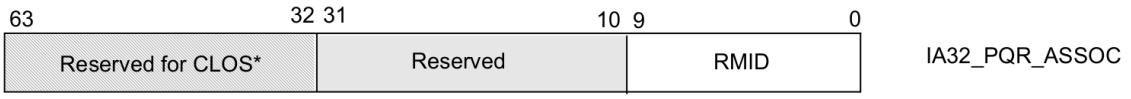


Figure 3.4: The MSR *IA32_PQR_ASSOC*, present in every CPU, allows to track each application's RMIDs through context switches and CPU migrations

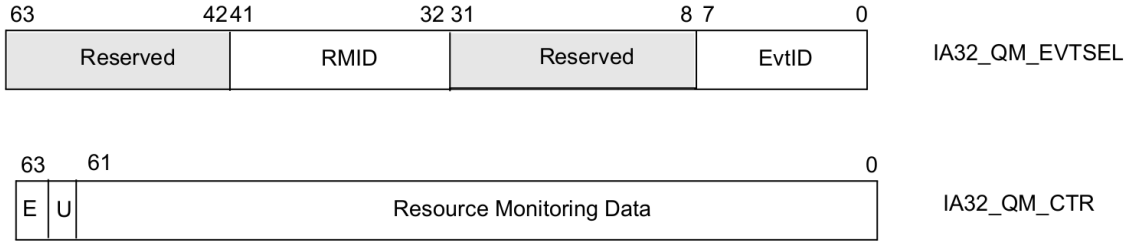


Figure 3.5: The MSRs *IA32_QM_EVTSEL* and *IA32_QM_CTR* in charge of selecting and reading a specific event supported by Intel RDT

making it possible for the operating system or VMM to read the LLC occupancy or memory bandwidth consumption for every application at any given moment. A full description of the Intel RDT functionality can be found in [9].

In order to track each application's LLC occupancy or memory bandwidth, the user of the Intel RDT interface must keep track of every thread's RMID that is monitored on the system. The RDT hardware interface exposes multiple privileged registers known as *Model Specific Registers* (MSRs) [141, 8], which are accessed through the assembly instructions `rdmsr` and `wrmsr`. For each core there is a register known as *IA32_PQR_ASSOC* that allows to keep track of which RMID is assigned to a thread that runs on a specific core. When a context switch happens, the operating system must update the RMID field of this CPU register with the RMID of the currently running thread.

At any given time, the interface user can query the cache space or memory bandwidth usage for any running application by manipulating two registers in a specific order. First, the register *IA32_QM_EVTSEL* must be written with the specific RMID we wish to consult alongside the corresponding event code (EvtID=0 for *cache occupancy*). Afterwards, it is possible to read the monitoring data by retrieving the least significant bits of the register *IA32_QM_CTR*. Figure 3.5 shows the bit layout of both of them.

3.2.4.2. Shared-resource allocation technologies

The main focus of Cache Allocation Technology is to enable resource allocation based on application priority or Class of Service (COS/CLOS). The processor exposes a set of COS, which can be used by multiple applications or individual threads. In other words, multiple RMIDs can be assigned to the same COS. This functionality allows to group different applications to share the same memory bandwidth cap or cache partition.

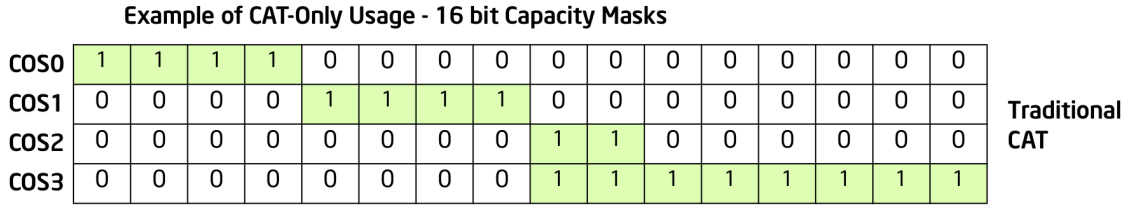


Figure 3.6: Capacity bitmasks used to establish which cache ways are shared by each CoS. Note that all the active ways in a CBM must be adjacent.

The allocation for the respective applications or threads is then restricted based on the class they are associated with. Each COS can be configured using a capacity bitmask (CBM) which indicates the cache ways that are enabled/disabled in the corresponding partition of the LLC. Hence, CBMs establish the degree of overlap and isolation between classes. For each logical processor a register exists (referred to as the *IA32_PQR_ASSOC* MSR or PQR) enabling the OS/VMM to specify a COS of the active thread/VM on a CPU. Figure 3.6 shows an example of cache allocations for different capacity bitmasks.

Recently in [36], PMCTrack functionality was further improved by implementing a monitoring module that provides support for accessing Intel RDT technologies [143]. Thanks to this support, we were able to access Intel RDT interface via wrapper functions provided by the Linux kernel, that basically makes use of inline assembly code to allow running the privileged instructions from C code.

There is another allocation technology apart from Intel CAT that is included in Intel RDT. The Memory Bandwidth Allocation (MBA) [14] technology allows the system software to limit the off-core access rate – to the LLC and, indirectly, to main memory of a specific application. The interface to Intel MBA works in a similar way to the one we have detailed in this section, but we will not give further details since this technology has not been used in any of the proposals made by this thesis.

3.3. Metrics

In this section we describe the metrics we considered in the evaluation of cache partitioning algorithms and asymmetry aware schedulers. Note that some of the metrics used on CMPs to measure throughput and fairness had to be adapted to work on AMPs.

3.3.1. Metrics on CMPs

Previous research on fairness for multicore systems [54, 171] defines a scheme as fair if equal-priority applications in a workload suffer the same slowdown as a result of sharing the system. To cope with this notion of fairness, we employ the *unfairness*

metric, which has been extensively used in previous work [54, 192, 160, 69]. For a workload consisting of n applications, this metric (lower-is-better) is defined as follows:

$$Unfairness = \frac{MAX(Slowdown_1,...,Slowdown_n)}{MIN(Slowdown_1,...,Slowdown_n)} \tag{3.1}$$

Notably, according to the definition of the unfairness metric, we could improve its value just by slowing down certain applications to achieve similar but potentially high slowdown figures. Clearly, this is unacceptable, as it may come at the expense of high throughput degradation. Therefore, the value of the unfairness metric must be always reported along with system throughput figures, as we do in this thesis.

To measure the performance degradation of an individual application in a multi-program workload we consider the *Slowdown* metric, defined as follows:

$$Slowdown_{app} = \frac{CT_{part,app}}{CT_{alone,app}} \tag{3.2}$$

where $CT_{part,app}$ denotes the completion time of application *app* when it runs sharing the system under a given cache-partitioning scheme, and $CT_{alone,app}$ is the completion time of the application when running alone on the CMP system.

The slowdown of a single-threaded application can be also defined in terms of the average number of instructions per cycle observed when it runs alone with all cache space available ($IPC_{alone,app}$) and that achieved when it runs with other applications in the workload under a certain cache-partitioning scheme ($IPC_{part,app}$):

$$Slowdown_{app} = \frac{IPC_{alone,app}}{IPC_{part,app}} \tag{3.3}$$

To quantify throughput on CMPs, previous works [171, 61] have employed the *System ThroughPut* (STP), which is defined as follows:

$$STP = \sum_{i=1}^{n} \left(\frac{CT_{alone,a_i}}{CT_{part,i}} \right) = \sum_{i=1}^{n} \left(\frac{1}{Slowdown_{a_i}} \right) \tag{3.4}$$

In order to measure global throughput, we discarded those metrics that depend on Instructions Per Second (IPS) or Instructions Per Cycle (IPC), since they might turn out to be misleading in the context of multithreaded applications. In specific, some parallel applications use active wait loops (spinning) to enforce synchronization; which may result in extremely high IPC figures and unrealistic throughput values reported during spinning periods in which no useful work has been done. Due to the IPC limitations, we did not use previous metrics such as Weighted Speedup [174] or Harmonic Mean Speedup [116].

3.3.2. Metrics on AMPs

Equations 3.1 and 3.3 are also valid for asymmetric systems. In this work, we assume that the IPS_{alone} on an AMP is maximized when a thread runs on a big core in isolation; that is the case across all the applications explored in our experimental platforms.

In quantifying throughput on AMPs, we turned to the *Aggregate Speedup* (ASP) metric [167, 159, 151], defined as follows:

$$ASP = \sum_{i=1}^{n} \left(\frac{CT_{small,i}}{CT_{sched,i}} - 1 \right) \tag{3.5}$$

where n is the number of applications in the workload, $CT_{small,i}$ represents the execution time of application i when it runs on slow cores alone on the system. Besides, $CT_{sched,i}$ is the execution time of application i when it is executed under a specific scheduling algorithm in the context of a multi-application workload. There have been similar metrics used by other authors, such as STP [183]. However, as pointed out in [151], ASP is better equipped to capture the differences in global throughput provided by the different scheduling algorithms on AMPs. Since ASP takes into account the global benefit that all applications in the workload gather when using the high performance big cores of the asymmetric platform.

Chapter 4

CAMPS: a Contention-aware scheduler for AMPS

The work summarized in this chapter primarily explores how to fairly schedule, at the OS level, a mix of unmodified applications on an AMP system, and does so by paying special attention to the impact of shared-resource contention on these architectures. To deliver fairness in this context, the scheduler must even out the progress made by the various applications as they run on the different core types throughout the execution [182, 167]. This requires the scheduler to be equipped with a mechanism enabling to measure the performance degradation accumulated by an application at runtime with respect to its solo execution (aka. *slowdown*). On AMPs, the slowdown depends on two main factors: (1) *performance asymmetry* and (2) *shared-resource contention*. *Performance asymmetry* refers to the fact that most applications derive a non-negligible speedup from using high-performance big cores relative to running on low-power small ones. When a thread runs on a small core, it slows down in proportion to its big-to-small speedup, which may differ greatly across applications and may vary over time through different program phases [24]. *Shared-resource contention* may also lead to substantial performance degradation. In current AMP hardware, clusters of cores of the same type (big or small) typically share a last-level cache [19, 78, 43] and other memory-related resources with the remaining cores. Applications running on the various cores may compete with each other for these shared resources, which could degrade their performance in an uneven and unpredictable way, as the hardware itself does not guarantee a fair usage of these resources [28, 192, 205, 197, 196, 54].

Previous scheduling proposals for AMPs, such as Equal-Progress [182] or ACFS [167], attempt to enforce fairness by just catering to performance asymmetry aspects, but they do not take shared-resource contention effects into account. As we demonstrate in this chapter, this leads to substantial performance/fairness degradation when several memory-intensive applications are present in the workload. Conversely, contention-conscious approaches that aim to deliver fairness [192, 63, 205] or strive to improve performance isolation [197, 194, 11] are not designed to work on systems that combine high-performance cores with low-power cores with differ-

ent microarchitectural features. Hence, these schemes do not factor in performance asymmetry.

To fill this gap, we propose CAMPS, an OS-level contention-aware scheduler for AMPs, which seeks to optimize fairness while maintaining acceptable system throughput. Our scheduler also exposes a configurable parameter enabling the user to trade fairness for throughput. As the vast majority of schedulers proposed for AMPs [24, 111, 164, 183, 182, 98, 167], our proposal primarily targets long-running compute-intensive workloads. In particular, in this thesis chapter we make the following main contributions:

- We devised a novel runtime mechanism to predict the slowdown that a thread in the workload experiences as it runs on the various cores of an AMP. Specifically, our scheduler approximates the current slowdown by monitoring various runtime metrics via performance monitoring counters (PMCs), and by comparing that information with the thread's past history gathered in low contention scenarios.
- Unlike other OS-level schedulers for AMPs [111, 167], CAMPS does not rely on platform-specific speedup prediction models, which typically entail the monitoring of a specific set of hardware PMC events that may differ substantially across processor models and architectures [111, 164, 152, 167]. Instead, our proposal employs a small and fixed set of performance metrics that can be easily gathered using PMCs available in commercial AMP hardware, thus making the scheduler portable across architectures.
- We implemented CAMPS in the Linux kernel, on top of the Completely Fair Scheduler (CFS), which is largely asymmetry agnostic. As we demonstrate in this chapter, the completion time of an application under the stock Linux scheduler may vary substantially across multiple runs of the same workload on an AMP. As a result, CFS and the HMP (Heterogeneous Multi-Processing) scheduler [154] –an extension of CFS for big.LITTLE platforms– constitute unfair scheduling schemes for asymmetric multicores, especially when compute-intensive applications are present in the workload. Notably, CAMPS delivers more consistent performance from run to run and higher degree of fairness for a wider spectrum of workloads.
- For our experimental evaluation, we employed the Intel QuickIA prototype [43] as well as commercial ARM-based asymmetric multicore platforms [19, 78]. We performed an extensive experimental comparison with previously proposed asymmetry-aware schemes [111, 164, 167]. Our analysis reveals that CAMPS improves fairness by up to 11% compared to a state-of-the-art fairness-aware scheduler [167], and at the same time improves throughput by up to 17%.

The remainder of this chapter is organized as follows. Firstly, Section 4.1 provides a brief rundown of the key aspects and background of this research topic and motivates our work. Section 5.2 discusses related work. Section 4.3 outlines

the design and implementation of our asymmetry-aware scheduling proposal. Finally, Section 4.4 showcases our experimental results and Section 4.5 presents the conclusions of this chapter.

4.1. Motivation

In this section we first introduce the notion of fairness employed in our work, and discuss the challenges associated with determining the slowdown at runtime. We then present an experimental study that showcases the main observation we exploit to determine the slowdown on-line on AMPs.

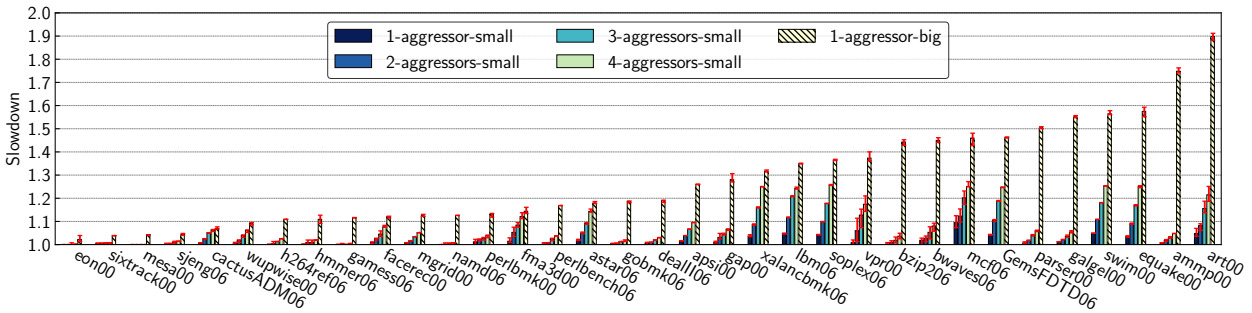
4.1.1. Fairness on AMPs

Delivering fairness entails ensuring that the slowdown accumulated by the various application threads throughout the execution remains as even as possible [54, 192, 167, 182], while maintaining acceptable throughput. To this end, the scheduler must be equipped with a mechanism to determine a thread's slowdown online. As defined in Equation 3.3, measuring the Slowdown requires knowing the IPS of a given application when running under a specific scheme. Notably, measuring the slowdown at runtime by using is difficult in practice; while a thread's IPS_{scheme} can be easily obtained via PMCs, accurately determining IPS_{alone} online is a challenging task, even on symmetric CMPs [205, 192]. For that reason, existing scheduling algorithms for symmetric CMPs typically rely on estimation models to approximate IPS_{alone} [192], or employ heuristics to determine the degree of performance degradation indirectly via contention-related metrics [205], such as the last-level cache (LLC) miss rate [28, 192]. Unfortunately, these scheduling algorithms are not designed to work on systems featuring different core types. Moreover, adapting them to AMP systems represents a challenging task, as these schedulers assume that the value of key performance metrics used to drive scheduling decisions (e.g. IPC or LLC miss rate) do not vary across cores when the application runs alone. On current AMP hardware [18, 19, 43], this assumption is not valid, as cores may exhibit different microarchitectural features and cache sizes [111, 167]. This fact further complicates determining the slowdown on an AMP.

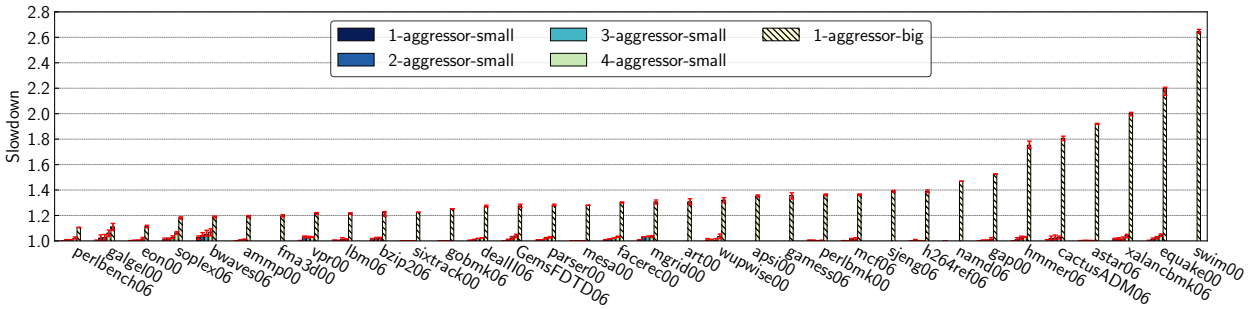
4.1.2. Impact of shared resource contention on AMPs

Recently proposed fairness-aware schedulers for AMPs [182, 167] implicitly rely on the assumption that a thread's slowdown is 1 (no performance degradation) when it runs on a big core, even if it runs simultaneously with other threads. In a similar vein, the thread's Speedup Factor is used by these schedulers to approximate the slowdown when the thread runs on a small core.

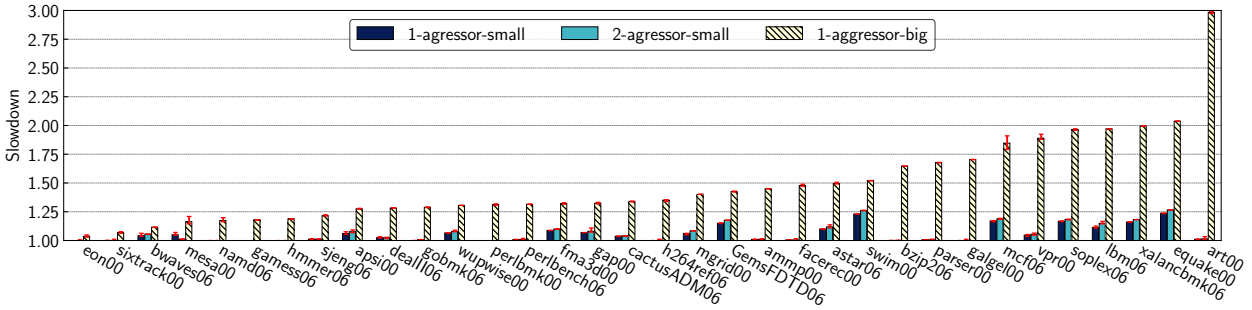
Assuming that a thread's slowdown is negligible when it runs on a big core (as done in [182, 167]) is unrealistic under shared resource contention. To illustrate this fact



(a) 2B-4S-Juno



(b) 4B-4S-Odroid



(c) 2B-2S-QuickIA

Figure 4.1: Average slowdown relative to solo execution experienced by various benchmarks when mapped to a big core and run simultaneously with several instances of an aggressor. The error bars report the minimum and maximum values gathered across the various runs (five executions). A suffix ("00" or "06") was appended to the application name to indicate the benchmark suite it belongs to (CPU2000 or CPU2006, respectively).

we experimented with diverse AMP platforms. Our analysis allowed us to draw two major insights:

1. Performance degradation due to resource sharing among big cores can be substantial on current AMP hardware (up to 2.98x on our experimental platforms), and should be accounted for when tracking the slowdown of a thread to enforce fairness as well as to ensure effective utilization of big cores. In fact, for some programs, the benefit that comes from running on a big core with respect to a small one could be substantially reduced – or even completely eliminated – due to the contention-related performance degradation.

2. Monitoring a thread's IPS when it runs on a big core, and in a low contention scenario across the neighboring big cores (i.e. those sharing the LLC) is typically a good estimate for IPS_{alone}. Essentially, the performance penalty that a thread mapped to a big core may suffer from placing multiple memory-intensive aggressors on small cores is usually very low compared to the one that comes from interference with memory-intensive threads running on neighboring big cores. This has to do with the memory-hierarchy organization on current AMP hardware, as well as with the fact that small cores typically utilize less memory bandwidth than big cores.

We draw these conclusions from experiments in which we measured the slowdown experienced by SPEC CPU applications when mapped to a big core, and running simultaneously with several instances of a memory-intensive aggressor application. We used benchmarks from both the CPU2000 and CPU2006 [1] suites to consider a wider diversity of working set sizes. As the aggressor, we used the `bandwidth` benchmark [196], which accesses memory in sequence with no data dependency between consecutive accesses. This allows the CPU generate multiple memory requests in parallel, maximizing memory level parallelism (MLP). This benchmark introduces substantial contention on the LLC, shared buses and DRAM controller. On our platforms, we observed that `bandwidth` causes even a higher degree of contention than that generated by highly memory-intensive programs from SPEC CPU, such as `lbm`.

A detailed explanation of the asymmetric platforms topologies and characteristics can be found in Section 3.1.1. Figure 4.1 shows the slowdown (relative to the solo execution) that different applications experience when running simultaneously with several instances of `bandwidth`. For each benchmark, which is always assigned to a big core in our experiments, we explored different scenarios. In the first one, denoted as "1-aggressor-big", the benchmark runs simultaneously with one instance of `bandwidth`, which is mapped to a different big core; the small cores remain idle in this case. In the other scenarios, labelled as "N-aggressors-small", N instances of `bandwidth` are mapped to small cores; thus, in leaving the remaining big cores

[1] Note that the SPEC CPU 2017 benchmarks did not exist when the proposal made by this chapter was created. We incorporated them to our experiment base and used them on later chapters.

unused, we remove contention on the LLC and the bus interface of the big core cluster, but not on the DRAM controller.

It is worth mentioning that a full-detailed description of the asymmetric platforms used in this thesis can be found in Section 3.1.1. For simplicity, we employ the nB-mS-$platformtag$ notation to refer to each configuration, where n and m denote the number of big and small cores, respectively. On *2B-4S-Juno* and *4B-4S-Odroid*, the set of cores of the same type (big or small), which make up a cluster, share a last-level cache (L2) and a bus interface with the remaining cores in the cluster. On *2B-2S-QuickIA* (a dual-socket system) a bus interface is shared between cores of the same cluster; a shared LLC exists on the big core cluster only. All platforms feature a single DRAM controller. Note that the big core cluster is made up of out-of-order cores, in contrast to the simpler and more energy-efficient small core cluster that features an in-order pipeline.

As is evident, the slowdown can be substantial (up to 1.9x on *2B-4S-Juno*, up to 2.65x on *4B-4S-Odroid*, and up to 2.98x on *2B-2S-QuickIA*) when both the benchmark and a single aggressor run simultaneously on big cores, even though small cores are unused. By contrast, when one aggressor runs on a small core the slowdown drops significantly; for most benchmarks it is below 10% in this case. Actually, if we populate all the small cores with aggressor instances, slowdown values are no greater than 26%, still much smaller than those obtained when the aggressor runs on a big core. There are two main reasons associated with this behavior. First, the contention on the LLC and on the shared bus (big-core cluster) is removed completely in the "N-aggressors-small" scenarios. Second, we observed that the pressure a single aggressor puts on the shared resources is higher when it runs on a big core than on a small one. This has to do with the fact that in-order small cores cannot handle multiple outstanding cache misses, leading to a lower bus and memory bandwidth utilization, and as a result to a smaller degree of contention.

Finally, we should highlight that not every application experiences noticeable slowdown due to contention when executed together with highly memory-intensive benchmarks such as `bandwidth`. For example, this is the case of `sixtrack`, `eon` or `mesa`. As pointed out in previous work [28, 205], CPU-intensive applications with a very small working set (which fits in a reduced portion of the LLC) and good cache locality, or those that do not put a lot of pressure on the memory hierarchy, do not experience significant performance penalty due to contention. As in [192], our scheduling proposal uses the bus transfer rate (BTR) to identify scenarios where threads are unlikely to suffer from contention when running on a big core cluster. In our platforms, the BTR is measured as follows:

$$BTR = \frac{bus\_ra * LLC\_cls * freq}{cycles} \tag{4.1}$$

where *bus_ra* is the amount of bus read accesses, *LLC_cls* represents the last level cache line size, *freq* depicts the processor frequency and *cycles* is total cycle count.

4.2. Related Work

The closest fairness-aware proposals to our CAMPS scheduler are the Equal-Progress [182] and ACFS [167] scheduling strategies. Both schedulers take per-thread SF values into consideration when tracking the slowdown that each thread in the workload experiences at run time and tries to enforce fairness by evening out observed slowdowns. Equal-Progress and ACFS exhibit important differences that are worth discussing. First, when determining a thread's slowdown, Equal-Progress does not factor in the past speedup phases the thread underwent. Instead, the slowdown is approximated by taking into account the total cycle count that the thread has consumed on each core type thus far and the current SF [182]. ACFS, on the contrary, maintains a per-thread counter that accumulates the total thread's progress based on the current and the past speedup application phases. In Section 4.3, we describe this progress-tracking mechanism in detail and illustrate the differences with the one used by CAMPS. Second, Equal-Progress was designed to achieve equal slowdown across threads, and so it only takes into account the SF of individual threads when computing slowdowns. ACFS, by contrast, takes into account the application-wide speedup to guarantee *equal slowdowns among applications*. This feature makes it possible for ACFS to provide a better support when multithreaded applications are included in the workload [167]. Third, ACFS supports user-defined priorities, while the Equal-Progress scheduler does not. Finally, note that Equal-Progress relies on either IPC sampling or PIE to obtain SFs online [182]. Since PIE is not available on existing asymmetric hardware, –as done in [167]– we evaluated the history-based variant of Equal-Progress, which is based on IPC sampling.

The main limitation of ACFS and previous proposals [167, 182, 120, 24] is the fact that they do not take shared-resource contention effects into consideration. As our experiments in Section 4.4 reveal, failing to cater to these effects leads the scheduler to exhibit unfair behavior when multiple memory-intensive programs are included in the workload.

Our scheduling proposal, predicts a thread's cross-core relative performance by measuring its actual IPS, and by comparing it with an estimate of the IPS_{alone} – approximated with big-core IPS values collected for different program phases in low-contention scenarios. This makes it possible to avoid the phase-related inaccuracies of IPC sampling [172], and allows us to cater to the potentially high variability of the IPS under different contention levels. Notably, the strategy proposed in this thesis, CAMPS, does not employ platform-specific SF prediction models, as ACFS does, but instead relies on the monitoring of a fixed set of high-level performance metrics (the same across platforms). This removes the need for conducting non-trivial offline analyses on each system to build speedup prediction models, thus improving the scheduler portability. In fact, training prediction models for different metrics could be a possible avenue for future work; considering the amount of possibilities in offline analysis offered by promising machine learning techniques such as deep learning.

To the best of our knowledge, the only existing contention-aware scheduling algorithms for AMPs are those proposed in [62] and [22]. Unlike CAMPS, which was de-

signed from the ground up as an OS-level scheduler, the strategies proposed in [62] and [22] constitute user-level scheduling prototypes, which perform thread-to-core mappings by leveraging CPU-affinity system calls. The scheduler proposed by Fan et al. [62] strives to improve the system throughput when using workloads consisting of single-threaded programs. It relies on two prediction models –*specific to each application and platform*– enabling the scheduler to approximate the degradation that the application suffers at runtime due to contention. Generating these prediction models requires to go through an offline training phase that entails running 80 workloads where the application is included. Our proposal –primarily designed to optimize fairness rather than throughput– does not rely on platform-specific or per-application prediction models, thus preventing the user from conducting the extensive offline profiling required to build those models [62]. Moreover, as opposed to our approach, [62] assumes that an application speedup factor is known beforehand (e.g. determined offline); this assumption is unrealistic on most practical scenarios. Barati et al. [22] propose a fairness-aware scheduler specifically tailored to asymmetric systems where cores differ in processor frequency only. It is well known that an application's degree of memory intensity is enough to approximate its slowdown when it runs on cores with the same microarchitecture but different frequency [172, 167]. For that reason, relying exclusively on the memory access rate is effective under frequency-based asymmetry [111]. However, previous work [111, 164] has demonstrated that this form of performance asymmetry differs substantially from that of commercial AMP hardware available today, where the various cores may exhibit profound microarchitectural differences and diverse cache sizes. In this scenario, other aspects beyond an application's degree of memory intensity must be taken into consideration for effective scheduling [111, 183, 167]. Unlike [22], our approach is implemented in the OS kernel and does not make any assumption about the form of performance asymmetry of the platform. This enables us to perform an extensive comparison with recent fairness-aware approaches [182, 167] by employing real AMP hardware.

Many of the research proposals that tackle contention have been evaluated as user-level prototypes. Even if they are completely valid approaches that may be adopted in production systems, they do not consider a completely realistic scenario. Some of the algorithms rely on complex statistical models that use costly floating point operations, that would not be viable at the operating system level. Even if you are able to access the partitioning interface from userspace, you get the overhead penalty of context switches every time. Given the fact that this approach entails making several system calls to access privileged resources such as performance monitoring and cache partitioning hardware. Besides, by operating at user level you miss important events from a thread's life cycle, such as fault pages and other blocking operations that might affect parallel applications. These issues complicate applying certain optimizations in scheduling schemes, since you are unable, for instance, to detect and accelerate critical sections. All in all, we consider that the system software is the most natural place to implement scheduling policies that gather online performance metrics or might make use of partitioning hardware extensions, allowing to minimize the impact that their implementations might have in the system latency and improving its responsiveness.

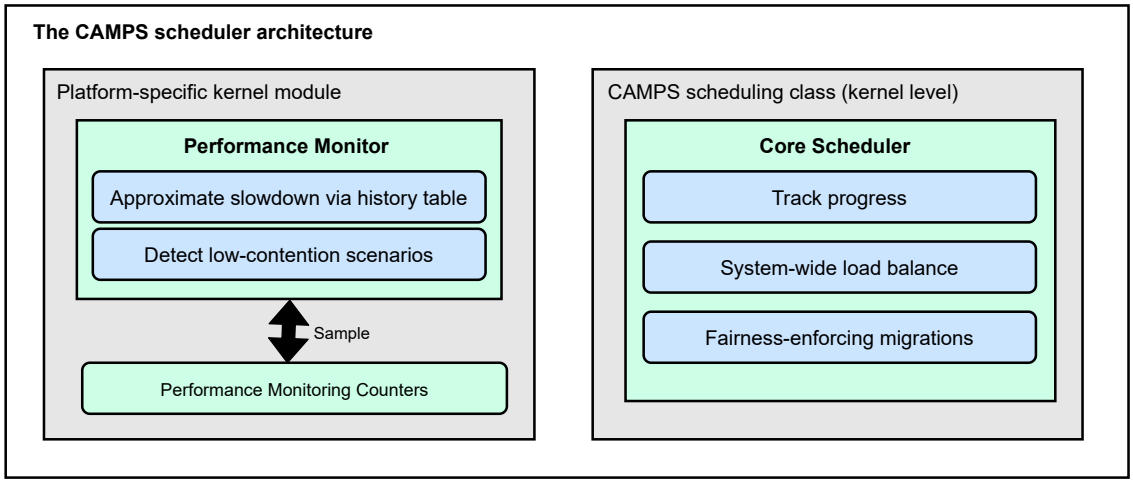


Figure 4.2: Diagram of the CAMPS architecture that summarizes the functionality of the the *core scheduler* and the *performance monitor*.

4.3. The CAMPS scheduler

CAMPS consists of two components: the *performance monitor* and the *core scheduler*. The performance monitor gathers the value of various runtime metrics for each thread in the workload using performance counters (PMCs), and feeds the core scheduler with critical information it needs, such as estimates of threads' slowdowns. The *core scheduler* assigns threads to big and small cores so as to preserve load balance, and swaps threads between cores when necessary to ensure that applications achieve similar progress on the AMP.

In this section we first present general aspects regarding our implementation of CAMPS in the Linux kernel. Then we outline the progress tracking mechanism and discuss how fairness is enforced via thread swaps. Next, we cover the non-work-conserving (NWC) mode of CAMPS, which may be triggered on special occasions to aid the performance monitor in approximating the slowdown of specific threads. Finally, we describe special features included in the scheduler to effectively deal with multithreaded applications and a special parameter that allows to trade fairness for throughput.

4.3.1. CAMPS in the Linux kernel

The Linux scheduler is equipped with multiple scheduling algorithms (CFS, FIFO, etc.), which are implemented as independent *scheduling classes*. CAMPS's core scheduler was bundled as a new scheduling class in the kernel. In creating this class, we started off with a fork of CFS (fair class), and implemented CAMPS's *core scheduler* on top of it. By contrast, the *performance monitor* (platform specific) was implemented in a loadable kernel module –bundled as a monitoring module of the PMCTrack tool [160]. Figure 4.2 depicts the CAMPS architecture, showing the main functions of both the *core scheduler* and the *performance monitor*.

It is worth noting that Linux CFS is largely asymmetry agnostic; as we show in Section 4.4.2, CFS may randomly assign an application to different core types in subsequent runs of the same workload, which leads to inconsistent performance across executions on an AMP system. Moreover, CFS is contention unaware [206] and does not feature any mechanism to keep track of the progress that a thread makes as it runs on the different core types throughout the execution. (Actually, to CFS, a *tick* consumed on a big core *is worth the same* as a tick consumed on a small core [120].) As a result, and unlike CAMPS, CFS does not guarantee similar progress (fairness) across applications on AMPs.

Our scheduling class just relies on the stock Linux scheduler for two main tasks: (1) to enforce load balance between cores of the same type (big cores and small cores separately), and (2) to multiplex CPU usage among threads assigned to the same CPU (i.e. the CFS algorithm is applied on a per-CPU basis). CAMPS's core scheduler, by contrast, takes care of enforcing system-wide load balance, and evens out relative progress among applications, by assigning threads to the different core types and by triggering migrations if necessary. Since CAMPS is based on CFS, it maintains per-CPU run queues of runnable threads. In addition, it employs two linked lists of runnable threads, with threads assigned to big cores and to small cores, respectively; each list is protected with a read-write spinlock. Note that these lists are manipulated much less often than per-CPU run queues (e.g. when a thread is migrated onto a different core type). We observed that this design approach is not subject to scalability issues on current AMPs which feature a limited number of cores (up to 8). Notably, previous research [140] has demonstrated that even relying on a single global run queue delivers more than sufficient scalability on current AMP platforms. Nevertheless, to make CAMPS more scalable for future AMPs with a higher core count, the scheduler could be reimplemented by leveraging the core-partition approach described in [172].

4.3.2. Determining the slowdown at runtime

The *performance monitor* approximates a thread's current slowdown by using Eq. 3.3; the actual IPS is measured with PMCs, and the IPS_{alone} is estimated by using a *history table* maintained for each thread at runtime. This table stores IPS values observed in past execution phases when the thread ran on a big core in a low-contention scenario. As shown in Section 4.1, on a big-core cluster, the performance degradation that comes from interference with threads running on the small core cluster is typically very small. Based on this observation, big-core low-contention IPS values recorded in the table are used to approximate IPS_{alone}.

To detect low-contention scenarios on a big core, we leverage the heuristics based on the bus transfer rate (BTR) metric proposed in [191, 192]. Essentially, a thread whose BTR is smaller than a given *low_btr* threshold is not likely to suffer noticeably from contention. In a similar vein, when the aggregate BTR in a core cluster falls below a given *high_btr* threshold the degradation due to contention is typically very low [192]. As shown in [191, 192], the thresholds can be easily determined for

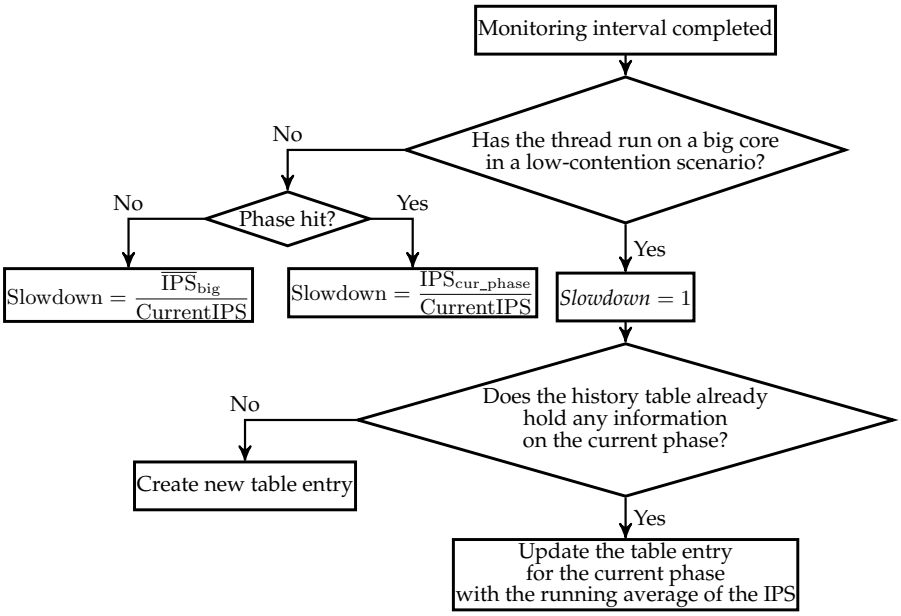


Figure 4.3: Mechanism used by CAMPS for approximating a thread's slowdown with help from the history table

any platform by using synthetic benchmarks. Essentially when the thread runs on a big core in this kind of low-contention scenarios we assume that its slowdown is 1 (no degradation). If these scenarios do not occur naturally as a result of the contention-aware thread assignments performed by CAMPS, the core scheduler will enter a *non-work-conserving mode* (described in Section 4.3.4), which introduces low-contention scenarios artificially.

Indexing a thread's history table, which is necessary to approximate the slowdown and to record new IPS samples, requires the performance monitor to figure out whether information on the current execution phase already exists in the table or not. To this end, we leverage a variant of the phase-detection mechanism employed in previous work [15]. Overall, the scheduler continuously monitors the percentage of instructions of different types (int/FP, load, store, branches, etc.) retired during the last monitoring interval, which make up a *instruction type vector* (ITV). If the Manhattan distance of the ITVs for two performance samples (collected at different intervals) is smaller than a threshold, both samples are assumed to belong to the same execution phase. Note that the Manhattan distance of two n-dimensional vectors (X and Y) is defined as $\sum_{i=1}^{n} |X_i - Y_i|$.

Unfortunately, this phase-detection scheme, whose effectiveness was evaluated on a simulator [15], cannot be implemented in the real AMP platforms we used (described in Section 3.1.1), as the performance monitoring unit is not equipped with the necessary performance events or with enough physical PMCs. To overcome this issue, we adapted the phase-detection approach by monitoring the thread's BTR and its IPS (required for our scheduling policy) along with two alternative *control metrics*: the number of L1 cache accesses per 1K instructions, and the percentage of branches retired over the total instruction count. As the ITV, the value of these control metrics for a specific phase remain the same under different levels

of shared resource contention, and they do not vary significantly across core types. Notably, the value of these metrics changes dramatically when an application enters a new phase exhibiting a different degree of memory intensity and branch-prediction related behavior. These two aspects have a great impact on cross-core relative performance on AMPs [111, 167]. These observations make the selected *control metrics* very suitable to index the table.

Figure 4.3 depicts how the *performance monitor* estimates a thread's slowdown and maintains its history table. The table is updated at the end of a monitoring interval in which the thread ran on a big core cluster in a low-contention scenario. If the table does not already hold information on the current phase, that IPS value is recorded in a new entry; otherwise the existing table entry is updated with a running average of the IPS values recorded for that phase. In either case, the scheduler estimates the slowdown to be 1 (no degradation). When the thread runs on a small core, or on a big core under potential contention, CAMPS accesses the history table to estimate the slowdown. If the IPS for the current phase is found in the table (i.e. *phase hit*), the slowdown is estimated with the ratio of the IPS value retrieved from the table ($\text{IPS}_{\text{cur\_phase}}$) and the current IPS value measured in the last monitoring interval. In case that no information is found for the current phase (i.e. *phase miss*), the estimated slowdown is the ratio of the average IPS across samples stored in the history table ($\overline{IPS}_{big}$) and the current IPS value.

To determine the most suitable size for the history table we conducted a sensitivity study by analysing performance traces gathered with PMCs for SPEC CPU applications. This sensitivity study can be found in Section 4.4.1. Based on the results of our analysis we opted to use history tables of 22 entries. This choice provides a good trade-off between slowdown estimation accuracy and memory utilization.

4.3.3. Progress tracking and enforcing fairness

CAMPS's core scheduler maintains a progress counter for each thread referred to as `amp_progress`. This counter tracks how much progress the thread has made thus far relative to the progress that would have resulted from running it on a big core the whole time in complete isolation (no contention). When a thread runs for a clock *tick* on a given core type, the scheduler increments `amp_progress` by $\Delta_{\texttt{amp\_progress}}$, defined as follows:

$$\Delta_{\texttt{amp\_progress}} = \frac{100 \cdot W_{\text{def}}}{CS \cdot W_t} \tag{4.2}$$

where W_t is the thread's weight, derived directly from the application priority (set by the user); W_{def} is the weight of applications with the default priority; and CS is the thread's current slowdown as estimated by the *performance monitor*.

To illustrate the main idea behind the definition of $\Delta_{\texttt{amp\_progress}}$, let us analyse the following scenario illustrated by Figure 4.4. Two sequential applications with the default priority (i.e. $W_{\text{t}} = W_{\text{def}}$) run on an AMP system with no contention present and their single runnable threads – A and B – are mapped to a big core and a small

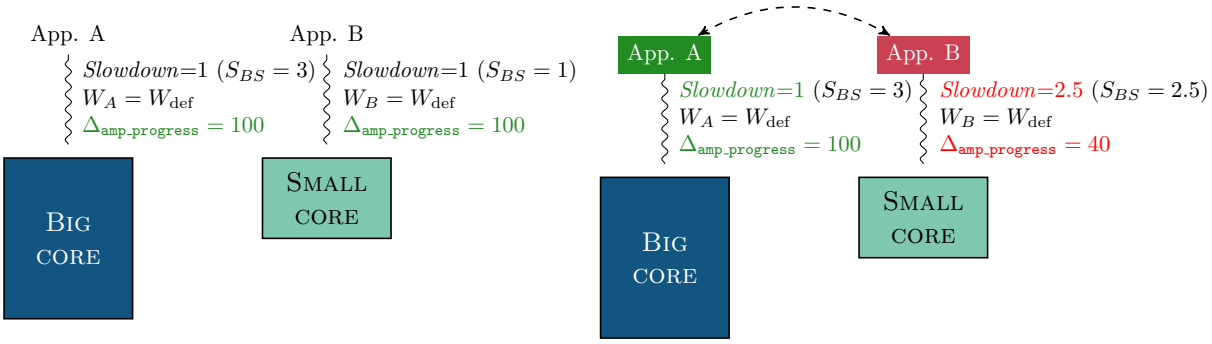


Figure 4.4: Example that illustrates CAMPS design when scheduling two applications at different points of their execution on an AMP system.

core, respectively. At a specific point of the execution, application A goes through a execution phase with ($S_{BS} = 3$), running three times faster on a big core relative to a small one. On the contrary, application B rubs on a small core and would not make any more progress on a big core($S_{BS} = 1$). In this scenario, the *CS* for both applications would be 1 since there is no performance degradation, so $\Delta_{\texttt{amp\_progress}}$ would be equal to 100. This indicates that an application is now making 100% of its maximum attainable progress, as it would achieve by running on a big core in isolation. Let us now consider a time where application B goes through a different program phase with a current slowdown of 2.5. In these circumstances, $\Delta_{\texttt{amp\_progress}}$ would be equal to 40; namely, the application only makes 40% of its maximum attainable progress that is achieved when running on a big core in isolation. Therefore, the lower the slowdown (*CS*), the faster a application's `amp_progress` counter will be incremented. Eventually, the difference between the *amp_progress* counters will increase over time and, upon reaching a certain threshold, the applications will be swapped to preserve fairness on the AMP system.

When a new thread enters the system, the *core scheduler* assigns it to the least loaded core on the AMP, so that the load balance across cores is preserved. In doing so, CAMPS picks big cores first, since this contributes to maximizing throughput [120, 165]. Notably, the `amp_progress` counter of a newly created thread is set to the maximum value for this counter observed among threads in the system at that point. This initial value enables a *fair* progress comparison among threads that entered the system at different points in time. Every thread also has to go through a warm-up period (10 sampling intervals in our experimental setting) right after being spawned. The first two samples collected during the warm-up period are discarded for slowdown estimation, so as to mitigate mispredictions associated with cold-start effects (e.g. the number of cache misses typically spikes intermittently at the beginning of the execution).

Note that the approach used by CAMPS to enforce fairness via progress tracking has several aspects in common with that of the ACFS scheme [167]. Despite the fact that both schedulers maintain per-thread progress counters, they employ different mechanisms to determine a thread's current slowdown (denoted as the *CS* factor in Eq. 4.3) at runtime. While CAMPS does take shared resource contention into consideration, as described in Section 4.2, ACFS does not. In fact, ACFS assumes

that a thread's slowdown is always 1 when it runs on a big core, and uses the thread SF (predicted via a platform-specific estimation model) to approximate its slowdown when the thread runs on a small core.

Like ACFS, CAMPS may also trigger thread swaps between cores every so often to enforce fairness. Essentially, threads mapped to big cores usually make faster progress than threads running on small ones, which causes unfairness. To even out the progress among threads via thread swaps, CAMPS follows a similar approach to that of ACFS [167]. Specifically, a thread running on a big core will be swapped with another thread running on a small core only when the difference of their progress counters exceeds a given threshold, referred to as `amp_threshold`. Specific instructions are provided in [167] for selecting the most appropriate value of this threshold for a given platform. For our experiments, we chose a value of this threshold so as to achieve an average migration rate of 400ms, which ensures negligible overheads in current AMP hardware [167].

It is worth highlighting that special care is taken with *sleeper* threads (i.e. those that wake up after a potentially long suspension). Essentially, the progress counter of a *sleeper* thread that just woke up could be much smaller than that of other threads in the system, as the thread's progress counter remains unmodified while it sleeps. This situation could lead *sleeper* lagging threads to monopolize big cores when waking up after a very long pause. To address this issue, CAMPS resets a thread's progress counter when it realizes that it has been blocked for a certain time period, which is application specific. This period corresponds to the time that it would take this thread when just migrated to a small core (due to a fairness-oriented swap) to be swapped back to a big core. This time period depends on `amp_threshold` and on the thread's average slowdown, which is maintained by CAMPS. The reset value for the counter is the minimum value for the progress counter observed among threads on the system.

We found that relying on the progress counters alone (as ACFS does) is ineffective in case that aggressor applications and contention-sensitive programs are mapped to the big-core cluster simultaneously. As shown in Section 4.1, this mapping may severely degrade the performance of contention-sensitive applications, which may backfire by decreasing the benefits from using a big core. To mitigate this issue, CAMPS uses the BTR-based heuristics proposed in [192] to detect potentially contentious scenarios, and favors those threads swaps that contribute to avoiding contention on the big core cluster. Algorithm 4.1 illustrates how CAMPS's core scheduler selects threads to be swapped. The algorithm is executed as soon as the scheduler detects that swap candidates exist on both core types (i.e. the progress counters of two threads running on opposite core types exceed `amp_threshold`). Specifically, CAMPS always selects the thread with the highest `amp_progress` counter running on a big core –denoted as T_B– to be migrated to a small core. In choosing its swap partner, small-core threads with a lower value of the `amp_progress` counter are considered first. If a contention-friendly swap is found (i.e. it leads to a low contention scenario on the big core cluster), the swap is performed. Otherwise, the thread with the lowest BTR is the one selected as the swap partner; this contributes to reducing the degree of shared-resource contention on the big core cluster as a

Algorithm 4.1: Selection of swap candidates in CAMPS.

Input: T_B is the runnable thread with the highest progress counter mapped to the big core, S is the set of runnable threads (T_S, i) assigned to small cores that constitute potential swap partners for T_B (i.e. `amp_progress`$(T_B)-$`amp_progress`$(T_S,i)\geq$`amp_thresh`). Note that $S \neq \varnothing$, and threads in S are sorted in ascending order by their `amp_progress` counter.

```
1  min_btr ← ∞; T_min-BTR ← NIL;
2  swap_performed ← false;
3  do
4      T_S, i ← Get first thread in S ;
5      if Swapping T_B and T_S, i leads to a low-contention scenario on the big core
         cluster || (amp_progress(T_B)−amp_progress(T_S, i) ≥ 2∗amp_threshold) then
6          Swap T_B and T_S;
7          swap_performed ← true;
8      else
9          Remove T_S, i from S;
10         if BTR(T_S, i) < min_btr then
11             min_btr ← BTR(T_S, i); T_min-BTR ← T_S, i ;
12         end
13     end
14 while !swap_performed && S ≠ ∅;
15 if !swap_performed then
16     Swap T_B and T_min-BTR;
17 end
```

result of the reduction in the cluster's aggregate BTR [192]. Note also that the scheduler forces the selection as a swap candidate of those threads that are lagging considerably behind the rest, namely, when the difference between T_B's progress counter and the thread's progress counter is greater than 2∗`amp_threshold`. This enables aggressor threads to eventually have a chance to run on big cores when the workload includes multiple memory-intensive applications.

4.3.4. Non-work conserving mode

As discussed earlier, CAMPS populates a thread's history table while it runs on a big core cluster during low contention scenarios. Unfortunately, when the number of memory-intensive threads in the workload is high, low contention scenarios might not occur that often for contention-sensitive programs. In these cases, CAMPS may transition into a *non-work-conserving (NWC) mode*, in which low contention scenarios are created artificially. To control transitions into this special mode, CAMPS operates as follows. Every time that a thread completes $n - intervals$ consecutive monitoring intervals, the scheduler retrieves the thread's phase hit rate as well as the number of IPS samples that have been inserted into the history table over that time period. If the phase-hit rate falls below 80%, and no IPS samples have been inserted in the history table during that period, the scheduler enters the NWC mode. We will refer to the thread that caused the transition into this mode as the *NWC thread*.

When in the NWC mode, fairness-oriented thread swaps are not performed. During this special mode, the main goal is to collect as many low contention big-core IPS samples as possible for the *NWC thread*. To this end, if the *NWC thread* was not running on a big core already, it will be swapped with a big-core thread. In doing so, CAMPS tries to select a memory-intensive (high-BTR) thread as the swap partner, so as to reduce contention on the big core cluster as a result of the swap. Once the NWC thread is mapped to the big core, CAMPS will attempt to gather big-core IPS samples for this thread. If at this point a low-contention scenario does not yet occur on the big core cluster, the scheduler will temporarily disable (for a very short period of time) as many big cores as necessary to create such a scenario. In practice, making this possible comes down to disabling only a few big cores: those where memory-intensive threads are currently running. Note that during the NWC mode, other threads (in addition to the NWC thread) may leverage low-contention scenarios to populate the history table.

The scheduler will transition back into the normal operating mode when (1) the NWC thread's phase hit rate is over 80% –after inserting a number of IPS samples in the history table–, or (2) when the NWC thread blocks or terminates. Notably, when in the NWC mode, CAMPS still keeps updating thread progress counters. This allows threads that did not benefit the NWC mode (e.g. those assigned to big cores that were temporarily disabled), to be compensated later accordingly. In addition, to prevent that specific threads force the transition into the NWC mode systematically, we take progress counters into consideration when controlling transitions; threads progressing much further ahead than the rest at some point cannot become NWC threads.

Note that throughout the entire warm-up period, a thread is not allowed to trigger the activation of CAMPS's NWC mode (described in Sec. 4.3.4). This is a control measure to remove the potentially negative interference caused by the presence of multiple short-lived memory-intensive threads, which could otherwise activate the NWC mode ineffectively; that would lead to unnecessary overheads without reaping any benefits, as CAMPS discards a thread's history table when it terminates.

In our implementation in the Linux kernel, big cores are temporarily disabled in the NWC mode (when needed) by selecting the *idle task* to run forcefully on the corresponding core (this action is performed in the `pick_next_task()` operation of our scheduling class), and by temporarily binding to that core any thread previously assigned to it. We found that using short core disabling periods, such as the 100ms setting used in our experimental platforms (2 monitoring intervals), allows the scheduler to have a fine-grained control when in the NWC mode. Essentially, this enables CAMPS to better adjust to the number of core disabling operations required by the current NWC thread.

Although CAMPS was designed primarily for long-running compute-intensive workloads, latency-sensitive memory-intensive applications could be negatively affected by core disabling actions in the NWC mode. To deliver more consistent tail latencies, CAMPS could be seamlessly modified to map this kind of applications to small cores, which are never disabled when in the NWC mode.

4.3.5. Special support for multithreaded applications

On AMPs, an application can be developed so as to explicitly leverage the features of the various cores by dividing the computation into multiple tasks or threads specifically designed to run effectively on a particular core type. These applications are typically run by manually binding the various threads/tasks to the core type where they are meant to run. CAMPS supports the execution of those applications, as it respects user-enforced CPU affinities. Nevertheless running such an application along with other programs would not guarantee system-wide fairness; CAMPS strives to deliver fairness only across those (unmodified) applications whose threads are allowed to run on different core types. Notably, affinities in general greatly limit the schedulability in most OS-level schedulers [40], not only that of CAMPS.

To provide better support for multithreaded programs that do not rely on affinities, CAMPS leverages two mechanisms: *spin notifications* and *per-application history tables*.

Spin notifications enable the scheduler to be aware of those situations where threads in a multithreaded program busy wait (or *spin*) rather than blocking while waiting in synchronization primitives, such as barriers. Busy waiting enables to substantially reduce the number of context switches performed by the OS scheduler [121], and it may also reduce the number of thread migrations on AMP systems [165]. Nevertheless, *spinning threads* must be properly handled by the scheduler. The main issue is that busy-waiting threads may achieve a high IPS despite not doing useful work [2]. Using these misleading IPS values under CAMPS, would lead to polluting the history table and, in turn, to serious slowdown mispredictions.

To address this issue, we leverage *spin notifications* from user space to the OS by using a variant of the technique proposed in previous work [165]. In our implementation, we maintain a memory region shared between each application thread and the OS. When a thread begins to spin, it activates a flag in the shared memory region, which is later disabled as soon at the thread stops spinning. We opted to use a shared memory region rather than system calls (as in [165]) for spin notifications, since the former approach provides negligible overhead. When a thread is spinning, the *performance monitor* discards the associated IPS samples, and always estimates the slowdown for the thread to be 1, as it is not doing useful work. In a similar vein, CAMPS's *core scheduler* avoids migrating spinning threads to big cores. Notably, issuing spin notifications from user space does not require making changes in the applications as long as they use the synchronization primitives provided by the threading library or the underlying runtime system.

As a proof of concept we implemented this mechanism in the OpenMP runtime system provided by GCC, by instrumenting the code of synchronization primitives. For applications that do not use standard, library-based synchronization primitives, spin notifications could be exploited by leveraging hardware-aided spin-detection approaches [121] or by manually instrumenting the code.

[2] Best practices in implementing spin locks dictate using algorithms where a thread spins on a local variable [121]; this leads to a high IPC, due to the effective utilization of the CPU pipeline.

In many multithreaded applications, the various threads do the same kind of processing but with different data. In this scenario, we can leverage the IPS samples stored in a thread's history table to aid in predicting the slowdown for the remaining threads in the application. To this end, we maintain two levels of history tables for multithreaded applications: the per-thread table (L1) –presented in Section 4.3.2, and a per-application history table (L2). Essentially, when a thread creates a new phase entry in its own table, it inserts this new entry into the per-application table too. In doing so, other threads in the application that incur a L1 phase miss, can potentially retrieve information for the current phase by accessing the L2 (application-wide) table. If a L2 phase hit occurs, the entry is copied onto the thread's private table (L1). This way we avoid future accesses to the same L2 table entry. Note that in limiting the number of read and write operations on the L2 table, we reduce potential contention that comes from accessing the L2 table (protected with a lock) simultaneously from multiple CPUs. Although using two levels of history tables is specially well suited to applications where all threads run the same code with different data, the scheme could be trivially augmented to other kind of multithreaded programs (such as those following the pipeline paradigm) where a few threads perform a specific task cooperatively, where others do a different kind of processing. In that case, a L2 history table would be shared by threads that do the same kind of processing, which could be identified by the function that they execute.

Lastly, we should highlight that, for multithreaded applications, CAMPS downscales the CS factor in Eq. 4.3 in proportion to the number of runnable threads in the application. Note that this is a proxy for the amount of thread-level parallelism and ACFS also employs this technique [167]. Previous work [164, 167] has demonstrated that, when multithreaded programs are included in the workload, this approach enables the scheduler to provide better performance and fairness than making decisions based exclusively on per-thread slowdowns (or SFs).

4.3.6. Trading fairness for throughput

The progress tracking mechanism used by CAMPS is inspired by that of the ACFS scheduler. One of the potential benefits that comes from factoring in application weights in progress tracking just like ACFS does, is that this approach has already proven effective when it comes to enforcing user priorities on real asymmetric hardware [167]. Another powerful feature of ACFS's progress-tracking mechanism is that it can be augmented with a configurable parameter (referred to as the `unfairness_factor`) that allows the scheduler to gradually increase throughput on the AMP system in scenarios where fairness constraints are relaxed. At the same time, using high values of this parameter enables us to configure the scheduler to optimize throughput rather than fairness.

To allow the system administrator to trade fairness for throughput when needed, we augmented the *base* implementation of CAMPS (described in Section 4.3) with the `unfairness_factor` (UF) knob. In order to fully understand the differences

between CAMPS's and ACFS's implementation of this knob, we first outline the common aspects. To improve system throughput on the AMP, the scheduler must grant a higher share of the available big core cycles to those applications that derive a high speedup from using this kind of cores. To this end, we employ a dynamic priority scheme, where a thread's actual priority depends upon its static priority or weight (set by the user) and the big-to-small speedup of the application (S_{BS}).

$$\Delta_{\texttt{amp\_progress}} = \frac{100 \cdot W_{\text{def}}}{CS \cdot W_t} \tag{4.3}$$

Implementing this scheme comes down to replacing the *static* weight (W_t) in Eq. 4.3 from Section 4.3 (or in the corresponding equation for ACFS [167]) with its dynamic weight (DW_t), which is defined as follows:

$$DW_t = W_t \cdot \left(1 + \frac{(\texttt{UF} - 1) \cdot (S_{BS} - S_{min})}{S_{max} - S_{min}}\right) \tag{4.4}$$

where S_{max} and S_{min} are the maximum and minimum speedups observed among applications in the workload.

When the UF knob is set at its default and lowest possible value (1.0), we have that $DW_t = W_t$, so the scheduler behaves as the base implementation, hence attempting to optimize fairness. For UF values > 1, the scheduler increases throughput at the expense of degrading fairness gradually. Essentially, by replacing the static weight (W_t) with its dynamic counterpart (DW_t), the progress counter of high-speedup threads is incremented at a slower pace than that of low-speedup threads, which results in a higher big-core share for high-speedup applications and, in turn, in higher system throughput.

In order to make it possible for the scheduler to calculate a thread's dynamic weight (DW_t), it must be equipped with a mechanism to determine the big-to-small speedup online. As described in [167] ACFS relies on platform-specific models to approximate a thread's speedup using hardware performance counters. This approach could be adopted in CAMPS as well, but that would greatly limit the portability of the scheduler. Essentially, relying of this kind of estimation models entails monitoring a specific set of performance events that largely depends on the form of performance asymmetry in the platform [164], and which is typically tied to the processor models present in the AMP system [111, 167].

To guarantee that our scheduler remains portable across AMP platforms, we use the IPS values collected on both core types for the thread to approximate its speedup. Notably, to improve the accuracy of the prediction, we leverage information in a thread's history table by using IPS values from the same phase whenever available. To this end, we add a new field in each entry of the history table, referred to as $IPS_{max,small}$, which stores the maximum IPS value observed for the program phase in question when the thread runs on a small core. Because a thread's performance is usually lower under contention, $IPS_{max,small}$ is just a lower bound for the IPS that would be observed when running the program phase alone on a small core. At the end of each monitoring interval, the performance monitor accesses the history table,

Table 4.1: Table that gathers CAMPS parameters used for the experiments on this chapter.

Name	Value
sampling_period	50 ms
warmup_period	10 sampling intervals
low_btr/high_btr	thresholds established with synthetic benchmarks
amp_threshold	Tuned to ensure 1 migration/400 ms in average
uf (unfairness factor)	1

and updates it if necessary. In the event of a hit in the history table, the thread's speedup is approximated as $\text{IPS}_{\text{cur\_phase}}/IPS_{max,small}$. Otherwise, the speedup is estimated as $\overline{IPS}_{big}/\overline{IPS}_{small}$, where $\overline{IPS}_{small}$ is the average of the IPS values gathered for a thread on a small core.

4.4. Experimental evaluation

We begin in Section 4.4.1 by analyzing how varying the CAMPS history table size affects its performance, measured by the phase hit rate.

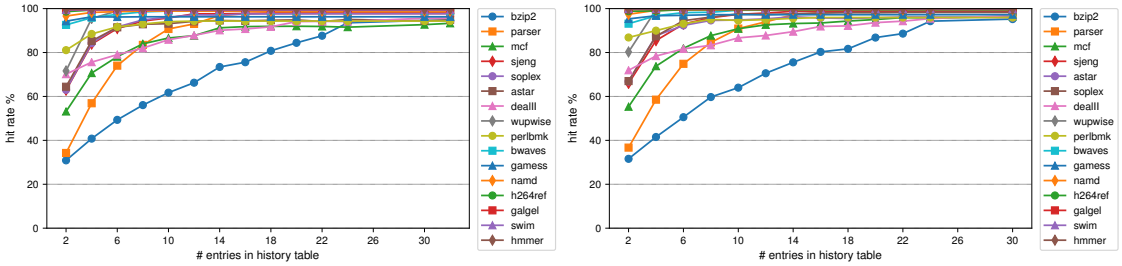
In Section 4.4.2 we compare the degree of fairness and other aspects of our scheduling proposal with that of the stock Linux scheduler (CFS) and with its extension for ARM big.LITTLE platforms (HMP [154]). To this end we experimented with a broad spectrum of workloads (long and short-running CPU-bound programs, IO-intensive and latency-sensitive benchmarks, etc.).

Afterwards, Section 4.4.3 studies the usefulness of the knob that allows to exchange fairness for throughput.

Finally we present in Section 4.4.4 the effectiveness of CAMPS with that of previously proposed asymmetry-aware schedulers [24, 111, 182, 167], which, as CAMPS, primarily target long-running compute-intensive applications. The schedulers were implemented as a scheduling class in the Linux kernel v3.10.104. By the time we started the implementation, that was the latest stable kernel version with official manufacturer support for the Odroid XU4 board. Variants of the vanilla v3.10.104 kernel were used on the other AMP systems considered, to maintain a common scheduler code base. Finally, Table 4.1 summarizes the set of CAMPS parameters that were experimentally established.

4.4.1. Determining the history table size

The effectiveness of our scheduling proposal is sensitive to the choice of the size of the thread's history table (maximum number of entries). As discussed in Section 4.2, this table is maintained by the *performance monitor* to approximate a thread's slowdown at runtime. Notably, in choosing this parameter, a trade-off must be made between memory utilization and accuracy in the slowdown prediction. An



(a) 20% of the samples for training (b) 40% of the samples for training

Figure 4.5: Phase hit rate for different maximum number of entries in history table.

important aspect is how to select the right size of the history table to achieve an acceptable phase hit rate for most applications, while eliminating unnecessary memory overhead.

To determine the most suitable size for the history table, we analysed performance traces gathered with hardware counters for 52 different applications in the SPEC CPU suite while running alone on a big core of the various AMP platforms used for our experiments. Since the conclusions of our study are largely similar on these platforms, we discuss the results for the *2B-4S-Juno* configuration (ARM Juno Board) only. To factor in the potential impact of shared resource contention in slowdown prediction –recall that, in potentially contentious scenarios, big-core IPS values are discarded by CAMPS–, we used only a fraction of the samples in the trace to populate the history table (i.e., *training* samples) and left the remaining ones (i.e. *test* samples) to approximate the phase hit rate for a particular table size. Note that training and test samples in our analysis are distributed all over the trace, and all samples are processed in order, to closely mimic the actual behavior of CAMPS's performance monitor.

Figures 4.5a and 4.5b show the phase hit rate for different applications and table sizes when using 20% and 40% of the samples for populating the history table, respectively. These two scenarios represent situations with a different degree of shared resource contention (a different fraction of samples is discarded). For the sake of clarity, the figures only display the data associated with 16 representative applications, which cover a wide range of trends observed during our analysis. The results reveal that for each program, a point can be reached in which increasing the size of the history table further does not provide any benefits. However, that point is application specific. For example, programs such as `hmmer` or `swim` can do just fine with a history table consisting of 2 entries (hit rate>99.4%), whereas for other programs, such as `mcf` or `bzip`, more than 16 entries are required to reach the saturation point.

In light of the results of our analysis, we opted to use 22-entry history tables in our experiments. That choice for the number of entries provides a good trade-off between the hit rate and memory utilization. When using 22 entries in the history table, 95% of the applications considered in the full set (52 programs) achieve a phase hit rate greater than 92% in the scenarios considered. Moreover, when the fraction of samples used to populate the history table is no smaller than 40%, 98% of

Table 4.2: Average reduction in unfairness and increase in throughput achieved by CAMPS over the other schemes on the Odroid XU4 board.

CAMPS vs. others	Reduct. in Unf.	Increase in throughput
HSP	32.75%	-16.03%
RR	16.89%	16.53%
Equal-Progress	18.67%	12.05%
ACFS	7.17%	4.51%

the applications considered obtain hit rates greater than 94.2%. For the remaining applications (e.g. `bzip`), using 22 entries in the history table guarantees a phase hit rate of at least 87.6%. Notably, such a history table occupies 1016 bytes on 64-bit platforms (e.g. Juno board) and 820 bytes on 32-bit platforms (e.g. Odroid XU4), which is roughly 15% of the size of the task structure.

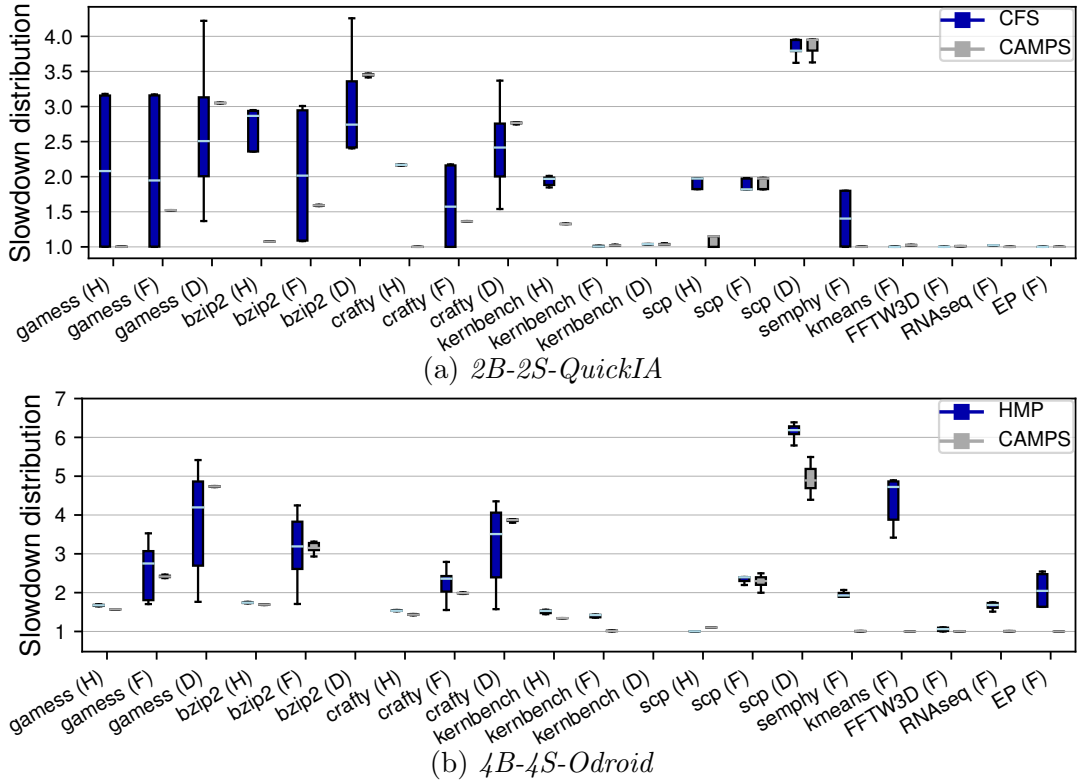
4.4.2. CAMPS vs CFS and HMP

We now illustrate how CAMPS compares with the stock Linux scheduler (*CFS*) and with *HMP* [154] in terms of performance variability across runs. In doing so, we consider diverse workloads, beyond those which CAMPS was optimized for. For the comparison against HMP, we employed the Odroid XU4 board (*4B-4S-Odroid*), as the kernel provided by the board's manufacturer uses HMP. To experiment with CFS, we used the *2B-2S-QuickIA* platform; CFS is the default scheduler on this platform (to the best of our knowledge, no implementation for HMP is currently available for x86 systems).

To measure how an application's completion time varies across multiple runs of the same workload, we used different types of programs: `gamess`, `bzip2` and `crafty` – long-running compute-intensive sequential applications; `kernbench` [4] – a kernel compilation benchmark that creates a mix of short-lived compute- and I/O-intensive single-threaded processes; `scp` – a sequential I/O intensive benchmark that transfers a 200MB file over the local network; and several multithreaded HPC programs (`semphy`, `kmeans`, `RNAseq`, `FFTW3D` and `EP`) with different scalability and synchronization patterns. We also experimented with `ebizzy` [4]–a micro-benchmark designed to generate a web server like workload (performance reported in terms of transactions per second)–, and with `schbench` –a benchmark [64] that measures the scheduler's tail (p99) latency.

In running these applications, we considered three homogeneous workload scenarios: *Full* or F –the total thread count (N_T) matches the number of CPUs (N_{CPUS})–, *Half* or H (N_T=N_{CPUS}/2), and *Double* or D (N_T=2·N_{CPUS}). For single-threaded programs we launched multiple simultaneous program instances to match the total thread count desired. Notably, for HPC compute-intensive parallel workloads, which are typically run with a total thread count that matches the number of CPUs, we experimented with full load only.

Figures 4.6a and 4.6b show the slowdown distribution of the various workloads under CAMPS, CFS and HMP on *2B-2S-QuickIA* and *4B-4S-Odroid*, respectively. For



(a) *2B-2S-QuickIA*

(b) *4B-4S-Odroid*

Figure 4.6: CFS vs. CAMPS on the Intel QuickIA (a) and HMP vs. CAMPS on the Odroid 4 XU board (b) when running the different benchmarks

each workload, we ran the program at least 20 times so as to capture the variance of the performance distribution under each scheduler. The slowdown is normalized with respect to the fastest run registered when running the program alone on the system.

We begin by discussing the results of the CFS scheduler (Figure 4.6a). In the H scenario, CFS provides worse performance (higher slowdown) and significantly higher variability than CAMPS across the board. This stems from the fact that CFS is asymmetry agnostic, and it randomly maps threads to any of the idle cores regardless of its type. Moreover, CFS tries to keep a thread running on the same CPU for as long as possible (to reduce the number of migrations), even if the thread is mapped to a small core. By contrast, CAMPS maximizes big core utilization so it maps all threads to big cores in this case (N_T=$N_{BigCores}$). As a result, it optimizes performance and evens out the slowdown (fairness).

In the F and D scenarios, both CFS and CAMPS evenly distribute threads across cores. The results, however, largely depend on the nature of the benchmark. For long-running compute-intensive programs CAMPS delivers repeatable completion times and similar unfairness values across runs, thanks in part to fairness-oriented thread swaps. By contrast, CFS exhibits a huge slowdown variability here; for instance, under the `gamess` (D) workload the slowdown ranges between 1.37x and 4.22x. That is because CFS does not make any effort to guarantee that threads make equal progress on an AMP, as discussed in Section 4.3.1. In fact, CFS may map an

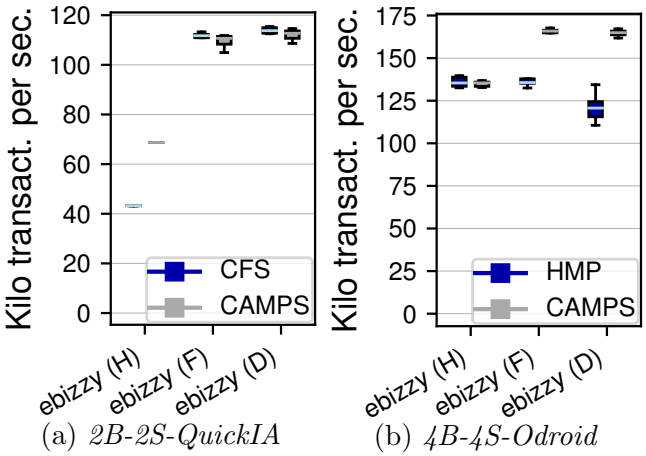
application to a big core in a whole run, and to a small core in another run. Our overarching conclusion is that for long-running compute-intensive workloads (like those considered in Section 4.4.4), the stock Linux scheduler does not constitute a good baseline for comparison due to its enormous performance variability. We should highlight that the root causes of this high variability, which we already discussed (i.e., a thread may be randomly mapped to any core type, no effort is made to accurately track and balance the progress on AMPs, etc.), are still present in the stock Linux scheduler implementation from kernel versions that are more recent than the one we used (v3.10.104). Despite the various changes made to the Linux scheduler in newer kernels, none of these changes was made to address these AMP-related issues. Note that we also conducted the same set of experiments with CFS on Linux v4.16.1 –the latest stable version available at the time these experiments were performed–, and observed the same huge performance variability.

For the `scp` benchmark, CAMPS and CFS yield similar slowdown figures in the F and D scenarios. We found that the (modest) variation in this case is not up to the OS scheduler itself (the benchmark is I/O intensive), but instead has to do with the disparities in the network bandwidth achieved by the different co-running instances of `scp`.

For the HPC workloads (last five groups of boxes in Figure 4.6a) both schedulers provide very similar slowdown (in a 2% range), with only one exception: the `semphy` program. This program goes through parallel phases and long-term serial execution phases, wherein the application exposes a single runnable thread to the OS. During sequential phases, CAMPS maps the single runnable thread to a big core, and as a result, it effectively mitigates this scalability bottleneck [83, 165]. CFS only accelerates serial phases in the event that the thread responsible from running serial code happened to be already mapped to a big core (by chance).

We now turn our attention to the results of the HMP scheduler on *4B-4S-Odroid* (Figure 4.6b). On this system, we could not gather the results for the `bzip` (D) and `kernbench` (D) workloads due to exceeding the platform's memory constraints (as discussed in Section 4.4.4.2). Unlike CAMPS, HMP is not designed to enforce system-wide fairness, but instead it extends CFS to provide a good trade-off between performance and energy consumptions for mobile workloads on ARM big.LITTLE systems. To this end, it devotes big cores to run compute-intensive code, and uses small cores for interactive or I/O intensive applications. While CAMPS exhibits a similar (low variability) profile as that observed on *2B-2S-QuickIA*, HMP does not behave exactly as CFS. In particular, in the H scenario, it nearly matches the slowdown distribution of CAMPS, as it uses big cores to run the various threads as soon as it detects they are going through a compute-intensive phase. In the F and D scenarios, HMP makes no effort to guarantee equal progress as opposed to CAMPS; HMP is subject to high variability, making it specially unsuitable to be considered as a baseline for comparison under long-running compute-intensive workloads.

For the I/O intensive workloads (`scp`), HMP delivers a smaller variance than our scheduling proposal (CAMPS was not optimized for I/O benchmarks), but it clearly delivers worse performance than CAMPS in the D scenario.



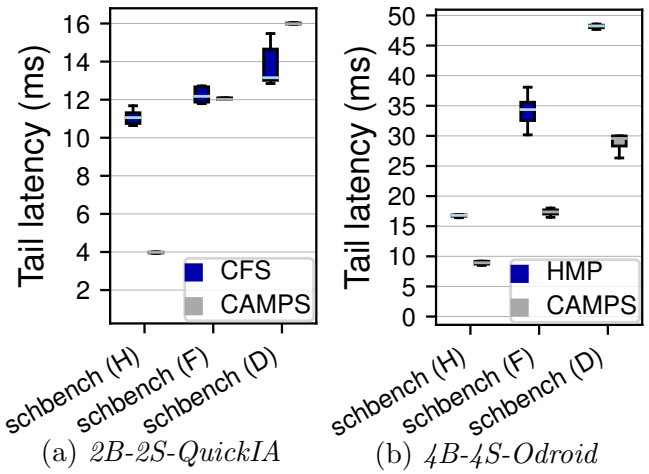
(a) *2B-2S-QuickIA* (b) *4B-4S-Odroid*

Figure 4.7: CFS vs. CAMPS on the Intel QuickIA (a) and HMP vs. CAMPS on the Odroid 4 XU board (b) when running the `ebizzy` benchmark

As for the multithreaded HPC programs (last five group of boxes), HMP yields very poor performance and is subject to large performance variability in some cases. Essentially, in these workloads HMP always maps all threads to big cores, thus introducing oversubscription (2 threads per big core) while leaving small cores idle. This mapping is very inappropriate in this context, as threads in some of these applications synchronize with each other frequently.

We now proceed to analyze the results of `ebizzy` on both platforms (Figures 4.7a and 4.7b). Since CAMPS performs no worse than HMP and CFS in the H scenario, we focus on the discussion of the remaining cases. Note that threads of this web-server-like application do not synchronize with each other, but instead attempt to complete as many requests as possible in parallel. In the F and D scenarios, CFS effectively utilizes both core types. The fact that threads do not make equal progress does not affect throughput; big-core and small-core threads do effective work, but at a different pace. CAMPS, which is not optimized for this workload type, delivers a slightly inferior throughput (1.6% less) than CFS under F and D. Because `ebizzy` threads are CPU bound, HMP assigns them all to big cores, leading to poor performance. CAMPS is able to outperform HMP by 21% and 37% on average in the F and D scenarios.

Finally, we examine the tail scheduler latency numbers (Figures 4.8a and 4.8b) obtained with multiple runs of `schbench`. To fully understand the results, it is worth recalling that CAMPS attempts to maximize big core utilization, and, to this end, it quickly moves threads to under-loaded big cores. In the H scenario, CAMPS maps all threads to the available big cores, while CFS and HMP may leave some threads running on small cores. Big core's superior performance translates into smaller latencies, thus enabling CAMPS to outperform the other schedulers: it achieves a 65% and 45% latency reduction vs. CFS and HMP, respectively. Our results also register more consistent tail latencies under CAMPS across different executions relative to CFS, but they also reveal a slightly worse tail latency (around 7% higher) under oversubscription (D). Lastly, we also observe that CAMPS's load balancing decisions in the F and D scenarios allow it to impressively reduce tail



(a) *2B-2S-QuickIA* (b) *4B-4S-Odroid*

Figure 4.8: CFS vs. CAMPS on the Intel QuickIA (a) and HMP vs. CAMPS on the Odroid 4 XU board (b) when running the `schbench` benchmark

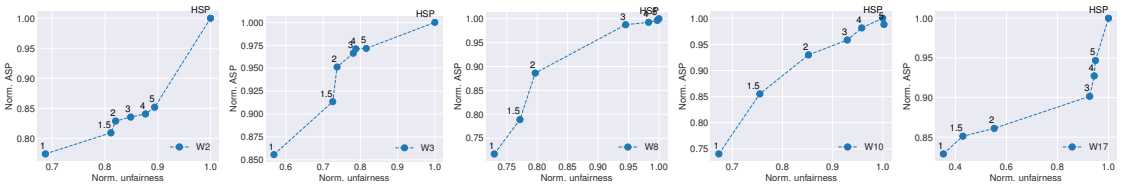


Figure 4.9: Normalized unfairness and throughput for different workloads and values of the UF parameter under ACFS

latencies (by up to 53%) w.r.t. HMP, which overloads big cores with compute-intensive threads.

4.4.3. Trading fairness for throughput

In previous work [167], ACFS's `unfairness_factor` was evaluated without considering the impact of cache and bus contention effects. We now assess the impact of varying the UF on fairness and throughput under ACFS and CAMPS running on the *2B-4S-Juno* AMP configuration, where applications may suffer performance degradation due to shared resource contention. For our experiment we selected those workloads considered in Section 5.1.1 where the HSP scheduler achieved considerably higher throughput than ACFS or CAMPS (see Figure 4.11 in the chapter). This way, as we gradually increase the unfairness factor under a particular workload, we would expect to see that the throughput figures provided by the CAMPS and ACFS schedulers get closer to those of HSP (as illustrated in [167] for ACFS).

Figures 4.9 and 4.10 show how the choice of the `UF` parameter (ranging between 1 to 5)[3] affects the unfairness and the system throughput for various workloads running on the *2B-4S-Juno* configuration under ACFS and CAMPS, respectively. The results reveal that throughput gains are obtained as we gradually increase the

[3]For most of the investigated workloads, increasing the UF value beyond 5 does not translate into noticeable throughput gains.

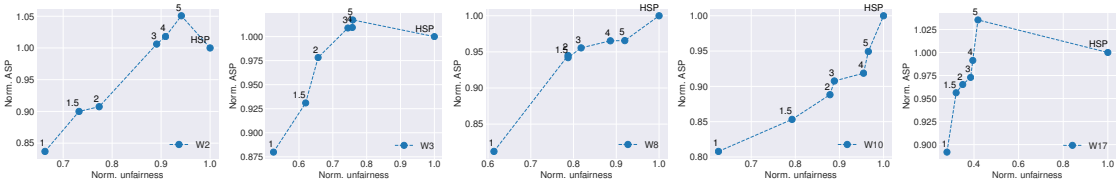


Figure 4.10: Normalized unfairness and throughput for different workloads and values of the UF parameter under CAMPS

value of the `unfairness_factor`; this also comes at the expense of increased fairness degradation. Moreover, as we increase the UF, the throughput and fairness figures of both schedulers get closer to those of the HSP scheme, which strives to optimize throughput.

We also observe that under a few workloads (W2, W3 and W17) CAMPS is capable to slightly outperform the HSP scheduler (up to 5% increase in throughput) when using high values of the UF parameter. This is clearly not the case under ACFS. To fully understand these results, it is worth recalling that HSP and ACFS do not deal with shared-resource contention effects. In the aforementioned program mixes, the applications with the highest big-to-small speedups in the workload are also subject to high contention-related performance degradation. We found that the HSP scheduler maps these contention-sensitive applications to big cores simultaneously for longer periods of time than CAMPS and ACFS when using the default setting of the `unfairness_factor` (1.0). As we increase the `unfairness_factor`, both fairness-aware schedulers grant a higher big-core share to high-speedup applications (the contention-sensitive programs in this case). Clearly, a modest increase of the big-core share of high-speedup memory-intensive applications results in higher performance for these application and, in turn, in throughput gains. However, increasing too much the amount of time that these applications spend on the big cores (what happens as a result of increasing the `unfairness_factor` beyond a certain point), lead high-speedup memory-intensive applications to be mapped simultaneously to big cores more often, which backfires by degrading the performance of individual applications. Unlike ACFS (contention unaware), CAMPS is able to deal with this issue effectively as it favors thread swaps that contribute to minimizing contention on the big core cluster. In other words, CAMPS reduces the amount of time that contention-sensitive high-speedup applications run together on the big core cluster. This aspect coupled with a high setting of the UF parameter (e.g. 5), which increases the big-core share allotted to high-speedup programs, enables CAMPS to obtain throughput improvements over a contention-unaware scheduler that strives to optimize throughput (HSP).

To sum up, our study has demonstrated that the `unfairness_factor` knob effectively enables us to improve system throughput in scenarios where fairness constraints are relaxed. Moreover, by using high values for this parameter as those used in our experiments, CAMPS can be configured to approximate the behavior of the HSP scheduler. In several cases, CAMPS is able to outperform this scheduler in both fairness and throughput. Hence, the main takeaway from our study is that CAMPS constitutes a versatile contention-aware scheme, as it enables the system

administrator to pursue two optimization goals with a single scheduling algorithm.

4.4.4. CAMPS vs. other asymmetry-aware schedulers

To assess the effectiveness of CAMPS we compared it with three previously-proposed fairness-aware schedulers for AMPs: ACFS [167], Equal-Progress [182] and an asymmetry-aware Round-Robin (RR) scheme [24]. We also experimented with a scheduler that attempts to optimize throughput by preferentially running on big cores those applications that derive a higher big-to-small speedup [111, 164]. We will refer to this scheduler as *HSP* (High SPeedup).

All the schedulers considered (except for RR) rely on performance monitoring counters (PMCs) to function. HSP and ACFS determine threads' SFs on-line by continuously monitoring different PMC events, and by feeding an estimation model with the obtained event counts. (More information on the mechanism employed to build the estimation model and to determine the associated events on our ARM-based experimental platforms can be found in [159].) The Equal-Progress scheduler [182], by contrast, leverages PIE [183] or IPC sampling [24] to determine thread SFs at runtime. Since the required hardware extensions for PIE are not available in commercial AMP hardware, we evaluate the history-based variant of Equal-Progress [182], based on IPC sampling. Under all schedulers, PMCs are sampled on a per-thread basis every 50ms; this sampling period enables the OS to detect coarse-grained program phases and to filter out many spikes in performance metrics that become apparent when using smaller sampling periods (due to fast oscillations in some metrics). Notably, we observed that the overhead associated with PMC-related processing at this rate becomes negligible for most applications (for a few programs we observed up to a 0.28% overhead). To reach a 1% overhead, the sampling period has to be reduced to a value as low as 5ms.

For our experiments, we used the *2B-4S-Juno* and *4B-4S-Odroid* AMP configurations presented in Section 3.1.1. Our evaluation targets workloads consisting of long-running compute-intensive benchmarks from diverse suites: SPEC CPU, PARSEC, Minebench and the NAS Parallel benchmarks. We also experimented with `FFTW3D` - a program performing the FFT. All programs were compiled with GCC (`-O3` switch) and by employing the `-mtune=cortex-a15.cortex-a7` (*2B-4S-Juno* only) and the `-mtune=cortex-a57.cortex-a53` (*2B-4S-Juno* only) compiler options to apply common big.LITTLE optimizations. The total thread count in each workload was set to match the total number of cores in the platform (including both big and little cores), as in previous work on AMPs [111, 164, 182]. We ensure that all applications in the mix are started simultaneously and when one of them terminates it is restarted repeatedly until the longest application in the set completes three times. We then measure unfairness and throughput, by using the geometric mean of the completion times for each program. To assess throughput we employed the *Aggregate Speedup* (ASP) metric as explained in Section 3.3.2. We ran each experiment five times, and report the average, minimum and maximum values of the unfairness and throughput in each case.

Table 4.3: Multi-application workloads for the *2B-4S-Juno* AMP configuration.

Name	Applications
W1	GemsFDTD,equake,soplex,milc,ammp,bzip2
W2	galgel,soplex,hmmer,lbm,fma3d,bzip2
W3	galgel,equake,gamess,lbm,bzip2,astar
W4	twolf,bwaves,equake,soplex,astar,gobmk
W5	GemsFDTD,bwaves,equake,povray,fma3d,astar
W6	bwaves,equake,gamess,lbm,fma3d,bzip2
W7	GemsFDTD,applu,perlbmk,sixtrack,astar,gzip
W8	bwaves,perlbmk,povray,fma3d,astar,gzip
W9	galgel,perlbmk,sixtrack,mgrid,astar,libquantum
W10	GemsFDTD,vortex,perlbmk,fma3d,astar,gzip
W11	bzip2,equake,hmmer,vortex,crafty,astar
W12	gamess,hmmer,soplex,art,astar,gzip
W13	GemsFDTD,bwaves,gamess,hmmer,crafty,astar
W14	bzip2,bwaves,hmmer,lucas,gobmk,gzip
W15	soplex,art,vortex,lbm,fma3d,gobmk
W16	galgel,equake,hmmer,lbm,fma3d,h264ref
W17	bwaves,equake,gamess,povray,astar,libquantum
W18	GemsFDTD,galgel,gamess,hmmer,astar,libquantum
W19	swim,mcf,perlbench,h264ref,gobmk,gzip
W20	galgel,equake,hmmer,povray,mgrid,gobmk
W21	galgel,equake,hmmer,bzip2,perlbench,h264ref
W22	galgel,equake,gamess,hmmer,sixtrack,povray
W23	gamess,art,bzip2,gobmk,sixtrack,vortex
W24	galgel,gamess,hmmer,povray,perlbench,gobmk

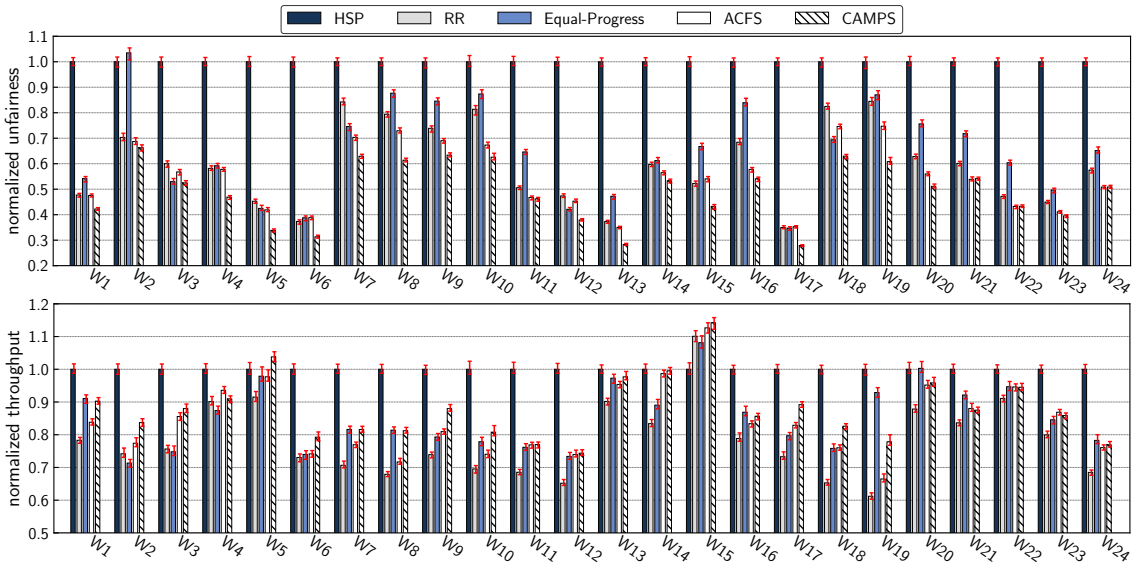


Figure 4.11: Unfairness (top) and throughput (bottom) for the workloads in Table 4.3 running on *2B-4S-Juno* under the various scheduling algorithms

In evaluating the various schedulers we built two different sets of workloads, shown in Tables 4.3 and 4.5. In the first one, each program mix is made up of six single-threaded applications running on the *2B-4S-Juno* configuration. The second set, which we ran on *2B-4S-Juno*, includes mixes consisting of both single-threaded and multithreaded programs.

4.4.4.1. Workloads for the 2B-4S configuration

We begin by analyzing the results of the first workload set, shown in Figure 4.11. The unfairness and throughput (ASP) values reported in the charts are normalized with respect to the results of the HSP scheduler. In building the workloads (Table 4.3), we divided SPEC CPU applications into two groups: *light-sharing* programs, whose performance does not suffer noticeably under contention; and *memory-intensive* programs, which are subject to high contention-related degradation or put significant pressure on the shared resources. We then generated 24 random program mixes by combining 29 SPEC benchmarks that cover a wide spectrum of speedup factors. Table 4.3 shows these program mixes which are displayed sorted in descending order by the number of memory-intensive programs included in the workload.

The results illustrate that optimizing one metric may lead to substantial degradation of the other one. This trend was also observed in previous work [167, 159], which illustrates that fairness and throughput are largely conflicting optimization goals on AMPs. As is evident, HSP, which strives to optimize throughput, achieves the best ASP values for most workloads, at the expense of the worst unfairness numbers (the higher, the worse) across the board. Conversely, the remaining schedulers (fairness aware), achieve substantial reductions in unfairness vs. HSP (up to a 72% reduction – CAMPS under W17), at the cost of potentially high throughput degradation (up

Table 4.4: Average reduction in unfairness and increase in throughput achieved by CAMPS over the other schemes on *2B-4S-Juno*.

CAMPS vs. others	**Reduct. in Unf.**	**Increase in throughput**
HSP	50.96%	-12.24%
RR	17.08%	13.19%
Equal-Progress	23.64%	3.31%
ACFS	10.71%	4.48%

to 38% – RR under W19).

The results of ACFS, RR and CAMPS exhibit a clear trend across the board. Specifically, for the vast majority of workloads ACFS delivers better throughput and higher reductions in unfairness than RR. This is the expected behavior since ACFS takes applications' big-to-small speedups into consideration when distributing big-core cycles among applications, whereas RR does not. Despite the higher throughput, the fact that ACFS does not factor in contention effects when making decisions, leads ACFS to similar unfairness figures to those of RR in some cases (e.g. W4-W6, W15 or W17). By contrast, our proposal is able to reduce unfairness even further: by up to 11% with respect to ACFS (W17) and by up to 28% relative to RR (W19). In addition, CAMPS is capable of reaping higher throughput gains: up to a 17% increase vs. ACFS (W19). Notably, under those workloads including a small number of memory-intensive applications (W20-W24), we observe that CAMPS and ACFS perform very similarly. This suggests that CAMPS is also suitable for low-contention scenarios, as it delivers similar unfairness and throughput figures to ACFS, the state-of-the-art fairness-aware scheme providing the best results under these circumstances [167]. All in all, as summarized in Table 4.4 CAMPS achieves an average 10.7% reduction in unfairness with respect to ACFS while improving throughput by 4.48%.

Now we zoom in on the results of the Equal-Progress scheme, which, as our proposal, also strives to optimize fairness. We observe that this scheduler is not able to obtain lower unfairness than CAMPS or ACFS for most workloads. More importantly, Equal-Progress's results reveal significant divergences across the board: for a few workloads, such as W4, W12 or W18, it obtains throughput and fairness figures closer to those of CAMPS and ACFS, whereas for others it exhibits a much unfairer behavior along with either throughput degradation relative to CAMPS (e.g. W2, W8-W10, W14, etc.) or with overall performance gains in some cases (e.g. W1, W19 or W20). As discussed in detail in [167], this somewhat inconsistent behavior of Equal-Progress stems from two main factors: (1) the inaccuracies associated with the mechanism it employs to track thread progress on AMPs, and (2) the fact that it relies on IPC sampling to determine thread's SFs online on commercial AMPs [182]. IPC sampling has been shown to lead to inaccurate SFs, since IPC values collected on each core type may belong to different program phases [172]. We observed that inaccuracies in the SF –obtained when measuring the IPC directly on both core types– are more frequent under contention, as the IPC may suffer profound oscillations (even within the same program phase) based on the degree of contention a thread is suffering. Inaccuracies prevent Equal-Progress from delivering even

progress across applications, and the performance it delivers is heavily affected by these inaccuracies: throughput increases when the scheduler happens to grant a higher big-core share to high-speedup applications. Although CAMPS also relies on measuring the IPC to determine a thread's slowdown, the reference values used to approximate run-alone performance (IPS_{alone}, stored in the history table for the different phases) are collected under low contention scenarios, as explained in Section 4.3. This makes it possible for CAMPS to overcome the aforementioned issue of Equal-Progress. On average, our proposal reduces unfairness by 23.6% compared to Equal-Progress.

The results also reveal that for some workloads CAMPS and ACFS achieve throughput values similar (in a 3% range) to those of HSP. Overall, we observe that the throughput degradation achieved by fairness-aware schedulers is significantly lower for workloads where the number of applications that experience a higher-than-average speedup exceeds the number of big cores (two). Under these circumstances (e.g. W1, W3, W13 or W17), CAMPS and ACFS grant a substantial amount of big core cycles to these specific applications (by triggering periodic swaps), whereas HSP usually maps only two high-speedup programs to big cores for a long time period. This leads ACFS and CAMPS to reduce unfairness in a greater extent than HSP (e.g. W13 and W5), while yielding a low throughput degradation.

Lastly, we should highlight that HSP is especially affected by contention effects under the W5 and W13-W15 workloads, where the two applications with the highest speedup (those listed at the beginning of each row in Table 4.3) are both highly memory intensive or constitute a pair consisting of a memory-intensive and a cache-contention sensitive program. The benefit that these applications derive from running on a big core comes in part due to the fact that this core type features a larger shared L2 cache than the small core. Unfortunately, when the scheduler maps two memory-bound programs on the big cores simultaneously, threads compete with each other for space in the shared cache as well as for bus bandwidth, which leads to non-negligible performance degradation for both applications, and in turn degrades system throughput. Specifically, under the aforementioned workloads, HSP maps memory-bound applications to big cores simultaneously for longer periods of time than fairness-aware schedulers, which –by contrast– swap threads between core types every so often. Swapping threads reduces the amount of time that the conflicting applications are mapped together to big cores, which contributes to improving both throughput and fairness. Specifically, the results reveal that all fairness-aware schedulers reap high normalized throughput figures under these program mixes (W5, W13-W15). More importantly, our proposal, is able to outperform HSP for some of these conflicting workloads (W5 and W15). This is possible thanks to the fact that CAMPS swaps threads based on their observed progress and by catering to the degree of contention.

4.4.4.2. Workloads for the 4B-4S-Odroid configuration

We now proceed with the discussion of the results for workloads we ran on the *4B-4S-Odroid* configuration. On this system, we attempted to analyse workload

Table 4.5: Multi-application workloads for the *2B-4S-Juno* AMP configuration.

Name	Applications
M1	art,galgel,libquantum,sixtrack,gamess,hmmer,soplex,gzip
M2	galgel,libquantum,hmmer,mcf,mgrid,crafty,parser,gzip
M3	mcf,lucas, galgel, soplex,h264ref,povray,perlbmk,gobmk
M4	galgel,mcf,h264ref,povray,perlbmk,crafty,gobmk,astar
M5	gamess,hmmer,mcf,mgrid,lucas,applu,namd,gobmk
M6	art,galgel,gamess,hmmer,mgrid,lucas,h264ref,astar
M7	hmmer,mcf,mgrid,lucas,soplex,applu,h264ref,gzip
M8	mgrid,lucas,soplex,fma3d,applu,ammp,h264ref,astar
M9	art,libquantum,sixtrack,h264ref,semphy(4)
M10	gamess,applu,povray,crafty,swaptions(4)
M11	soplex,povray,namd,gobmk,semphy(4)
M12	fma3d,h264ref,povray,equake,kmeans(4)
M13	FFTW3D(4),kmeans(4)
M14	semphy(4),EP(4)
M15	blackscholes(4),EP(4)
M16	blackscholes(4),kmeans(4)

scenarios with a wide diversity of SFs among applications and a varying degree of competition for the available big cores. Note that, in building the program mixes, we had to pay special attention to the aggregate memory footprint of the workload, which should not exceed the limited amount of physical memory available on the Odroid XU4 board (2GB) to prevent Linux's Out-of-Memory killer from kicking in during the experiments. Due to this constraint, we had to discard some application mixes for the different categories considered.

Overall, the workloads we explored –shown in Table 4.5– can be grouped in three broad categories. The first one combines 8 single-threaded programs (M1-M8) that exhibit a varying degree of memory intensity and cover a wide spectrum of SF values. Workloads in the second category (M9-M12) couple 4 single-threaded applications with a parallel program. Notably, the sequential programs derive a higher benefit from using a single big core most of the time than the multithreaded program. Catering to the amount of TLP (thread-level parallelism) in the application under these circumstances is crucial to identify those application phases that really benefit from using a handful of big cores (e.g. serial execution phases) [165, 164]. Finally, workloads in the third category (M13-M16) combine two parallel applications with different scalability features. Specifically, the `FFTW3D`, `semphy` and `blackscholes` programs exhibit sequential phases that span over 20% of their execution time, whereas `EP` and `kmeans` constitute highly parallel applications.

Figure 4.12 shows the results for workloads in Table 4.5. Despite the profound differences between the composition of these workloads and those evaluated on the *2B-4S-Juno* configuration, the results exhibit very similar trends to those discussed earlier. Essentially, CAMPS achieves the highest reduction in unfairness (up to 55%

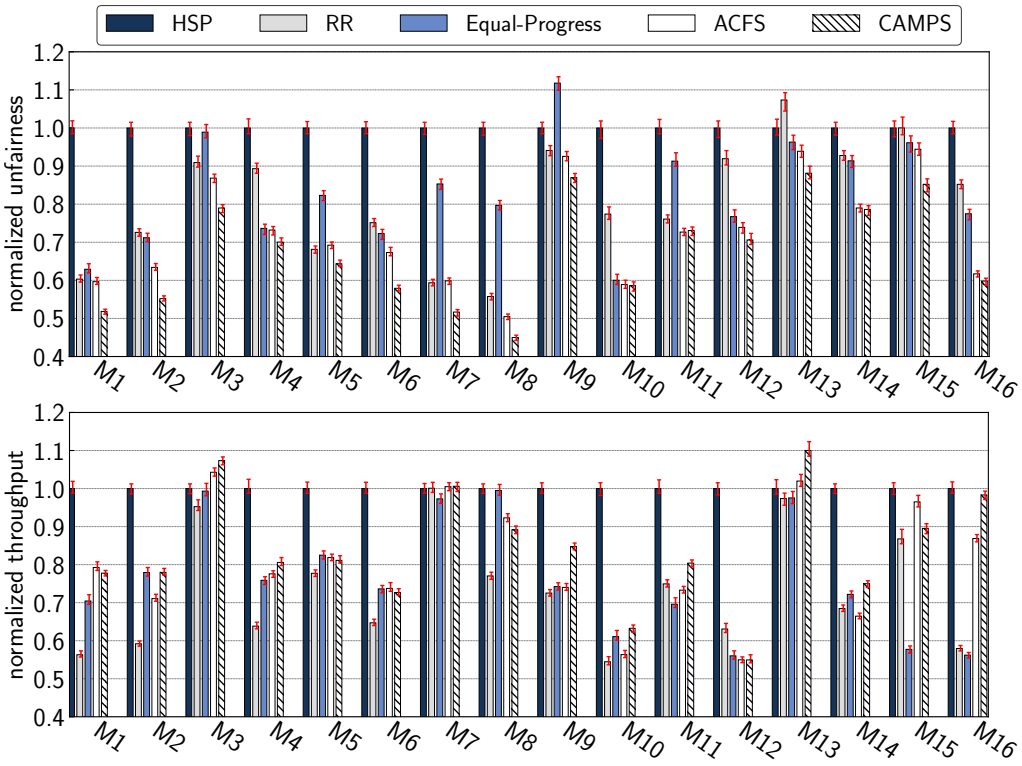


Figure 4.12: Unfairness (top) and throughput (bottom) for the workloads in Table 4.5 running on *4B-4S-Odroid* under the various scheduling algorithms

vs. HSP) for the vast majority of workloads. At the same time, ACFS is usually the scheme that provides closest fairness figures to those of CAMPS, followed by RR and Equal-Progress. Again, we observe that ensuring fairness comes at the expense of significant throughput degradation in some cases (up to 45% - M12).

Results in Table 4.2 indicate that CAMPS still achieves substantial average reductions in unfairness w.r.t. the other schemes on *4B-4S-Odroid* (32.7% w.r.t. HSP, and 7% relative to ACFS). These overall gains are slightly smaller than those achieved on *2B-4S-Juno* (see Table 4.4). This has to do with the lower degree of memory intensity of the workloads we ran on *4B-4S-Odroid*, which stems from the impossibility (due to the memory constraints) to consider mixes with multiple highly memory-intensive programs with a large memory footprint.

In spite of obtaining more modest fairness improvements in this scenario, CAMPS reaps considerably higher throughput gains relative to RR and Equal-Progress – over 16% and 12% respectively. This is due to the higher speedup diversity present in these program mixes, which stems from two factors. First, the SF range across sequential programs is significantly wider on this platform (from 1.36x to 6.63x) than on *2B-4S-Juno* (from 1.5x to 4.4x). RR does not take SFs into consideration when making scheduling decisions, thus failing to obtain decent throughput figures in this context. Second, some program mixes combine single-threaded programs, which derive non-negligible benefits from using a single big core, with multi-threaded programs that only derive significant benefits from big cores in the event that all of its active threads are mapped simultaneously to big cores for some time

(due to synchronization). Under these circumstances, devoting big cores to run low-TLP phases (e.g. serial code) brings higher benefits than mapping threads-to-cores based on the per-thread slowdown [83, 165, 167]. Unlike RR and Equal-Progress, the other schedulers (including CAMPS) take this aspect into consideration indirectly by downscaling the thread's slowdown factor (or speedup [164]) with the number of runnable threads in the application (a proxy for the amount of TLP), as stated in Section 4.3.5. Failing to cater to the amount of TLP in the application leads RR and Equal-Progress to high throughput degradation in some cases (e.g. over 40% degradation under M16).

4.5. Conclusions

In this chapter, we have proposed CAMPS, an OS-level fairness-aware scheduler for asymmetric single-ISA multicores. Unlike other fairness-conscious asymmetry-aware schemes [24, 182, 167], our approach effectively caters to the performance degradation that comes from contention on the shared resources among cores, such as the last-level cache or the memory bus. CAMPS accurately tracks the progress that the various threads in the workload make when running on the different core types throughout the execution, and enforces fairness by evening out the progress across threads.

CAMPS's progress tracking scheme relies on approximating the current slowdown of an application thread by comparing its actual performance with the performance observed in the past for the thread when it ran on a big core in a low contention scenario. In doing so, the scheduler factors in the contention-related performance degradation as well as the slowdown that the thread normally experiences when it is mapped to a small core rather than to a big one. Notably, our approach does not require special hardware extensions [183, 182] or platform-specific speedup-prediction models [111, 167] to function. Instead, CAMPS relies on the gathering of a set of performance metrics that can be easily measured online in commercial AMP hardware via performance counters. This makes the scheduler highly portable across different processor models and CPU architectures.

We implemented CAMPS in the Linux kernel and assessed its effectiveness on real asymmetric hardware. An extensive comparison was performed with other existing schemes that aim to optimize fairness [24, 182, 167]. Our experimental results reveal that CAMPS outperforms the state-of-the-art fairness-aware scheme for AMPs –the ACFS scheduler [167]– in both fairness and throughput.

Another important contribution of this work is the analysis provided in 4.4.2. The results illustrate the high variability delivered by CFS and HMP on AMPs for long-running compute-intensive workloads. Besides, these experiments reveal that CAMPS is more capable of handling various application types on AMPs than CFS and HMP.

Chapter 5

PBBCache: A parallel simulator for rapid prototyping and evaluation of cache-partitioning policies

Cache clustering constitutes a generalization of strict cache partitioning, where each partition can be shared by a group of applications (aka *cluster*). Partitioning the cache optimally for a certain optimization objective is an NP-hard problem [133], but determining the optimal cache-clustering solution adds a new level of complexity, as a decision must be made on how to best group applications into clusters, and how to optimally distribute cache space across clusters. Previous work has pointed out that cache clustering proves more effective than strict cache partitioning as the number of applications increases [171], specially on systems supporting a reduced number of coarse-grained cache partitions (i.e., in the order of megabytes). This is partly due to the finer grained distribution of the cache space that naturally results from sharing cache ways between applications.

To aid in the rapid design and evaluation of cache-clustering policies we have designed and implemented PBBCache, an open-source [68] parallel simulator that is described in this chapter. PBBCache relies on offline-collected application performance data (e.g., instructions per cycle, memory bandwidth consumption, etc.) to approximate the degree of throughput, fairness or other relevant metrics for a workload under a particular partitioning approach. Our simulator has been designed to achieve two main goals. Firstly, it is meant to serve as a tool for rapid prototyping and evaluation of partitioning policies. To increase programming productivity, PBBCache has been implemented in Python, which is one of the most widely used programming languages today [38]. Secondly, the simulator allows researchers to guide the design process of their algorithms and, more importantly, to easily discard unpromising approaches without having to go through the tedious development process in the system software. The simulator has been built to enable the assessment of the real potential of partitioning algorithms and to identify their limitations,

by providing a comparison with the optimal solution. Even though many cache-partitioning and clustering approaches have been proposed [171, 153, 58, 195], it still remains unclear how close they perform relative to the optimal solution. To fill this gap, PBBCache has the ability to efficiently determine the optimal solution for different optimization objectives.

In designing and implementing PBBCache we made the following main contributions:

- Our simulator is equipped with a slowdown-prediction model enabling to determine the performance degradation that an application suffers due to cache-sharing and memory-bandwidth contention. To approximate bandwidth contention for a certain distribution of cache space across applications in a workload, we extended the probabilistic model proposed in [136] to factor in how sensitive an application is with its effective bandwidth consumption at runtime.
- To determine the optimal cache space distribution in a reasonable amount of time, the simulator implements a novel parallel branch-and-bound (B&B) solver that effectively distributes the computation across cores on one or multiple computing nodes. This solver has been specifically designed for the optimization problems that arise in the context of cache partitioning. A key design aspect is the mechanism used to break down the work to be done in parallel into tasks (referred to as *subnodes*) with a similar computational complexity, which provides good scalability. The effectiveness of the bounding functions we devised for various optimization objectives, also contributes to the success of the B&B approach. These bounding functions are able to prune over 95% of the search space, for big problem sizes.
- To the best of our knowledge, our proposal is the first parallel approach to solve the optimal cache-partitioning problem by factoring in both cache-sharing and memory-bandwidth contention. Specifically, we studied two optimization objectives – fairness and throughput optimization, and found that each associated optimization problem can be expressed as a mixed-integer non-linear program. Notably, state-of-the-art non-linear solvers [6, 2, 1, 5] fail to provide a solution.
- To evaluate the effectiveness of PBBCache we implemented existing partitioning policies [153, 58, 195] on top of it, and compared the results it provides with the actual figures observed on commercial hardware equipped with cache partitioning support. Moreover, to assess the performance and scalability of the parallel B&B algorithm we conducted experiments using single-node and multi-node machine configurations.

The remainder of the chapter is organized as follows. Section 5.1 presents background on cache partitioning. Section 5.2 discusses related work. Section 5.3 introduces PBBCache's design. Afterwards, Section 5.4 formalizes the optimal cache-partitioning problem. Section 5.5 showcases our strategy to determine the optimal

cache-partitioning solution via parallel B&B. Finally, Section 5.6 covers the experimental evaluation, and Section 5.7 summarizes the chapter.

5.1. Background

In this section we formally introduce the problems of optimal cache partitioning and optimal cache clustering.

5.1.1. Optimal cache-partitioning problem

Before presenting a formal problem definition, it is worth describing how current cache-partitioning algorithms typically operate. For simplicity in the explanation we focus on way-partitioning, since this is the specific hardware implementation found in our experimental platforms (Intel CAT). Essentially, a partitioning algorithm has to distribute the available cache ways among applications based on their runtime properties so as to accomplish the objective it was designed to achieve (e.g., maximizing throughput, minimizing energy consumption, etc.). Notably, this kind of algorithms, as well as the corresponding optimization problem we describe next, just determine the number of ways that are allotted to each application, but not which exact ways are assigned. An important challenge is that certain applications may go through different program phases, which may lead to time-changing cache behavior. Under these circumstances, partitioning the cache statically (i.e. fixed cache way distribution throughout the execution) may not constitute the best solution. To cope with application's phase changes, partitioning algorithms are usually invoked periodically [171, 153, 58]. Specifically, the system software continuously monitors application behavior by using (for example) performance counters; the more recent values of the gathered performance metrics are used as input to the partitioning algorithm (invoked every so often) so as to determine the partitioning for the next execution interval, where the applications are likely to exhibit a similar behavior to that reflected by the recent collected data.

Our goal is to determine the optimal cache partitioning for a workload in a certain execution interval where the applications exhibit a stable behavior Researchers can use the optimal solution generated off-line with our simulator for a certain optimization objective to quickly assess the effectiveness of their algorithms. Determining an optimal solution for a workload considering program phases is a much more complex problem, especially because phase transitions do not happen at the same time in multiple applications. That would require to break down the whole workload execution into stages of stable behavior across applications, and apply our proposed method to detect the optimal solution for each and every "stable stage". Making a comparison with multiple application mixes enables to quickly assess the real effectiveness of a partitioning algorithm. More importantly, this allows to identify potentially conflicting workload scenarios where the algorithm fails to achieve good results, thus providing valuable insights to guide the algorithm's design process.

The optimal cache-partitioning problem can be generally formulated as a Mixed Integer Program (MIP). Let A be a workload consisting of N applications $\{a_1, a_2, \cdots, a_N\}$ that run on a system featuring a W-way last-level cache with $W \geq N$, and let K be $\{1..W\}$. The set DV of decision variables is defined as $\{w_{a,k} \mid \forall a \in A, k \in K\}$, where each decision variable $w_{a,k}$ is a binary variable indicating whether an application $a \in A$ is assigned k ways or not. The associated MIP is formulated by considering a generic optimization function f, as follows:

$$\textit{Minimize: } f(DV) \tag{5.1}$$

Subject to:

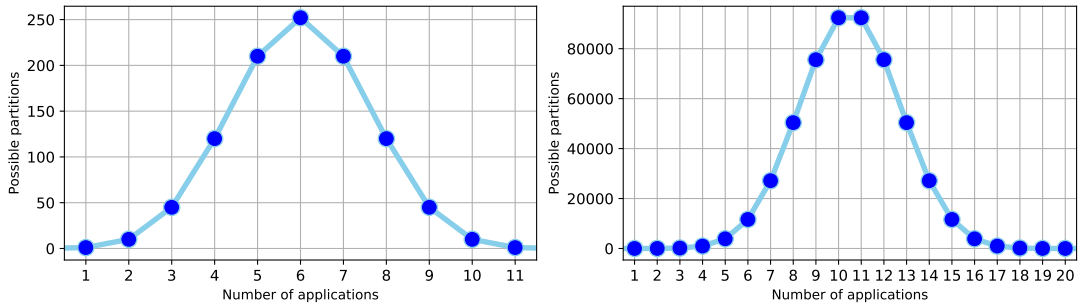
$$\sum_{k \in K} w_{a,k} = 1, \forall a \in A \quad \textit{Only 1 way assignment per application} \tag{5.2}$$

$$\sum_{a \in A} \sum_{k \in K} k \cdot w_{a,k} = W \quad \textit{No cache ways remain unused} \tag{5.3}$$

$$1 \leq \sum_{k \in K} k \cdot w_{a,k} \leq W - N + 1, \forall a \in A \quad \textit{Each application gets at least 1 way} \tag{5.4}$$

In this work we focus on two specific optimization problems, whose definition entail incorporating non-linear constraints to the set of Equations 5.1 to 5.4. The first problem, referred to as *Opt-STP*, is system throughput optimization, and it comes down to maximizing the STP metric. The second one, denoted as *Opt-Unf*, strives to find the cache space distribution that minimizes the Unfairness metric, so as to optimize system-wide fairness. As stated in Section 3.3, both the STP and Unfairness metrics depend on the slowdown experienced by each application; the Slowdown in turn depends on the cache-way distribution (decision variables in the generic MIP). Notably, *Opt-STP* and *Opt-Unf* constitute non-linear optimization problems. This stems from the fact that evaluating either optimization function (STP or Unfairness) for any feasible cache-way distribution requires determining the slowdown of each application by means of a prediction model that factors in the combined performance degradation due to cache and bandwidth contention. In our proposed model, this entails solving a set of non-linear equations, as we describe in Section 5.3.2. The detailed formalization of *Opt-STP* and *Opt-Unf* as Mixed Integer Non-Linear Problems (MINLPs) can be found in Section 5.4. Notably, we created AMPL implementations [1] for these problems and tested them with different state-of-the-art non-linear solvers [6, 2, 1, 5], but found that all of them failed to provide a solution. More importantly, even for small workloads – where PBBCache finds the optimal solution sequentially in less than one minute – they also failed to find a local minima after 1 hour of execution. We also tried feeding the solvers with an initial solution determined earlier via a heuristic algorithm, which was unable to provide the optimal solution for the workloads considered. In this case, the solvers were not even able to find a better solution in one hour of execution (we configured the solvers so that the execution was aborted automatically if a

[1] *A Mathematical Programming Language*, is a modelling language used to formalize optimization problems and implement highly complex mathematical algorithms.



(a) 11 ways (b) 20 ways

Figure 5.1: Number of possible ways to partition a LLC as we increase the number of applications for 11 and 20 cache ways, respectively. The number of ways and applications considered are based on the features of Platforms A and B, described in Section 3

near-optimal solution was not found within that time period). By contrast, for the largest workloads we explored on our experimental platforms (see Section 5.6), our simulator is capable of finding an optimal solution in less than 9 minutes by using a sequential algorithm, and in less than 34 seconds by leveraging a parallel B&B strategy on a 28-core server platform.

For the sake of completeness, the following recursive definition provides the number of possible ways P to partition a W-way LLC for N applications:

$$P(W,N) = \begin{cases} 1 & W = N \text{ or } N = 1 \\ N & W = N+1 \\ \sum_{i=1}^{W-N+1} P(W-i, N-1) & otherwise \end{cases} \tag{5.5}$$

As shown in Figure 5.1, the P function reaches the maximum when $N = \lceil W/2 \rceil$. The number of possible ways to partition the LLC rapidly increases with the application count (N), but it drops back symmetrically towards 1 when $N > \lceil W/2 \rceil$. Due to the vast search space when W is high, determining the best solution via extensive exploration becomes largely impractical.

5.1.2. Optimal cache-clustering problem

Optimal cache clustering constitutes a generalization of the optimal cache-partitioning problem. When cache clustering is used, applications in the workload are grouped into a number of sets, each one referred to as a *cluster*; the cache space is divided into separate partitions, one for each cluster. So, applications in the same cluster share the same cache partition.

To formally define the optimal cache clustering for a certain workload A consisting of N applications, we first introduce some basic terminology. We use the term *cluster set* to refer to any possible way to break down A into *clusters* so that the available cache space (W ways) can then be distributed across clusters (i.e. each cluster gets at least one way). A cluster set CS is one of the possible partitions

of the A set (i.e. grouping of the set's elements into non-empty subsets) with a number of items $\leq W$. Let $\mathbb{C}_A$ be the set with all possible cluster sets of A. Note that $|\mathbb{C}_A| \leq \mathcal{B}_N$, where $\mathcal{B}_N$ denotes the Bell number, namely the number of possible partitions of the A set. Let f be the objective function to be minimized in the optimization problem. We define $OPT_{CS,f}$ for a certain cluster set $CS \in \mathbb{C}_A$, as the value of f associated with the distribution of cache ways across clusters in CS that minimizes f.

For workload A and a certain optimization function f, we define the optimal cache clustering as the cluster set in $\mathbb{C}_A$ that exhibits the optimal (minimal) $OPT_{CS,f}$ value. A potential way to determine the solution to the optimal cache-clustering problem is to generate $\mathbb{C}_A$ and then identify the optimal solution by comparing the $OPT_{CS_i,f}$ values for each $CS_i \in \mathbb{C}_A$. In turn, determining $OPT_{CS_i,f}$ for any CS_i constitutes an instance of the optimal cache-partitioning problem, where cache space is distributed among clusters rather than among applications. Given the non-linear nature of Opt-STP and Opt-Unf, determining the optimal cache clustering for the throughput and fairness optimization objectives is also a non-linear problem.

5.2. Related Work

In discussing related work we first consider partitioning policies. Next, we cover previous research on parallel B&B.

5.2.1. Cache-partitioning and cache-clustering policies

A large body of work has studied the cache-partitioning problem and proposed different approaches to determine promising solutions using approximate algorithms [153, 58, 195, 137]. A recent survey [133] discusses the most effective approaches for various optimization objectives, such as maximizing throughput or reducing energy consumption. Specifically, in our simulator we implemented the UCP [153] and Yu-Petrov [195] algorithms – described in Section 5.3.4, which primarily strive to optimize system throughput. More recently, different cache-clustering algorithms have been proposed [171, 58]. We implemented the KPart [58] in PBBCache. This policy is also described in detail in Section 5.3.4. We should highlight that none of these works [171, 153, 58, 195] provide an explicit comparison of the corresponding proposal with a reference optimal cache-partitioning solution.

5.2.2. Parallel Branch-and-Bound

The B&B method constitutes a classical approach to solve combinatorial optimization problems. It can be considered a search space enumeration that explores a subset of feasible solutions. The effectiveness of an implementation of the B&B method for a specific problem largely depends on several design aspects, such as

Algorithm 5.1: Sequential B&B algorithm. (variant of the one defined in [50]).

```
1  incumbent ← initial solution provided by a heuristic algorithm
2  upper_bound ← incumbent.Cost();
3  prio_q ← [root node];
4  while prio_q is not empty do
5      node ← pop highest priority node from prio_q
6      foreach child of node do
7          if child.isSolution() && child.Cost() < upper_bound then
8              (incumbent, upper_bound) ← (child, child.Cost());
9              Remove nodes from prio_q whose cost > upper_bound;
10         else
11             lower_bound ← bounding_function(child);
12             child.setLowerBound(lower_bound);
13             if child is a feasible node && lower_bound < upper_bound then
14                 prio_q.append(child);
15             end
16         end
17     end
18 end
```

the bounding function used for pruning, the rule to select the next node to be processed, or the mechanism to determine the initial solution [72]. In minimization problems, the bounding function returns a *lower bound* of the cost (i.e. value of the optimization function used) of the best solution reachable from a specific node. B&B algorithms for minimization problems maintain a variable with the cost of the best solution found thus far (aka incumbent), which is an *upper bound* of the cost of the optimal solution. Conversely, in maximization problems, the bounding function returns an *upper bound* of the cost of the best solution reachable from a node, and a variable is used to maintain a *lower bound* of the cost of the optimal solution.

As an illustrative example, Algorithm 5.1 depicts the pseudo-code of a sequential implementation of the B&B method for a minimization problem. A heuristic algorithm is used to determine the initial solution, which is used for the initialization of the *upper bound* (lines 1-3). A priority queue is used to keep nodes as they are expanded during the search (line 14). Specifically, the node in the queue with the smallest lower bound is processed first.

An important contribution of this work is the parallel B&B implementation that our simulator leverages to determine the optimal cache-partitioning solution. As we explain in Section 5.5, it is a distributed-memory strategy that follows a master-slave pattern. The parallelization of B&B has been widely studied [72, 50, 55]. There are critical implementation challenges that must be faced [50, 55], such as: the initial definition of the search space, the choice of the work allocation policies, the communication of key information between processes, the minimization of idleness and the maximization of useful work. Our parallel strategy has been carefully designed to cope with many of these challenges that also arise in context of the Opt-STP and Opt-Unf non-linear problems. To the best of our knowledge, ours constitutes the first attempt to efficiently solve the optimal cache-partitioning problem via parallel

B&B when factoring in both cache-sharing and memory-bandwidth contention.

Crainic et al. [50] group parallel B&B algorithms into two main categories: *tree-based* and *node-based* strategies. Algorithms in the first category aim to build and explore the search space tree in parallel. By contrast, *node-based* approaches aim to accelerate a particular operation mainly at the node level, such as evaluation or bounding. The vast majority of the proposed parallel B&B algorithms exploit a *tree-based* strategy or a combination of node and tree parallelization [73, 148, 82, 118, 131]. Many frameworks have also been proposed to simplify the development of parallel B&B algorithms [50]. A well-known example is Bobpp [82, 131, 67], which allows the implementation of combinatorial problem solvers on both shared and distributed-memory architectures. Notably, the use of a framework is not always the best approach when dealing with a specific problem since custom solutions can take into account some problem-specific characteristics that contribute to improve performance [82]. Indeed, our implementations feature specific optimizations tailored to the Opt-Unf and Opt-STP problems. A key aspect of our approach is the fact that it considers *subnodes* as the work unit. As we explain in Section 5.5.3, the subnode abstraction allows us to break down the associated processing of a single node into tasks with similar computational complexity. This and many other design aspects of our strategy are motivated by the high computational cost associated with evaluating the bounding and optimization functions, which require solving a set of non-linear equations.

5.3. Design of the PBBCache simulator

In this section we provide an overview of the simulator design. We begin by introducing the structure of the simulator's input data and the interaction with it via the command line. Then, we describe the technique used to approximate the slowdown for each application in a workload based on the amount of cache space allotted and the degree of memory-bandwidth contention. Finally, we outline the partitioning algorithms implemented in our cache-partitioning simulator, and showcase some implementation details.

5.3.1. Input data and command-line options

PBBCache is a command-line tool. As depicted in Figure 5.2 it accepts as input two text files: a *workloads* file and a *metrics* file. The *workloads* file specifies the composition of the workloads that will be used in the simulation; each workload (one per line) is encoded as a comma-separated list of application names. The *metrics* file stores a table, where each row contains the values of various runtime metrics (e.g. instructions per cycle, cache miss rates on different cache levels, memory bandwidth consumption, memory-related stall cycles, etc.) for a specific application, which have been gathered offline with PMCs when the application runs alone on a certain platform with a fixed number of cache ways. Essentially, this file summarizes the

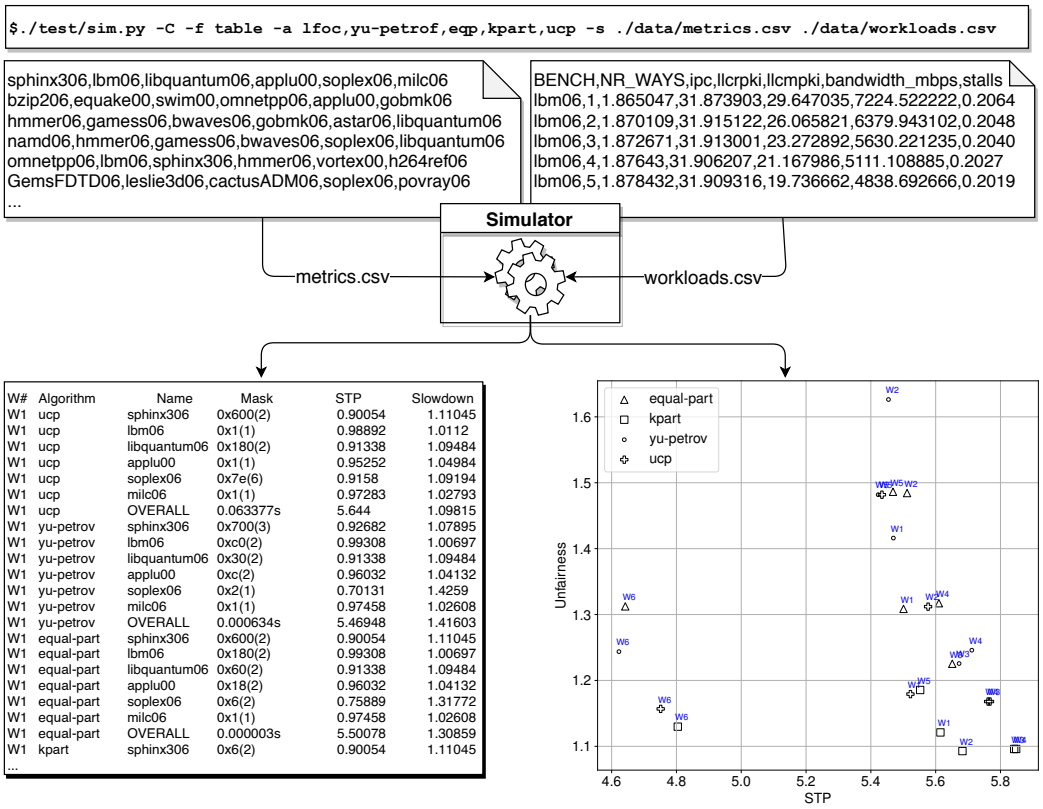


Figure 5.2: Simulator's diagram that shows an example of the user interaction with the simulator via command line, the data input and the generated output.

behavior of each application with every possible cache way count. The various metrics, which can be easily gathered on Intel processors that support Intel CAT, are used as input to different partitioning algorithms (as discussed in Section 5.3.4 each algorithm uses different metrics), and are also required to determine both the slowdown (see Section 5.3.2) and the amount of cache space each application gets inside a cluster (see Section 5.3.3). In creating the *metrics* file, the user may decide to include the information only for a particular program stage (e.g. first K billion instructions), a specific execution phase or the average registered for each metric throughout the application's execution.

The simulator text output, which can be presented in different formats (`-f` option), shows the amount of cache ways allotted by each partitioning algorithm considered for the simulation (as indicated with `-a`) as well as other values, such as the per-application slowdown. In using the sample command of Figure 5.2, where the `-C` option is used, a chart is also generated making it possible to compare the effectiveness of the various algorithms regarding fairness and throughput at first glance[2]. Simulations are performed sequentially by default, but because they are independent from one another they can be also launched in parallel (when the `-P` option is provided) by leveraging multiple slave processes running on one or

[2]Because the value of the unfairness metric only depends on the maximum and minimum slowdown observed across applications, reporting the value of the STP metric as well is crucial to properly assess the effectiveness of a partitioning approach.

multiple machines; details on our parallel programming framework can be found in Section 5.3.5. A complete discussion of PBBCache's command-line options can be found in [68].

5.3.2. Determining the slowdown under cache-partitioning

For each workload and partitioning algorithm indicated by the user in the command line, PBBCache determines the slowdown of each application in a workload, which is necessary to assess the degree of fairness and throughput delivered. Each partitioning algorithm decides how the various LLC cache ways are distributed among applications in a workload. The cache space distribution has an important impact on performance, but also determines the level of bandwidth contention present on the system, which may lead to performance degradation. Therefore, to accurately determine the slowdown of each application both the allotted cache ways and the degree of bandwidth contention should be considered. Specifically, let A be a workload consisting of N applications $[a_1, \cdots, a_N]$ running on a system that features a W-way last-level cache, and under a certain cache-partitioning algorithm *part*. Our simulator approximates the slowdown of each application a_i as follows:

$$Slowdown_{a_i} = SC_{\mathrm{part,a_i}} \cdot SB_{\mathrm{part,a_i}} \tag{5.6}$$

where $SC_{\mathrm{part,a_i}}$ indicates how much the application slows down due to the amount of cache space granted by *part* to it (w_i ways); $SB_{\mathrm{part,a_i}}$ is the slowdown that a_i suffers exclusively due to bandwidth contention (see Section 5.3.2.1).

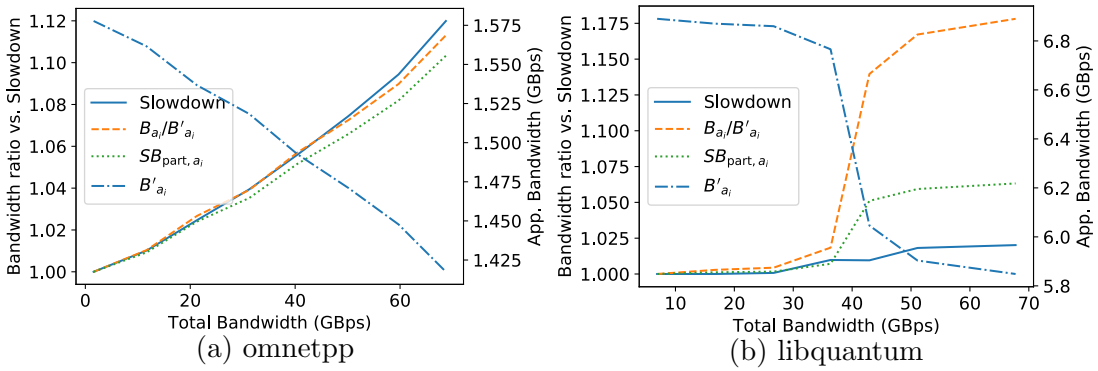
PBBCache approximates $SC_{\mathrm{part,a_i}}$ with the ratio of instructions per cycle (IPC) observed for a_i when using W and w_i ways:

$$SC_{\mathrm{part,a_i}} = \frac{\mathrm{IPC}_{a_i}(W)}{\mathrm{IPC}_{a_i}(w_i)} \tag{5.7}$$

Determining $SB_{\mathrm{part,a_i}}$ is a more challenging task for two reasons. First, the amount of memory bandwidth consumed by a_i at runtime depends on w_i and on the bandwidth consumption of the remaining programs [136]. So, the behavior of each application under the cache-way distribution made by *part* has to be taken into consideration to determine $SB_{\mathrm{part,a_i}}$. Second, we found that this performance degradation (slowdown factor) depends on how sensitive the application is to bandwidth contention. We now proceed to describe the bandwidth model that PBBCache leverages to approximate $SB_{\mathrm{part,a_i}}$.

5.3.2.1. Modeling Memory Bandwidth Contention

To illustrate the effects of bandwidth contention on our experimental platforms, we conducted several experiments where an application runs simultaneously with an increasing number of aggressor benchmarks; for the aggressor we used the bandwidth benchmark also used in Section 4.1.2. For each application we measured how



(a) omnetpp (b) libquantum

Figure 5.3: Memory bandwidth vs. slowdown observed for `omnetpp` and `libquantum` as increasing the total memory bandwidth consumption. The slowdown prediction provided by Morad's model (B_{a_i}/B'_{a_i}) and PBBCache's model (SB_{part,a_i}) is also reported.

its bandwidth and slowdown – w.r.t. the solo execution – varies as we add more instances of the aggressor application (increasing the total bandwidth consumption of the workload). To effectively track the slowdown that comes primarily from memory bandwidth contention in the experiments, we assigned separate cache partitions for the application under study and for the aggressors. Figure 5.3 shows the results for two SPEC CPU2006 applications gathered on the *Skylake* platform (which is described in Chapter 3). Note that B_{a_i} denotes the bandwidth consumption of the application when it runs alone on the platform (constant), and B'_{a_i} represents its actual bandwidth when running with the remaining applications in the workload.

As shown in Figure 5.3, B' drops as we increase the total bandwidth consumption, whereas the slowdown increases. Clearly, the observed slowdown is not negligible (e.g., up to 1.12x for `omnetpp`), so bandwidth-contention related degradation must be factored in to accurately determine the slowdown for individual applications. Notably, on the *Broadwell-EP* platform, where we also conducted the same experiments we observed considerably higher slowdowns due to bandwidth contention (up to 1.7x). As pointed out in Section 5.6.2, *Broadwell-EP* has roughly half the available bandwidth of *Skylake*, hence the larger observed slowdowns.

To properly account for bandwidth contention effects, PBBCache employs a variant of the probabilistic model proposed by Morad et al. [136]. Essentially, this model enables to approximate – from offline-collected information of individual applications running alone – (i) the bandwidth that each application would exhibit when running simultaneously with others and (ii) the slowdown that comes exclusively from memory-bandwidth contention. Essentially to determine (i) for a workload consisting of N applications, the following system of $N+1$ non-linear equations must be solved:

$$\left\{ \overline{B'}_{a_i}^2 \cdot \left(1 - \frac{1}{\overline{B}_{a_i}}\right) + \overline{B'}_{a_i} \cdot \left(1 - \frac{1}{\overline{T}}\right) \cdot \left(1 - \frac{1}{\overline{B}_{a_i}}\right) + 1 - \frac{1}{\overline{T}} = 0 \right\}_{i=1}^{N} \tag{5.8}$$

$$\sum_{i=1}^{N} \overline{B'}_{a_i} = \overline{T} \tag{5.9}$$

where $\overline{B}_{a_i}$ is the bandwidth observed for each application a_i when running alone on the platform (with the same amount of cache space as that allotted in the workload), $\overline{B'}_{a_i}$ is the actual bandwidth for a_i when running simultaneously with the other applications in the workload, and $\overline{T}$ is the total bandwidth consumption of the workload. Note that $\overline{B}_{a_i}$, $\overline{B}'_{a,i}$ and $\overline{T}$ are normalized to the maximum memory bandwidth of the platform.

To determine the application slowdown due to bandwidth contention (ii) Morad's model uses the ratio $\overline{B}_{a_i}/\overline{B'}_{a,i}$, which is based on the observation that the application bandwidth consumption and its performance naturally decreases due to contention, and so does its performance. We observed that this approach to approximate the slowdown is accurate for highly bandwidth-intensive applications (i.e., over 90% of its stall cycles are dominated by long-latency demand cache misses) such as `omnetpp` (see Figure 5.3a). However, for the remaining applications, the reduction in memory bandwidth consumption does not correlate linearly with the performance degradation, thus obtaining inaccurate slowdown estimates with the model (such as on Figure 5.3b). Essentially, some applications can generate a lot of prefetching-related memory requests and cache write-back operations, which may result in high memory bandwidth consumption; however, a reduction of the bandwidth consumption of these applications due to contention does not directly translate into linear performance degradation. To mitigate this issue in calculating the slowdown of an application a_i, PBBCache factors in the stall cycles due to long-latency demand cache cycles (MS_{a_i}) and the total number of stall cycles (TS_{a_i}) observed in the solo execution as follows:

$$SB_{part,a_i} = \frac{TS_{a_i} + MS_{a_i} \cdot \left(\frac{\overline{B}_{a_i}}{\overline{B'}_{a_i}} - 1\right)}{TS_{a_i}} \tag{5.10}$$

5.3.3. Determining the slowdown for cache-clustering policies

As explained in Section 5.1.2, cache-clustering policies group applications into clusters; each cluster is assigned a separate cache partition with a certain size. To apply the slowdown prediction model presented in Section 5.3.2 (Equations 5.6-5.10), the simulator must determine first how much cache space each application gets inside the assigned cluster. This is a challenging task, as the effective cache space an application gets largely depends on its co-runners in the cluster [205].

Caches in modern processors typically implement a variant of the pseudo-LRU replacement policy [205, 25][3]. Under these circumstances, the amount of cache space that an application gets is usually proportional to its rate of demand (frequency of cache misses) in competition with the rate of demand of the co-runners [153]. Based on this observation Mukkara et al. [137] proposed a simple model to rapidly estimate the effect in the cache miss rate curves when several applications share a cache, and

[3]The last-level cache, however, may incorporate specific optimizations to increase effective associativity without adding ways [25].

to approximate the cache space that each application would get for different way counts. The KPart clustering policy relies on this model [58], which leverages per-application MPKI (cache Misses Per Kilo-Instruction) tables. Essentially, for each number of cache ways the model determines the fraction of that cache space that will be assigned to each application. Intuitively, the higher the MPKI of an application for a certain way count, the more space the application gets when sharing a portion of cache with that number of ways.

By comparing the prediction provided by Mukkara's model with the actual cache usage reported by the Intel Cache Monitoring Technology on our experimental platforms, we observed that using the MPKI for the cache space prediction may lead to substantial inaccuracies that stem from the fact that the MPKI is not a good proxy of the rate of cache demand. Specifically, two applications with the same MPKI value but different performance (IPC) have a different rate of cache demand (in terms of misses per cycle). The application with the higher IPC in this case has a higher rate of demand, and so it typically obtains more cache space.

To overcome this shortcoming, our simulator employs a variant of Mukkara's model that uses MPKC (Misses Per Kilo Cycles) tables instead of MPKI tables. We refer to this model as the *cache-space* model. Note that the predicted amount of cache space for an application in a cluster may not be a multiple of the way size (e.g. 1.5 ways). In this case, linear interpolation (as in [58, 137]) is used to determine the value of the different metrics (i.e. IPC_{a_i}, $\overline{B}_{a_i}$, TS_{a_i} and MS_{a_i}) required to apply our slowdown-prediction model. So for example, if an application is expected to receive 1.5 ways of cache space inside a certain cluster, the value of each metric would be obtained via linear interpolation based on the corresponding metric values for 1 and 2 ways, which can be found in the *metrics* file.

5.3.4. Partitioning policies

The initial version of our PBBCache simulator implemented four cache-partitioning schemes: Equal-Part, UCP [153], Yu-Petrov [195], KPart [58]. PBBCache employs the offline-collected metrics found in the *metrics* file as input to each partitioning algorithm. Note that real implementations in the system software of most of these approximate algorithms may gather runtime application metrics online using performance counters.

The **Equal-Part** approach is a naive approach used only for comparison purposes that assigns all applications a separate partition with the same size. As we prove later on in Section 5.6, this partitioning policy does not gauarantee fairness or throughput on the system, since it fails to cater to the specific degree of cache sensitivity that each application shows.

UCP aims at improving fairness and overall throughput by minimizing the total number of misses incurred by all applications in the workload on the shared last-level cache. UCP does not attempt to determine the optimal solution but instead employs an approximate algorithm referred to as *lookahead* [153], which uses as input the

Algorithm 5.2: Pseudo-code of the lookahead partitioning algorithm

Input: *apps* represents the set applications that will receive a cache partition and *nr_ways* is the number of ways of the LLC available to assign.

```
1  function lookahead(apps,nr_ways)
2  remaining = nr_ways;
3  while remaining > 0 do
4  |   for (app, i) in apps do
5  |   |   alloc = allocations[i];
6  |   |   max_mu[i] = get the max marginal utility by estimating for every alloc ;
7  |   |   ways_req[i] = min ways required to get max_mu[i] for the app;
8  |   end
9  |   winner = application with greater max_mu;
10 |   allocations[winner]+ = ways_req[winner];
11 |   remaining− = ways_req[winner];
12 end
13 return allocations
```

MPKI table of each application. This table stores the application's MPKI value for any possible cache size. *Lookahead* behaves in a similar way to a greedy algorithm, it iteratively selects the best next way assignment based on their *marginal utility*; an estimation of how beneficial two different assignments are for a specific application that is based on the number of misses they incur. A pseudo-code of the original algorithm can be observed in Algorithm 5.2.

The algorithm proposed by **Yu and Petrov** [195] strives to reduce system bandwidth pressure. To this end, it partitions the LLC so as to minimize the total bandwidth. The algorithm relies on per-application bandwidth consumption measurements with different cache sizes gathered offline.

KPart [58] constitutes a cache-clustering approach designed for throughput optimization. KPart implements an iterative algorithm that creates and merges application clusters via hierarchical clustering. To decide which clusters must be merged on each iteration of the loop and how to distribute the available ways among clusters (inter-cluster way-partitioning), the scheme leverages the distance metric proposed in [137] as well as the UCP's *lookahead* algorithm [153]. The application of UCP and the evaluation of the distance metric relies on the ability to determine MPKI tables and IPC tables (i.e. number of Instructions Per Cycle for different cache sizes) online for each application.

5.3.5. Notes on the simulator implementation

PBBCache has been completely implemented in Python and relies on libraries available for multiple operating systems, hence, it is a multi-platform tool.

To leverage parallelism in the simulator implementation we use *ipyparallel* [10]. This framework, which relies on Python's *multiprocessing* module, enables us to perform master-slave distributed-memory parallel processing. The main features of ipyparallel are as follows:

- Master and slave (aka *engine*) processes *do not share memory*.
- The framework allows the master process to submit work (*tasks*) to be executed by one or several engines. Two complementary mechanisms are available to do so: the *Load Balanced* and the *Direct* view. With the first one, the set of engines is treated as a pool of workers; the programmer does not have to explicitly determine which engine ultimately executes the task. Instead, an underlying scheduling algorithm is in place to assign tasks to engines dynamically and to quickly assign tasks to idle engines. The second mechanism, by contrast, exposes individual engines to the programmer. Regardless of the mechanism used, ipyparallel allows the master to submit tasks in a synchronous or an asynchronous way (i.e., in the latter the master does not remain blocked until the task or the set of tasks submitted completes).
- In ipyparallel's programming model, engines do not communicate with each other. However, other Python modules can be used to establish explicit inter-engine communication. Because this kind of communication is required in PBBCache, we turned to the ZeroMQ messaging library [7], which is also used in ipyparallel's implementation.

The ipyparallel framework allows applications based on it to seamlessly distribute the computation across cores present in one or several machines, while maintaining a single implementation. Notably, using this approach to leverage parallelism (i.e., multiple processes that cooperate with one another) in our simulator provides much better performance and scalability on a shared-memory machine than using an ad hoc Python multithreaded application. This stems from the fact that, due to implementation issues in CPython –the reference implementation of the Python interpreter–, the truly parallel execution of multiple CPU-bound threads is not allowed within the interpreter [23]. The execution of CPU-bound threads is instead serialized, thus making multithreading an unsuitable choice for applications like our simulator.

We should highlight that Jython –an alternative Python implementation written in Java– is not subject to this multithreaded-related issue. Unfortunately, its JyNi compatibility layer [3], which enables the utilization of a few Python modules written for CPython from Jython, does not currently support the *pandas* or *matplotlib* modules. Because these two widely-used modules are key building blocks of our open-source simulator [68] for seamless visualization and data manipulation, we opted to use CPython, the default Python implementation.

Finally, it is worth noting that, to solve the set of non-linear equations required for the evaluation of our bandwidth-contention model (see Section 5.3.2.1, we use the *simpy* library. This library offers a high-level and flexible mechanism to specify sets of equations in the source code; hence, making it easier to others to create and test their own models.

5.4. Formalization of Opt-STP and Opt-Unf as MINLPs

Here we provide a detailed formalization of the Opt-STP and Opt-Unf problems for a workload A consisting of N single-threaded applications $\{a_1, a_2, \cdots, a_N\}$ that run on a system featuring a W-way last-level cache with $W \geq N$.

We first present the parameters and decision variables which are common to both optimization problems. The **parameters** represent performance metrics of each application in the workload. These values are gathered offline as an application runs alone on the system under different way assignments:

$$IPC_{a\in A,k\in K} \quad \textit{IPC alone} \tag{5.11}$$
$$B_{a\in A,k\in K} \quad \textit{Bandwidth alone} \tag{5.12}$$
$$TS_{a\in A,k\in K} \quad \textit{Total stalls} \tag{5.13}$$
$$MS_{a\in A,k\in K} \quad \textit{Memory stalls} \tag{5.14}$$

Other parameters are calculated internally based on the others:

$$pIPC_a = IPC_{a,W}, \forall a \in A \quad \textit{IPC with all available ways} \tag{5.15}$$
$$pS^c_{a,k} = \frac{pIPC_a}{IPC_{a,k}}, \forall a \in A, \forall k \in K \quad \textit{Relative performance degradation} \tag{5.16}$$

The **decision variables** are the following:

$$w_{a\in A,k\in K} \in \{0,1\} \quad \textit{Way assignment} \tag{5.17}$$
$$\overline{vB}_{a\in A} \in \mathbb{R} \quad \textit{Effective Bandwidth alone} \tag{5.18}$$
$$\overline{vB'}_{a\in A} \in \mathbb{R} \quad \textit{Bandwidth shared} \tag{5.19}$$
$$\overline{vT} \in \mathbb{R} \quad \textit{Total bandwidth shared} \tag{5.20}$$
$$vSC_{a\in A} \in \mathbb{R} \quad \textit{Effective slowdown due to cache-sharing} \tag{5.21}$$
$$vSB_{a\in A} \in \mathbb{R} \quad \textit{Bandwidth Slowdown} \tag{5.22}$$
$$vTS_{a\in A} \in \mathbb{R} \quad \textit{Effective total stalls} \tag{5.23}$$
$$vMS_{a\in A} \in \mathbb{R} \quad \textit{Effective memory stalls} \tag{5.24}$$

$$vS_{a\in A} \in \mathbb{R} \quad \textit{Effective Slowdown} \tag{5.25}$$

The **Opt-STP** problem can be formulated as follows:

$$\text{Maximize}: \ \sum_{a\in A} \frac{1}{vS_a} \tag{5.26}$$

subject to these linear constraints:

$$\overline{vT} = \sum_{a \in A} \overline{vB'}_a \tag{5.27}$$
$$\sum_{k \in K} w_{a,k} = 1, \forall a \in A \tag{5.28}$$
$$\sum_{a \in A} \sum_{k \in K} k \cdot w_{a,k} = W \tag{5.29}$$
$$1 \leq \sum_{k \in K} k \cdot w_{a,k} \leq W - N + 1, \forall a \in A \tag{5.30}$$
$$\overline{vB}_a = \sum_{k \in K} B_{a,k} \cdot w_{a,k}, \forall a \in A \tag{5.31}$$
$$vSC_a = \sum_{k \in K} pS^c_{a,k} \cdot w_{a,k}, \forall a \in A \tag{5.32}$$
$$vTS_a = \sum_{k \in K} TS_{a,k} \cdot w_{a,k}, \forall a \in A \tag{5.33}$$
$$vMS_a = \sum_{k \in K} MS_{a,k} \cdot w_{a,k}, \forall a \in A \tag{5.34}$$

And also **subject to** the following set of non-linear constraints for the evaluation of the bandwidth model. Specifically, $\forall a \in A$, we have that:

$$(\overline{vB'}_a)^2 \cdot \left(1 - \frac{1}{\overline{vB}_a}\right) + \overline{vB'}_a \cdot \left(1 - \frac{1}{\overline{vT}}\right) \cdot \left(1 - \frac{1}{\overline{vB}_a}\right) + 1 - \frac{1}{\overline{vT}} = 0 \tag{5.35}$$
$$vSB_a = \frac{vTS_a + vMS_a \cdot \left(\frac{\overline{vB}_a}{\overline{vB'}_a} - 1\right)}{vTS_a} \tag{5.36}$$
$$vS_a = vSC_a \cdot vSB_a \tag{5.37}$$

The **Opt-Unf** problem can be formulated as a MINLP, as follows:

$$\text{Minimize}: \ \frac{S_{\max}}{S_{\min}}, S_{\max} \geq vS_a, \forall a \in A, S_{\min} \leq vS_a, \forall a \in A \tag{5.38}$$

subject to the constraints specified by Equations 5.27 to 5.37.

5.5. Determining the optimal solution

This section describes the features of the various B&B algorithms used by PBB-Cache to determine the solution of the Opt-STP and Opt-Unf optimization problems. Currently, four B&B algorithms exist: Opt-STP-S, Opt-STP-P, Opt-Unf-S and Opt-Unf-P. The name of each algorithm encodes the optimization problem the algorithm solves (Opt-STP or Opt-Unf) and indicates whether the algorithm is sequential or parallel (via the -S and -P suffixes, respectively). Opt-STP-S and Opt-Unf-S follow the structure of the sequential B&B algorithm depicted in Algorithm 5.1. They are primarily used to assess the effectiveness of the proposed bounding functions, and as a baseline to quantify the scalability of the corresponding parallel version.

All these B&B algorithms have several things in common. First, they all use best-first search. Second, the sequential and parallel algorithms for the same optimization problem share the same bounding function and the same heuristic approach to

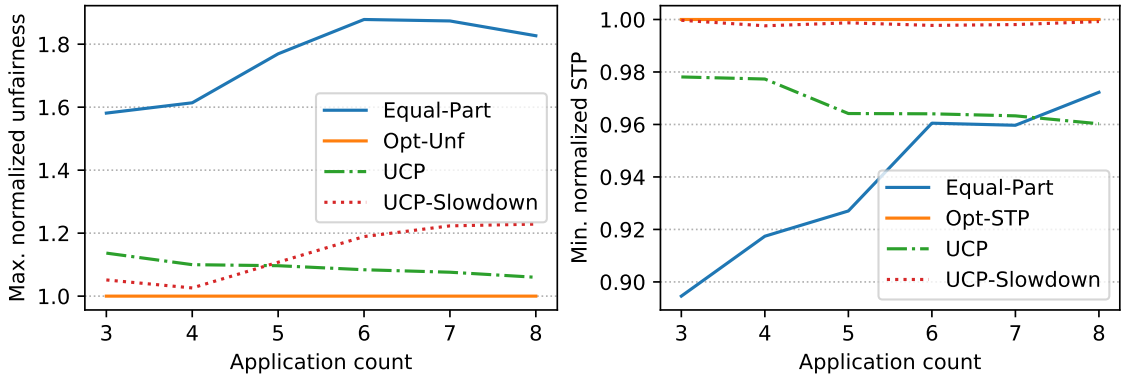


Figure 5.4: Comparison of various approximate algorithms and the optimal solution.

determine the initial solution. Third, despite the fact that we identify four B&B algorithms – for the sake of clarity in the explanation – two generic functions of our simulator are used to implement them. Among other things, these two functions accept as a parameter the bounding function to be applied and a flag that indicates whether it is a maximization or minimization problem. This allows us to implement the Opt-STP-P and Opt-Unf-P algorithms by invoking a single generic function; the same applies to Opt-STP-S and Opt-Unf-S. Note that this also makes it easier to extend the PBBCache with support for optimal cache partitioning under other optimization objectives (e.g., energy efficiency minimization).

In the remainder of this section we first discuss the approach to determine the initial solution (Section 5.5.1), and then proceed to present the bounding functions we use for the Opt-STP and Opt-Unf optimization problems (Section 5.5.2). Next, we describe the generic distributed-memory parallel approach used by Opt-STP-P and Opt-Unf-P (Section 5.5.3). Finally, we discuss different strategies to determine the solution of the optimal cache-clustering problem (Section 5.5.4).

5.5.1. Initial solution for B&B

To find a suitable strategy to determine the initial solution in the B&B algorithms, we considered 3 simple partitioning approaches: *Equal-Part*, *UCP* and *UCP-Slowdown*. The first two schemes were described in Section 5.3.4; *UCP-Slowdown* is a variant of UCP [153] that uses per-application slowdown tables (i.e. slowdown for different number of ways) instead of per-application MPKI tables. In using slowdown tables, UCP-Slowdown attempts to minimize the summation of slowdown across applications, by considering exclusively the performance degradation that comes from cache sharing.

Figure 5.4 illustrates how the three approaches perform relative to the optimal solution for the Opt-STP and Opt-Unf optimization problems when using workloads with different application counts. Each point in the chart represents the worst value obtained for the metric in question (normalized Unfairness or STP) across 10 randomly generated workloads with the same application count (indicated on the y-axis). In light of the results, we opted to choose UCP to obtain the initial

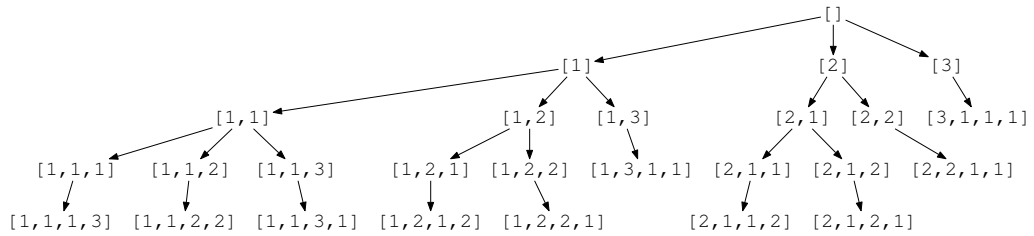


Figure 5.5: Search space tree for the optimal cache-partitioning problem with 4 applications and 6 ways. Equation 5.5 indicates the different cases to be considered in expanding each node based on W (remaining ways) and N (remaining applications). A node has only one child (leaf) when $W{=}N$ or $N{=}1$. Otherwise it has as many as $W{-}N{+}1$ children, that come from inserting a number $\in \{1 \,..\, W{-}N{+}1\}$ at the end of the node's list.

solution in the Opt-Unf-S and Opt-Unf-P B&B algorithms, as it exhibits the closest behavior to the solution of the Opt-Unf optimization problem. By contrast, for the Opt-STP-S and Opt-STP-P algorithms we employ UCP-Slowdown instead, as the STP it provides is in less than a 0.3% range of that of Opt-STP, thus clearly outperforming the other approaches. Note that both UCP and UCP-Slowdown, employ the *lookahead algorithm*, which, as reported in [153] has a worst-case time complexity of $\frac{W^2}{2}$, where W denotes the total number of cache ways.

5.5.2. Bounding functions

Before describing the bounding functions, we introduce the notation used to represent solutions in the optimal cache-partitioning problem. Figure 5.5 shows the corresponding search-space tree for 4 applications and 6 ways. The leaf nodes of the tree represent the complete, feasible solutions available; intermediate nodes represent partial solutions. For simplicity, each solution is represented as a list where each i-th item indicates the number of ways allotted to application i. So for example, the solution associated with the leftmost node of the tree is `[1,1,1,3]`, namely, the first three applications in the workload get 1 way and the last one gets 3.

Henceforth, we will refer to the bounding function used by the OPT-STP-S/P algorithms as *bound_stp*, and to that of the OPT-Unf-S/P algorithms as *bound_unf*. Both bounding functions accept as a parameter the partial solution (*PS*) associated with the node being explored, as well as the number of remaining ways to assign (R). The *bound_stp* function determines an upper bound of the STP value for the best solution for throughput reachable from *PS*; *bound_unf* provides a lower bound of the Unfairness metric for the best solution for fairness reachable from *PS*. Several factors make determining these bounds a very complex problem. First, both metrics (see Section 3.3.1) are defined in terms of the slowdown of each application in a workload. Note that an application's slowdown depends on both the number of cache ways allotted to it, and on the degree of bandwidth contention on the system, which, in turn, varies with the distribution of the remaining cache ways among the rest of applications in the workload. Second, determining the fraction of the slowdown that comes from bandwidth contention alone entails solving the set of non-linear equations of our model (Equations 5.8 and 5.9). In our setting, this may

take hundreds of milliseconds, so exploring multiple candidate solutions reachable from PS to determine a bound is largely impractical; the costly bandwidth model would have to be applied multiple times (one for each candidate solution) thus incurring the associated overhead.

In defining the bounding functions, we rely on the observation that the slowdown of an application decreases as we assign more cache ways to it. Hence, the higher the number of ways available on the platform (W), the higher the value of the STP metric; a higher way count also contributes to reducing unfairness in most cases. Due to the complexity of determining the bounds, coupled with the high cost associated with the bandwidth model evaluation, we opted to relax the cache partitioning problem (just for this purpose) by removing the constraint on the total number of ways that can be allotted. Specifically, both bounding functions rely on determining an *ideal* solution reachable from PS that optimizes STP. The ideal solution is a feasible solution for the relaxed cache-partitioning problem, where the total number of ways assigned to the applications may be greater than W.

The ideal solution is obtained by allotting each and every application not considered in the partial solution PS, the maximum amount of ways found in any feasible solution reachable from PS. Specifically, let S be the number of applications to be considered for the distribution of those remaining ways. In any possible distribution of R ways, any application gets $R - S + 1$ ways at the most. Hence, the ideal solution results from completing PS by assigning $R - S + 1$ ways to the remaining R applications. For example, let us consider a system consisting of a 10-way LLC and a workload made up of four applications. For PS=`[3,2]`, the ideal solution would be `[3,2,4,4]`.

The upper bound provided by *bound_stp* is the STP value of that ideal solution. To determine a lower bound in *bound_unf* for the Unfairness metric –defined in Equation 3.1– we have to follow a different approach. A trivial lower bound L for a node can be obtained as follows: $L = \frac{M}{m}$, where M and m are the maximum and minimum slowdowns, respectively, observed across applications in PS (partial solution of the node). The unfairness of any solution reachable from PS will be $\geq L$ since, in assigning ways to the rest of applications, the new maximum slowdown will be $\geq M$ and the new minimum slowdown will be $\leq m$. Notably, *bound_unf* determines a less optimistic lower bound L' defined as $\frac{M'}{m}$, where M' is the maximum slowdown found in the aforementioned ideal solution (i.e., PS filled with $R - S + 1$ values). Note that $L' \geq L$, and L' is still a lower bound, as the ideal solution guarantees the lowest possible slowdown for the remaining applications, hence minimizing the ratio, as we keep the same denominator m.

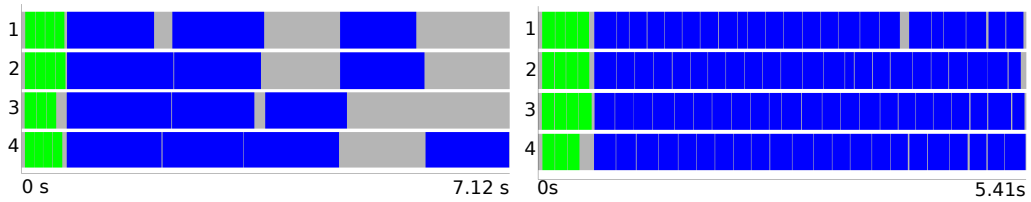
Despite the simplicity of the bounding approach, our experimental results in Section 5.6.3 reveal that the *bound_stp* and *bound_unf* functions lead to very effective pruning. Moreover, because the bandwidth model has to be evaluated just once, the approach is affordable in terms of computational cost.

5.5.3. Parallel distributed-memory B&B algorithms

The Opt-STP-P and Opt-Unf-P algorithms follow the same parallel strategy. For the sake of simplicity in the explanation, we describe the strategy assuming a minimization problem, as it is done in [50].

We begin by discussing the main challenges that arise in attempting to solve the Opt-Unf and Opt-STP problems via parallel B&B. First, as shown in Figure 5.5, the search space tree is largely unbalanced. A possible approach to parallelizing the search consists in breaking down the full tree into subtrees – preferably with a similar node count, and processing these subtrees in parallel. However, this approach does not necessarily provide good scalability due to the unpredictable effect of pruning; the number of nodes of a particular subtree that ultimately have to be processed depend on the pruning effectiveness in that area of the search space. Therefore, considering an entire subtree as the work unit for parallel processing does not constitute a good approach, as it leads to load imbalance. Secondly, calculating the cost of a solution (leaf node of the tree) or determining the lower bound of any node entails solving the set of non-linear equations of the bandwidth model, which, as stated earlier, may have a substantial overhead (in the order of hundreds of ms.). Note that when processing a node of the tree, these tasks usually have to be performed several times. Because this kind of processing by itself has substantial computational load, treating one individual node (rather than a subtree) as the work unit for parallel computation constitutes a promising approach. Our strategy is based on this idea.

Our implementation uses a distributed-memory master-slave pattern. The work unit to be processed by slave processes is the *subnode*; we use this term to denote the processing that has to be done for a subset of children of a certain node in the tree. Recall that in processing a node, the B&B method has to determine the lower bound for all of its children; children nodes with a lower bound greater than the upper bound are pruned, and the remaining nodes are either considered when updating the upper bound (leaf nodes) or left for further processing. We observed that using the node as the work unit leads to load imbalance, as the higher the number of children – which largely varies across nodes of the tree – the higher the computational cost associated with processing the node. To overcome this problem, we divide the node-level processing into groups of children, each one with a children count not greater than `max_children` – a configurable parameter of our algorithm. Specifically, a node consisting of C children is divided into $\lceil \frac{C}{\texttt{max\_children}} \rceil$ subnodes. Breaking down the node-level processing in this manner allows us to create smaller tasks with similar granularity, which enable a more even distribution of the work especially when pruning is working very effectively (just a few nodes to process). To illustrate this fact, Figures 5.6a and 5.6b show the task granularity that comes from using the node and the subnode as the work unit. Clearly, using subnodes leads to a higher number of smaller, more uniform tasks. This contributes to reducing load imbalance and improves performance. The default value of the `max_children` parameter is 3, which makes it possible to obtain fine-grained tasks but with enough computational load so that it is worth it to send them to slaves for processing even



(a) Node-based processing (b) Subnode-based processing

Figure 5.6: Traces for Opt-STP-P obtained with Paraver [33] (6 applications and 4 slave processes). Tasks in blue denote node (a) and subnode (b) processing; idle periods appear in gray; light-green tasks represent the parallel initialization of the subnode queue.

on a different computing node, like the setting explored in Section 5.6.

Algorithm 5.3: Simulator's master B&B code.

Input: A is the workload considered for cache partitioning, W is the number of cache ways available on the platform, *slave_count* is the number processes in *slave_pool*

```
1  incumbent ← initial solution for A provided by heuristic algorithm (see Section 5.5.1)
2  upper_bound ← incumbent.Cost();
3  U ← slave_count * initial_load ; queue_limit ← slave_count * 2 ; pending ← [ ]; prio_q ← [ ]
4  Send A and other global data to slave_pool
   // Initialization of the subnode queue (prio_q)
5  NS ← unroll a number of nodes no smaller than U from A's tree via breadth-first traversal
6  Calculate lower bound in parallel for each node N_i ∈ NS using slave_pool
7  Remove nodes from NS whose lower bound≥ upper_bound (those will not be processed)
8  foreach node n_i in NS do prio_q.concat(break_into_subnodes(n_i, W, max_children)) end
9  while !prio_q.isEmpty() || !pending.isEmpty() do
       // Submit subnodes to the slave pool asynchronously
10     while !prio_q.isEmpty() && len(pending) ≤ queue_limit do
11         subnode ← pop highest prio subnode from prio_q
12         ptask ← slave_pool.async_submit(subnode, subnode.lower_bound)
13         pending.append(ptask, subnode.lower_bound)
14     end
15     completed_tasks ← Remain blocked until at least one task in pending list completes
16     (local_ub, local_inc) ← (upper_bound, incumbent)
       // Retrieve remote upper bounds (ZeroMQ), update and prune if necessary
17     (incumbent, upper_bound) ← process_remote_info(prio_q, pending, local_ub, local_inc)
       // Process completed tasks
18     foreach t_i in completed_tasks do
19         foreach subnode s_j in t_i.promising_child_subnodes() do append s_j to prio_q if
             s_j.lower_bound < upper_bound end
20         Remove t_i from pending
21     end
22 end
```

Algorithms 5.3 and 5.4 outline the behavior in a minimization B&B of the master process and the subnode processing by a slave. The master submits subnodes to the slave pool; upon completion of a subnode processing request the slave process returns a list of promising child subnodes back to the master – to be processed later. The distributed algorithm terminates when there are no subnodes left to process. Because master and slave processes do not share memory, each process keeps a local copy of the upper bound; when any process finds a better solution, the new identified upper bound is notified to the rest of the processes by using the publish-subscribe communication pattern of the ZeroMQ messaging library [7]. Specifically, each process acts as an independent publisher and, in turn, is subscribed to the notifications issued by the remaining processes. Upon receiving a notification, if the remote upper bound is better than the local one, the process updates the local

Algorithm 5.4: Simulator's slave B&B code for subnode processing.

Input: A: workload considered for cache partitioning; W is the number of cache ways available on the platform; *subnode* to be processed, LB: lower bound of *subnode*. (Note that *upper_bound* and *incumbent* are global variables, but local to each slave process)

```
1  promising ← [ ] // Initialize list of promising subnodes to be returned to master
2  if subnode.isSolution() || subnode.ReachableSolutions() == 1 then
3      (solution, cost) ← subnode.getSolution()
4      if cost < upper_bound then
5          (incumbent, upper_bound) ← (solution, cost)
6          zeromq_publish(incumbent, upper_bound)
7      end
8  else
9      foreach child in subnode.getChildren() do
10         (local_ub, local_inc) ← (upper_bound, incumbent)
           // Retrieve remote upper bounds (ZeroMQ), update and prune if necessary
11         (incumbent, upper_bound) ← process_remote_info(promising, [ ], local_ub, local_inc)
12         if LB > upper_bound then return [ ]
13         if child.isSolution() || child.ReachableSolutions() == 1 then
14             (solution, cost) ← child.getSolution()
15             if cost < upper_bound then
16                 (incumbent, upper_bound) ← (solution, cost)
17                 zeromq_publish(incumbent, upper_bound)
18             end
19         else
20             lower_bound ← bounding_function (child);
21             child.setLowerBound(lower_bound);
22             if lower_bound < upper_bound then
23                 (incumbent, upper_bound) ← (solution, cost)
24                 promising_q.concat(break_into_subnodes(child, W, max_children))
25             end
26         end
27     end
28 end
29 return promising
```

upper bound, which will be used for pruning from then on. Note that using ZeroMQ for this task provides a simple and efficient implementation, and does not create additional library dependencies for our simulator, as ipyparallel's implementation already relies on ZeroMQ.

As it is shown in Algorithm 5.3, the master process maintains a priority queue of subnodes yet to be submitted (line 8), and a list with pending subnodes currently being processed by slaves (line 13). Subnodes in both data structures are sorted in ascending order by its lower bound, so as to perform pruning operations more efficiently. To follow a best-first approach, the most promising subnode, located at the front of the queue is submitted first. When the upper bound is updated in the master process unpromising subnodes in the queue are simply pruned by removing them from the queue (line 17). The master may also prune unpromising subnodes being currently processed by slaves; the corresponding task in the slave process is immediately cancelled remotely and the associated subnode is removed from the pending list (line 20).

As shown in Algorithm 5.3, the master process continuously submits subnodes to the slave pool asynchronously. The load balancing algorithm of the ipyparallel framework [10] automatically maps work to specific slaves so as to balance the load in the slave pool. The submission of subnodes is temporarily stalled by the master when the total number of subnodes being processed by slaves exceeds twice the number of slave processes. We found that increasing the number of pending subnodes beyond that point leads to submitting a higher number of potentially unpromising subnodes, that would have been otherwise pruned locally by the master, rather than cancelled. This degrades performance as it may keep slave processes busy for a longer period of time doing useless work. Conversely, considering fewer pending subnodes causes slaves to go idle more frequently, as they wait more often for the master to submit new work, thus negatively impacting scalability. Therefore, our choice of the maximum number of pending tasks provides a good trade-off between pruning effectiveness and scalability.

Finally, we zoom in on the initialization of the subnode queue (lines 3-8 – Algorithm 5.3). A simple way to initialize it would be to insert the root node of the tree only, as done in the sequential algorithm depicted in Algorithm 5.1. However, using this approach here degrades scalability substantially as it does not keep all slave processes busy from the beginning of the execution. Specifically, it takes some time to expand enough subnodes to make this happen. Our algorithm instead unrolls a certain number of tree nodes via breadth-first traversal (line 5). In doing so, the master builds a list of nodes that belong to a certain level of the tree l or to two consecutive levels (l and $l + 1$); nodes in the upper levels of the tree ($< l$) are automatically discarded by the B&B algorithm so as to remove the substantial overhead associated with the evaluation of the bounding function in the master process. Nodes on the list are submitted to slave processes so as to compute their lower bound in parallel; nodes whose lower bound is higher than the upper bound are pruned. Finally, promising nodes are broken down into subnodes (line 8), which are used to populate the queue. Note that the number of nodes unrolled is no smaller than `initial_load` $*$ `slave_count`, where `slave_count` denotes the

number of slave processes, and `initial_load` is a configurable parameter of the algorithm, whose default value is 2. That value typically ensures that the queue is initialized with enough subnodes to keep slaves busy.

5.5.4. Determining the optimal cache-clustering solution

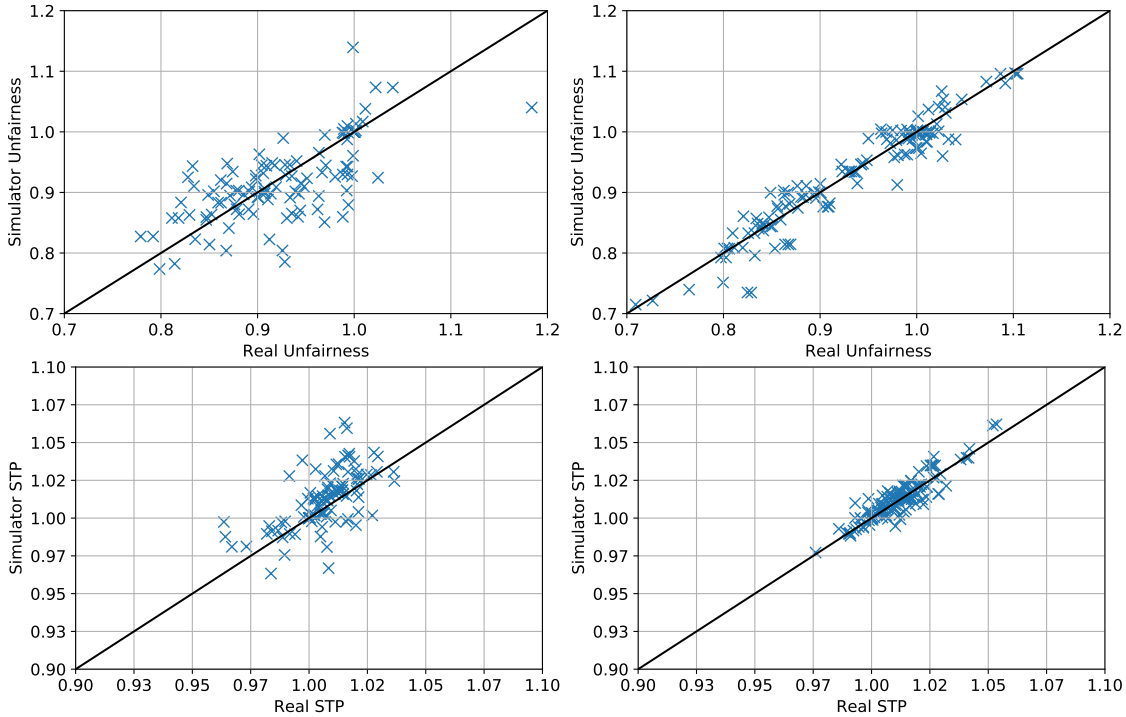
To determine the optimal clustering for a workload A, we must consider all *cluster sets* of A. For each cluster set, the optimal distribution of the cache space among clusters must be determined. The optimal solution is the best one observed across cluster sets.

In this problem, each possible cluster set can be explored in parallel. In turn, we may also exploit parallelism to determine the optimal cache partitioning for a given cluster set. Exploiting these two complementary levels of parallelism, however, is not possible with ipyparallel, as nested parallelism is not currently supported. PBBCache implements a master-slave algorithm that evaluates multiple cluster sets in parallel. The master generates every possible cluster set for the workload and submits them to the slave pool for processing by leveraging a mechanism to restrict the number of pending tasks similar to the one described in Section 5.5.3. Slaves employ the Opt-Unf-S or Opt-STP-S algorithms to sequentially determine the optimal cache partitioning for a given cluster set under the fairness and STP optimization objectives, respectively. Finally, it is worth highlighting that the exploitation of the outermost level of parallelism (what we do here) is generally more effective than considering the inner one, because in many of the cluster sets that must be explored, the corresponding search space tree for the optimal cache-partitioning problem is very small. Processing these small trees in parallel does not bring substantial performance gains relative to doing it sequentially.

5.6. Experiments

5.6.1. Experimental Setup

In this chapter we turned to platforms *Broadwell-EP* and *Skylake* for the simulator's validation, since they are the only ones that support cache partitioning (via Intel CAT technology). To preserve comparability among platforms, all experiments were conducted with workloads of up to 8 applications, as it is the maximum number of cores of platform *Broadwell-EP*. Besides, we experimented with platforms *Haswell* and *SandyBridge-EP* to assess the simulator scalability on machines with a higher core count. Note that the experiments were performed with hyperthreading (SMT) and turbo boost disabled. A full description of the platforms used in these experiments can be found in Chapter 3).



(a) *Broadwell-EP* (b) *Skylake*

Figure 5.7: Real vs. simulator-provided values for the STP and Unfairness metrics on *Broadwell-EP* (left) and *Skylake* (right), normalized to the results of the Equal-Part scheme.

5.6.2. Validation of the simulator

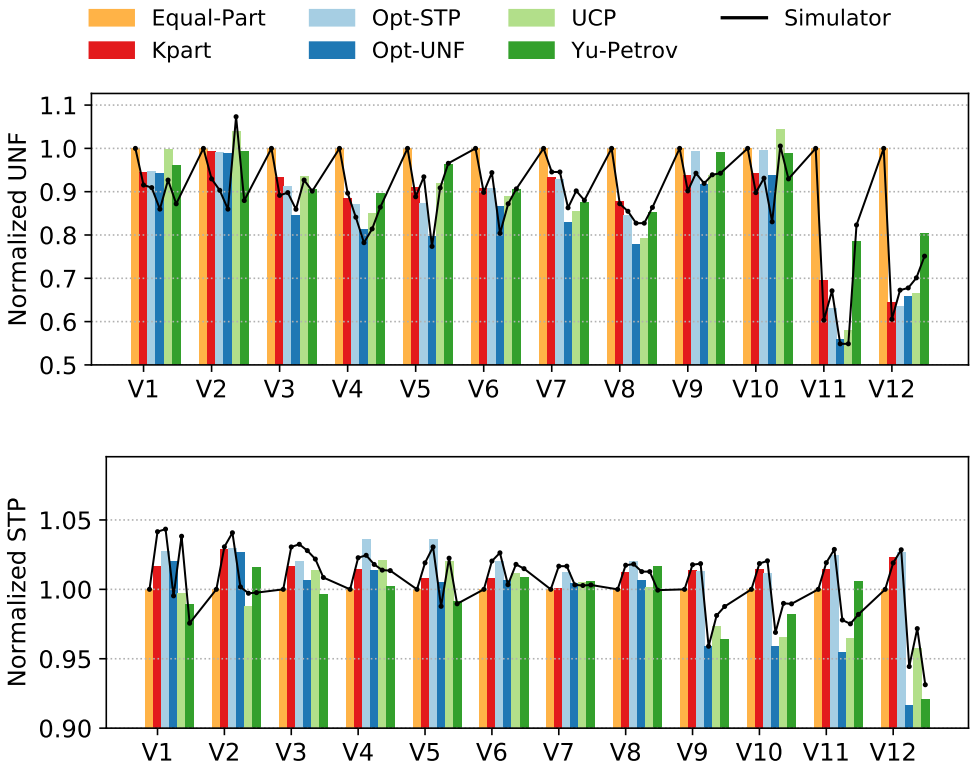
For the validation experiments we used 25 randomly generated workloads consisting of 6 and 8 programs each. In building the workloads, which exhibit a different degree of shared-resource contention, we used 25 different benchmarks from SPEC CPU. For each workload we collected the STP and Unfairness values provided by PBBCache for various partitioning algorithms: UCP [153], Yu-Petrov [195], Equal-Part and KPart [58]. We also gathered the corresponding values for the optimal solutions provided by PBBCache for the Opt-Unf and Opt-STP problems. To validate these results we compared them with those observed when applying the same partitioning statically on the real machine where the corresponding performance information for the simulation was obtained (*Broadwell-EP* and *Skylake*).

For the experiments on the actual machine, we rely on the PMCTrack tool [160], which makes it possible to establish per-process cache partitions from user-space on systems equipped with Intel CAT. Essentially, prior to the execution of the workload under a certain partitioning approach we run the simulator to retrieve the associated cache partitioning provided by the algorithm, and map each application in the workload to its corresponding partition. Note that, for simplicity, the partitions remain the same (static) throughout the workload's execution. We ensure that all applications in the mix are started simultaneously and when one of them terminates it is restarted repeatedly until the longest application in the set completes three times. We then measure unfairness and STP, by using the geometric mean of the completion times for each program.

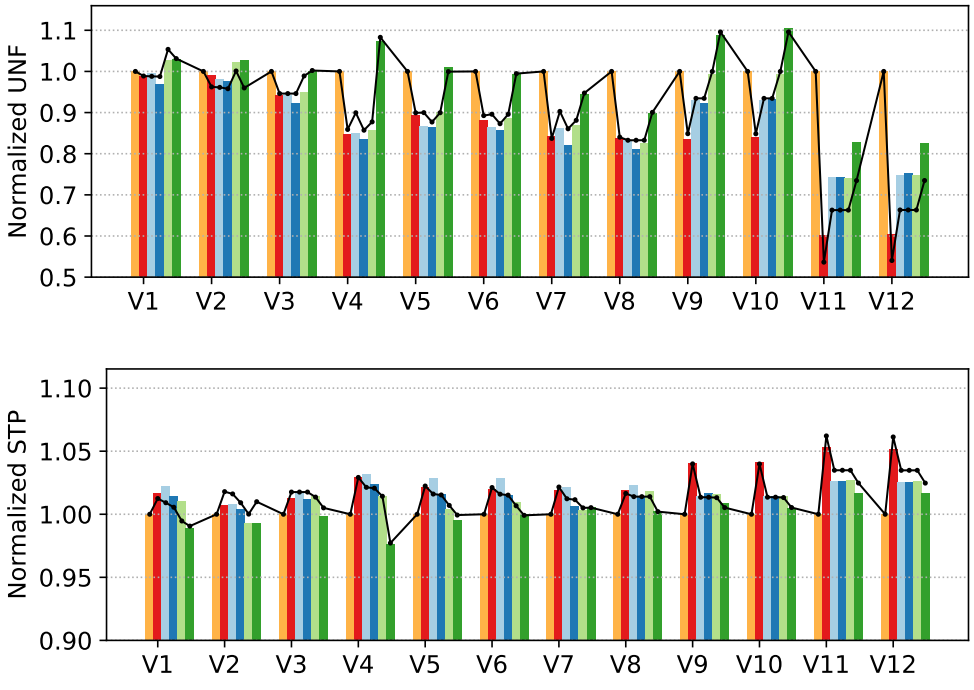
Figures 5.7a and 5.7b show the comparison of the real vs. simulator-provided values for the STP and Unfairness metrics on *Broadwell-EP* and *Skylake*, respectively. Both metrics have been normalized to the results of the Equal-Part policy. We observe that the Unfairness and STP values provided by the simulator closely track those of the actual platforms. Specifically, the average error rate observed on *Broadwell-EP* is 3% and 1% for Unfairness and STP respectively, and on the *Skylake* is 2% and 0.4%. We found that the main reason behind the less accurate simulations on *Broadwell-EP* has to do with inaccuracies in the bandwidth model that became apparent for some bandwidth-intensive applications. In particular, these model inaccuracies are due to the fact that PBBCache was fed with the average bandwidth registered across the execution to predict the slowdown due to contention. In some cases, this average does not represent the benchmark behavior in certain program phases where we find spikes in bandwidth consumption that are substantially higher than the average. Consequently, the actual slowdown is underpredicted due to serious bandwidth contention in these cases, leading to lower Unfairness and STP values. Additionally, the theoretical maximum memory bandwidth on *Broadwell-EP* is 68.3GB/s, whereas on *Skylake* this bandwidth is 128GB/s. As a result, bandwidth contention is substantially higher on the first platform and this issue contributes to exacerbate the previously discussed model inaccuracies.

A potential way to address this issue, which did not prevent us from gathering good predictions on *Skylake*, would be to make PBBCache aware of the different program phases of bandwidth-intensive applications. Because phase transitions do not occur at the same time in multiple programs, this would require to break down the whole workload execution into stages of stable behavior across applications, and apply the particular partitioning scheme on each stage. Unfortunately, this approach would make it difficult to obtain a solution in a reasonable amount of time, which stands in contrast with the main goal of the simulator: a tool for rapid prototyping and evaluation. Note that to determine an exact solution the B&B algorithm for the Opt-STP (or Opt-Unf) non-linear optimization problem would have to be invoked for each execution stage one after another, as the progress each application makes (which must be determined to detect the next phase change) depends in turn of the partitioning applied in the previous stage.

Figure 5.8 shows the accuracy of the simulator predictions on *Broadwell-EP* and *Skylake* for some representative workloads, which summarize the trends observed in all our experiments. The results are reported separately for each individual partitioning scheme. Clearly, PBBCache is capable of capturing the relative benefit in STP and Unfairness that a partitioning scheme obtains over the others, and enables us to identify the best performing schemes in each scenario. Notably, dividing the LLC into same-sized partitions (Equal-Part) does not constitute a good approach regarding fairness or throughput, as, in doing so, we do not cater to the degree of cache sensitivity of each application. For instance, when assigning the same cache space to an application whose data can fit entirely on the private cache levels, you are effectively removing it from the rest of the applications in the system that could take a real advantage of this space. Even if the Yu-Petrov's algorithm provides



(a) *Broadwell-EP*



(b) *Skylake*

Figure 5.8: Real vs. simulator-provided values for the STP and Unfairness metrics on Platforms *Broadwell-EP* and *Skylake*, normalized to the results of Equal-Part.

Application count	**3**	**4**	**5**	**6**	**7**	**8**
Number of nodes	357	2056	8295	24955	58071	106963

Table 5.1: Number of nodes of the search space tree for different workloads on *Broadwell-EP*.

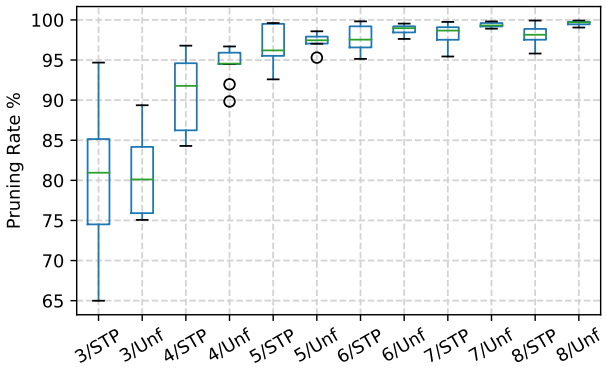
better results in a few cases, the offline-collected bandwidth consumption measurements do not prevent it from suffering from high fairness/throughput degradation – remaining close with the naive Equal-Part approach for workloads V1,V6 and V7. Moreover, we observe that UCP and KPart are more effective in general, but are still far from the optimal cache-partitioning solutions Opt-Unf and Opt-STP in most cases. As pointed out in previous work [171], cache-clustering policies can outperform strict cache-partitioning policies. This is the case for workloads V9-V12 on *Skylake*, where the KPart cache-clustering policy provides better throughput and fairness than Opt-STP and Opt-Unf.

5.6.3. Effectiveness of the bounding functions

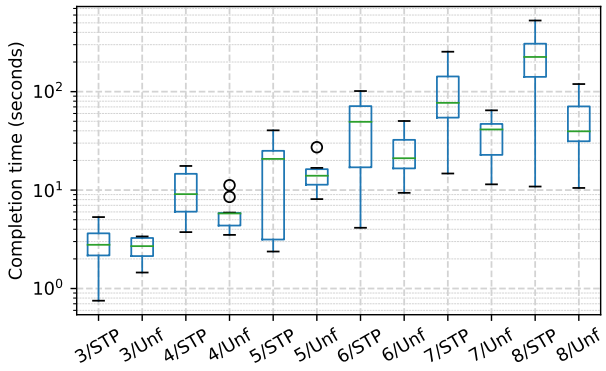
One of the key aspects of the proposed B&B algorithms is the effectiveness of the *bound_stp()* and *bound_unf()* bounding functions, described in Section 5.5.2, and used for the Opt-STP and Opt-Unf problems, respectively. To compare the efficacy of both bounding functions and analyse their impact in performance we randomly built 66 workloads consisting of a number of applications that range between 3 and 8 (the core count of *Broadwell-EP*). Eleven workloads were considered for each application count. For these experiments we fed PBBCache with data gathered on *Broadwell-EP* where the number of possible solutions is substantially higher than on *Skylake* (see Section 5.1.1).

Figure 5.9 shows the pruning rate and the completion time for the different workload sets (one for each application count) and optimization metrics (i.e. STP and Unfairness) obtained by the Opt-STP-S and Opt-Unf-S sequential B&B algorithms. The pruning rate is defined as the percentage over the total number of nodes in the search space tree that were discarded by pruning. The results reveal that the pruning rate largely depends on the nature of the workload. For example, for 3-application workloads under Opt-STP-S, the pruning rate ranges between 65% and 94.7%. We also observe that, as we increase the number of applications in the workload, the variability decreases, and the average pruning rate improves substantially; it is greater than 96.7% when the application count is >5. This indicates that the effectiveness of the bounding functions increases with the problem size, which is a good property of our proposed B&B algorithms.

As expected, the pruning rate has an enormous impact on the completion time. In particular, the slight superiority of *bound_unf()* over *bound_stp()* for 6-8 applications (see Figure 5.9a) leads to substantially smaller completion times due to pruning (Figure 5.9b). Notably, because the number of nodes in the search space tree – shown in Table 5.1 – grows exponentially with the application count, a small increase in the pruning rate may have a significant impact on the completion time.

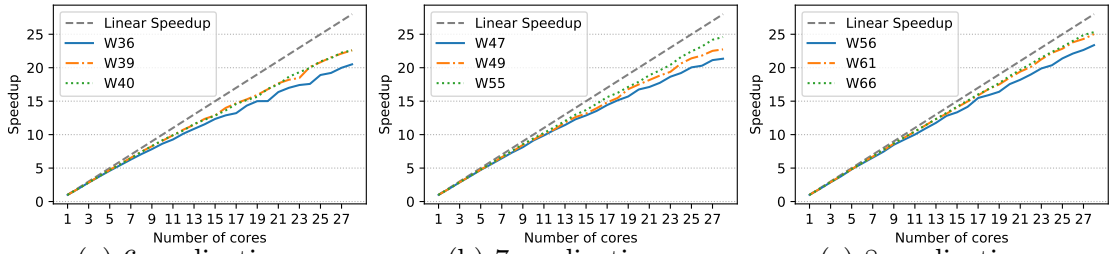


(a)



(b)

Figure 5.9: Pruning rate (a) and completion time (b) for sequential B&B algorithms under different sets of workloads. Labels X axis, with format *n/target* indicate the number of applications in the workload (n) in the corresponding set, and the optimization metric.



(a) 6 applications (b) 7 applications (c) 8 applications

Figure 5.10: Scalability for different workload sets consisting of 6, 7 and 8 applications

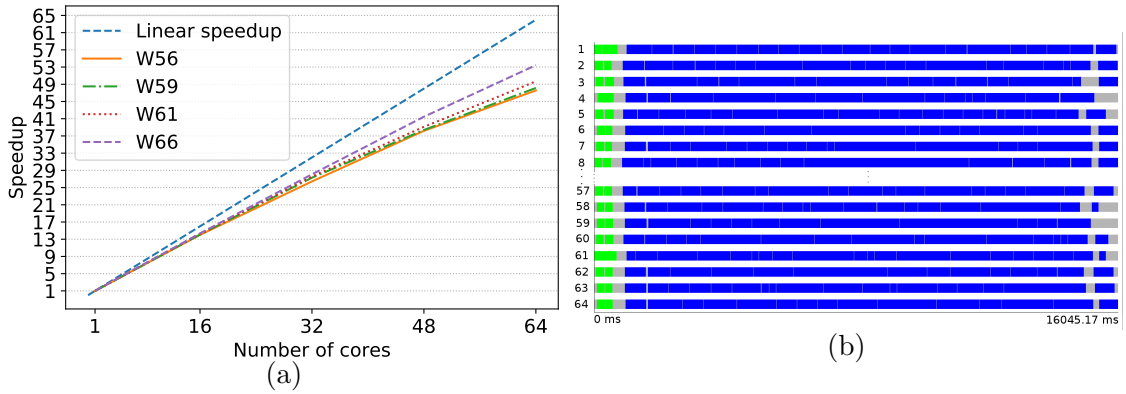
Specifically, an increase of 0.5% of the pruning rate for an 8-application workload can accelerate the execution by a factor of 2x. All in all, we observe that for the workloads explored the Opt-Unf-S algorithm provides the optimal solution in less than 2 minutes, and the Opt-STP-S algorithm in less than 9 minutes. Even though this execution time is relatively low for a single workload, greater workload sizes would exponentially increase the execution time. Hence, the quick evaluation and prototyping of cache partitioning strategies would become largely impractical.

5.6.4. Scalability of the distributed-memory parallel B&B strategy

To assess the scalability of proposed parallel B&B strategy on a single multicore server we used *Haswell* (as shown in Table 3.2). Specifically, we focused on the study of a subset of the workloads explored in Section 5.6.3 where the Opt-STP-S effectiveness of the bounding function, along with the pruning, is not capable of providing an optimal solution.

Figure 5.10 shows the speedup achieved with the Opt-STP-P algorithm for the different application mixes explored (consisting of 6, 7 and 8 applications, respectively) as we vary the number of cores from 1 to 28. As stated in Section 5.5.3, the `max_children` and `initial_load` parameters of the algorithm were set to 3 and 2, respectively. Note that the speedup is reported relative to the completion time of the sequential B&B algorithm. The linear speedup was also included in the charts for comparison purposes.

The results reveal that the maximum speedup registered is 25.2x (W55 with 28 cores), which corresponds to a parallel efficiency of 0.9. Overall, we also observe that the speedup of our parallel approach gets closer to the linear speedup as the workload size increases. This is a positive trend, since the parallel strategy becomes more effective in utilizing multiple cores as the sequential B&B algorithm begins to exhibit substantially longer completion times (up to 9 minutes as shown in Figure 5.9b for the STP metric). Hence, by leveraging parallelism on this platform our proposed simulator is able to determine the optimal solution for any of these workloads in less than 34s. Finally, we should highlight that the reason of the lower scalability observed for 6-application workloads (Figure 5.10a) has to do with the fact that at the end of the execution of the parallel algorithm the number of remaining subnodes to process is smaller than the number of cores, leaving a few cores idle



(a) (b)

Figure 5.11: (a) Speedup for different workload sets using from one to four nodes (16 cores each) on *SandyBridge-EP*. (b) Excerpt of the execution trace for W66 with 64 cores. Note that idle periods are denoted in grey as in traces shown in Section 5.5.3

for a short time period of time. As the problem size increases (higher application count) the fraction over the total execution affected by this imbalance scenario is smaller, which provides better scalability. We next elaborate on this aspect as it also becomes apparent in our multi-node experiments.

For the performance evaluation of the parallel simulator using multiple computing nodes, we used a cluster consisting of four identical 16-core server system, each one following the specifications of *SandyBridge-EP*. Figure 5.11a reports the speedup observed for different workloads as we increase the number of cluster nodes from 1 to 4 (i.e. from 16 to 64 cores).

The results reveal that the parallel strategy is capable of obtaining substantial performance gains relative to the sequential approach by effectively utilizing multiple cores on different nodes. We also observe that the speedup achieved for up to two nodes (32 cores) is very close to the linear speedup, but it drops slightly when using 3 or more nodes. Specifically, for three nodes (48 cores) the parallel efficiency for the various workloads ranges between 0.8 and 0.87. We found that this trend is caused by the issue of load imbalances described previously; as the number of cores increase, some of them remain idle for a short period of time at the end of the execution due to the shortage of subnodes to process. Figure 5.11b illustrates this fact by means of a sample execution trace of the algorithm with 64 cores. We observed that in increasing the problem size, the speedup gets closer to its linear counterpart. A potential approach to obtain higher scalability in maintaining the problem size constant is to start using finer-grained subnodes as the algorithm begins to reach the end of the execution. That would entail (1) maintaining a counter in the master process that keeps track of the number of feasible solutions remaining to explore – by leveraging Equation 5.5 – and (2) devising a policy to adjust the value of the `max_children` parameter dynamically as we get closer to the end of the execution. We plan to implement that promising optimization as part of a future release of our open-source simulator [68].

5.7. Conclusions

In this chapter we have presented PBBCache, an open-source [68] parallel simulator written in Python whose primary goal is to enable rapid prototyping and evaluation of cache partitioning and clustering policies. PBBCache allows researchers to quickly compare novel approaches with the optimal solution or with existing cache-partitioning schemes, making it possible to determine if a new approach is promising even before starting its implementation in the system software and the associated evaluation on real hardware, which can be a time-consuming task.

A key aspect of our simulator is the mechanism it employs to determine the relative performance degradation (i.e. slowdown) that an application suffers when running simultaneously with others on a multicore system. This mechanism factors in the slowdown that comes from contention on two critical shared resources: the last-level cache and memory bandwidth. To account for memory bandwidth contention we extended the model proposed in [136] with awareness on how sensitive application performance is to a reduction of the available bandwidth. Evaluating this model entails solving a set of non-linear equations, which is computationally expensive, and makes determining the optimal cache partitioning that maximizes system throughput or fairness a mixed-integer non-linear problem. To efficiently solve these optimization problems by exploiting parallelism, PBBCache implements a distributed-memory branch-and-bound (B&B) algorithm specifically tailored for optimal cache partitioning. The load-balancing strategy coupled with the effectiveness of the bounding functions specifically designed for the optimization problems considered, gives rise to a scalable simulator that effectively utilizes multiple cores on one or several computing nodes, as we demonstrate in our experiments.

For the validation of PBBCache's simulation model we conducted experiments on commercial platforms that support Intel CAT and Memory Bandwidth Monitoring. Our analysis reveals that the simulator succeeds in identifying what partitioning approach is the most effective for particular workloads. After all the insights exposed in this chapter an important question arises, is this simulator helpful enough to design a novel cache-partitioning policy?

Chapter 6

LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores

In this chapter we show how we take advantage of PBBCache to guide the design and implementation of LFOC, a Lightweight Fairness-Oriented Cache-clustering OS-level policy. LFOC employs Intel CAT's hardware extensions to dynamically create a number of LLC partitions (*clusters*) based on the features of the workload, and maps applications to different clusters by catering to their degree of cache sensitivity and contentiousness. Our policy continuously monitors applications' runtime metrics with hardware performance counters and classifies applications into different categories based on cache behavior. The collected performance information is used as input to an efficient clustering algorithm.

By proposing LFOC, we make the following contributions:

- LFOC leverages a lightweight online mechanism to approximate the degree of cache sensitivity of an application that avoids costly periodic monitoring operations (i.e. measuring application performance for different cache sizes at runtime) used by other approaches [58], whenever possible.
- We conduct a theoretical analysis of the cache-clustering problem that optimizes fairness for different workload scenarios. The insights from this analysis revealed that the key to enforce fairness lies in identifying contentious cache-insensitive (aka *streaming*) applications and confine them to a reduced set of small cache partitions.
- LFOC attempts to approximate the cache clustering enforced by the optimal solution provided by PBBCache. So as to devote the vast majority of space in the LLC to cache-sensitive applications.
- We implemented LFOC in the Linux kernel and evaluated it on a real system featuring an Intel Skylake processor. In our experiments, we compare

its effectiveness to that of two previously proposed policies –Dunn [171] and KPart[58]–, which optimize fairness and throughput, respectively. Our analysis reveals that LFOC is able to deliver up to a 20.5% reduction in unfairness (9% on average) relative to Dunn (fairness-oriented clustering), and delivers higher throughput and fairness than every analyzed scheme for the vast majority of the workload scenarios considered.

The remainder of the chapter is organized as follows. Section 6.1 presents background on cache partitioning and also discusses related work. Section 6.2 presents the analysis of the optimal solution that motivates our proposal. Section 6.3 outlines the design and inner workings of LFOC. Section 6.4 covers the experimental evaluation, and Section 6.5 concludes the chapter.

6.1. Background

6.1.1. Related Work

Many researchers have attempted to mitigate the contention problem in the LLC via software and hardware techniques [205, 133, 124, 58, 171, 202, 161]. A large body of work has addressed this problem via cache-partitioning or cache-clustering approaches equipped with approximate algorithms [153, 195, 58, 137]. A recent survey [133] discusses the most effective solutions available to target various optimization objectives.

Cache partitions can be created on systems with specific hardware support (such as Intel CAT) or by means of software-based solutions, most of which rely on page-coloring [173, 193, 198, 170]. While page-coloring can be applied to off-the-shelf multicore platforms [196] is known to be subject to a number of limitations, which can be overcome by using hardware-based cache partitioning [171]. Among the different hardware alternatives, the main differences essentially lie on how to manage the number of ways for the different applications: some proposals are based on the cache replacement policy [127, 102, 185] while others use set sampling and duplicate cache tags to guide cache partitioning [153, 179]. In this work we propose an OS-level (also extensible to the virtual machine monitor) cache-clustering scheme that leverages hardware-based way-partitioning.

In the remainder of this section we discuss the cache-partitioning and cache-clustering policies closer to our LFOC approach.

6.1.1.1. Cache partitioning proposals

One of the algorithms that has had major influence on the design of cache partitioning algorithms is UCP, presented in Section 5.3.4, relies on hardware extensions to determine per-application MPKI tables at runtime. Unfortunately, more than a

decade later of the original proposal, these hardware extensions have not yet been adopted in commercial platforms. Our LFOC approach relies on the *lookahead* algorithm to distribute the vast majority of the space in the LLC among cache-sensitive applications, by using the per-application slowdown tables (i.e., slowdown for different number of ways relative to using the total way count) as input to the algorithm instead of MPKI tables; this enables us to provide better system-wide performance.

As we pointed out in 5.3.4, The Yu and Petrov [195] cache-partitioning algorithm strives to reduce system bandwidth pressure. However, it relies on bandwidth consumption measurements with different cache sizes gathered offline for the various applications. As opposed to this approach, LFOC does not require offline-collected application data to function.

6.1.1.2. Cache clustering proposals

More recently, different cache-clustering algorithms have been proposed [58, 171] as a more flexible alternative to strict cache-partitioning. As pointed out in Section 5.3.4, KPart is a partitioning policy based on hierarchical clustering that strives to optimize throughput. This algorithm, heavily relies on the ability to determine MPKI and IPC tables online for each application. By contrast, LFOC requires to gather a smaller amount of performance information than KPart while avoiding to perform costly cache way sweeps periodically, thus effectively reducing the sampling overhead. Further details on this technique are specified in Section 6.3

Selfa et al. [171] propose the Dunn cache-partitioning policy, designed to improve fairness. This strategy groups applications into clusters by applying the *k-means* clustering algorithm, using the fraction of core stalls caused by L2 cache misses incurred by the applications as the metric to guide clustering. In our experimental platform this information can be obtained with the `STALLS_L2_MISS` performance counter event. We should highlight that this strategy does not strictly constitute a pure cache-clustering approach as defined in Section 5.1.2, since the cache partitions it creates may overlap with each other. This overlapping can create unpredictable interactions between applications that belong to different clusters [58].

In our experimental evaluation, we compare the effectiveness of LFOC to the Dunn and KPart approaches, and demonstrate that LFOC delivers higher reductions in unfairness than these policies across the board. We should highlight that Dunn and KPart are user-level clustering approaches, unlike LFOC, which was implemented in the OS kernel. User-level solutions may incur higher overheads since they make extensive use of system calls to access privileged resources such as performance monitoring hardware and cache partitioning facilities, which are handled by the OS. LFOC, by contrast, accesses these facilities directly via a lightweight kernel-level API. Moreover, because using floating-point (FP) is problematic at the kernel level [126], our implementation of LFOC is free of any FP operations, as opposed to KPart's [57], which heavily relies on it.

Table 6.1: Classification of applications based on cache behavior

Type	Criterion
Streaming	($Slowdown \leq 1.03$ and $LLCMPKC \geq 10$) in at least one way assignment, and $Slowdown < 1.06$ in all way assignments
Sensitive	If not streaming and $Slowdown \geq 1.05$ for a number of ways ≥ 2
Light sharing	Not streaming and not sensitive

6.2. Analysis of the optimal cache-clustering solution

The design of our approach is inspired by the behavior of the optimal cache-clustering solution that optimizes fairness. In this section we provide an analysis on the optimal solution, which we could approximate for different workload scenarios by using the PBBCache simulator [68].

To carry out our analysis with the simulator, we used performance counters to gather the average value of different runtime metrics with varying cache sizes for applications from the SPEC CPU2006 and CPU2017 suites running alone on a real system featuring an Intel Skylake processor with an 11-way 27.5MB LLC. As well referred to as platform *Skylake*, more information on this platform can be found in Chapter 3. The offline-collected metric values, which correspond to the execution of the first 150 billion instructions of the aforementioned benchmarks, are used as input to the simulator. This information is used to determine the optimal clustering solution for fairness, namely, the solution to the optimal cache-clustering problem that obtains the optimal (minimal) unfairness value for the maximum throughput (STP) attainable.

For our experiments we considered randomly-generated multiprogram workloads including different number of SPEC CPU applications (from 4 to 16). According to the performance data collected offline we classify applications into three classes based on their degree of cache sensitivity and contentiousness: *Cache-sensitive*, *light sharing* and *streaming* programs. At a high level, the cache-sensitive category is used for those programs that experience significant performance drops as we reduce the number of cache ways allotted to them; this is not the case for *light sharing* and *streaming* applications. Streaming programs are characterized by exhibiting a low slowdown for almost all way allocations, while incurring a high number of LLC misses per cycle. Applications of this kind are cache insensitive, and typically act as aggressor programs to cache-sensitive applications co-located on the same cache cluster, as the performance of the latter can be degraded substantially. Light-sharing programs are neither cache sensitive nor aggressive to others (the working set typically fits in the per-core private cache levels). Table 6.1 summarizes the criteria we followed to make this classification on our experimental platform, which is based on two offline-collected metrics: the application slowdown –relative per-

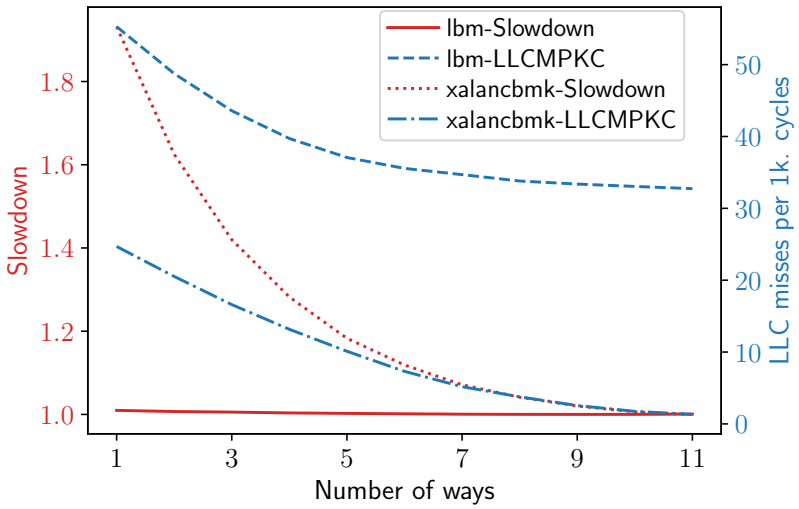


Figure 6.1: Slowdown and LLCMPKC for different way counts

formance with respect to using the entire LLC space– and the number of LLC Misses Per Kilo Cycles (LLCMPKC). Figure 6.1 illustrates the behavior differences of a streaming application (`lbm`) and that of a cache-sensitive one (`xalancbmk`). It highlights how the slowdown and the LLCMPKC varies with the amount of ways allocated to each application.

After a thorough analysis of the optimal cache-clustering and optimal cache-partitioning solutions provided by the simulator for the various workloads, we draw the following major insights:

- In most cases, the cache-clustering solution that optimizes fairness isolates all streaming applications in a reduced set of ways (no greater than 2 in any workload). In many scenarios, a single 1-way cluster is used to confine all streaming programs.
- This same solution maps light-sharing programs onto different clusters following a hardly predictable pattern. More importantly, by conducting additional analyses with the simulator, we observed that moving individual light-sharing applications to different clusters has very little impact on throughput and fairness.
- As expected, the amount of ways assigned to cache-sensitive applications is critical for both throughput and unfairness. Recall that the unfairness metric factors in the maximum slowdown observed across applications in the workload, and sensitive benchmarks typically experience a very high performance degradation if their cache size requirements are not fulfilled.
- The benefit that comes from assigning separate cache partitions (even optimally) to individual applications decreases dramatically as the number of applications gets closer to the number of cache ways. As an illustrative example, Figure 6.2 shows the average unfairness delivered by the optimal partitioning solution, normalized to that of the optimal clustering solution for

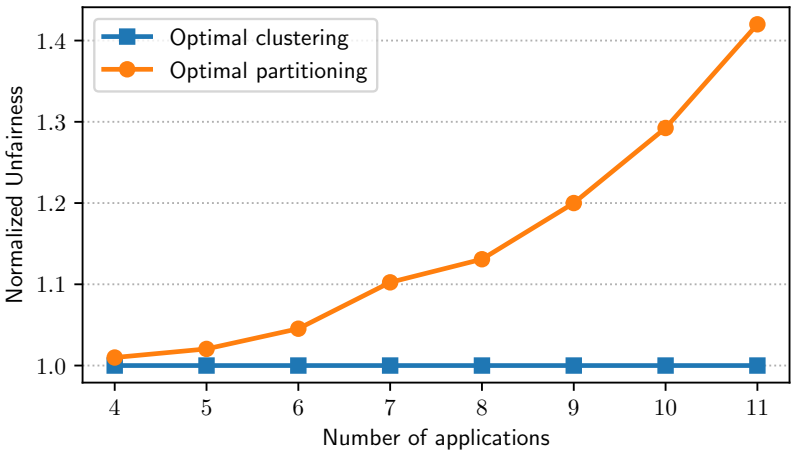


Figure 6.2: Comparison of optimal clustering vs optimal partitioning.

different workload sizes. As observed, optimal cache-partitioning suffers from increased unfairness as the workload size grows. When the application count matches the number of ways, each application can be assigned only one way under strict cache-partitioning – this is the only feasible option, which gives rise to high unfairness in most workloads. Our overarching conclusion is that cache-clustering policies are clearly superior to cache-partitioning approaches as the ratio of the number of ways to the number of applications decreases.

To further illustrate the general behavior of the optimal clustering solution, Figure 6.3 reports the average application count per cluster size, as well as the total number of clusters –grouped by its size (in ways)– that the solution builds for a subset of the workloads we explored: 20 randomly selected program mixes made up of 10 applications each. The data reported in the figure confirms the first three aforementioned observations. First, streaming applications are typically confined in clusters with just one way allocated to them. In relative numbers, more than 87% of streaming application instances are assigned to this kind of clusters, while the remaining ones are allocated to 2-way clusters. Second, we can observe that light sharing applications are mapped to clusters with very different size; however, the vast majority of these programs are mapped to 1-way clusters. Third, the results reveal that cache-sensitive applications are predominantly present in big cache clusters. Specifically, more than 77% of the sensitive application instances are assigned to clusters with 4 or more ways. Finally, as is evident, 1-way clusters with a high number of applications are often present in the optimal solution for the various workloads.

6.3. Design and Implementation

In this section we begin by describing how our clustering algorithm works at a high level. Then we proceed to indicate how applications are classified at runtime by leveraging data from hardware performance monitoring counters.

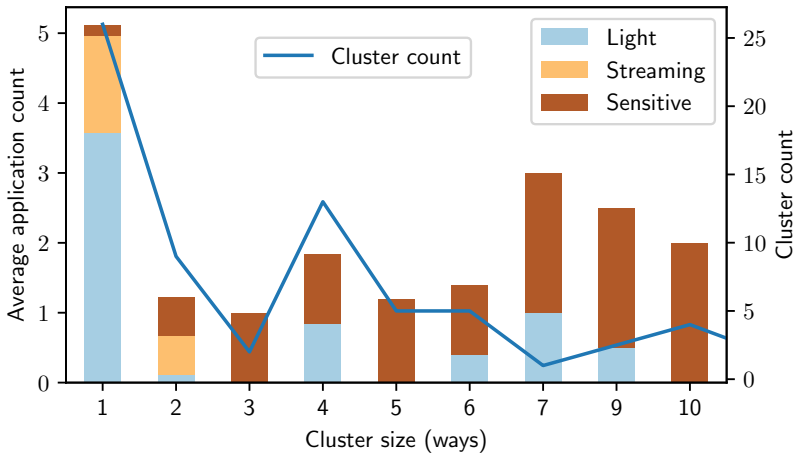


Figure 6.3: Cluster count and breakdown of applications into the different categories for each cluster size.

6.3.1. Algorithm outline

LFOC has been implemented on Linux as an extension of the OS scheduler. Specifically, it has been bundled in a loadable kernel module as a *monitoring plugin* of the PMCTrack tool [160], described in Section 3

LFOC classifies applications at runtime into three classes based on its cache behavior –*light sharing*, *streaming* and *sensitive*– and assigns each application to a certain cache partition whose size is determined dynamically based on the properties of the workload.

When an application enters the system its cache behavior is unknown. To this end, a special *unknown* class is assigned to the application right after being spawned. At the beginning of the execution, each thread has to go through a warm-up period (3 sampling intervals in our experimental setting). Any performance information gathered with hardware counters during the warm-up period is not used to classify applications, so as to mitigate mispredictions associated with cold-start effects.

Periodically, our scheduling extension activates the partitioning scheme depicted in Algorithm 6.1, which relies on the conclusions of the analysis presented in Section 6.2. Overall, the algorithm reserves up to two cache ways to map streaming programs. The remaining cache ways are distributed among cache sensitive applications, which are then assigned to separate cache partitions. The size of these partitions is determined by means of the lookahead algorithm [153], using as input the slowdown curve for each application built by using IPC values obtained online (i.e., slowdown registered for different cache ways). With this cache-way distribution for cache-sensitive applications, LFOC attempts to fulfil their cache requirements based on the degree of cache sensitivity. Finally, light sharing applications are distributed among the various partitions, by attempting to populate partitions with streaming applications first, as the optimal solution typically does.

Algorithm 6.1: Cache-clustering algorithm used by LFOC

Input: ST, CS, and LS represent the sets of streaming, cache-sensitive and light-sharing applications, respectively; $max\_streaming\_way$ and $gaps\_per\_streaming$ are configurable parameters of LFOC (default value 5 and 3, respectively), nr_ways is the number of ways of the LLC.

function `LFOC_partitioning`(ST,CS,LS,nr_ways):
if $|CS| == 0$ **then**
  Create a single cluster S consisting of nr_ways;
  Map all applications in $ST \cup LS$ to S;
  return $\{S\}$
end
$Clusters \leftarrow \varnothing$;
$ways\_for\_streaming \leftarrow min(2, |ST|/max\_streaming\_way)$;
$r = \lceil |ST|/ways\_for\_streaming \rceil$;
for $i \leftarrow 1$ ***to*** $ways\_for\_streaming$ **do**
  Add a new 1-way cluster C to $Clusters$;
  Map up to r apps from ST to C;
  Remove assigned apps from ST;
end
{ Use slowdown tables of CS apps. as input to lookahead }
$W \leftarrow$ `lookahead`($CS, nr\_ways - ways\_for\_streaming$);
for $i \leftarrow 1$ ***to*** $|CS|$ **do**
  Add a new cluster C with $W[i]$ ways to $Clusters$;
  Map application i in CS to C;
end
$idx \leftarrow 0$;
while $|LS| > 0$ ***and*** $idx < ways\_for\_streaming$ **do**
  $TargetC \leftarrow Clusters[idx]$;
  $gaps\_available \leftarrow r - |TargetC| * gaps\_per\_streaming$;
  if $gaps\_available > 0$ **then**
    Map up to $gaps\_available$ apps from LS to $TargetC$;
    Remove assigned apps from LS;
  end
end
Distribute remaining applications in LS in a round-robin fashion among non-streaming clusters;
return $Clusters$

6.3.2. Application Classification

Once the warm-up period for a particular application has finished, LFOC enters a *sampling mode* whose goal is to determine the application class based on its performance sensitivity to the amount of space assigned in the LLC. This is crucial to decide on the share of the total cache space to be allotted to the application, as well as to determine what co-runners in the workload (if any) must be assigned to the same cache partition [133].

The sampling mode is inspired by the technique proposed in [58], which operates as follows. Two non-overlapping complementary cache partitions covering the entire LLC space are created; the first one, referred to as the *sampling partition*, is reserved for the application that triggered the transition into sampling mode, and the other one is devoted to the remaining applications. To determine the application class –based on the classification criteria presented in Section 6.2– the value of var-

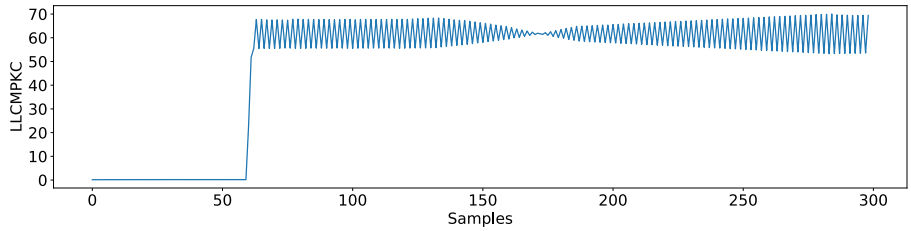


Figure 6.4: LLCMPKC captured at the beginning of the execution of `fotonik3d`.

ious hardware events (i.e., number of instructions retired, cycles and LLC misses) is gathered with PMCs as we vary the size of the sampling partition. Notably, for sensitive applications we also obtain the slowdown curve, which is required to create partitions for these applications, as depicted in Algorithm 6.1. Once the sampling process terminates, LFOC transitions back into the normal operating mode described earlier.

In the original approach [58], the size of the first partition is varied from the number of ways minus one to 1, whereas the size of the other partition (complementary) increases accordingly. This full sweep is required by the dynamic version of the KPart clustering approach [58], which relies on the ability to accurately determine the IPC and the number of LLC Misses Per Kilo Instructions (LLCMPKI) for each way count and for *every application* in the workload. We observed that this approach introduces substantial overheads due to the fact that the cache assignment enforced during the sampling mode is typically suboptimal. The sampling application receives a progressively smaller amount of cache space, while the remaining applications share a increasingly bigger cluster. This usually leads to performance/fairness degradation especially when cache sensitive applications and streaming programs are included in the workload.

To overcome these shortcomings, LFOC immediately puts a stop to the sampling process –performed in the opposite direction (i.e. the size of sampling partition increases gradually rather than decreasing)– in scenarios where varying the size of the sampling partition further provides no useful information to the clustering algorithm. Firstly, when the LLC miss rate falls below a certain low threshold, performance does not increase much when allotting more cache space to the application, so we expect IPC values –used to construct slowdown tables– to remain very close beyond that point. Secondly, streaming applications typically exhibit a very low increase in performance when granting more cache space to them. In these scenarios, LFOC interrupts the sampling process and proceeds to determine the application class. In practice, to successfully identify many streaming and light sharing applications –whose slowdown curves are not needed by LFOC– only a few way counts must be explored. When the sampling process is cancelled (due to the first criterion) for a sensitive application, LFOC uses the last IPC sample gathered to approximate the performance with higher way counts, which is necessary to build the entire slowdown table.

Because an application may exhibit different program phases at runtime, the initial classification may not be representative throughout the execution. For example,

Figure 6.4 shows the behavior of the LLCMPKC metric of the streaming `fotonik3d` application over time, where a short light-sharing phase precedes the streaming behavior that the program exhibits for the vast majority of the execution. Failing to determine application classes accurately could lead to suboptimal cache-partitioning for certain time periods, and hence to unfairness. Triggering the sampling mode periodically helps to mitigate this issue, but, unfortunately, it backfires by introducing substantial overheads.

To determine application classes at runtime in a lightweight manner, LFOC triggers a transition into the sampling mode only in the event that the application class has likely changed. To this end, the OS continuously monitors for each application the value of the LLCMPKC metric and the fraction of pipeline stall cycles incurred due to long-latency memory accesses [1], and leverages a few heuristics to capture class changes. In particular, a class change is signalled for a light sharing application if it enters a memory-intensive phase, namely, the average LLCMPKC measured over the last five monitoring periods exceeds a `high_threshold` (10 in our experimental setting, as reported in Table 6.1 for streaming-like behavior) or the average fraction of long-latency memory-access stalls is greater than 25%. This approach filters out spikes in the aforementioned metrics while effectively identifies memory-intensive phases. Conversely, for streaming programs, which LFOC typically assigns to cache clusters consisting of one way, the sampling mode is engaged if its average LLCMPKC falls below a `low_threshold` (defined as 30% of `high_threshold`). Finally, for sensitive applications, LFOC associates a *critical size*, defined as the amount of cache space where the slowdown falls below 5%. The *critical size* is determined during the last sampling period triggered by the application. Essentially, a class change is signaled for sensitive applications when they enter a stable non-memory intensive phase (inverse of the criterion presented earlier for light-sharing applications) for effective cache allocations[2] smaller than the critical size, or when the average LLCMPKC is higher than `high_threshold` for an amount of cache space bigger than the critical size.

6.4. Experiments

To assess the effectiveness of our OS-level cache-clustering approach we implemented it in the Linux kernel v4.9.160. For the experiments we used the *Skylake* platform. The processor of this platform integrates an 11-way 27.5MB last level (L3) cache that supports way-partitioning; each core features a 64KB L1 cache and a 1MB L2 cache (private levels). More details of this platform can be found in Chapter 3

On this platform we carried out a experimental comparison of LFOC with the stock Linux kernel –it does not partition the LLC–, and with the Dunn [171] and KPart [58] cache-partitioning policies, specifically designed to optimize fairness and

[1] approximated via the `STALLS_L2_MISS` performance counter event, also used in [171].

[2] The amount of cache space used by an application is gathered by leveraging the Intel Cache Monitoring Technology.

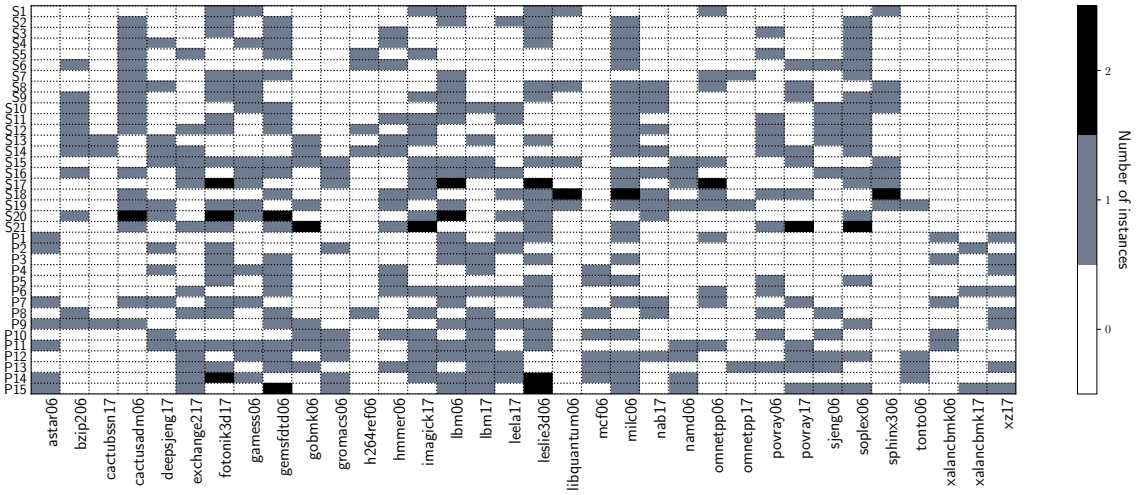


Figure 6.5: Multiprogram workloads used for our experiments. Each matrix cell indicates the number of instances of a benchmark (x-axis) in a workload (y-axis)..

system throughput respectively. To perform a fair comparison with previous approaches we used a similar methodology as that described in the corresponding articles [171, 58]. Essentially, we conduct experiments with HPC multiprogram workloads consisting of a mix of single-threaded benchmarks from SPEC CPU, and run each program for a fixed number of instructions (150 billion instructions in our setting). Specifically, we ensure that all applications in the mix are started simultaneously, and when one of them completes the corresponding instructions, the program is restarted repeatedly until the longest application in the set completes three times. We then measure unfairness and STP (throughput), by using the geometric mean of the completion times for each program.

Figure 6.5 depicts the composition of the 36 randomly generated workloads we used in our experiments, which are made of benchmarks from the SPEC CPU2006 and CPU2017 suites. Note that we selected applications from both suites to experiment with a wider range of streaming and cache-sensitive programs, as most benchmarks in both suites exhibit a light sharing, cache-insensitive execution profile on *Skylake*. This is caused in part due to the coarse granularity of the cache partitions we can create on this system: the smallest partition is as big as 2.5MB. We considered workloads of 8, 12 and 16 applications each, so as to analyze the impact that the workload size has on the fairness improvement achieved by each approach.

In this section we first evaluate the effectiveness of the cache-clustering algorithms associated with the KPart, Dunn and LFOC policies. We then proceed to analyze how well dynamic clustering techniques deal with the time-changing behavior of the applications in different workloads.

6.4.1. Evaluation of Clustering Algorithms

Our goal is to measure the degree of fairness and throughput delivered by a certain clustering strategy alone (i.e. how applications are grouped into shared or separate

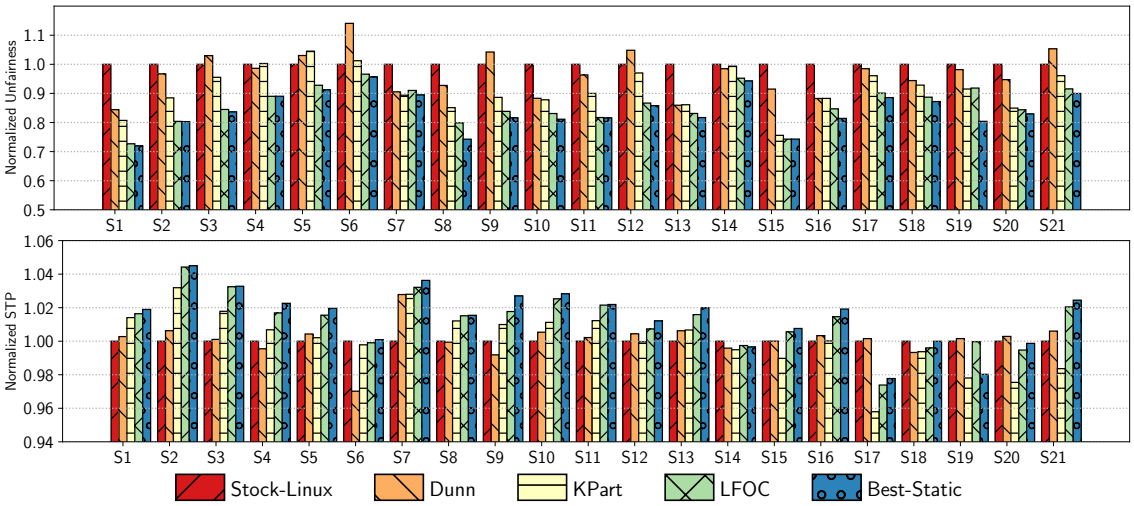


Figure 6.6: Normalized unfairness and STP values obtained by the static version of the various clustering algorithms.

clusters according to their runtime properties) putting aside the associated overheads due to algorithm execution, performance monitoring and cache allocation. We do account for these overheads in the experiments of the next section.

To assess the effectiveness of each clustering algorithm, we consider workloads consisting of applications whose behavior falls in a clear class (cache sensitive, streaming or light sharing) for the vast majority of the execution (Si workloads in Figure 6.5). For the analysis, we implemented the clustering algorithms used by KPart, Dunn and LFOC on top of the PBBCache simulator, which accepts as input the average value of various performance metrics gathered offline for different cache sizes. To conduct the corresponding experiments, we launch the simulator prior to the execution of each workload to retrieve the cache-partitions and application-to-partition mappings imposed by a certain clustering strategy. Then, we enforce the corresponding cache partitions on a per-process manner from user-space, using the PMCTrack tool [160], and proceed to launch the workload, which will use the same *static* cache configuration throughout the execution. For comparison purposes, we have also gathered the results of an ideal cache-clustering policy, referred to as Best-Static, which establishes the cache-partitions and application-to-cluster mappings based on the optimal fairness solution determined by the simulator.

Figure 6.6 shows the degree of unfairness and throughput delivered by the different clustering strategies; the values have been normalized to the results of Stock-Linux (no cache partitioning). The results reveal that the Dunn approach, designed to optimize fairness, exhibits a non-uniform behavior across workloads; for some program mixes it is capable to reduce unfairness up to 15.5% , but for others it causes substantial fairness degradation (by a factor of up to 1.14x) relative to Stock-Linux. We found that this is due to its exclusive reliance on the `STALLS_L2_MISS` performance event; the higher the value of this event, the higher the number of cache ways allotted by Dunn to the application [171]. More specifically, we observed that some streaming (aggressor) cache-insensitive applications, such as `GemsFDTD` or `fotonik3d` exhibit high values of this event, as their performance is greatly af-

fected by memory accesses. These applications can be mapped together to the same (or overlapping) cache partitions with highly sensitive programs, such as `soplex` or `omnetpp`, for which the event reaches similar figures, leading to performance degradation and unfairness. Based on this insight, we conclude that using the `STALLS_L2_MISS` event alone is not enough to drive fairness-aware partitioning policies.

We also observe that KPart's clustering algorithm, designed to optimize throughput, brings modest throughput gains[3] in the workloads we explored (up to 3%). However, this approach does bring substantial unfairness reductions (8.6% on average). Nonetheless, we observe that LFOC's simpler and more lightweight partitioning algorithm provide substantially better fairness than KPart for the vast majority of the workloads (up to 27.3%, and 14% on average relative to Stock Linux). At the same time, LFOC achieves higher throughput than KPart across the board, and performs in a close range (1.8% on average) of the *Best Static* approach (our approximation to the optimal policy in these workload scenarios).

6.4.2. Study of the dynamic policies

For the evaluation in this section we used our OS-level implementation of LFOC, and also created a user-level implementation of Dunn, as it was originally proposed [171] as a user-level cache-clustering policy. A good property of Dunn is the fact that it only requires the continuous monitorization of the `STALLS_L2_MISS` performance event for the various applications over time. The simplicity of the Dunn approach stands in contrast to the higher complexity of KPart, which relies on the ability to accurately gather a substantial amount of performance information online for each application (LLCMPKI and IPC values for every possible cache-way count) in order to apply the clustering algorithm.

Table 6.2: Average execution time (in ms) of the KPart and LFOC algorithms

#Apps.	4	5	6	7	8	9	10	11
LFOC	0.00151	0.00154	0.00163	0.00174	0.00174	0.00182	0.00191	0.00216
KPart	0.51800	0.79600	1.21800	1.48100	2.01200	2.74200	3.32000	4.14000

In an attempt to evaluate the dynamic version of KPart –referred to as KPart-Dynaway [58]– we considered the user-level implementation created by the authors [57]. Unfortunately, this implementation, which consists of roughly 4K lines of C++ code and makes intensive use of the Armadillo linear algebra library, was specifically tailored to the hardware platform where the authors conducted the experiments [58], and makes numerous assumptions that do not apply to our experimental setting (e.g., the number of cache ways should be no smaller than the number of applications in the workload). Due to these platform-specific assumptions and other issues – as yet unidentified, the execution of KPart-Dynaway crashes shortly

[3]In the original paper [171], the authors report a 24% average increase in throughput on a different platform, with workloads whose composition was not disclosed.

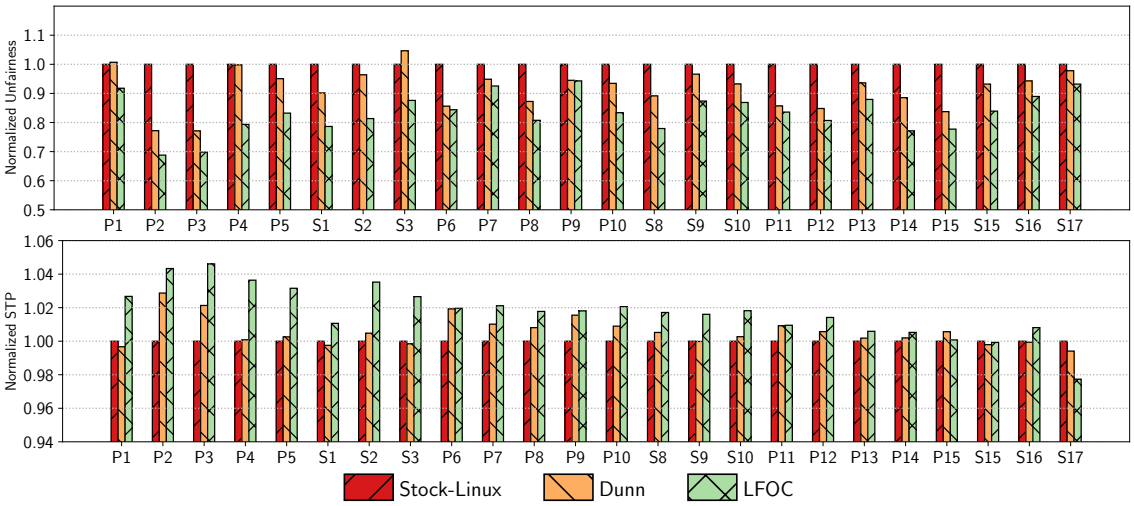


Figure 6.7: Normalized unfairness and STP values delivered by the dynamic cache clustering approaches

after the partitioning algorithm is executed for the first time, preventing us from launching any of the workloads we considered for the evaluation. We leave for future work the adaptation of this somewhat complex implementation for our platform, and its complete evaluation. Nevertheless, to highlight the enormous difference between the complexity of KPart's partitioning algorithm and the one used by our approach for different number of applications, Table 6.2 shows the execution time for both algorithms (compiled with aggressive optimizations) for different workload sizes. We were able to gather that information from KPart's implementation by instrumenting the code of the partitioning algorithm that completes successfully (for workloads with less than twelve applications) right before the program's crash. As it can be seen, LFOC's execution time (2μs) is up to three orders of magnitude smaller than KPart's, which can take over 4ms to complete for 11 applications (slightly longer than the default timer tick in the Linux kernel). As we showed in the previous section, this increased complexity does not enable KPart to provide better fairness than our lightweight approach.

In our OS-level implementation of LFOC we sample performance counters every 100M instructions during the normal operation mode (see Section 6.3) and every 10M instructions during the sampling mode. Using a shorter instruction window for the sampling mode makes it possible to reduce the time required to complete sampling. Notably, we observed that in most cases full cache-way sweeps are not required during LFOC's sampling mode as the partitioning algorithm (as explained in Section 6.3) does not require detailed per-way metrics for all applications, as opposed to KPart. In our experiments, the partitioning algorithm for both Dunn and LFOC is executed every 500ms, as this is the setting used in the original evaluation of the Dunn approach [171].

Figure 6.7 shows the normalized unfairness and throughput values delivered by the Dunn and LFOC (dynamic) approaches for different workloads. Note that in this case, we considered additional program mixes (Pi workloads) that include applications such as `xz`, `astar`, `mcf` or `xalancbmk`, which exhibit distinct long-

term program phases with varying degree of memory intensity. Some of these applications go through highly cache sensitive phases, so Stock-Linux delivers higher unfairness values in these scenarios. That is the reason why Dunn exhibits a slightly fairer behavior under these circumstances relative to the scenario considered in the previous section. Yet, LFOC is capable to provide better throughput than Dunn, and improves fairness over Dunn across the board (up to 20.5% for P4, and 9% on average). With respect to Stock-Linux, LFOC reduces unfairness by 16.7% on average.

6.5. Conclusions

In this chapter we have presented LFOC, an OS-level cache-clustering approach that leverages cache-partitioning support in Intel-CAT enabled multicore processors to improve fairness while maintaining acceptable throughput. LFOC classifies applications into three categories according to their degree of memory-intensity and cache sensitivity, and ensures that streaming aggressor benchmarks are confined to small cache partitions so they are isolated from cache-sensitive benchmarks, which are assigned an amount of cache space in accordance to its sensitivity. In doing so, LFOC tries to mimic the behavior of the optimal cache clustering solution, which we approximated by means of a simulator. We implemented LFOC in the Linux kernel and assessed its effectiveness on a commercial multicore platform featuring an Intel Skylake processor. Our experiments reveal that LFOC is able to deliver an average 16.7% fairness improvement relative to stock Linux. At the same time, LFOC clearly outperforms two other existing cache-clustering approaches, one of which was specifically designed to deliver fairness [171].

Key aspects of LFOC are its lightweight clustering algorithm, the online heuristics it leverages to classify applications online, and its ability to fairly share the space on the LLC among applications by using limited monitoring information, which can be obtained at runtime without collecting performance data for every possible way count.

Chapter 7

Conclusions

Chip multicore processors (CMPs) have spread across a wide range of general-purpose computing systems. Cores in a CMP typically share a last-level cache (LLC) and other memory-related resources with the remaining cores, such as a DRAM controller or an interconnection network. This may induce co-running applications to intensively compete for their use, leading to substantial and uneven performance degradation. Despite the increasing core counts and cache sizes of every new technological iteration, the issue of shared-resource contention, which has been present since the advent of multicores, still poses a big challenge.

Previous research [205, 134] has proved that the system software can be extremely useful in mitigating contention. This thesis tackles the negative effects of shared-resource contention at the OS-level by taking advantage of scheduling and resource management techniques, leveraging the specific features available on different multicore platforms.

One of the main proposals of this thesis focuses on Asymmetric single-ISA Multicore Processors (AMPs), which combine high-performance big cores with low-power small cores. On these platforms contention not only comes from applications competing for *big* core usage but also from the shared resources. Previous research proposed asymmetry-aware schedulers that strived to optimize system throughput, fairness and energy efficiency; nonetheless, none of these existing proposals take shared-resource contention effects into account. To fill this gap, this thesis proposes CAMPS, a contention-aware fair scheduler for AMPs that does not require special hardware extensions or platform-specific prediction models to function. Our scheduler accurately tracks the progress that the various threads of a workload make when running on different core types, and enforces fairness by evening out the progress across threads throughout the execution. Our main contributions to contention-conscious scheduling for AMPs are as follows:

- We conduct an exhaustive experimental study to assess the impact on contention on real asymmetric multicore platforms that integrate high-performance out-of-order big cores and low-power in-order small cores, where cores of the

same type are grouped into clusters sharing a last-level cache (LLC). An important insight from our analysis is the fact that factoring in the contention that arises on high-performance big-core clusters is fundamental when trying to improve fairness and system throughput on AMPs.

- We devised a novel runtime mechanism to predict the slowdown that a thread in the workload experiences as it runs on the various cores of an AMP. Specifically, our scheduler approximates the current slowdown by monitoring various runtime metrics via performance monitoring counters (PMCs), and by comparing that information with the thread's past history gathered in low contention scenarios.
- For the experimental evaluation of CAMPS, we employed the Intel QuickIA prototype [43] as well as commercial ARM-based asymmetric multicore platforms [19, 78]. We performed an extensive experimental comparison with previously proposed asymmetry-aware schemes [111, 164, 167]. The experiments reveal that CAMPS outperforms the state-of-the-art fairness-aware scheme for AMPs –the ACFS scheduler [167]– in both fairness and throughput. Besides, we prove that CAMPS ensures consistent performance and repeatable completion times for a wide range of application types, and also show that this repeatability and consistency is not present under the default Linux scheduler (i.e. CFS) even with the *HMP* patch, which specifically extends CFS for commercial AMP platforms.

In this thesis we also investigated how to reduce the negative impact of shared-resource contention by leveraging cache-partitioning techniques. After years of research on cache-partitioning [133], hardware manufacturers like Intel [142] or AMD [12] finally added hardware support for partitioning the LLC in some of their commercial multicore processors. To advance the state of the art on cache-partitioning, we first tried to devise means to determine the optimal solution for strict cache-partitioning (i.e., assigning a separate LLC partition to each application) and for cache-clustering (i.e., creating separate LLC partitions, which may be shared by several applications) considering different optimization objectives. Previous work has pointed out that cache clustering (aka. partition sharing [32]) proves more effective than strict cache partitioning as the number of applications increases [171], and especially on current CMPs, which support a reduced number of coarse-grained cache partitions (i.e., in the order of megabytes). While partitioning the cache optimally with separate LLC partitions for each application is an NP-hard problem [133], determining the optimal cache-clustering solution adds a new layer of complexity. After all, under cache-clustering a decision must be made not only on how to best group applications into clusters, but also on how to optimally distribute LLC space across clusters. This complexity coupled with the exponential growth of the search space that increases with greater workload sizes, makes its very difficult to obtain the optimal solution in a reasonable amount time.

To make it possible to efficiently determine the optimal solution (for both strict cache-partitioning and cache-clustering), and to enable rapid prototyping and evaluation of cache partitioning and clustering policies, we developed the PBBCache

simulator. This open-source simulator [68] is equipped with parallel algorithms that make it possible to obtain the optimal cache-partitioning and cache-clustering solution for different optimization objectives. The ability to quickly compare novel approaches with existing cache-partitioning schemes or with the optimal solution is a key feature of PBBCache. In designing and evaluating the simulator we made the following contributions:

- PBBCache is equipped with a slowdown-prediction model enabling to determine the performance degradation that an application suffers due to cache-sharing and memory-bandwidth contention. To approximate bandwidth contention for a certain distribution of cache space across applications in a workload, we extended the probabilistic model proposed in [136] to factor in the degree of sensitivity to bandwidth contention that an application has, which does not depend on its effective bandwidth consumption only.
- The optimal strict-cache partitioning problem (for both system throughput and fairness optimization) constitutes a mixed-integer non-linear optimization problem. To determine the optimal solution in a reasonable amount of time, PBBCache implements a novel parallel branch-and-bound (B&B) solver that effectively distributes the computation across cores on one or multiple computing nodes. To the best of our knowledge, our proposal is the first parallel approach to solve the optimal cache-partitioning problem by factoring in both cache-sharing and memory-bandwidth contention. Notably, we found that state-of-the-art non-linear solvers [6, 2, 1, 5] fail to provide a solution to this optimization problem.
- A key design aspect of our B&B algorithm is the mechanism used to break down the work to be done in parallel into tasks (referred to as *subnodes*) with a similar computational complexity, which provides good scalability. The effectiveness of the bounding functions we devised for various optimization objectives, also contributes to the success of the B&B approach.
- To evaluate the effectiveness and accuracy of PBBCache we implemented existing partitioning policies [153, 58, 195] on top of it, and compared the results it provides with the actual figures observed on commercial hardware equipped with cache partitioning support. Moreover, to assess the performance and scalability of the parallel B&B algorithm we conducted experiments using single-node and multi-node machine configurations.

After carrying out an extensive study of the optimal cache-clustering solution on different scenarios, we proceeded to design LFOC, a novel OS-level cache-clustering policy that leverages cache-partitioning support to improve fairness while maintaining acceptable throughput. LFOC classifies applications into differently categories according to their degree of memory-intensity and cache sensitivity. To deliver fairness, LFOC ensures that applications that may be potentially aggressive to others in terms of LLC contention are confined in small cache partitions, thus ensuring that they are effectively isolated from cache-sensitive programs. In designing, implementing and evaluating LFOC we made the following contributions.

- We conducted a profound analysis of the optimal (fairness-wise) cache-clustering solution obtained for a wide range of multi-program workloads with PBBCache. This analysis enabled us to identify key patterns observed in the optimal solution, which were crucial to guide the design of LFOC's cache-clustering algorithm. Specifically, one of the key aspects to enforce fairness is to effectively identify contentious cache-insensitive (aka *streaming*) applications and confine them in a reduced set of small LLC partitions.
- LFOC leverages a lightweight online mechanism to approximate the degree of cache sensitivity of an application that avoids costly periodic monitoring operations (i.e. measuring application performance for all possible LLC sizes [58]), whenever possible.
- We implemented LFOC in the Linux kernel and evaluated it on a real system featuring an Intel Skylake processor. Our experiments reveal that LFOC is able to deliver an average 16.7% fairness improvement relative to stock Linux. At the same time, LFOC clearly outperforms two other existing cache-clustering approaches, one of which was specifically designed to deliver fairness [171].

7.1. Future work

This thesis has proposed multiple techniques that contribute to mitigating shared-resource contention effects on multicore processors. However, there are still many challenges that remain unsolved and so they represent promising avenues for future research. Some of the most relevant ones are as follows:

- **Mitigating shared-resource contention effects in upcoming architectures**. Due to their outstanding energy efficiency, mobile devices have recently pushed forward asymmetric multicore configurations and some of them are seeping through to the desktop market, especially on the laptop segment. There is still work to do on this topic. Of special attention would be to assess the impact of contention on recent AMP platforms, such as the Intel Lakefield SoC (System on Chip) or the Apple M1 SoC [17]. Likewise, designing cache-clustering approaches for recent AMD server processors –such as EPYC Rome and Milan– would be also very interesting. On these processors multiple logically independent LLCs –each one shared by a different subset of cores– are present on the same chip [13]. Although contention-aware thread-to-LLC mapping schemes have been proposed [29, 205, 69], no previous proposal have yet addressed fairness-optimized cache-clustering coupled with effective thread-to-LLC mapping. Optimizing fairness on these AMD processors is a more complex problem than on single-LLC CMPs–the ones considered in the thesis–, so this constitutes a promising research avenue.
- **Dealing with bandwidth contention**. In distributing LLC space, our LFOC cache-clustering strategy prioritizes cache-sensitive applications over

streaming aggressor programs, which are granted a small portion of the LLC. We observed that in this context, if the workload aggregate bandwidth consumption is very high, these streaming applications may begin to suffer significant performance degradation as a result of bandwidth contention. A promising way to improve our cache partitioning strategy would be to factor in the effects of applying per-application memory-bandwidth consumption caps via hardware mechanisms, such as the recent Intel Memory Bandwidth Allocation (MBA) technology. Using this technology would enable to have greater control over bandwidth contention, and would make it possible to design policies that simultaneously exploit the Intel CAT and MBA features.

- **Other optimization goals**. The parallel algorithms enabling to determine the optimal solution in PBBCache for both strict cache-partitioning and cache-clustering, can be configured to optimize many different optimization objectives. The ability to explore the optimal solution for other optimization targets beyond fairness – such as maximizing system throughput or delivering better energy efficiency – creates opportunities to guide the design of additional cache-clustering policies that strive to improve these aspects. At the same time, this PBBCache support would also enable to assess the impact that optimizing a specific metric has on other metrics, thus opening up opportunities for research on multiobjective optimization.

- **Improving the performance of the parallel methods to determine the optimal solution under both strict cache partitioning and cache-clustering**. It would also be interesting to explore alternative ways to determine optimal solutions more efficiently in PBBCache. This would allow to study the behavior of the optimal for bigger problem sizes, which becomes something necessary on future platforms, where the core count is expected to increase and so will the workload size. Of special attention is the case of the optimal cache clustering problem, whose parallel method in PBBCache currently requires a complete enumeration of all possible ways to group applications into clusters. Designing an efficient method to reach a near-optimal solution in this context would be extremely helpful.

- **Add cache-partitioning support for multithreaded applications**. Multithreaded applications introduce another level of complexity to cache-partitioning; a joint decision should made on both the inter-application LLC-space distribution, and the intra-application LLC partitioning – whether to partition the LLC among threads from the same application or not, and, if so, how to distribute the cache space among these threads. In this context, some ideas from our proposed cache-clustering method could be applied. However, it would be also necessary to conduct an extensive study to assess the potential rewards of applying cache-partitioning to multithreaded programs.

Resumen en Español

Los procesadores multinúcleo o CMPs (*Chip Multicore Processors*) son actualmente la arquitectura más usada por la mayoría de sistemas de computación de propósito general, y muy probablemente se mantendrán en esa posición dominante en el futuro cercano. Los avances tecnológicos han permitido integrar progresivamente en el mismo chip más cores y aumentar los tamaños de los distintos niveles de cache. No obstante, la contención de recursos compartidos en CMPs –presente desde la aparición de estas arquitecturas– todavía representa un reto importante que afrontar. Los cores en un CMP comparten en la mayor parte de los diseños una cache de último nivel o LLC (*Last-Level Cache*) y otros recursos, como el controlador de DRAM o una red de interconexión. La existencia de dichos recursos compartidos provoca en ocasiones que cuando se ejecutan dos o más aplicaciones simultáneamente en el sistema, se produzca una degradación sustancial y potencialmente desigual del rendimiento entre aplicaciones.

En investigaciones previas [205, 134] se ha demostrado que el software del sistema puede ser de gran ayuda a la hora de mitigar estos problemas. En esta tesis se han desarrollado diversas estrategias a nivel del sistema operativo –implementadas en el kernel Linux– para lidiar de forma eficaz con los efectos adversos de la contención y optimizar la justicia global del sistema.

La primera aportación importante de la tesis ha sido un planificador consciente de la contención para procesadores multicore asimétricos o AMPs (*Asymmetric Multicore Processors*). Estas arquitecturas combinan en la misma plataforma cores complejos (*big*) de alto rendimiento, con cores más simples (*small*) de bajo consumo, funcionando todos ellos bajo un mismo repertorio de instrucciones. Aunque en las últimas décadas se han propuesto diversas técnicas para mitigar los efectos adversos de la contención, ninguna de ellas es específica para AMPs. En estas arquitecturas, a diferencia de los multicore tradicionales, una aplicación puede sufrir una degradación sustancial del rendimiento, relativa a su ejecución aislada, debido al efecto combinado de la competición por el uso de cores *big* y de la contención de recursos compartidos. En esta tesis proponemos CAMPS, un planificador consciente de la contención que optimiza justicia en sistemas asimétricos. Las principales contribuciones de la tesis en este ámbito son las siguientes:

- Se ha realizado un estudio experimental extenso donde se demuestra la importancia de considerar la contención en los cores big de alto rendimiento para mejorar la justicia y el rendimiento global en AMPs.

- Se ha diseñado un mecanismo ligero y eficiente –basado en el uso de contadores hardware– que permite registrar el progreso de cada aplicación en los distintos tipos de core y bajo distintos niveles de contención. Con esta información, CAMPS distribuye de forma justa el tiempo de uso de los cores big entre las distintas aplicaciones.
- Se ha realizado una evaluación experimental exhaustiva utilizando distintas arquitecturas asimétricas de ARM [19, 78], así como el prototipo experimental QuickIA de Intel [43]. Dicha evaluación demuestra que CAMPS es capaz de gestionar cargas de trabajo con distintos tipos de aplicaciones más eficientemente que el planificador por defecto de Linux (CFS), e incluso que su variante más extendida específica para dispositivos móviles con procesadores asimétricos (parche *HMP*). Además, CAMPS es capaz de superar también a todas las estrategias que se habían propuesto hasta la fecha en la literatura para mejorar justicia en AMPs, como es el caso del planificador ACFS [167].

A nivel micro-arquitectónico se han propuesto diversos mecanismos para mitigar los problemas de contención derivados de la interferencia entre aplicaciones. Entre los que han venido despertando mayor interés investigador destacan los mecanismos y estrategias de particionado de cache [133]. Los primeros trabajos de particionado exploraron técnicas en el software de gestión de memoria basadas en el coloreado de páginas de memoria y pusieron de manifiesto las ventajas de reducir la interferencia por el uso de los niveles compartidos de memoria. Posteriormente se han ido diseñando y evaluando diferentes mecanismos hardware adicionales para soportar el particionado. Este interés ha llegado ya a los propios fabricantes, que han empezando a incorporar dicho soporte en algunos de los procesadores comerciales recientes, especialmente para los procesadores del segmento de servidores [142, 12].

En esta tesis hemos explorado este soporte hardware y hemos analizado la utilidad de estrategias de *particionado basado en clusters*, donde se permite el uso compartido por varias aplicaciones de una misma partición. Propuestas anteriores [171] han demostrado que las ventajas del particionado en clusters son mayores al aumentar el número de aplicaciones en la carga de trabajo. Además, estas ventajas son especialmente notables en los sistemas actuales, que soportan la creación de un número reducido de particiones de grano grueso (en el orden de megabytes). No obstante, encontrar la solución optima de particionado –lo cual es esencial para guiar el diseño de nuevas estrategias– es un problema difícil de resolver. De hecho, encontrar una solución óptima, incluso para políticas de particionado estricto –donde cada partición es asignada a una sola aplicación– es ya un problema NP-hard [133]. Permitir la compartición de una o varias particiones entre varias aplicaciones (particionado en clusters), amplia aún más las dimensiones del espacio de búsqueda, y dificulta aún más encontrar soluciones óptimas al problema.

Tras estudiar algunas de las propuestas existentes [153, 58], nos preguntamos cuánto se aproximaban a las estrategias óptimas. Para poder responder a esa pregunta desarrollamos PBBCache, un simulador paralelo de código abierto que permite agilizar el proceso de diseño y evaluación de políticas de particionado de cache, tanto estrictas como en clusters.

Las principales contribuciones de PBBCache son las siguientes:

- PBBCache es capaz de estimar la degradación del rendimiento relativa (*slowdown*) que sufren diferentes aplicaciones que se ejecutan simultáneamente bajo una estrategia de particionado determinada. Dicha estimación tiene en cuenta no solo la degradación proveniente por la contención de la LLC, sino también por la contención por el ancho de banda con memoria. Para tener en cuenta este segundo factor se extendió un modelo de estimación no lineal propuesto en la literatura [136].
- Para reducir los tiempos de simulación, PBBCache implementa la primera estrategia paralela de memoria distribuida que permite determinar de forma efectiva la solución óptima de particionado estricto para diferentes objetivos de optimización. La eficiencia paralela de dicha estrategia es satisfactoria, consiguiéndose buenos resultados incluso en máquinas con múltiples nodos.
- Para validar la precisión de PBBCache se implementaron varias estrategias de particionado propuestas recientemente [153, 58, 195], y se compararon los resultados con datos reales obtenidos en procesadores comerciales de Intel con soporte hardware para particionado. Los resultados obtenidos han sido satisfactorios.

Gracias a PBBCache y a un análisis intensivo de las soluciones óptimas proporcionadas por este simulador, fuimos capaces de guiar el diseño de LFOC, una nueva estrategia de particionado en cluster a nivel del sistema operativo que optimiza la justicia manteniendo un buen rendimiento global en la plataforma. LFOC clasifica en tiempo de ejecución las aplicaciones en varias categorías en base a su grado de sensibilidad a la contención cache, y a su demanda efectiva de espacio en ésta. Para mejorar la justicia, LFOC prioriza las aplicaciones de tipo *cache-sensitive* en el reparto de espacio en la LLC, y confina aquellas aplicaciones que tienen un comportamiendo de tipo *streaming* (aplicaciones agresoras) en un conjunto reducido de particiones pequeñas de cache. Las principales contribuciones asociadas al diseño, implementación y evaluación de LFOC son las siguientes:

- Se ha realizado un exhaustivo análisis basado en simulación del problema de particionado de cache en clusters. Este estudio revela que la clave para optimizar la justicia recae en (1) identificar aquellas aplicaciones insensibles a la contención de la LLC pero que a su vez logran atesorar mucho espacio en ésta, y (2) aislar estas aplicaciones en un conjunto reducido de particiones de la LLC de pequeño tamaño. De este modo puede dedicarse una mayor fracción de espacio en la LLC a las aplicaciones *cache sensitive*.
- LFOC emplea un mecanismo ligero que permite aproximar en tiempo de ejecución el grado de sensibilidad que una aplicación tiene al compartir la LLC con otras. Con este mecanismo se reduce la frecuencia de las costosas operaciones de monitorización empleadas en otras estrategias de particionado [58], que requieren recopilar múltiples métricas de cada aplicación para distintos

tamaños de cache realizando un barrido de todas las asignaciones de vías posibles.

- Implementamos LFOC en el kernel Linux y lo evaluamos en un sistema que incorpora un procesador Intel Skylake con soporte de particionado de cache. Nuestro análisis experimental revela que LFOC mejora la justicia en un 16.7% en media, con respecto a Linux (sin particionado de la LLC), y además ofrece un mayor grado de justicia que la estrategia de particionado en clusters propuesta más recientemente para optimizar la justicia [171].

Bibliography

[1] BARON: A general purpose global optimization software package. http://archimedes.cheme.cmu.edu/?q=baron. Accessed: 2019-07-15.

[2] BONMIN: Basic open-source nonlinear mixed integer programming. https://www.coin-or.org/Bonmin/. Accessed: 2019-07-17.

[3] JyNI – jython native interface:compatibility/wish list. https://jyni.org/#compatibility-wish-list. Accessed: 2019-11-21.

[4] Linux test project. https://github.com/linux-test-project/ltp.

[5] The NEOS server. https://neos-server.org/neos/. Accessed: 2019-07-18.

[6] SCIP: solving constraint integer programs. https://scip.zib.de/. Accessed: 2019-07-18.

[7] Zeromq: An open-source universal messaging library. https://zeromq.org/. Accessed: 2019-9-4.

[8] Intel® 64 and ia-32 architectures developer's manual: Vol. 3b, 2014. http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html.

[9] Intel® 64 and ia-32 architectures software developer's manual: Volume 3, 2018. https://www.intel.es/content/www/es/es/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html.

[10] IPyparallel. using IPython for parallel computing. https://ipyparallel.readthedocs.io/, 2018. Accessed: 2019-03-19.

[11] A. Alhammad and R. Pellizzoni. Trading cores for memory bandwidth in real-time systems. In *22nd Real-Time Embedded Tech. and Applications Symp. (RTAS 16)*, pages 1–11, April 2016.

[12] AMD. AMD64 technology platform quality of service extensions. Accessed: 2019-09-19.

[13] AMD. High Performance Computing: Tuning guide for AMD EPYC 7002 series processors. https://developer.amd.com/wp-content/resources/56827-1-0.pdf, 2020. Accessed: 2021-05-21.

[14] J. Herdrich Andrew. Introduction to memory bandwidth allocation, 2019. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-memory-bandwidth-allocation.html.

[15] Arunachalam Annamalai, Rance Rodrigues, Israel Koren, and Sandip Kundu. An opportunistic prediction-based thread scheduling to maximize throughput/watt in amps. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 63–72, 2013.

[16] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 298–309, 2005.

[17] Apple. Apple m1 chip. https://www.apple.com/mac/m1/, 2020. Accessed: 2020-12-5.

[18] ARM. Benefits of the big.LITTLE Architecture. http://www.arm.com/files/downloads/Benefits_of_the_big.LITTLE_architecture.pdf. Accessed: 2015-01-10.

[19] ARM. Juno platform. http://infocenter.arm.com/help/topic/com.arm.doc.subset.boards.juno/index.html. Accessed: 2017-3-9.

[20] R. Balasubramonian, D. Albones, A. Buyuktosunoglu, and S. Dwarkadas. Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures. In *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000*, pages 245–257, 2000.

[21] M. Banikazemi, D. Poff, and B. Abali. Pam: A novel performance/power aware meta-scheduler for multi-core systems. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[22] S. Barati and H. Hoffmann. Providing fairness in heterogeneous multicores with a predictive, adaptive scheduler. In *2016 Int'l Parallel and Distrib. Processing Symp. Workshops*, pages 38–49, 2016.

[23] David Beazley. Understanding the python GIL. In *In PyCON'10 Python Conference*, 2010.

[24] Michela Becchi and Patrick Crowley. Dynamic Thread Assignment on Heterogeneous Multiprocessor Architectures. In *3rd Int'l Conf. Computing Frontiers (CF 06)*, pages 29–40, 2006.

[25] N. Beckmann and D. Sanchez. Modeling cache performance beyond lru. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 225–236, March 2016.

[26] R. Bitirgen, E. Ipek, and J. F. Martinez. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 318–329, 2008.

[27] S. Blagodurov, A. Fedorova, S. Zhuravlev, and A. Kamali. A case for numa-aware contention management on multicore systems. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 557–558, 2010.

[28] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, December 2010.

[29] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. Contention-aware scheduling on multicore systems. *ACM Trans. Comput. Syst.*, 28(4):8:1–8:45, December 2010.

[30] M. T. Bohr and I. A. Young. CMOS scaling trends and beyond. *IEEE Micro*, 37(6):20–29, 2017.

[31] Mark T. Bohr, Robert S. Chau, Tahir Ghani, and Kaizad Mistry. The high-k solution. *IEEE Spectrum*, 44(10):29–35, 2007.

[32] Jacob Brock, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. Optimal cache partition-sharing. In *2015 44th International Conference on Parallel Processing*, pages 749–758, 2015.

[33] BSC. Paraver: a flexible performance analysis tool. `https://tools.bsc.es/paraver`, 2018. Accessed: 2019-03-19.

[34] T. Burd, N. Beck, S. White, M. Paraschou, N. Kalyanasundharam, G. Donley, A. Smith, L. Hewitt, and S. Naffziger. "Zeppelin": An soc for multichip architectures. *IEEE Journal of Solid-State Circuits*, 54(1):133–143, 2019.

[35] Doug Burger, James R. Goodman, and Alain Kägi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, page 78–89, New York, NY, USA, 1996. Association for Computing Machinery.

[36] Jorge Casas Hernán. Infraestructura de simulación para evaluación de estrategias de particionado de caché en procesadores equipados con la tecnología intel rdt. 2019.

[37] Calin Cascaval, Luiz DeRose, David A. Padua, and Daniel A. Reed. Compile-time based performance prediction. In Larry Carter and Jeanne Ferrante, editors, *Languages and Compilers for Parallel Computing*, pages 365–379, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[38] Stephen Cass and Parthasaradhi Bulusu. Interactive: The Top Programming Languages 2020. `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020`, 2020. Accessed: 2020-09-19.

[39] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13, 2016.

[40] Felipe Cerqueira, Arpan Gujarati, and Björn B. Brandenburg. Linux's processor affinity API, refined: Shifting real-time tasks towards higher schedulability. In *2014 IEEE Real-Time Systems Symposium*, pages 249–259, 2014.

[41] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In *HPCA '05*.

[42] X. E. Chen and T. M. Aamodt. A first-order fine-grained multithreaded throughput model. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 329–340, 2009.

[43] Nagabhushan Chitlur, Ganapati Srinivasa, Scott Hahn, P K Gupta, Dheeraj Reddy, David Koufaty, Paul Brett, Abirami Prabhakaran, Li Zhao, Nelson Ijih, Suchit Subhaschandra, Sabina Grover, Xiaowei Jiang, and Ravi Iyer. QuickIA: Exploring heterogeneous architectures on real prototypes. In *IEEE International Symposium on High-Performance Comp Architecture*, pages 1–8, 2012.

[44] Kallia Chronaki, Miquel Moretó, Marc Casas, Alejandro Rico, Rosa M. Badia, Eduard Ayguadé, and Mateo Valero. On the maturity of parallel applications for asymmetric multi-core processors. *Journal of Parallel and Distributed Computing*, 127:105–115, 2019.

[45] Kallia Chronaki, Alejandro Rico, Rosa M. Badia, Eduard Ayguadé, Jesús Labarta, and Mateo Valero. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, page 329–338, New York, NY, USA, 2015. Association for Computing Machinery.

[46] W. Cohen. Tuning programs with oprofile. *Wide Open Magazine*, 1:53–62, 2004.

[47] Cynthia A. Colinge, Jean-Pierre Colinge, and Isabelle Ferain. Multigate transistors as the future of classical metal-oxide-semiconductor field-effect transistors. *Nature*, 479:310–6, 11 2011.

[48] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. Accelerator-rich architectures: Opportunities and progresses. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[49] R. Courtland. Intel now packs 100 million transistors in each square millimeter. *IEEE Spectrum*, March 2017.

[50] Teodor Gabriel Crainic, Bertrand Le Cun, and Catherine Roucairol. Parallel branch-and-bound algorithms. *Parallel combinatorial optimization*, 1:1–28, 2006.

[51] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Scheduler-based dram energy management. In *Proceedings 2002 Design Automation Conference (IEEE Cat. No.02CH37324)*, pages 697–702, 2002.

[52] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, 1974.

[53] Jack Dongarra. Report on the sunway taihulight system. Tech Report University of Tennessee: UT-EECS-16-742, 2016. Accessed: 2020-09-24.

[54] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 335–346, New York, NY, USA, 2010. Association for Computing Machinery.

[55] Stefan Edelkamp and Stefan Schroedl. *Heuristic Search: Theory and Applications*. Morgan Kaufmann, 2012.

[56] D. Eklov, D. Black-Schaffer, and E. Hagersten. Statcc: A statistical cache contention model. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 551–552, 2010.

[57] N. El-Sayed et al. Source code of kpart. https://github.com/Nosayba/kpart, 2018. Accessed: 2019-02-20.

[58] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. KPart: A hybrid cache partitioning-sharing technique for commodity multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 104–117, 2018.

[59] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[60] Daniel Etiemble. 45-year cpu evolution: one law and two equations, 2018.

[61] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3):42–53, May 2008.

[62] X. Fan, Y. Sui, and J. Xue. Contention-aware scheduling for asymmetric multicore processors. In *Proc. 2015 Int'l Conf Parallel and Dist. Syst. (ICPADS 15)*, pages 742–751, Dec 2015.

[63] Josué Feliu, Julio Sahuquillo, Salvador Petit, and José Duato. Perf fair: A progress-aware scheduler to enhance performance and fairness in smt multicores. *IEEE Transactions on Computers*, 66(5):905–911, 2017.

[64] Matt Fleming. A survey of scheduler benchmarks. `https://lwn.net/Articles/725238/`, 2017. Accessed: 2018-01-20.

[65] R. C. Frye, K. L. Tai, M. Y. Lau, and T. J. Gabara. Trends in silicon-on-silicon multichip modules. *IEEE Design Test of Computers*, 10(4):8–17, 1993.

[66] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. Ginseng: Market-driven llc allocation. In *Proceedings of the 2016 USENIX Annual Technical Conference*, USENIX ATC '16, pages 295–308, 2016.

[67] François Galea and Bertrand Le Cun. Bob++: a framework for exact combinatorial optimization methods on parallel machines. In *International Conference High Performance Computing & Simulation*, pages 779–785, 2007.

[68] A. Garcia-Garcia, J. Casas, and J. C. Saez. PBBCache: A parallel branch-and-bound based cache-partitioning simulator. https://github.com/pbbcache/cachesim, 2019. Accessed: 2019-05-10.

[69] A. Garcia-Garcia, J. C. Saez, and M. Prieto-Matias. Contention-aware fair scheduling for asymmetric single-ISA multicore systems. *IEEE Transactions on Computers*, 67(12):1703–1719, Dec 2018.

[70] Adrian Garcia-Garcia, Juan Carlos Saez, Fernando Castro, and Manuel Prieto-Matias. LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[71] Adrian Garcia-Garcia, Juan Carlos Saez, José Luis Risco-Martin, and Manuel Prieto-Matias. PBBCache: An open-source parallel simulator for rapid prototyping and evaluation of cache-partitioning and cache-clustering policies. *Journal of Computational Science*, 42:101102, 2020.

[72] Bernard Gendron and Teodor Gabriel Crainic. Parallel branch-and-branch algorithms: Survey and synthesis. *Operations research*, 42(6):1042–1066, 1994.

[73] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens. A GPU-based branch-and-bound algorithm using integer–vector–matrix data structure. *Parallel Computing*, 59:119 – 139, 2016. Theory and Practice of Irregular Applications.

[74] Samuel Greengard. The future of semiconductors. *Commun. ACM*, 60(3):18–20, February 2017.

[75] L. Gwennap. TSMC 7nm approaches Intel's prowess. *Microprocessor Report.The Linley Group*, January 2017.

[76] Linley Gwennap. TSMC ships first 5nm processors. `https://www.linleygroup.com/mpr/article.php?id=12347`, 2012. Accessed: 2020-09-28.

[77] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. Exploiting heterogeneity for tail latency and energy efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, page 625–638, New York, NY, USA, 2017. Association for Computing Machinery.

[78] Hardkernel. Odroid XU4 board. `http://odroid.com/dokuwiki/doku.php?id=en:odroid-xu4`, 2016. Accessed: 2016-6-22.

[79] A. Hartstein and Thomas R. Puzak. The optimum pipeline depth for a microprocessor. *SIGARCH Comput. Archit. News*, 30(2):7–13, May 2002.

[80] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[81] John L. Hennessy and David A. Patterson. A new golden age for computer architecture. *Commun. ACM*, 62(2):48–60, January 2019.

[82] Juan F. R. Herrera, José M. G. Salmerón, Eligius M. T. Hendrix, Rafael Asenjo, and Leocadio G. Casado. On parallel branch and bound frameworks for global optimization. *Journal of Global Optimization*, 69(3):547–560, Nov 2017.

[83] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *IEEE Computer*, 41(7):33–38, 2008.

[84] Glenn Hinton, Dave Sager, Mike Upton, Darrell Boggs, Desktop Platforms Group, and Intel Corp. The microarchitecture of the pentium 4 processor. *Intel Technology Journal*, 5, 2001.

[85] M. Horowitz. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[86] Lisa R. Hsu, Steven K. Reinhardt, Ravishankar Iyer, and Srihari Makineni. Communist, utilitarian, and capitalist cache policies on cmps: caches as a shared resource. PACT '06, pages 13–22, New York, NY, USA, 2006. ACM.

[87] Ibrahim Hur and C. Lin. Adaptive history-based memory schedulers. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 343–354, 2004.

[88] Intel. https://ark.intel.com/content/www/es/es/ark/products/64590/intel-xeon-processor-e5-2650-20m-cache-2-00-ghz-8-00-gt-s-intel-qpi.html, 2012. Accessed: 2021-09-21.

[89] Intel. https://ark.intel.com/content/www/es/es/ark/products/81057/intel-xeon-processor-e5-2695-v3-35m-cache-2-30-ghz.html, 2014. Accessed: 2021-09-21.

[90] Intel. https://ark.intel.com/content/www/es/es/ark/products/92986/intel-xeon-processor-e5-2620-v4-20m-cache-2-10-ghz.html, 2016. Accessed: 2021-09-21.

[91] Intel. https://ark.intel.com/content/www/es/es/ark/products/120476/intel-xeon-gold-6138-processor-27-5m-cache-2-00-ghz.html, 2017. Accessed: 2021-09-21.

[92] A. Jaleel, W. Hasenplaugh, M. Qureshi, J. Sebot, S. Steely, and J. Emer. Adaptive insertion policies for managing shared caches. In *2008 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 208–219, 2008.

[93] S. Jarp, R. Jurga, and A. Nowak. Perfmon2: a leap forward in performance monitoring. *Journal of Physics: Conference Series*, 119:042017, 2008.

[94] Yunlian Jiang, Xipeng Shen, Jie Chen, and Rahul Tripathi. Analysis and approximation of optimal co-scheduling on chip multiprocessors. pages 220–229, 01 2008.

[95] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*, pages 367–378, 2008.

[96] Ivan Jibaja, Ting Cao, Stephen M. Blackburn, and Kathryn S. McKinley. Portable performance on asymmetric multicore processors. In *2016 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 24–35, 2016.

[97] Jose Joao, Aater Suleman, Onur Mutlu, and Yale Patt. Bottleneck identification and scheduling in multithreaded applications. *ACM SIGPLAN Notices*, 47:223–234, 03 2012.

[98] José A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt. Utility-based acceleration of multithreaded applications on asymmetric CMPs. *SIGARCH Comput. Archit. News*, 41(3):154–165, June 2013.

[99] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. A domain-specific architecture for deep neural networks. *Commun. ACM*, 61(9):50–59, August 2018.

[100] D. Kanter. Samsung 10nm takes density crown. *Microprocessor Report.The Linley Group*, January 2017.

[101] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict qos for latency-critical workloads. *SIGARCH Comput. Archit. News*, 42(1):729–742, February 2014.

[102] Samira Khan, Alaa R. Alameldeen, Chris Wilkerson, Onur Mutluy, and Daniel A. Jimenezz. Improving cache performance using read-write partitioning. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 452–463, 2014.

[103] S. Khushu and W. Gomes. Lakefield: Hybrid cores in 3d package. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–20, 2019.

[104] Changdae Kim and Jaehyuk Huh. Fairness-oriented OS scheduling support for multicore systems. In *Proc. 2016 Int'l Conf. Supercomputing (ICS 16)*, pages 29:1–29:12, 2016.

[105] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings. 13th International Conference on Parallel Architecture and Compilation Techniques, 2004. PACT 2004.*, pages 111–122, 2004.

[106] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[107] Y. G. Kim, M. Kim, and S. W. Chung. Enhancing energy efficiency of multimedia applications in heterogeneous mobile multi-core processors. *IEEE Transactions on Computers*, 66(11):1878–1889, Nov 2017.

[108] Rob Knauerhase, Paul Brett, Barbara Hohlt, Tong Li, and Scott Hahn. Using os observations to improve performance in multicore systems. *IEEE Micro*, 28(3):54–66, May 2008.

[109] J. U. Knickerbocker, F. L. Pompeo, A. F. Tai, D. L. Thomas, R. D. Weekly, M. G. Nealon, H. C. Hamel, A. Haridass, J. N. Humenik, R. A. Shelleman, S. N. Reddy, K. M. Prettyman, B. V. Fasano, S. K. Ray, T. E. Lombardi, K. C. Marston, P. A. Coico, P. J. Brofman, L. S. Goldmann, D. L. Edwards, J. A. Zitz, S. Iruvanti, S. L. Shinde, and H. P. Longworth. An advanced multichip module (mcm) for high-performance unix servers. *IBM Journal of Research and Development*, 46(6):779–804, 2002.

[110] Masaaki Kondo, Hiroshi Sasaki, and Hiroshi Nakamura. Improving fairness, throughput and energy-efficiency on a chip multiprocessor through dvfs. *SIGARCH Computer Architecture News*, 35:31–38, 03 2007.

[111] David Koufaty, Dheeraj Reddy, and Scott Hahn. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 125–138, New York, NY, USA, 2010. ACM.

[112] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, pages 81–92, 2003.

[113] R. Kumar, D. M. Tullsen, and N. P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *2006 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 23–32, 2006.

[114] R. Kumar, D.M. Tullsen, P. Ranganathan, N.P. Jouppi, and K.I. Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings. 31st Annual International Symposium on Computer Architecture, 2004.*, pages 64–75, 2004.

[115] Viren Kumar and Alexandra Fedorova. Towards better performance per watt in virtual environments on asymmetric single-ISA multi-core systems. *SIGOPS Oper. Syst. Rev.*, 43(3):105–109, 2009.

[116] Kun Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in smt processors. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software. ISPASS.*, pages 164–171, 2001.

[117] Antony Leather. Intel's surprise ryzen killer: Alder lake hybrid processors offer biggest performance leap in 14 years. `https://www.forbes.com/sites/antonyleather/2020/08/13/intels-surprise-ryzen-killer-alder-lake-hybrid-processors-offer-biggest-performance#42ed29f24f5f`, 2020. Accessed: 2020-08-20.

[118] L. Li, H. Liu, H. Wang, T. Liu, and W. Li. A parallel algorithm for game tree search using gpgpu. *IEEE Transactions on Parallel and Distributed Systems*, 26(8):2114–2127, Aug 2015.

[119] Tong Li, Dan Baumberger, David A. Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *In Proc. of SC '07*, pages 53:1–53:11, 2007.

[120] Tong Li, Paul Brett, Rob Knauerhase, David Koufaty, Dheeraj Reddy, and Scott Hahn. Operating system support for overlapping-ISA heterogeneous multi-core architectures. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[121] Tong Li, Alvin R. Lebeck, and Daniel J. Sorin. Spin detection hardware for improved management of multithreaded systems. *IEEE Trans. Parallel Distrib. Syst.*, 17(6):508–521, June 2006.

[122] C. Liu, Anand Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 176–185, 2004.

[123] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 450–462, New York, NY, USA, 2015. Association for Computing Machinery.

[124] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 450–462, New York, NY, USA, 2015. Association for Computing Machinery.

[125] G. H. Loh. 3d-stacked memory architectures for multi-core processors. In *2008 International Symposium on Computer Architecture*, pages 453–464, 2008.

[126] R. Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[127] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic shared cache management (prism). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 428–439, 2012.

[128] T. Marinakis and I. Anagnostopoulos. Performance and fairness improvement on cmps considering bandwidth and cache utilization. *IEEE Computer Architecture Letters*, 18(2):1–4, 2019.

[129] Nikola Markovic, Daniel Nemirovsky, Osman Unsal, Mateo Valero, and Adrian Cristal. Thread lock section-aware scheduling on asymmetric single-ISA multi-core. *IEEE Computer Architecture Letters*, 14(2):160–163, 2015.

[130] 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.

[131] Tarek Menouer. Solving combinatorial problems using a parallel framework. *Journal of Parallel and Distributed Computing*, 112:140 – 153, 2018.

[132] Andreas Merkel, Jan Stoess, and Frank Bellosa. Resource-conscious scheduling for energy efficiency on multicore processors. In *Proceedings of EuroSys 10*, pages 153–166, Paris, France, 13-16 April 2010. ACM, New York.

[133] S. Mittal. A survey of techniques for cache partitioning in multicore processors. *ACM Comput. Surv.*, 50(2):27:1–27:39, May 2017.

[134] Sparsh Mittal. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput. Surv.*, 48(3):45:1–45:38, February 2016.

[135] Jeffrey C. Mogul, Jayaram Mudigonda, Nathan Binkert, Parthasarathy Ranganathan, and Vanish Talwar. Using Asymmetric Single-ISA CMPs to Save Energy on Operating Systems. *IEEE Micro*, 28(3):26–41, 2008.

[136] Tomer Y. Morad, Noam Shalev, Idit Keidar, Avinoam Kolodny, and Uri C. Weiser. EFS: Energy-friendly scheduler for memory bandwidth constrained systems. *Journal of Parallel and Distributed Computing*, 95:3–14, 2016. Special Issue on Energy Efficient Multi-Core and Many-Core Systems, Part I.

[137] A. Mukkara, N. Beckmann, and D. Sanchez. Whirlpool: Improving dynamic cache management with static data classification. In *Proc. of the 21st Int'l Conf. on Arch. Support for Programming Lang. and Oper. Syst.*, ASPLOS '16, pages 113–127, 2016.

[138] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO 07)*, pages 146–160, 2007.

[139] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *2008 International Symposium on Computer Architecture*, pages 63–74, 2008.

[140] Mohannad Nabelsee, Anselm Busse, Helge Parzyjegla, and Gero Mühl. Load-aware scheduling for heterogeneous multi-core systems. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, page 1844–1851, New York, NY, USA, 2016. Association for Computing Machinery.

[141] K. Nguyen. Intel's cache monitoring technology software-visible interfaces. `https://software.intel.com/en-us/blogs/2014/12/11/intel-s-cache-monitoring-technology-software\-visible-interfaces`, 2014. Accessed: 2015-02-10.

[142] K. Nguyen. Introduction to cache allocation technology in the intel xeon processor e5 v4 family. `https://software.intel.com/en-us/articles/introduction-to-cache-allocation-technology`, 2016. Accessed: 2019-03-20.

[143] K. Nguyen. Introduction to the intel(r) resource director technology features in intel(r) xeon(r) processors e5 v4. `https://software.intel.com/en-us/articles/introduction-to-the-intel-resource-director-technology-features-in-intel-xeon-proce` 2016. Accessed: 2016-12-27.

[144] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[145] Els Parton and Peter Verheyen. Strained silicon — the key to sub-45 nm CMOS. *III-Vs Review*, 19(3):28 – 31, 2006.

[146] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta. Cellss: Making it easier to program the cell broadband engine processor. *IBM Journal of Research and Development*, 51(5):593–604, 2007.

[147] Perf. Perf wiki tutorial on perf. `https://perf.wiki.kernel.org/index.php`, 2015. Accessed: 2015-01-20.

[148] Tiago Pessoa, Jan Gmys, Francisco de Carvalho-Junior, Nouredine Melab, and Daniel Tuyttens. GPU-accelerated backtracking using cuda dynamic parallelism. *Concurrency and Computation: Practice and Experience*, 30(9):e4374, 2018.

[149] Vinicius Petrucci, Michael A. Laurenzano, John Doherty, Yunqi Zhang, Daniel Mossé, Jason Mars, and Lingjia Tang. Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 246–258, 2015.

[150] Vinicius Petrucci, Orlando Loques, Daniel Mossé, Rami Melhem, Neven Abou Gazala, and Sameh Gobriel. Energy-efficient thread assignment optimization for heterogeneous multicore systems. *ACM Trans. Embed. Comput. Syst.*, 14(1):15:1–15:26, January 2015.

[151] Adrián Pousa. *Optimización de rendimiento, justicia y consumo energético en sistemas multicore asimétricos mediante planificación*. PhD thesis, Universidad Nacional de La Plata, 2017.

[152] Mihai Pricopi, Thannirmalai Somu Muthukaruppan, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Power-performance modeling on asymmetric multi-cores. In *Proceedings of the 2013 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '13, pages 15:1–15:10, Piscataway, NJ, USA, 2013. IEEE Press.

[153] M.K. Qureshi and Y.N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of MICRO 06*, pages 423–432, 2006.

[154] Morten Rasmussen. Task placement for heterogeneous MP systems. `https://lwn.net/Articles/517250/`, 2012. Accessed: 2016-07-06.

[155] Dheeraj Reddy, David Koufaty, Paul Brett, and Scott Hahn. Bridging functional heterogeneity in multicore architectures. *SIGOPS Oper. Syst. Rev.*, 45(1):21–33, February 2011.

[156] Rakesh Reddy and Peter Petrov. Eliminating inter-process cache interference through cache reconfigurability for real-time and low-power embedded multi-tasking systems. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES

'07, page 198–207, New York, NY, USA, 2007. Association for Computing Machinery.

[157] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, page 128–138, New York, NY, USA, 2000. Association for Computing Machinery.

[158] Ganesh T. S. Memory frequency scaling on intel's skull canyon nuc - an investigation. `https://www.anandtech.com/show/10602/memory-frequency-scaling-on-skull-canyon`, 2016. Accessed: 2020-05-20.

[159] J C Saez, A Pousa, A E de Giusti, and M Prieto-Matias. On the interplay between throughput, fairness and energy efficiency on asymmetric multicore processors. *The Computer Journal*, 61(1):74–94, 2018.

[160] J. C. Saez, A. Pousa, R. Rodriíguez-Rodriíguez, F. Castro, and M. Prieto-Matias. PMCTrack: Delivering performance monitoring counter support to the os scheduler. *The Computer Journal*, 60(1):60–85, 2017.

[161] J.C. Saez, J.I. Gomez, and M. Prieto. Improving priority enforcement via non-work-conserving scheduling. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 99–106, 2008.

[162] Juan Carlos Saez, Jorge Casas, Abel Serrano, Roberto Rodríguez-Rodríguez, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matias. An OS-oriented performance monitoring tool for multicore systems. In *Proc. of Euro-Par 2015: Parallel Processing Workshops*, pages 697–709, Cham, 2015. Springer International Publishing.

[163] Juan Carlos Saez, Fernando Castro, and Manuel Prieto-Matias. Enabling performance portability of data-parallel openmp applications on asymmetric multicore processors. In *49th International Conference on Parallel Processing - ICPP*, ICPP '20, New York, NY, USA, 2020. Association for Computing Machinery.

[164] Juan Carlos Saez, Alexandra Fedorova, David Koufaty, and Manuel Prieto. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. *ACM Trans. Comput. Syst.*, 30(2), April 2012.

[165] Juan Carlos Saez, Alexandra Fedorova, Manuel Prieto, and Hugo Vegas. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*, CF '10, page 31–40, New York, NY, USA, 2010. Association for Computing Machinery.

[166] Juan Carlos Saez, A. Pousa, F. Castro, D. Chaver, and M. Prieto-Matias. Exploring the throughput-fairness trade-off on asymmetric multicore systems.

In *Proceedings of Euro-Par 14: Parallel Processing Workshops*, pages 326–337, Porto, Portugal, 25-26 August 2014. Springer-Verlag, Berlin.

[167] Juan Carlos Saez, Adrian Pousa, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matias. Towards completely fair scheduling on asymmetric single-isa multicore processors. *Journal of Parallel and Distributed Computing*, 102:115–131, 2017.

[168] Juan Carlos Saez, Daniel Shelepov, Alexandra Fedorova, and Manuel Prieto. Leveraging workload diversity through OS scheduling to maximize performance on single-ISA heterogeneous multicore systems. *J. Parallel Distrib. Comput.*, 71:114–131, January 2011.

[169] Juan Carlos Sáez Alcaide. *Planificación de procesos en sistemas multicore asimétricos = Thread Scheduling on Asymmetric Multicore Systems*. PhD thesis, Madrid, May 2011. Tesis de la Universidad Complutense de Madrid, Facultad de Informática, Departamento de Arquitectura de Computadores y Automática (Arquitectura y Tecnología de Computadores e Ingeniería de Sistemas y Automática), leída el 22-02-2011.

[170] A. Scolari, D.B. Bartolini, and M.D. Santambrogio. A software cache partitioning system for hash-based caches. *ACM Trans. Archit. Code Optim.*, 13(4):57:1–57:24, December 2016.

[171] Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit, and María E. Gómez. Application clustering policies to address system fairness with Intel's cache allocation technology. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 194–205, 2017.

[172] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. HASS: A scheduler for heterogeneous multicore systems. *SIGOPS Oper. Syst. Rev.*, 43(2):66–75, April 2009.

[173] T. Sherwood, B. Calder, and J. Emer. Reducing cache misses using hardware and software page placement. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, pages 155–164, 1999.

[174] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreaded processor. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, page 234–244, New York, NY, USA, 2000. Association for Computing Machinery.

[175] S. Srikantaiah, R. Das, A. K. Mishra, C. R. Das, and M. Kandemir. A case for integrated processor-cache partitioning in chip multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, 2009.

[176] Shekhar Srikantaiah, Mahmut Kandemir, and Mary Jane Irwin. Adaptive set pinning: Managing shared caches in chip multiprocessors. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 135–144, New York, NY, USA, 2008. Association for Computing Machinery.

[177] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, Sep. 1992.

[178] L. T. Su, S. Naffziger, and M. Papermaster. Multi-chip technologies to unleash computing performance gains over the next decade. In *2017 IEEE International Electron Devices Meeting (IEDM)*, pages 1.1.1–1.1.8, 2017.

[179] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 62–75, 2015.

[180] G. E. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings Eighth International Symposium on High Performance Computer Architecture*, pages 117–128, 2002.

[181] Kai Tian, Yunlian Jiang, and Xipeng Shen. A study on optimally coscheduling jobs of different lengths on chip multiprocessors. pages 41–50, 01 2009.

[182] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. Fairness-aware scheduling on single-isa heterogeneous multi-cores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 177–187, 2013.

[183] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, pages 213–224, 2012.

[184] Pablo Viana, Ann Gordon-Ross, Edna Barros, and Frank Vahid. A table-based method for single-pass cache optimization. In *Proceedings of the 18th ACM Great Lakes Symposium on VLSI*, GLSVLSI '08, page 71–76, New York, NY, USA, 2008. Association for Computing Machinery.

[185] R. Wang and L. Chen. Futility scaling: High-associativity cache partitioning. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pages 356–367, 2014.

[186] S. Wang and L. Wang. Thread-associative memory for multicore and multithreaded computing. In *ISLPED'06 Proceedings of the 2006 International Symposium on Low Power Electronics and Design*, pages 139–142, 2006.

[187] X. Wang, S. Chen, J. Setter, and J. F. Martínez. Swap: Effective fine-grain management of shared last-level caches with minimum hardware support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 121–132, 2017.

[188] Jonathan Weinberg and Allan Edward Snavely. Accurate memory signatures and synthetic address traces for hpc applications. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, ICS '08, page 36–45, New York, NY, USA, 2008. Association for Computing Machinery.

[189] X. Xiang, B. Bao, C. Ding, and Y. Gao. Linear-time modeling of program working set in shared cache. In *2011 International Conference on Parallel Architectures and Compilation Techniques*, pages 350–360, 2011.

[190] Yuejian Xie and Gabriel Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. volume 37, pages 174–183, 01 2009.

[191] Di Xu, Chenggang Wu, and Pen-Chung Yew. On mitigating memory bandwidth contention through bandwidth-aware scheduling. In *19th Int'l Conf. Parallel Arch. Compilation Tech. (PACT 10)*, pages 237–248, 2010.

[192] Di Xu, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Zhenjiang Wang. Providing fairness on shared-memory multiprocessors via process scheduling. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 295–306, New York, NY, USA, 2012. Association for Computing Machinery.

[193] Ying Ye, Richard West, Zhuoqun Cheng, and Ye Li. COLORIS: A dynamic cache partitioning system using page coloring. In *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 381–392, 2014.

[194] Ying Ye, Richard West, Jingyi Zhang, and Zhuoqun Cheng. Maracas: A real-time multicore vcpu scheduling framework. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 179–190, 2016.

[195] C. Yu and P. Petrov. Off-chip memory bandwidth minimization through cache partitioning for multi-core platforms. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 132–137, 2010.

[196] Heechul Yun, Renato Mancuso, Zheng-Pei Wu, and Rodolfo Pellizzoni. PALLOC: Dram bank-aware memory allocator for performance isolation on multicore platforms. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 155–166, 2014.

[197] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memory bandwidth management for efficient performance isolation in multicore platforms. *IEEE Transactions on Computers*, 65(2):562–576, 2016.

[198] X. Zhang, S. Dwarkadas, and K. Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 89–102, 2009.

[199] Ying Zhang, Lide Duan, Bin Li, Lu Peng, and Srinivasan Sadagopan. Cross-architecture prediction based scheduling for energy efficient execution on single-ISA heterogeneous chip-multiprocessors. *Microprocess. Microsyst.*, 39(4):271–285, June 2015.

[200] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst.*, 31(6), August 2009.

[201] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. *SIGARCH Comput. Archit. News*, 32(5):177–188, October 2004.

[202] H. Zhu and M. Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *Proc. of the 21st Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 33–47, 2016.

[203] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing Cache Contention in Multicore Processors Via Scheduling. In *15th Int'l Conf. Architectural Support Programming Lang. and Oper. Syst. (ASPLOS 10)*, pages 129–142, 2010.

[204] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. volume 45, pages 129–142, 03 2010.

[205] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1), December 2012.

[206] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1), December 2012.