UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Informática
de Sistemas y Computadores

Contention-Aware Scheduling for SMT Multicore Processors

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

Doctor of Philosophy
(Computer Engineering)

Author

Josué Feliu Pérez

Advisors

Prof. Julio Sahuquillo Borrás
Prof. Salvador V. Petit Martí

February 2017

Doctoral Committee

- Prof. J. Angel Gregorio Monasterio
  Universidad de Cantabria, Santander, Spain

- Prof. Manuel E. Acacio Sánchez
  Universidad de Murcia, Murcia, Spain

- Prof. Lieven Eeckhout
  Universiteit Gent, Ghent, Belgium

Agraïments

Xicotetes decisions, aparentment intranscendents, acaben marcant una vida. En la meua, una d'aquestes decisions fou entrar al despatx de Julio un 15 de juliol de 2010. Simplement buscava informació per a fer un projecte final de carrera i poc més que vaig eixir amb un doctorat planificat.

En primer lloc aquesta tesi té molt que agrair a Julio i Salva. Per confiar en mi en aquelles primeres reunions de 2010 quan no era més que un estudiant d'enginyeria informàtica. Per tot el que m'han ensenyat sobre l'arquitectura de computadors i la investigació. I per tot el treball intensíssim que ha portat aquesta tesi i, principalment, les publicacions en les que es basa. Moltes hores extres de treball repartides entre nits, caps de setmana i vacances. És una obvietat que sense ells aquesta tesi no existiria.

En segon lloc vuic agrair la col·laboració a la resta de professors que han contribuït al desenvolupament d'aquesta tesi. A José Duato, per la seva col·laboració en part dels articles, així com per ajudar-nos a obtenir la beca amb la que he realitzat aquest doctorat. A Stijn Eyerman, per l'acollida tan bona que vaig tenir en la Ghent University, així com per la seva enorme contribució en la publicació més important d'aquesta tesi. I a Lieven Eeckhout, pels seus precisos comentaris i col·laboració en part del treball realitzat.

A continuació vuic agrair a tots els companys que he tingut durant el desenvolupament de la tesi al grup d'arquitectures paral·leles, destacant aquells amb els que he compartit més dinars i debats. Amb un ambient de treball tan bo es rendeix molt millor. També agrair als companys de la Ghent University que em feren sentir com a casa.

Per últim, però no menys important, a la meua família pel suport que m'han donat. En especial a ma mare, per exigir sempre arribar un poquet més lluny, i a la qui segurament també he d'agrair-li totes les hores que va dedicar als meus estudis quan jo era ben menut. A la meua novia Amparo, per recolzar-me sempre (incloent les dos estades a Ghent), així com per aguantar-me en els moment de major estrés, on ha sigut complicat separar recerca i vida. I finalment als meus amics, que sense saber-ho han contribuït a que puga desconnectar el cap de setmana i començar cada dilluns amb forces.

Contents

List of Figures

List of Tables

Abbreviations and Acronyms

ANTT	**A**verage **N**ormalized **T**urnaround **T**ime
CMP	**C**hip **M**ulticore **P**rocessor
CPI	**C**ycles **P**er **I**nstruction
DRAM	**D**ynamic **R**andom-**A**ccess **M**emory
IABW	**I**deal **A**verage **B**and**W**idth
IPC	**I**nstructions **P**er **C**ycle
LLC	**L**ast-**L**evel **C**ache
L1	First-level (cache)
L2	Second-level (cache)
L3	Third-level (cache)
MPKI	**M**isses **P**er **K**ilo-**I**nstruction
NUMA	**N**on-**U**niform **M**emory **A**ccess
SMT	**S**imultaneous **M**ulti**T**hreading **M**emory
ST	**S**ingle-**T**hreaded
PA	**P**rocess **A**llocation
PS	**P**rocess **S**election
OATR	**O**nline **A**verage **T**ransaction **R**ate
OS	**O**perating **S**ystem
QoS	**Q**uality **o**f **S**ervice
ROB	**R**e**O**rder **B**uffer
STP	**S**ystem **T**hroughput
TLB	**T**ransaction **L**ookaside **B**uffer
UMA	**U**niform **M**emory **A**ccess

Abstract

The recent multicore era and the incoming manycore/manythread era generate a lot of challenges for computer scientists going from productive parallel programming, over network congestion avoidance and intelligent power management, to circuit design issues. The ultimate goal is to squeeze out as much performance as possible while limiting power and energy consumption and guaranteeing a reliable execution. The increasing number of hardware contexts of current and future systems makes the scheduler an important component to achieve this goal, as there is often a combinatorial amount of different ways to schedule the distinct threads or applications, each with a different performance due to the inter-application interference. Picking an optimal schedule can result in substantial performance gains.

This thesis deals with inter-application interference, covering the problems this fact causes on performance and fairness on actual machines. The study starts with single-threaded multicore processors (Intel Xeon X3320), follows with simultaneous multi-threading (SMT) multicores supporting up to two threads per core (Intel Xeon E5645), and goes to the most highly threaded per-core processor that has ever been built (IBM POWER8). The dissertation analyzes the main contention points of each experimental platform and proposes scheduling algorithms that tackle the interference arising at each contention point to improve the system throughput and fairness.

First we analyze contention through the memory hierarchy of current multicore processors. The performed studies reveal high performance degradation due to contention on main memory and any shared cache the processors implement. To mitigate such contention, we propose different bandwidth-aware scheduling algorithms with the key idea of balancing the memory accesses through the workload execution time and the cache requests among the different caches at each cache level.

The high interference that different applications suffer when running simultaneously on the same SMT core, however, does not only affect performance, but can also compromise system fairness. In this dissertation, we also analyze fairness in current SMT multicores.

To improve system fairness, we design progress-aware scheduling algorithms that estimate, at runtime, how the processes progress, which allows to improve system fairness by prioritizing the processes with lower accumulated progress.

Finally, this dissertation tackles inter-application contention in the IBM POWER8 system with a symbiotic scheduler that addresses overall SMT interference. The symbiotic scheduler uses an SMT interference model, based on CPI stacks, that estimates the slowdown of any combination of applications if they are scheduled on the same SMT core. The number of possible schedules, however, grows too fast with the number of applications and makes unfeasible to explore all possible combinations. To overcome this issue, the symbiotic scheduler models the scheduling problem as a graph problem, which allows finding the optimal schedule in reasonable time.

In summary, this thesis addresses contention in the shared resources of the memory hierarchy and SMT cores of multicore processors. We identify the main contention points of three systems with different architectures and propose scheduling algorithms to tackle contention at these points. The evaluation on the real systems shows the benefits of the proposed algorithms. The symbiotic scheduler improves system throughput by 6.7% over Linux. Regarding fairness, the proposed progress-aware scheduler reduces Linux unfairness to a third. Besides, since the proposed algorithm are completely software-based, they could be incorporated as scheduling policies in Linux and used in small-scale servers to achieve the mentioned benefits.

Resumen

La actual era multinúcleo y la futura era *manycore/manythread* generan grandes retos en el área de la computación incluyendo, entre otros, la programación paralela productiva o la gestión eficiente de la energía. El último objetivo es alcanzar las mayores prestaciones limitando el consumo energético y garantizando una ejecución confiable. El incremento del número de contextos hardware de los sistemas hace que el planificador se convierta en un componente importante para lograr este objetivo debido a que existen múltiples formas diferentes de planificar las aplicaciones, cada una con distintas prestaciones debido a las interferencias que se producen entre las aplicaciones. Seleccionar la planificación óptima puede proporcionar importantes mejoras de prestaciones.

Esta tesis se ocupa de las interferencias entre aplicaciones, cubriendo los problemas que causan en las prestaciones y equidad de los sistemas actuales. El estudio empieza con procesadores multinúcleo monohilo (Intel Xeon X3320), sigue con multinúcleos con soporte para la ejecución simultanea (SMT) de dos hilos (Intel Xeon E5645), y llega al procesador que actualmente soporta un mayor número de hilos por núcleo (IBM POWER8). La disertación analiza los principales puntos de contención en cada plataforma y propone algoritmos de planificación que mitigan las interferencias que se generan en cada uno de ellos para mejorar la productividad y equidad de los sistemas.

En primer lugar, analizamos la contención a lo largo de la jerarquía de memoria. Los estudios realizados revelan la alta degradación de prestaciones provocada por la contención en memoria principal y en cualquier cache compartida. Para mitigar esta contención, proponemos diversos algoritmos de planificación cuya idea principal es distribuir los accesos a memoria a lo largo del tiempo de ejecución de la carga y las peticiones a las caches entre las diferentes caches compartidas en cada nivel.

Las altas interferencias que sufren las aplicaciones que se ejecutan simultáneamente en un núcleo SMT, sin embargo, no solo afectan a las prestaciones, sino que también pueden comprometer la equidad del sistema. En esta tesis, también abordamos la equidad en los actuales multinúcleos SMT. Para mejorarla, diseñamos algoritmos de planificación que

estiman el progreso de las aplicaciones en tiempo de ejecución, lo que permite priorizar los procesos con menor progreso acumulado para reducir la inequidad.

Finalmente, la tesis se centra en la contención entre aplicaciones en el sistema IBM POWER8 con un planificador simbiótico que aborda la contención en todo el núcleo SMT. El planificador simbiótico utiliza un modelo de interferencia basado en pilas de CPI que predice las prestaciones para la ejecución de cualquier combinación de aplicaciones en un núcleo SMT. El número de posibles planificaciones, no obstante, crece muy rápido y hace inviable explorar todas las posibles combinaciones. Por ello, el problema de planificación se modela como un problema de teoría de grafos, lo que permite obtener la planificación óptima en un tiempo razonable.

En resumen, esta tesis aborda la contención en los recursos compartidos en la jerarquía de memoria y el núcleo SMT de los procesadores multinúcleo. Identificamos los principales puntos de contención de tres sistemas con diferentes arquitecturas y proponemos algoritmos de planificación para mitigar esta contención. La evaluación en sistemas reales muestra las mejoras proporcionados por los algoritmos propuestos. Así, el planificador simbiótico mejora la productividad, en promedio, un 6.7 % con respecto a Linux. En cuanto a la equidad, el planificador que considera el progreso consigue reducir la inequidad de Linux a una tercera parte. Además, dado que los algoritmos propuestos son completamente software, podrían incorporarse como políticas de planificación en Linux y usarse en servidores a pequeña escala para obtener los beneficios descritos.

Resum

L'actual era multinucli i la futura era *manycore/manythread* generen grans reptes en l'àrea de la computació incloent, entre d'altres, la programació paral·lela productiva o la gestió eficient de l'energia. L'últim objectiu és assolir les majors prestacions limitant el consum energètic i garantint una execució confiable. L'increment del número de contextos hardware dels sistemes fa que el planificador es convertisca en un component important per assolir aquest objectiu donat que existeixen múltiples formes distintes de planificar les aplicacions, cadascuna amb unes prestacions diferents degut a les interferències que es produeixen entre les aplicacions. Seleccionar la planificació òptima pot donar lloc a millores importants de les prestacions.

Aquesta tesi s'ocupa de les interferències entre aplicacions, cobrint els problemes que provoquen en les prestacions i l'equitat dels sistemes actuals. L'estudi comença amb processadors multinucli monofil (Intel Xeon X3320), segueix amb multinuclis amb suport per a l'execució simultània (SMT) de dos fils (Intel Xeon E5645), i arriba al processador que actualment suporta un major nombre de fils per nucli (IBM POWER8). Aquesta dissertació analitza els principals punts de contenció en cada plataforma i proposa algoritmes de planificació que aborden les interferències que es generen en cadascun d'ells per a millorar la productivitat i l'equitat dels sistemes.

En primer lloc, estudiem la contenció al llarg de la jerarquia de memòria en els processadors multinucli. Els estudis realitzats revelen l'alta degradació de prestacions provocada per la contenció en memòria principal i en qualsevol cache compartida. Per a mitigar la contenció, proposem diversos algoritmes de planificació amb la idea principal de distribuir els accessos a memòria al llarg del temps d'execució de la càrrega i les peticions a les caches entre les diferents caches compartides en cada nivell.

Les altes interferències que sofreixen las aplicacions que s'executen simultàniament en un nucli SMT, no obstant, no sols afecten a las prestacions, sinó que també poden comprometre l'equitat del sistema. En aquesta tesi, també abordem l'equitat en els actuals multinuclis SMT. Per a millorar-la, dissenyem algoritmes de planificació que

estimen el progrés de les aplicacions en temps d'execució, el que permet prioritzar els processos amb menor progrés acumulat para a reduir la inequitat.

Finalment, la tesi es centra en la contenció entre aplicacions en el sistema IBM POWER8 amb un planificador simbiòtic que aborda la contenció en tot el nucli SMT. El planificador simbiòtic utilitza un model d'interferència basat en piles de CPI que prediu les prestacions per a l'execució de qualsevol combinació d'aplicacions en un nucli SMT. El nombre de possibles planificacions, no obstant, creix molt ràpid i fa inviable explorar totes les possibles combinacions. Per resoldre aquest contratemps, el problema de planificació es modela com un problema de teoria de grafs, la qual cosa permet obtenir la planificació òptima en un temps raonable.

En resum, aquesta tesi aborda la contenció en els recursos compartits en la jerarquia de memòria i el nucli SMT dels processadors multinucli. Identifiquem els principals punts de contenció de tres sistemes amb diferents arquitectures i proposem algoritmes de planificació per a mitigar aquesta contenció. L'avaluació en sistemes reals mostra les millores proporcionades pels algoritmes proposats. Així, el planificador simbiòtic millora la productivitat una mitjana del 6.7% respecte a Linux. Pel que fa a l'equitat, el planificador que considera el progrés aconsegueix reduir la inequitat de Linux a una tercera part. A més, donat que els algoritmes proposats son completament software, podrien incorporar-se com a polítiques de planificació en Linux i emprar-se en servidors a petita escala per obtenir els avantatges mencionats.

Chapter 1

Introduction

This chapter introduces some concepts and presents the motivation for the work developed in this thesis. First, contention points on the memory hierarchy of single-threaded multicore processors are identified. Next, contention on simultaneous multithreading multicores is explained. After that, the chapter discusses how contention not only affects performance but also fairness. Finally, the objectives and main contributions of this thesis are described, and a summary about how the rest of this dissertation deals with contention on the different processor architectures is presented.

1.1 Background

1.1.1 Chip Multiprocessor

Multicore processors have become the common implementation for high-performance microprocessors. A chip multiprocessor (CMP) incorporates additional cores on the same chip with each technology generation, and has the potential to provide higher levels of processing performance than its single-core counterparts, while attacking power, cooling, and package costs problems. These advantages certainly explain the success of CMPs to such an extent that the use of these systems is currently spread from high performance to mobile and embedded systems.

One of the main performance bottlenecks in CMPs lies in the interconnection between the computational cores of the chip and the main memory. The most important component of this bottleneck has typically been the main memory latency. However, as the number of cores and their multithreading capabilities increase, the contention for the available main memory bandwidth becomes a major concern since it might negatively affect the scalability of current and future manycore designs.

Recent research work has shown that scheduling is a simple and powerful way of addressing main memory bandwidth contention. For instance, when the number of available tasks[1] exceeds the number of hardware contexts[2], bandwidth-aware scheduling strategies can help to reduce main memory bandwidth contention by avoiding concurrent execution of memory-hungry applications. These strategies take into account the total bandwidth required by applications and schedule a set of them to execute concurrently, ensuring that the accumulated bandwidth requirements of the co-runners[3] do not exceed

[1] In this work the terms task, process, job, program, and application are used as synonyms. With the exception of Section 8.2, where parallel applications are discussed, thread is also used as synonym of the above terms.

[2] A processor has as many hardware contexts as processes it can simultaneously run. In single-threaded processors, this number is equal to the number of cores. In SMT multicores, the number of hardware contexts is equal to the number of cores times the number of threads that each core can simultaneously run.

[3] The term co-runner is used to refer to the processes that concurrently run, interfering in the shared resources. In single-threaded processors, all the processes running concurrently are co-runners, regardless of the core on which they run, since they mainly interfere in the main memory. In SMT processors, we use the term to refer to the processes simultaneously running on the same core, since the strongest interference appears among them. Nonetheless, we will put the term in context when it can cause confusion.

the available bandwidth. Otherwise, performance could severely be damaged due to the interference that arises when accessing main memory.

In order to hide, as much as possible, the large memory latencies that current DRAM memories present, microprocessor architects are designing processors that implement huge last level caches (LLC), alongside other microarchitectural mechanisms. These LLC caches provide higher bandwidth, but they are accessed much more frequently than main memory, which might shift the primary interference point from the main memory to the LLC. For instance, the IBM POWER8 processor [1] implements a large L3 cache, with huge latencies (several tens of cycles), that can be accessed by up to eighty concurrent threads (in a 10-core POWER8 processor). Thereby, cache contention is a major design concern and is expected to exacerbate in future microprocessor generations.

Despite the current trend in the cache hierarchy design consists of implementing L1 and L2 private caches per core, and an L3 cache shared among all the cores, some processors have also implemented multiple shared caches on different levels of the cache hierarchy. This is the case, for example, of the Intel Dunnington and IBM POWER5 processors. The Intel Xeon E7450 is a six-core processor whose L2 cache level is composed of three L2 caches, each one shared by two cores. In a similar way, the IBM POWER5 implements eight cores and includes multiple L2 and L3 shared caches. Figure 1.1 depicts its memory hierarchy. The second cache level is composed of four L2 caches, each one shared by

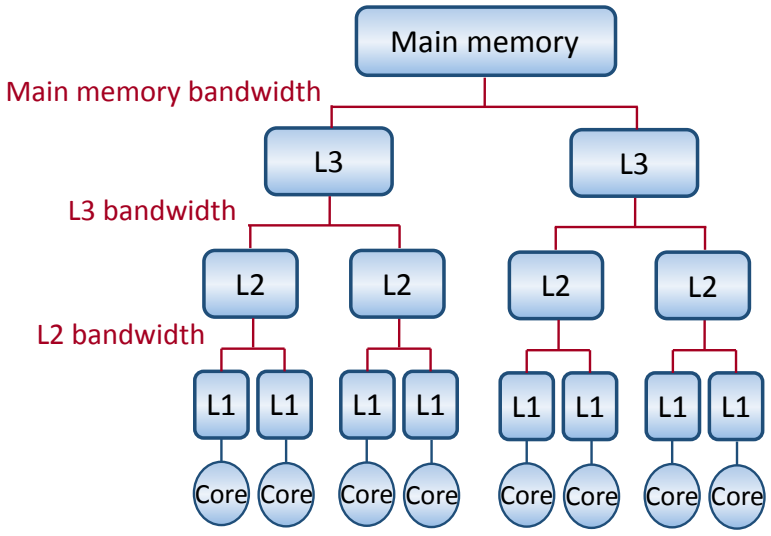


FIGURE 1.1: Memory hierarchy of the IBM POWER 5 processor.

a pair of cores, and the third level implements two L3 caches, which are shared by four cores. These cache hierarchy designs further allow smart scheduling policies to reduce cache bandwidth contention by assigning the processes to the cores balancing the accesses performed to each cache.

In summary, current caches are commonly shared by an increasing number of threads, which means that bandwidth contention can appear at any level of the cache hierarchy. Therefore, these potential contention points must be tackled by the scheduling policy in order to maximize the system performance.

1.1.2 Simultaneous Multithreading

Simultaneous multithreading (SMT) was proposed in 1995 by Tullsen et al. [2] as a way of improving the utilization and throughput of a single core. SMT, however, increases the area and power consumption of a core (5% to 20% [3, 4]), mainly due to replicating some architectural and performance-critical structures. Fortunately, the performance benefits outweigh these disadvantages and the most prevalent architecture for current high-end processors is a multicore processor consisting of SMT cores.

Recently, Eyerman and Eeckhout [5] show that a multicore processor consisting of SMT cores has an additional benefit other than increasing throughput. SMT is flexible when thread count varies: if thread count is low, per-thread performance is high because only one or a few threads execute concurrently on one core, whereas if thread count is high, it can increase throughput by executing more threads concurrently. As such, a multicore consisting of SMT cores performs as well as or even better than a heterogeneous multicore that has a fixed proportion of fast big cores and slow small cores.

SMT processors exploit both instruction-level and thread-level parallelism by issuing instructions from different threads in the same cycle. Thread-level parallelism increases the chance of issuing instructions, improving the utilization of the issue logic, but at the same time issued instructions from different threads continuously share core resources. The deep resource sharing makes threads interfere at several processor structures, which causes that the performance of SMT cores strongly depends on how these resources are shared among threads. If at any point of the execution, the demand for a given resource exceeds what that resource can provide, overall processor performance can be

damaged. Thus, scheduling algorithms that smartly allocate applications to cores can help to alleviate shared resource contention and improve performance and/or fairness in SMT multicore processors.

Two kinds of shared resources can be distinguished in SMT multicore processors: intra-core and inter-core resources, which are the shared resources within the core and in the uncore part of the system, respectively. Shared intra-core and inter-core resources vary with the processor architecture. The instruction queue, L1 cache, and execution units are typical examples of shared intra-core resources, while the LLC and main memory are resources commonly shared among cores.

As we have discussed before, bandwidth through the memory hierarchy is a critical shared resource in any current multicore system. Unlike single-threaded multicores, where L1 caches are not shared among processes, in SMT multicores the processes allocated to the same SMT core share the L1 cache among them. Thus, L1 caches emerge as a new contention point that should be tackled by bandwidth-aware process schedulers.

However, L1 bandwidth is not the only contention point in SMT processors. The way in which threads running on the same SMT core use the multiple shared components (e.g., execution units, instructions queues, etc.) can make the difference in terms of performance. This fact has also been addressed by several works [6] [7] through scheduling algorithms for SMT processors, typically known as symbiotic schedulers. These schedulers somehow estimate or measure how well a set of applications are going to co-run on the SMT processor in order to select the schedules[4] that achieve the highest throughput.

In summary, current microprocessors are able to concurrently run several applications, sharing some processor structures among the co-running processes. In these systems, selecting which applications to run and on which cores they should run has an impact on performance because cores deploy resources for which threads compete. As such, threads can interfere with each other, causing performance variations for other threads. Smart schedulers should be aware of this issue and reduce the negative interference as

[4]The term schedule is used to refer to one of the possible combinations of applications and the allocation of the applications to the cores. Note that not all the running threads share the same resources, since it depends on which core they are allocated to. For instance, all cores on a chip usually share the memory system, but a given cache can be shared by a smaller set of cores, and threads on an SMT core share almost all of the core resources.

much as possible by scheduling complementary or symbiotic tasks to share the different resources.

1.1.3 System Fairness

Sharing is convenient for several processor resources that might present low utilization when they are private and their usage is restricted to a single process. Moreover, sharing is an efficient and flexible approach. A single process can make use of the full capacity of a shared resource when it is the only process running or when the other processes perform a very scarce use of that resource. Notice that allowing a single process to achieve the same capacity from a resource would require to over-provision the resource.

These benefits have yield current SMT multicores to share most of their resources. Hence, processes compete among them at run-time for shared resources and sharing policies are implemented to regulate their usage. These policies should provide performance and fairness to concurrently running applications. However, designing fair sharing policies is challenging due to two main issues. First, processes present different requirements for the multiple shared resources, and second, the shared use of a resource affects differently the individual performance of distinct processes.

Therefore, while resource sharing allows the processor to achieve higher throughput, it can certainly affect the system fairness. In this dissertation, a system is considered fair when all the running processes present the same slowdown with respect to their isolated execution. The lack of fairness, known as unfairness, causes important undesirable behaviors on the system [8] [9] [10]: i) it complicates priority-based scheduling since jobs with lower priorities can achieve more progress than those with higher priorities, ii) it makes difficult to guarantee worst-case execution times (WCET), which is particularly important in embedded systems, iii) it reduces performance predictability, which complicates the analysis and optimization of both hardware and software implementations, and iv) it enables denial of service attacks.

Despite the current relevance of fairness, it is not commonly acceptable to improve it at the expense of overall workload performance. However, targeting fairness and performance at the same time is not an easy task. For example, a prevalent approach to improve performance consists in balancing the memory requests of a multiprogram

workload along its execution time [11] [12]. In contrast, to improve fairness, the processes with less accumulated progress should be prioritized over processes with superior progress. Unfortunately, both strategies can easily conflict. In such a case, preference should be given to one of the targets (performance or fairness), penalizing the other. To overcome these issues we propose the use of progress-aware schedulers to tackle performance and fairness simultaneously.

1.2 Objectives of the Thesis

The main objective of this dissertation is to address contention in the shared resources of current multicore processors (with different architectures), which affects the performance of individual processes and reduces the system throughput and fairness. This goal requires from identifying, quantifying, and analyzing such contention. Then, based on the results provided by these studies, we will design, implement, and evaluate multiple scheduling algorithms, each one aimed at alleviating the interference on different contention points of a target architecture. Their final goal is to improve both throughput and fairness of the evaluated systems when running multiprogram workloads formed by single-threaded applications.

Through the dissertation we sometimes refer to the proposed scheduling algorithms as schedulers. Nonetheless, our goal is not to replace the scheduler(s) of the Linux kernel with our algorithms, but to develop new scheduling algorithms that could be implemented as scheduling policies of the kernel scheduler. In this way, either the user or the operating system can select these policies to schedule the adequate workloads. Thereby, our scheduling algorithms do not need to handle all use cases that an operating system scheduler needs to consider such as short interactive processes, input/output bounded tasks, or parallel applications, among others.

The dissertation starts with multicore single-threaded processors, where bandwidth contention at the memory system, ranging from the main memory to the shared caches, strongly affects performance. Next, the study focuses on multicore SMT processors, where the processes running on the same SMT core share most of the core resources, including the execution units and the L1 bandwidth, enabling the design of smart process

allocation policies that address contention within the core. All the described experiments are carried out in real systems and the performance and fairness achieved with the proposed algorithms is compared with respect to Linux..

1.3 Main Contributions of the Thesis

The four major contributions of this thesis are described below:

- We study how bandwidth contention through the cache hierarchy of current multicore processors affects performance, finding out that not only contention at main memory, but also on the shared caches, can strongly deteriorate system performance. To deal with bandwidth contention along the memory hierarchy, **we propose a memory-hierarchy bandwidth-aware scheduler** which evenly distributes the memory and cache accesses along the workload execution time and across the available caches, respectively. The scheduling algorithm is further improved by favoring the execution of the processes more sensitive to bandwidth contention in scenarios with lower bandwidth consumption.

- When the processor implements SMT cores, the L1 cache and bandwidth are shared by the threads running on the same core, adding a critical contention point. Our analyses show how the L1 bandwidth utilization of a process is related with its performance and how both of them are affected by the interference caused by the L1 utilization of a co-running process. To address L1 bandwidth contention, **we propose an L1 bandwidth-aware process allocation policy** that balances the L1 requests of a workload across the L1 caches of the processor. The policy is then combined with a memory bandwidth-aware process selection algorithm to build an entire scheduler that addresses bandwidth contention on each memory contention point of SMT multicores.

- Interference in the shared resources not only affect performance but also system fairness, which is also a desirable characteristic. To improve system fairness **we propose a progress-aware scheduler** that estimates, at run-time, the progress made by each process of a multiprogram workload, favoring the execution of the processes with lower accumulated progress to improve system fairness. In addition,

we also present a progress-aware scheduler that simultaneously addresses fairness and performance.

- Finally, to deal with contention in all the shared resources of SMT multicores, **we propose a symbiotic scheduler**. This scheduler uses an SMT interference model, based on CPI stacks, to estimate the slowdown of a given schedule without actually running it. Using this model, the symbiotic scheduler can quickly explore the space of possible schedules and select the optimal one.

1.4 Thesis Outline

This dissertation consists of eight chapters. Chapter 1 has introduced the thesis, objectives, and contributions. Chapter 2 discusses the related work. Chapter 3 presents the experimental platforms and common aspects of the evaluation methodology. Chapter 4 studies the impact of bandwidth contention through the cache hierarchy and proposes the Memory-hierarchy bandwidth-aware scheduler. Chapter 5 analyzes how L1 bandwidth contention affects the performance of SMT multicores and presents the L1 bandwidth-aware process allocation policy and a bandwidth-aware scheduler for this architecture. Chapter 6 describes how progress can be estimated at run-time and introduces the progress-aware schedulers that address fairness and performance. Chapter 7 discusses the SMT interference models and proposes the Symbiotic scheduler. Finally, Chapter 8 summarizes this thesis, discusses future work, and enumerates the related publications.

Chapter 2

Related Work

This chapter discusses the state-of-the-art and important work related with the topics covered in this dissertation. First, we introduce the related work regarding main memory and cache contention on multicore processors. Second, we revise the related work that deals with contention on SMT and multicore SMT processors. Most related work is focused on improving system performance. Nonetheless, through the different sections we also discuss previous work that tackle the addressed contention focusing on fairness and on performance models that help taking smarter scheduling decisions.

2.1 Main Memory Contention

Multicore processors can concurrently run multiple processes, which potentially increases the number of memory accesses performed per unit of time. However, they were introduced without an equivalent growth on the main memory bandwidth. Since the memory bandwidth in a multicore processor is shared among the co-running processes, when the requirements of these processes are high, memory contention can rise and affect the performance significantly. To tackle this problem, important research work focusing on main memory bandwidth contention has been done.

Antonopoulos et al. [13] [14] propose some of the first scheduling policies based on the memory bus bandwidth consumption of the processes running at the same time. In [13], the bus bandwidth consumption values are obtained by modifying the source code of the running applications, while in [14], less intrusive implementations based on processor performance counters are explored. In both cases, the proposed policies try to match the total bandwidth requirements of the co-runners to the peak memory bus bandwidth.

More recently, Xu et al. [11] prove that irregular memory access patterns can produce fine-grained contention when the required bandwidth is close to the peak bandwidth. To deal with this situation, they propose the use of the average bandwidth requirements of the applications instead of the peak bandwidth. Authors estimate the Ideal Average Bandwidth (IABW) of a workload as the number of main memory accesses divided by the total execution time. By scheduling the applications to match the IABW each quantum, contention is greatly reduced.

Other works also deal with main memory bandwidth contention in other contexts. For instance, Koukis et al. [15] propose an scheduling algorithm addressing symmetric multiprocessing (SMP) clusters, which considers bandwidth contention at the network additionally to the main memory bandwidth. Pinel et al. [16] also perform main memory bandwidth-aware scheduling on multicore processors, exploring the trade-off between energy consumption and execution time to perform *green* scheduling.

Even though performance is usually the main goal, other works focus on system fairness. To provide fair sharing of the memory resources some works tackle the memory

controller. Mutlu et al. [8] propose a memory access scheduler that balances the DRAM-related slowdown experienced by the co-scheduled processes. A similar approach is followed by Nesbit et al. [17], who use concepts from network fair queuing to design a fair queuing memory system.

Unfortunately, fairly sharing a single resource or a set of resources does not provide system fairness. Ebrahimi et al. [18] present a global solution and propose achieving fairness via source throttling, a mechanism that addresses unfairness on the entire memory system. Authors propose to estimate unfairness in the shared memory system. For this purpose, they throttle down cores causing unfairness by limiting both the number of requests and the frequency at which they can be injected into the system. Other works try to improve system fairness by focusing on process scheduling. For instance, Xu et al. [19] mainly target main memory bandwidth contention and propose a process scheduler that monitors the progress of the processes at run-time, which is used to increase the priority of the processes progressing at a slower pace.

Finally, Subramanian et al. [20] combine performance predictability with fairness-oriented main memory request scheduling. The authors first present a model that estimates the slowdowns caused by memory interference by modifying the priority scheme of the memory controller. Then, they use the model as the base of two different memory request scheduling schemes that provide quality-of-service (QoS) and maximize fairness, respectively.

2.2 Cache Hierarchy Contention

2.2.1 Contention-Aware Scheduling

As for main memory contention, process scheduling is a simple yet effective way of addressing cache contention. Knauerhase et al. [21] show that the operating system (OS) can obtain task behavior data at run-time using performance counters, and use the gathered information to ameliorate performance variability and more effectively exploit multicore processor resources with a smart observation-based scheduling policy. Zhuravlev et al. [22] investigate how contention for shared resources can be mitigated via process scheduling. Authors propose a classification scheme that determines how

the processes affect each other on the shared resources considering contention on the cache space, memory controller, memory bus, and hardware prefetch. They design a scheduling algorithm that does not only improve the performance of a workload as a whole, but it can also improve the quality of service or provide performance isolation for individual applications.

Regarding memory contention on datacenters, Tang et al. [23] study the impact of sharing memory resources on the performance of datacenter applications. They analyze the impact of thread-to-core mappings according to the memory behavior of the applications considering the shared memory resources. Authors found that there is both a sizable benefit, but a potential performance degradation when improperly sharing resources. They also present an heuristic-based and an adaptive approach to enhance thread-to-core assignment of the datacenter applications. Finally, Sato et al. [24] observe that some threads cause severe performance degradation due to inter-thread cache conflicts and shortage of capacity on the shared cache. Based on these observations, they propose a scheduling policy that can prevent multiple threads from requesting a large cache capacity when sharing a limited cache, hence avoiding severe performance degradation.

Other works address cache contention with different approaches. For instance, Qureshi et al. [25] observe that memory level parallelism (MLP) benefits differ across cache misses. Since isolated cache misses have a stronger impact on performance than parallel cache misses, they propose an MLP-aware cache replacement mechanism that considers both the MLP-based cost of cache misses and data recency into account. Kaseridis et al. [26] propose a global solution that tackles the bandwidth contention that arises at each level of the memory hierarchy. To do this, they rely on additional hardware-based resource profilers and cache partitioning algorithms to avoid cache contention.

With the main goal of providing fairness to co-running processes, Fedorova et al. [27] propose a *cache-fair* scheduling algorithm that gives more execution time to the processes that are more affected by unbalanced cache sharing. The goal is to ensure that the applications run as quickly as they would do under a fair cache allocation, regardless of how the cache is actually being allocated. In a follow on work, Fedorova et al. [28] propose the use of the LLC miss rate as a scheduling heuristic that acts as a good predictor for all types of contention related with the LLC.

2.2.2 Resource Partitioning

Another way to tackle cache contention is cache partitioning. In this regard, several cache partitioning mechanisms have been proposed to mitigate cache contention and maximize throughput and/or improve fairness among the co-runners. Qureshi et al. [29] propose utility-based cache partitioning, a run-time mechanism with low overhead that partitions a shared cache among multiple processes depending on the reduction in cache misses that each process is likely to obtain for a given amount of cache resources. The proposed partitioning mechanism targets performance, but authors state that a similar approach can be used to improve system fairness.

Some cache partitioning proposals also deal with bandwidth contention. Moretó et al. [30] partition the LLC of CMPs to increase memory level parallelism and reduce workload imbalance. Cache partitioning algorithms like SHARP [31] and PriSM [32] manage the LLC cache by using formal control and probability theories, respectively. However, as pointed out by Sato et al. in [24], cache partitioning mechanisms can severely limit the overall performance if applications with cache requirements exceeding the cache capacity are co-scheduled.

Cache partitioning techniques are not only proposed to improve performance, but some mechanisms try to provide a fair cache access to the processes sharing the same cache structure. Suh et al. [33] estimate the isolated miss rate of the processes to improve the partitioning. Kim et al. [34] dynamically partition L2 caches based on metrics that correlate with execution-time fairness. Later, Chang et al. [35] introduce the use of multiple time-sharing cache partitions to improve throughput and fairness while maintaining QoS, by allowing one thread to temporarily shrink the cache capacity assigned to other threads.

Finally, other proposals are not restricted to cache partitioning and present wider resource sharing mechanisms. Nesbit et al. [36] propose a resource sharing mechanism that provides QoS to the running processes. In particular, authors present an arbiter that guarantees a minimum bandwidth to each process to provide QoS. A similar mechanism is designed by Colmenares et al. [37], who implement the Adaptive Resource Centric Computing (ARCC) in the Tessellation OS. Using ARCC, resources can be distributed among the processes providing performance isolation and predictability.

2.2.3 Performance Models

Performance and/or interference models are also used to enhance the scheduler. This approach allows the scheduler to take smarter decisions and improve performance and fairness. Eyerman et al. [38] propose a performance counter architecture for computing CPI components and develop a performance model for superscalar out-of-order processors based on these CPI components. Eklov et al. [39] present a method for measuring application performance and main memory bandwidth utilization as a function of the available shared cache capacity. Similarly, Casas et al. [40] present a methodology to predict the performance of an application when the available bandwidth and space through the memory hierarchy are reduced.

2.3 SMT Core Contention

The importance of intelligently selecting applications that should run together on an SMT core was recognized quickly after the SMT introduction [6]. The performance improvement heavily depends on the characteristics of the co-running applications, and some combinations may even degrade total throughput, for example due to cache trashing [41]. Snavely and Tullsen [6] were the first to propose a mechanism to decide which applications should run on the same core to obtain maximum throughput. At the beginning of every scheduler quantum, they shortly execute all (or a subset of) the possible combinations, and select the best performing combination for the next quantum. Because of the number of possible combinations quickly grows with the number of applications and hardware contexts, the overhead of sampling the performance quickly becomes large and/or the fraction of combinations that can be sampled becomes small. To overcome the sampling overhead, Eyerman and Eeckhout [7] propose model-based scheduling. A fast analytical model predicts the slowdown each application encounters when co-scheduled with other applications, and the best performing combination is selected.

Other authors have also studied the symbiosis between applications with different approaches. For instance, Čakarević et al. [42] characterize different types of resource sharing in an UltraSPARC T2 processor and improve the execution of multi-threaded

applications with a resource sharing aware scheduler. Acosta et al. [43] show that processor throughput is highly dependent on thread allocation and propose an allocation policy that combines computation and memory bounded processes in each core.

Other studies have explored the use of models and profiling to estimate SMT benefits. Cazorla et al. [9] propose a novel strategy to allow the operating system to run jobs at a certain percentage of their maximum speed, regardless of the system load. Moseley et al. [44] use regression on performance counter measurements to estimate the speedup of an SMT processor when co-executing two applications. Settle et al. [45] predict job symbiosis using offline profiled cache activity maps. More recently, Eyerman et al. propose a cycle accounting mechanism [46] and a probabilistic symbiotic scheduler [7] for SMT processors, while Porter et al. [47] estimate the speedup of a multi-threaded application when enabling SMT, based on performance counter events and machine learning. Mars et al. [48] use microbenchmarks called *bubbles* to measure first, the pressure on the memory subsystem that the applications generate, and second, how much the applications suffer from different levels of memory contention introduced by the *bubbles*. Using this information, obtained during a characterization phase, the complexity of finding good applications to core allocations is reduced. In a follow-up work, Zhang et al. [49] propose a similar methodology to predict the interference among threads on an SMT core. They develop microbenchmarks called *rulers* that stress different core resources, and by co-running each application with each ruler in an offline profiling phase, the sensitivity of each application to contention in each of the core resources is measured.

Finally, other works deal with fairness in SMT fetch policies. Luo et al. [50] and Eyeraman et al. [51] propose SMT fetch policies that enhance both performance and fairness considering the pipeline status and memory-level parallelism, respectively.

Chapter 3

Scheduling Framework, Experimental Platforms, and Evaluation Methodology

This chapter first presents the scheduling framework designed to facilitate the implementation of new scheduling algorithms and their evaluation on real systems. Next, the chapter describes the three experimental platforms used to carry out the experiments performed in this dissertation. Then, it discusses the evaluation methodologies used to properly evaluate process selection policies, process allocation policies, and entire schedulers. Finally, it introduces the metrics used to compare the different schedulers.

3.1 Scheduling Framework

To ease the implementation and evaluation of the scheduling algorithms that this thesis proposes, we design a scheduling framework. The framework is built as a user-level program and runs above the Linux operating system scheduler. It makes easier and faster to implement the proposed scheduling algorithms since they share most of the code. This framework also allows a fair evaluation by ensuring equal overhead due to process management or handling of performance counters across the studied scheduling algorithms. In short, different scheduling policies can be quickly implemented and fairly compared in the developed environment.

The designed framework schedules the processes in two main steps, which correspond with the process selection and process allocation policies. The process selection policy determines which processes should run the next quantum when the set of available processes exceeds the number of hardware contexts. Notice that the number of hardware contexts of a processor corresponds with its number of cores times the number of simultaneous threads per core it supports. The implemented process selection policies only need to select the set of processes that should run on the next quantum (according to any developed algorithm) and let the framework schedule them. The processes selected to run are resumed with the SIG_CONT signal, while the remaining processes are kept stopped. Once the quantum expires, the framework stops all the running processes again with the SIG_STOP signal.

The process allocation policy decides on which core each selected process runs. Process allocation is particularly important when the cores are SMT since the processes assigned to a given core are going to share critical core resources. Nonetheless, in the case of multicore single-threaded processors, the process allocation policy can also be used to balance the bandwidth through the different shared caches. A process allocation policy only needs to choose the core on which each selected process should run (i.e., it defines the final schedule), and then the framework is responsible of enforcing this schedule. The framework uses a process parameter on Linux called *core affinities*, which establishes the set of cores on which a process can run. By setting the *sched_setaffinity* attribute of each process to a single core, the framework enforces the schedule given by the process allocation policy.

In addition to the different scheduling algorithms proposed in this dissertation and part of the related work, the scheduling framework implements two scheduling algorithms: a random scheduler and a Linux-based scheduler. The implementation of the random scheduler is trivial. Each quantum, the scheduler randomly selects as many processes as hardware contexts in the experimental platform (process selection). Then, the selected processes are randomly assigned to the cores (process allocation). To implement the Linux-based scheduler, the scheduling framework selects all the processes to run on the next quantum (process selection), and allows the selected processes to be allocated to any hardware context (process allocation). This setup lets Linux scheduler decide at every moment which processes should run and the cores where they should be allocated to.

Finally, the framework also manages the access to hardware performance counters using the *libpfm* library, which supports independent measurements for co-running processes at run-time. The set of supported events depends on the experimental platform, but typically includes, among many others, the number of unhalted cycles, committed instructions, as well as requests and misses for the different caches of the memory hierarchy. During the development of the work performed in this thesis, *libpfm* has received several updates to support the latest architectures and to correct bugs. Hence, the framework has used different versions of the library, from *libpfm* 3.1 to 4.6. The scheduling framework allows the user to define the set of events to monitor, and then manages the configuration of the library and events, reads the counters for the processes that were run during a quantum, and updates the related scheduling variables. These variables, generally, are used by the proposed scheduling policies to smartly schedule the processes.

3.2 Experimental Platforms

3.2.1 Single-Threaded Multicore: Intel Xeon X3320

As an example of a multicore processor with single-threaded cores, we use the shared-memory quad-core Intel Xeon X3320 processor [52]. The main system characteristics are presented in Table 3.1. The processor implements four single-threaded cores, runs at 2.5 GHz, and is equipped with 4 GB of DDR2 RAM.

CPU	Intel Xeon X3320
Frequency	2.5 GHz
Number of cores	4
Multithreading	No
L1 cache	Code L1: 4 x 32 KB Data L1: 4 x 32 KB
L2 cache	2 x 3 MB
Memory	4 GB (2 GB x 2) DDR2

TABLE 3.1: Characteristics of the Intel Xeon X3320 system.

The operating system of this platform is a Fedora Core 10 Linux distribution with kernel 2.6.29. We installed the *perfmon2* patch to provide the system with performance monitoring support. Performance counters are managed through the *libpfm* 3.10.0 library.

Figure 3.1 presents the cache hierarchy of the Intel Xeon X3320. It consists of two 3 MB L2 caches (LLC), each one shared by a pair of cores. Each core implements a private L1 cache, with 32 KB for data and 32 KB for instructions. The main memory and each L2 shared cache of the hierarchy are potential contention points since the caches of the immediately higher level (L2 and L1, respectively) share the available bandwidth to access them.

The cache hierarchy of the Intel Xeon X3320 resembles the cache hierarchy implemented in more recent processors with greater number of cores, and deeper and wider memory hierarchies, such as the Intel Dunnington processors or the IBM POWER5, whose memory hierarchy is shown in Figure 1.1. As observed, the higher the number of cores

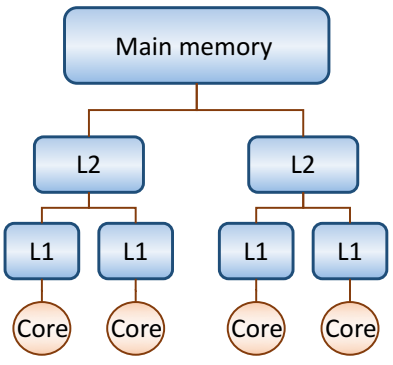


FIGURE 3.1: Memory hierarchy of the Intel Xeon X3320 processor.

and caches, the higher the number of contention points. Therefore, bandwidth-aware scheduling algorithms like the ones proposed in this work should provide better performance enhancements on these processors as well as future manycore processors that could implemented similar memory hierarchies.

3.2.2 SMT Multicore: Intel Xeon E5645

As an example of current SMT multicores we use the Intel Xeon E5645 processor. Table 3.2 presents the main system characteristics. The Intel Xeon E5645 is composed of six dual-threaded SMT cores. Each core includes two levels of private caches, a 32 KB L1 data cache and a 256 KB L2 cache. A third-level 12 MB cache is shared by the private L2 caches. The system is equipped with 12 GB of DDR3 RAM and runs at 2.4 GHz. The Intel Turbo Boost mode is disabled to prevent uncontrolled frequency increases when only one thread is running on a core. The system has a Fedora Core 10 distribution installed with Linux kernel 3.11.4, and uses the *libpfm* 4.3.0 library to handle hardware performance counters.

Figure 3.2 depicts the memory hierarchy of the Intel Xeon E5645. Notice that L1 caches are shared by the two hardware threads of each SMT core which can cause L1 cache contention among the processes running on the same SMT core. Main memory is the other critical contention point of the hierarchy. It receives the memory requests of all the running processes and presents higher latency than the L3 cache.

CPU	Intel Xeon E5645
Frequency	2.4 GHz
Number of cores	6
Multithreading	Yes, up to 2 threads per core
L1 cache	Code L1: 6 x 32 KB Data L1: 6 x 32 KB
L2 cache	6 x 256 KB
L3 cache	12 MB shared,
Memory	12 GB (4 GB x 3) DDR3

TABLE 3.2: Characteristics of the Intel Xeon E5645 system.

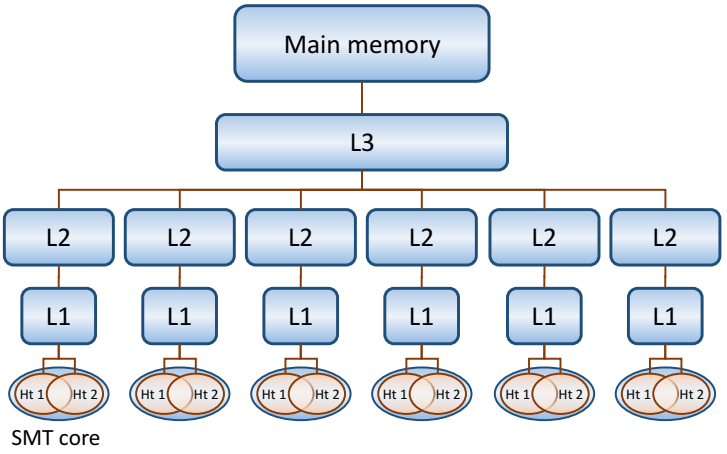


FIGURE 3.2: Memory hierarchy of the Intel Xeon E5645 processor.

3.2.3 IBM POWER8 System

Our third experimental platform is an IBM Power System S812L machine, which consists of an IBM POWER8 processor with ten cores where each core can execute up to 8 hardware threads simultaneously. Cores can work in single-threaded mode, SMT2 mode, SMT4 mode, or SMT8 mode. Mode transitions are done automatically by the processor according to the number of active hardware threads. The high degree of multithreading is challenging from a scheduling point of view since the sharing degree of most resources is expected to be high and to generate a great interference.

The remaining features of our IBM POWER8 system are presented in Table 3.3. The processor implements private L1 data and L2 caches of 64 KB and 512 KB, respectively, per core and shares a huge 80 MB last level cache. The memory hierarchy closely resembles the one shown in Figure 3.2 (with ten cores and eight threads per core). The system has 32 GB of RAM memory and runs at a maximum frequency of 3.7 GHz. Our setup uses an Ubuntu 14.04 Linux distribution with kernel 3.16.0 and manages performance counters with the *libpfm* 4.6.0 library.

We focus our experimental evaluation in SMT2 and SMT4 modes with multiprogram SPEC CPU2006 workloads. We do not evaluate the SMT8 mode since we did not notice performance benefits with the Linux scheduler in SMT8 mode over SMT4 mode running our target workloads. On average across ten 32-application workloads running on four cores Linux performs, in terms of system throughput (see Section 3.4), slightly better

CPU	IBM POWER8
Frequency	3.7 GHz
Number of cores	10
Multithreading	Yes, up to 8 threads per core
L1 cache	Code L1: 10 x 32 KB Data L1: 10 x 64 KB
L2 cache	10 x 512 KB
L3 cache	80 MB
Memory	32 GB (32 GB x 1) DDR3

TABLE 3.3: Characteristics of the IBM POWER8 system.

(0.9%) in SMT8 mode than in SMT4 mode. However, as the number of cores grows, the performance benefits are reduced when running in the SMT8 mode and turn into performance losses. Thereby, on average, across ten 80-application workloads ran on ten cores, Linux performs 7.8% worse in SMT8 mode than in SMT4 mode. This behavior should be related to the fact that SPEC benchmarks aim to stress the processor and the memory subsystem. In contrast, the SMT8 mode is expected to provide performance benefits to other types of workloads such as multi-threaded scale-out applications that share a considerable amount of code and present a small memory footprint.

3.2.3.1 NUMA Effects in the IBM POWER8

The IBM POWER8 processor has eight on-chip memory controllers. However, to reduce costs, our setup only has a single 32 GB DDR3 module connected to one of them. This apparently minor issue presents strong implications. The IBM POWER8 processor is implemented as a dual-chip module (DCM) processor but it works as a single chip processor [53]. More precisely it is built by mounting two chips (chiplets) containing half the number of cores each. Both chiplets are interconnected by fast local SMP links and each one implements four memory controllers. Figure 3.3 shows a block diagram of the processor and the memory subsystem. This design implies that our system includes two non-uniform memory access (NUMA) nodes. The first node comprises cores 0 to 4, while the second node contains the remaining five cores. Since the system only includes a single DRAM module connected to one of the NUMA nodes, the memory performance observed

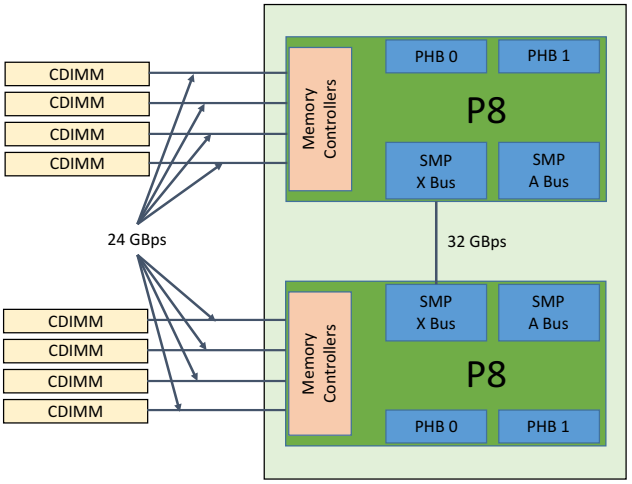


FIGURE 3.3: Logical diagram of the IBM POWER8 and memory subsystem.

by the cores varies depending on the node the core belongs to. To confirm this thesis, we use the LMbench [54] and STREAM [55] benchmarks and measure the DRAM latency and bandwidth, respectively. These applications aim to stress the memory subsystem by accessing the elements of data arrays whose size reaches up to 1792 MB.

Figure 3.4 presents the memory latency that the cores of each NUMA node experience for each tested array size. Memory requests access the array in 128-byte strides, which matches the POWER8 cache line size. We did not appreciate any latency difference between cores in the same NUMA node. The latency is identical for both NUMA nodes when the array fits in the L1, L2, or L3 caches. However, when the array exceeds the cache size and the main memory is accessed, the cores in the NUMA node 0, where the

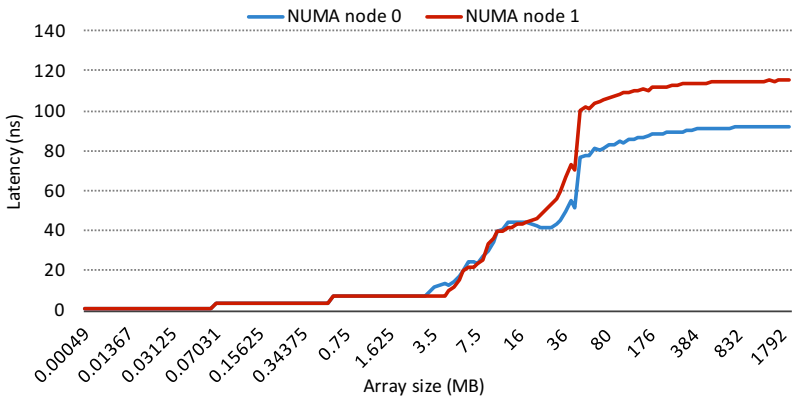


FIGURE 3.4: Memory latency varying the array size.

Function	Kernel	NUMA node 0	NUMA node 1
Copy	$a(i) = b(i)$	17.7 GB/s	16.8 GB/s
Scale	$a(i) = q \times b(i)$	17.3 GB/s	16.5 GB/s
Add	$a(i) = b(i) + c(i)$	24.3 GB/s	22.4 GB/s
Triad	$a(i) = b(i) + q \times c(i)$	24.3 GB/s	22.4 GB/s

TABLE 3.4: Bandwidth reported by the STREAM benchmark for the two NUMA nodes.

memory slot is plugged in, experience a latency around 20% lower than the cores on the NUMA node 1.

Regarding memory bandwidth differences between NUMA nodes, Table 3.4 presents the average bandwidth observed by the five cores in each node when running the STREAM benchmark. The results, broken down by kernel, show that the cores on the NUMA node 0 are able to consume between 5.1% and 8.5% more memory bandwidth, depending on the executed kernel. It is also worth noting that the cores on the NUMA node 0 almost reach the theoretical maximum memory bandwidth, which is 24 GB/s.

Linux seems to be aware of the system being a NUMA system. For instance, the *lscpu* command identifies two NUMA nodes, the first one including logical CPUs from 0 to 39, and the second one including logical CPUs from 40 to 79[1]. Since the kernel version 3.8, Linux is able to perform NUMA-aware scheduling [56] and allocates the applications that more frequently access the main memory to the NUMA node closest to the main memory, where most of the application data resides. In our system, this NUMA node is always the node 0, since it is the only one with a DRAM module installed. This scheduling behavior turns into performance improvements that should be taken into account in the experimental evaluation.

3.3 Evaluation methodology

All the experiments carried out in this dissertation are performed using benchmarks from the SPEC CPU2006 benchmark suite, which is a standard suite widely used by industry and academia to evaluate and compare the performance of processors and memory

[1]Each core of the POWER8 accounts for 8 logical CPUs in Linux. Logical CPUs 0 to 7 identify the 8 threads that can be run in core 0 with SMT8 mode, logical CPUs 8-15 identify those threads of core 1, and so on.

systems. Given the fact that we want to evaluate scheduling algorithms for multicore processors, the experimental evaluation targets multiprogram workloads composed of SPEC benchmarks.

3.3.1 Process Selection Methodology

A problem we had to face related with multiprogram workloads composed of SPEC benchmarks is that the stand-alone execution time of the benchmarks widely varies among them. While some benchmarks complete their execution within a minute, others can easily take more than ten minutes to finish. This fact complicates performing an adequate evaluation of scheduling algorithms. For instance, Xu et al. [11] observed that a scheduling policy that prioritizes the longest jobs could provide the best turnaround time in most workloads when the benchmarks experience widely different execution times. Another important drawback is that benchmarks with different execution time will have different weights in the mix execution, which might not be correctly reflected in some performance and fairness evaluation metrics. Finally, it could also limit the ability of a smart scheduler to perform better scheduling if, for example, a mix quickly completes the execution of most of its applications and the workload execution continues with a few processes for a long time.

To overcome these problems, we decide to equalize the execution time of the benchmarks when running alone [11, 19]. Hence, we measure the number of instructions that each benchmark completes when running alone in the system during x seconds [2]. The benchmarks with shorter execution time are relaunched, after they finish, until they reach this this period. We record the number of instructions that each benchmark executes during the x seconds, and set it as its *target number of instructions*. From now on, when we talk of executing or running a benchmark, we refer to the execution of its target number of instructions. In the experiments, the scheduling framework is in charge of relaunching the benchmarks that finish before completing their target number of instructions. Then, the framework also kills them when they reach this number of instructions to conclude their execution. Proceeding in this way, we avoid the problems that arise when the applications present widely different execution time.

[2] The number of seconds varies depending on the experimental system and it is indicated in the evaluation setup section of each chapter.

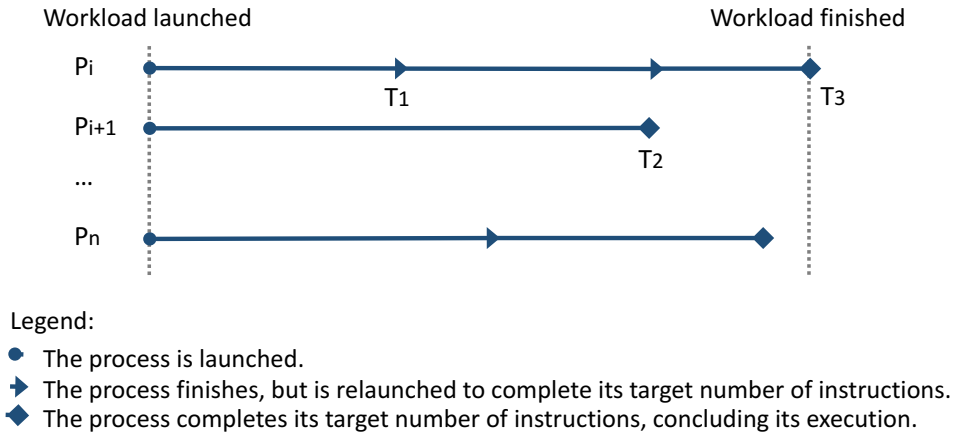


FIGURE 3.5: Timing chart under the process selection methodology.

To evaluate process selection policies and entire schedulers, we run multiprogram workloads where the number of processes exceeds the number of hardware contexts of the experimental platform. Otherwise, all the processes could be run each quantum and process selection would not be required. In addition, to carry out such experiments, we devise the process selection methodology, which is illustrated in Figure 3.5. As explained before, a processes does not conclude its execution until it completes its target number of instructions. Thus, a process can be relaunched (for example, T_1 in Figure 3.5) until it completes its target number of instructions. At this point, the frameworks kills the process and saves per-process metrics such as execution time or individual IPC (for instance, T_2 in Figure 3.5). The experiment continues until the last process of the workload finishes (T_3 in Figure 3.5). This is the point where the experiment ends and the framework obtains workload-related metrics such as the turnaround time of the mix or different IPC-aggregated metrics.

3.3.2 Process Allocation Methodology

The process selection methodology is not adequate to evaluate process allocation policies. The first change required is the number of applications of the workloads. Process allocation policies determine on which core each selected process should run and thus they require that the number of available processes matches (or is below) the number

of hardware contexts. Thus, to evaluate process allocation policies multiprogram workloads should include the same number of processes as hardware contexts the experimental platform has.

Notice that process allocation policies are particularly interesting for SMT multicores, since applications running on a given core share most of the core resources and can strongly interfere. In this context, a more important issue arises (assuming SMT cores that can run up to two threads): once the first process of the workload finishes, one core will run a single process, while the other processes will be run in pairs in the remaining cores. This scenario can *artificially* increase the performance of the processes running alone over the ones co-running on the same core. Thereby, it makes the evaluation of the policies difficult since it might be not possible to identify when the performance differences come from a smarter process allocation and when they are caused by the processes running alone on some cores for a fraction of the experiment.

To deal with this problem we devise a new evaluation methodology named process allocation methodology and illustrated in Figure 3.6. Under this methodology, all the applications of the workload are kept running until the last one finishes (i.e., it completes its target number of instructions as described in Section 3.3.1). Hence, the scheduling framework relaunches the processes that finish to keep the number of running applications constant during the entire experiment (for example, at T_1 in Figure 3.6, where the dashed line shows the execution of the process after concluding and being re-launched).

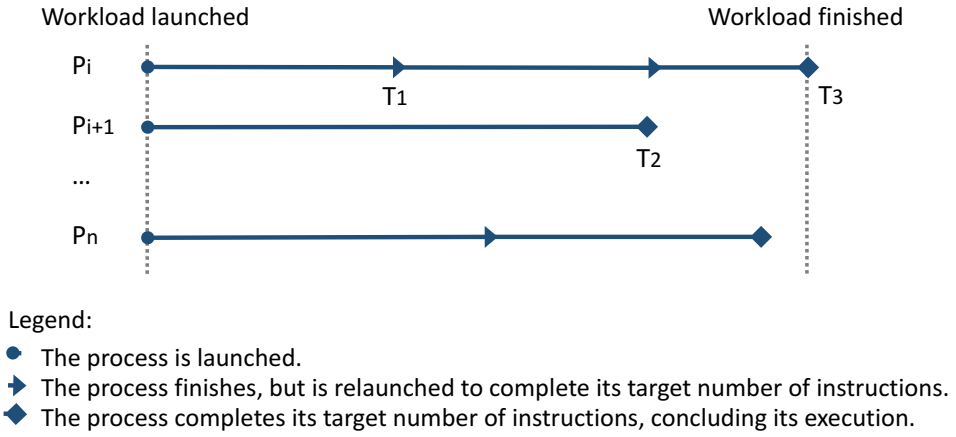


FIGURE 3.6: Timing chart under the process selection methodology.

When the last one finishes, the framework kills all the processes and the experiment concludes.

Despite the processes are relaunched, we need to measure the per-process performance metrics when the applications finish for the first time. Since the processes can progress at different paces, depending on the process allocation performed each quantum, there is no guarantee that at the end of the workload all the processes would have executed the same number of instructions. Thus, taking the per-process metrics when a given process finishes (it completes its target number of instructions) and then relaunch it, solves the mentioned problems, keeping uniform the number of applications of the workload and ensuring that the comparison is performed over the same number of instructions executed by each benchmark.

3.4 Metrics

A wide set of metrics has been used to analyze the performance and fairness of the proposed scheduling algorithms.

- **Turnaround time**. It is defined as the maximum turnaround time among the processes of a given workload (Equation 3.1). This metric measures the elapsed time since a workload is launched to execution until the last process completes its execution.

$$Turnaround\ time = Max\ (Turnaround\ time_i)\ \forall\{i\} \in \{1, N\} \tag{3.1}$$

  Note that in the context of process scheduling, the turnaround time of a process is the time since the process is launched until it concludes its execution, including the time where the process is not scheduled [19]. Otherwise, by pausing threads contention could be reduced and the turnaround time of the individual processes improved.

- **Average IPC**. This is the plain metric to compare throughput and is calculated as the arithmetic mean of the IPC of all the processes that form the workload (Equation 3.2). Unfair scheduling strategies may favor this metric if they prioritize

the execution of those benchmarks with highest IPC [6], so that the total number of instructions executed increases.

The unfair scenarios, however, are not allowed in our evaluation methodologies. In the process selection methodology all the applications execute their target number of instructions. In the process allocation methodology, the applications are relaunched after they complete their target number of instructions, but the individual metrics of the process are obtained at this point. Thus, unfair scheduling cannot affect the average IPC metric following our methodologies.

$$Average\ IPC\ =\ \frac{\sum_{i=0}^{N} IPC_i}{N} \tag{3.2}$$

- **Harmonic mean of the per-program IPC speedup**. Luo et al. [50] propose taking the harmonic mean of the individual thread speedups across all the applications of the workload (Equation 3.3), instead of using the arithmetic mean. They argue that the harmonic mean of speedups can be used as a metric that simultaneously captures both performance and fairness, since it tends to be lower when one or more threads have a significantly lower speedup (there is much variance).

$$Harmonic\,mean\,of\ IPC\ speedups\ =\ \frac{N}{\sum_{i=0}^{N} \frac{1}{Speedup_i}} \tag{3.3}$$

The speedup is defined as shown in Equation 3.4, where $IPC_{Together}$ is the IPC of the application running in a schedule and IPC_{Alone} is the IPC of the application running alone.

$$Speedup\ =\ \frac{IPC_{Together}}{IPC_{Alone}}\ =\ \frac{\frac{Instructions}{Turnaround\ cycles_{Together}}}{\frac{Instructions}{Turnaround\ cycles_{Alone}}} \tag{3.4}$$

- **System throughput (STP)** is a metric defined by Eyerman and Eeckhout [57] to quantify the accumulated single-program progress under multiprogram execution. It is calculated, using Equation 3.5, as the sum of the normalized progress over isolated execution (i.e., speedup) of all the applications that compose the workload.

$$STP\ =\ \sum_{i=0}^{N} Speedup_i \tag{3.5}$$

- **Average normalized turnaround time (ANTT)**. Eyerman and Eeckhout [57] propose to quantify the user-perceived performance using the ANTT metric. ANTT is calculated as the arithmetic mean, across all the applications of the workload, of the turnaround time of each application normalized over its stand-alone execution. ANTT is essentially a measure of the average per-application performance, but since it is inversely proportional to the harmonic mean of the per-program IPC speedup, which tends to be lower when there is much variance, it also incorporates a notion of fairness. It is a lower-is-better metric.

$$ANTT = \frac{\sum_{i=0}^{N} \frac{Turnaround\ time_i^{Together}}{Turnaround\ time_i^{Alone}}}{N} \tag{3.6}$$

- **Unfairness**. Running multiprogram workloads, fairness related metrics are used to estimate if performance benefits or losses are balanced across all the processes and do not concentrate only on a few of them. The unfairness metric has been used in several works [18, 19, 58] and is defined as the maximum slowdown divided by the lowest slowdown across all the processes (N) of the workload, as shown in Equation 3.7. The slowdown of a process corresponds with the inverse of the speedup defined in Equation 3.4. Notice that it is a lower-is-better metric and an unfairness equal to 1 means that the system is completely fair.

$$Unfairness = \frac{Max\ Slowdown_i}{Min\ Slowdown_j}\ \forall\{i,j\} \in \{1,N\} \tag{3.7}$$

Chapter 4

Bandwidth-Aware Scheduling on Multicore Processors

Several works have identified the main memory bandwidth of current processors as an important performance bottleneck. This chapter goes further these studies, analyzing the bandwidth contention through the full memory hierarchy of current multicore processors, and proposing scheduling algorithms to mitigate its negative effects on performance.

This chapter is organized as follows. First, the impact on performance of bandwidth contention through the memory hierarchy is studied. Next, the Memory-hierarchy bandwidth-aware scheduling algorithm is proposed to address bandwidth contention. This scheduler is then improved by favoring the execution of the processes more sensitive to bandwidth contention in less contentious schedules. Finally, the performance achieved by the proposed schedulers is discussed.

4.1 Performance Degradation Analysis

This section explores the bandwidth contention through the memory hierarchy of current single-threaded multicore processors. As an example of such architecture, we use a quad-core Intel Xeon X3320 processor. Please refer to Section 3.2.1 for the most relevant systems features.

The performance behavior analysis is carried out using the benchmarks of the SPEC CPU2006 benchmark suite with reference inputs. First, we characterize the benchmarks when running alone in the experimental platform. Then, we study their performance degradation due to L2 and main memory bandwidth contention. Finally, we measure their performance degradation under bandwidth-aware schedulers, which typically schedule the processes to keep a uniform bandwidth utilization during the workload execution.

To carry out the performance degradation analysis, each benchmark is concurrently launched with synthetic microbenchmarks, measuring the number of execution cycles, retired instructions, L2 and L1 cache misses. The microbenchmark is designed to inject synthetic traffic in the memory hierarchy and, depending on the requirements, it can mimic the behavior of either a main memory-bounded or L2-bounded application. Hence, this microbenchmark design allows us to study different workload conditions by setting different microbenchmark configurations.

In addition to bandwidth, cache space also acts as an important contention point. Both bandwidth contention and cache contention contribute to performance degradation. Nevertheless, the use of cache misses is also a good indicator of how contentious the cache usage is. The more contentious it is, the more misses occur, which translate into memory requests to the next level of the memory hierarchy.

4.1.1 Benchmarks Characterization

In order to avoid interference from other co-runners[1], each benchmark is characterized running alone in the system according to three main performance indexes: IPC, Transaction Rate on L2 (TR_{L2}), and Transaction Rate on main memory (TR_{MM}), both

[1] In this chapter the term co-runner refers to all the processes that run concurrently on the multicore processor.

presented in transactions per microsecond. The transaction rate refers to the number of transactions occurred at a given level of the memory system. If the memory hierarchy of the experimental platform included an L3 cache level, it would also become contention point and the TR_{L3} should be characterized. Notice that to quantify the TR of a given cache level, that is, the bandwidth requirements of the running processes in that level, we need to measure the number of transactions a process experiences between the cache level and its immediately upper level. Nevertheless, when the processor does not offer the accounting of such events, the TR can be accurately obtained by measuring the misses that the processes experience in the upper cache level. For instance, the TR_{L2} of a process can be calculated with the number of misses in the L1 cache. This is the case of the Xeon X3320 processor, and thus we calculate the TR values from the miss values provided by performance counters. More precisely, together with the *unhalted_core_cycles* event, we use the *last_level_cache_misses* and *L2_rqsts* events to measure TR_{MM} and TR_{L2}, respectively.

Figure 4.1 depicts the IPC for the studied benchmarks, while Figure 4.2 and Figure 4.3 show their TR_{MM} and TR_{L2}, respectively. A high correlation between IPC and TR_{MM} is observed since the five benchmarks with the lowest IPC (*mcf*, *astar*, *milc*, *soplex*, and *lbm*) present relatively high TR_{MM} values. TR_{L2} presents a lower impact, although when it surpasses 40 transactions/microsecond the IPC is usually lower than 1 (*mcf*, *cactusADM*, *leslie3d*, *soplex*, *gemsFDTD*, and *lbm*), except for *libquantum* and *bwaves*.

A given benchmark can be classified as memory-bounded when its TR_{MM} is high enough

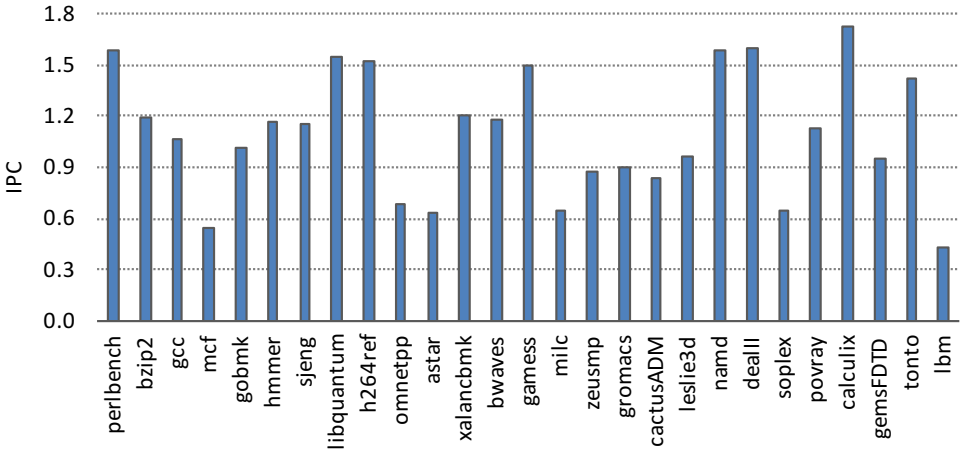


FIGURE 4.1: IPC for each SPEC CPU2006 benchmark.

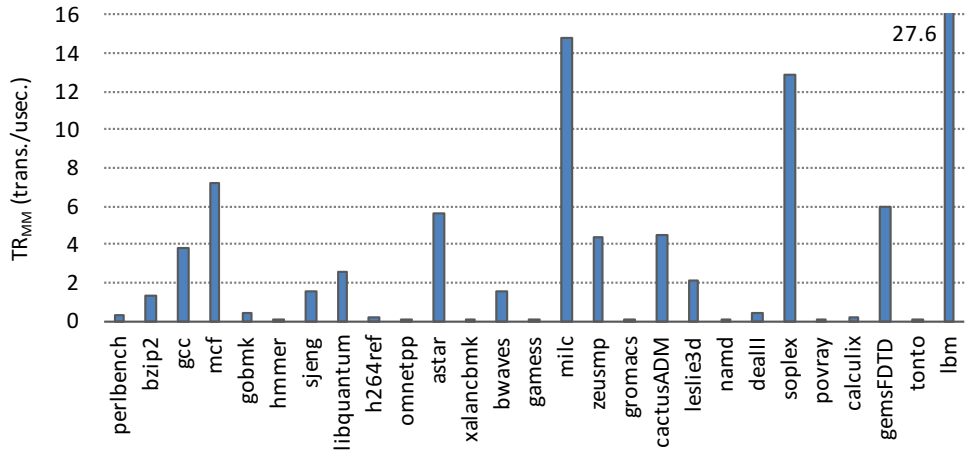


FIGURE 4.2: TR_{MM} for each SPEC CPU2006 benchmark.

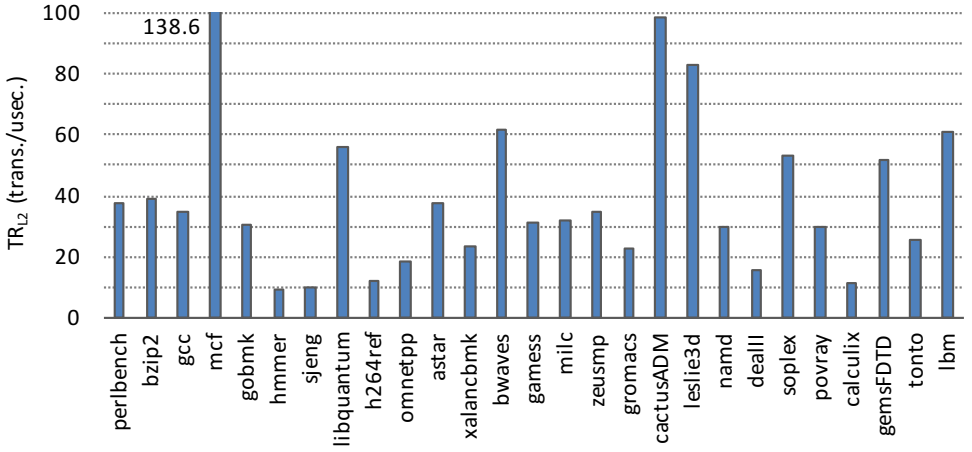


FIGURE 4.3: TR_{L2} for each SPEC CPU2006 benchmark.

to significantly increase main memory bandwidth contention. In such a case, the benchmark will show a low IPC and will potentially affect the IPC of the co-runners. Likewise, a benchmark is considered to be L2-bounded when its TR_{L2} can cause L2 bandwidth contention, which will affect the performance of those applications sharing the same L2. Note that L2-bounded does not necessarily mean memory-bounded. This is the case of *leslie3d*, with a TR_{L2} by about 80 transactions/microsecond but a TR_{MM} around 2 transactions/microsecond.

To remark that the effect of cache hierarchy bandwidth contention is expected to grow in future manycore processors where the LLC cache structures are being shared by an increasing number of cores, most of them implementing multithreading capabilities.

4.1.2 Microbenchmark Design

To analyze the performance degradation that contention causes in a given benchmark, we designed a synthetic microbenchmark (based on the microbenchmark used in [11]), which is used as co-runner. This microbenchmark creates contention by injecting synthetic traffic in an infinite loop. To parametrize the induced contention, the microbenchmark includes as argument the number of *nop* instructions that each iteration of the loop executes. The higher the number *nop* operations included, the lower the contention induced.

The designed microbenchmark can mimic the behavior of either a main memory-bounded or an L2-bounded application. Each iteration of this program executes a memory instruction that misses in a target level L_i of the memory hierarchy and hits in the next level L_{i+1}. Thus, the microbenchmark enables the study of how the performance of a given applications is affected by bandwidth contention at a target level, which is the one where the memory requests hit (i.e., L2 or main memory in the experimental platform). To sum up, the microbenchmark is used to evaluate how the IPC of a given benchmark degrades due to either main memory contention or cache bandwidth contention.

Algorithm 1 presents the core loop of the microbenchmark code. Parameter N refers to the number of lines of the target cache and must be properly tuned according to the cache geometry in order to force continuous misses in that level. The *STRIDE* parameter controls the number of accessed sets in the cache. In order to precisely control this number, the array A is allocated using *huge pages* [59], as explained below.

When using virtual memory, the address translation mechanism translates the virtual addresses used by the processes into physical addresses to access the caches. Virtual

Algorithm 1 Microbenchmark pseudocode

```
Require: N, nops, STRIDE
1: char A[N][CACHE_LINE_SIZE]
2: while (1) do
3:    for (i=0; r<100; i+=STRIDE) do
4:       A[i][0] = 1;
5:    end for
6:    for (i=0; i<#nops; i++) do
7:       asm("nop");
8:    end for
9: end while
```

addresses can be logically split into virtual page number and page offset. In the address translation process, the TLB translates the virtual page number to the physical page number, while the offset of the virtual page is kept for the physical address.

For common 4 KB pages typically used in Linux, the page offset depends on the 12 less significant bits of the address, while the 20 most significant bits are used to identify the virtual page. In this way, assuming a 64-byte cache line size, when accessing a cache with more than 64 sets, a process can not precisely control which set will be accessed since some bits that identify the set depend on the physical page number provided by the TLB, which is unknown to the running process. Therefore, the accessed sets are unpredictable when crossing page boundaries. To deal with this shortcoming, we use *huge pages* of 2 MB instead of typical 4 KB pages, which in the experimental platform allows a user process to determine the cache set that is accessed each iteration just modifying the page offset.

We configure the stride to access 25% of the cache sets. By using such a stride, the maximum impact on the L2 miss ratio during the experiments is only about 3%. Note that accessing a smaller percentage of the cache sets will increase this miss ratio because more conflicts will arise (the microbenchmark will replace the blocks faster). On the other hand, accessing a higher percentage implies using smaller strides, which causes the hardware prefetcher to interfere in the access pattern of the microbenchmark, modifying its parametrized TR and consequently affecting the experiment results.

To ensure the desired microbenchmarks behavior, loop indexes are mapped to registers using the C language *register* keyword and the code is compiled with the *-O0* optimization flag. The hardware prefetcher is disabled to check that the microbenchmark exhibits the desired behavior, but it is enabled again to perform the performance degradation analysis and the experimental evaluation.

4.1.3 Degradation due to Main Memory Contention

To check the performance degradation caused by main memory contention, we designed two experiments. The first experiment is aimed at checking the impact of the traffic created by the co-runners on the performance of a given benchmark. The second studies

how the number co-runners and the core they are launched to affect the performance of the benchmarks.

The first experiment is designed assuming that the system is fully loaded; that is, each core is busy running a process. To this end, each benchmark is concurrently launched with three memory-bounded instances of the microbenchmark. To explore the effects of having different traffic amounts, the microbenchmark is configured to obtain TR_{MM} values ranging from 5 to 70 transactions/microsecond for each instance. The highest value of the range is the maximum number of main memory transactions/microsecond the microbenchmark can perform in the experimental platform.

Figure 4.4 presents the results of this experiment. As observed, the amount of memory traffic generated by the microbenchmark can strongly affect the performance of the applications. In some cases, performance drops exceed 50%. This is the case of *mcf*, *libquantum*, *milc*, *soplex*, and *lbm*, when the three instances of the microbenchmark are tuned to have a TR_{MM} equal to 70 transactions/microsecond. Few applications, like *hmmer*, *gamess*, *nand*, or *povray*, are lightly affected since they show very low transaction rate between L2 and main memory. As expected, the lower the TR_{MM} of the microbenchmark the smaller the performance degradation. However, some benchmarks, like *libquantum*, *milc*, *soplex*, and *lbm*, show important performance drops (greater than or close to 10%) even for a TR_{MM} of the microbenchmark equal to 5 transactions/microsecond.

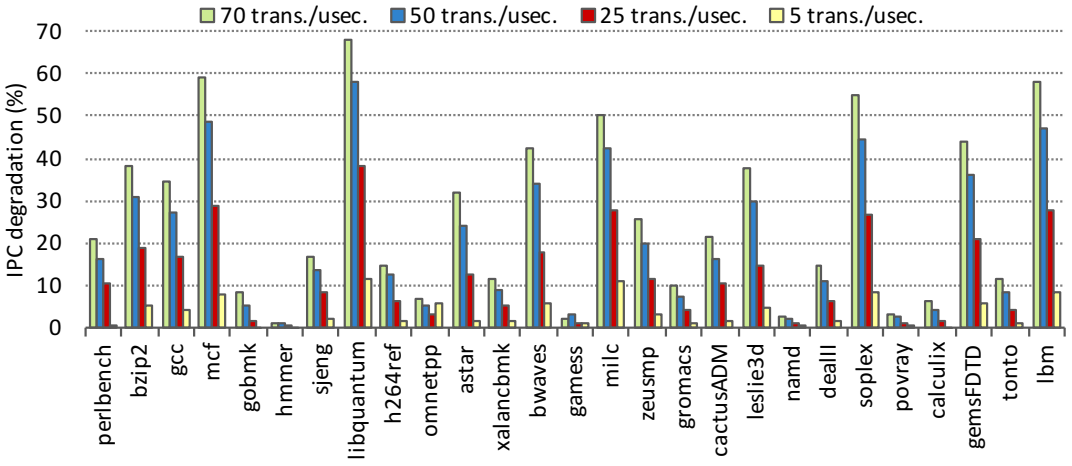


FIGURE 4.4: IPC degradation due to main memory contention varying the TR_{MM} of the co-runners.

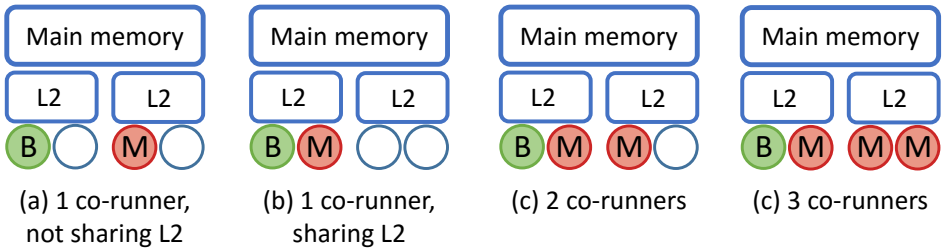


FIGURE 4.5: Analyzed microbenchmarks scenarios.

The second experiment varies the number of co-runners as well as the core on which they are executed. The microbenchmark instances are launched with a TR_{MM} equal to 50 transactions/microsecond. Figure 4.5 shows the four scenarios analyzed and Figure 4.6 presents the results.

Notice that the benchmarks experience a performance degradation in scenario *d* similar to that of scenario *c*, despite there is one extra instance of the microbenchmark running in scenario *d*. This means that memory bandwidth is already saturated with two instances of the microbenchmark for almost all the studied benchmarks. Regarding the scenarios with one co-runner (*a* and *b*), most benchmarks suffer higher IPC degradation when the microbenchmark runs on a core that shares the LLC with the core running the benchmark (scenario *b*), since in this case the processes are affected by both L2 cache and main memory bandwidth contention. Only a few memory-bounded benchmarks (*milc*, *GemsFDTD*, and *lbm*) suffer higher performance degradation when the microbenchmark

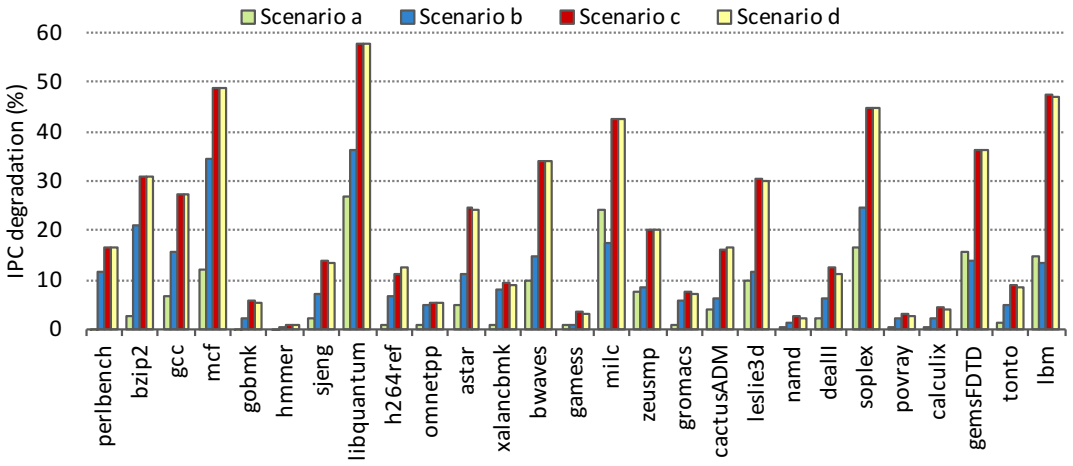


FIGURE 4.6: IPC degradation due to contention in the four studied scenarios.

does not share the LLC (scenario *a*). A priori, scenario *a* should cause lower performance degradation than scenario *b*, where the processes share the LLC cache additionally to the main memory. However, when the LLC cache is not shared and there is not L2 bandwidth contention, the microbenchmark effectively reaches higher main memory transaction rates, increasing the contention at this point. This fact explains why some benchmarks suffer higher performance degradation in scenario *a* than in scenario *b*.

In short, some benchmarks suffer additional degradation from the cache hierarchy contention (scenario *b*) while others are mainly affected by memory bandwidth contention (scenario *a*). Therefore, it is critical to consider both of them when scheduling on machines with a complex memory hierarchy.

4.1.4 Degradation due to L2 Contention

To evaluate the performance degradation caused by L2 contention, the microbenchmark parameters are tuned to stress the L2 cache but not the main memory. That is, the memory accesses will miss on the L1 cache and hit on the L2 cache. Since each L2 cache is shared by a pair of cores, experiments focus only on a single L2 cache. Two processes are launched together, one SPEC benchmark and one L2-bounded instance of the microbenchmark. Hence, there is no benchmark running on the other pair of cores. We vary the induced TR_{L2} of the co-runner from 20 to 290 transactions/microsecond, which is the maximum value reachable in the platform.

Figure 4.7 shows the performance degradation suffered by the benchmarks in this experiment. As observed, the IPC of some benchmarks like *mcf* and *soplex* is strongly affected (IPC degradation is even higher than 10%) by the traffic created by other processes competing for the L2 cache. In addition, twelve benchmarks from twenty seven have a degradation higher than or close to 5% when they are co-scheduled with an L2-bounded instance of the microbenchmark with TR_{L2} equal to 290 transactions/microsecond. This means that some benchmarks are highly sensitive to the L2 accesses of the co-runners. In fact, in some benchmarks like *bzip2*, *h264ref*, *omnetpp*, *xalancbmk*, or *povray* the IPC degradation due to L2 contention can be higher than the caused by main memory contention, when the corresponding benchmark runs concurrently with one instance of the microbenchmark. For example, the performance degradation of

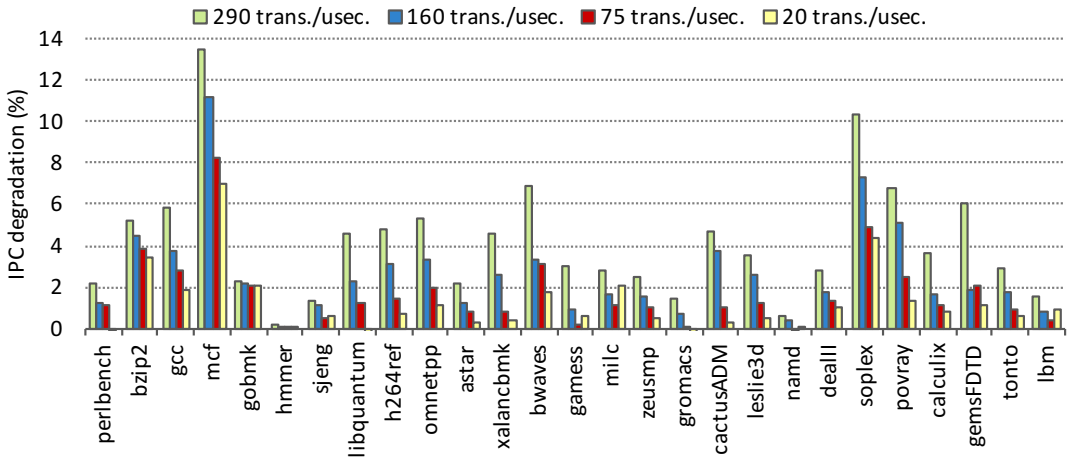


FIGURE 4.7: IPC degradation due to L2 contention varying the TR_{L2} of the co-runners.

bzip2 caused by main memory contention when running concurrently with one memory-bounded microbenchmark is 2%, while one L2-bounded microbenchmark can degrade its performance up to 5%.

Therefore, in this work we claim that, since the current industry trend is to increase the number of cores as well as their multithreading capabilities, a bandwidth-aware scheduling policy for each level of the cache hierarchy can help the scheduler to improve the system performance.

4.1.5 Degradation Running in Bandwidth-Aware Scheduling Scenarios

The last experiment analyzes the IPC degradation suffered by the benchmarks assuming a fixed main memory bandwidth utilization generated by all the processes running concurrently. The IPC degradation is evaluated for a bandwidth utilization of 30 transactions per microsecond, which is the average IABW of the evaluated mixes (see Section 4.3.2). This experiment reproduces the common situation created by state-of-the-art bandwidth-aware schedulers, which try to achieve a constant bandwidth utilization as close as possible to the IABW. Therefore, the experiment obtains an IPC degradation that approaches to the suffered by each benchmark when it is executed under this kind of schedulers. In order to simulate the described situation, the benchmarks are executed concurrently with three instances of the microbenchmark. The TR_{MM} of the

microbenchmark is tuned to reach an overall amount of 30 memory transactions/microsecond, considering the benchmark plus the three instances of the microbenchmark.

Figure 4.8 shows the results of this experiment. The observed degradation is highly correlated with the TR_{MM} presented by the benchmarks (see Figure 4.3). Benchmarks with low TR_{MM} are not sensitive to the contention between L2 and main memory since their main memory accesses are not frequent. In fact, benchmarks with TR_{MM} lower than 2 transactions/microsecond suffer an IPC degradation below 5% (except *dealII*, *bzip2*, and *libquantum*, although the TR_{MM} of the last two is close to 2 transactions/microsecond). In contrast, all the benchmarks with TR_{MM} above 2 transactions/microsecond suffer a higher IPC degradation, which surpasses 10%, with the only exception of *astar*.

Depending on the degradation level, benchmarks can be classified in two categories. The *little sensitive* group includes the processes with an IPC degradation below 5%, which are little affected by the bandwidth-aware scheduling. On the other hand, benchmarks with an IPC degradation between 5% and 35% are included in the *sensitive* category, since their degradation due to bandwidth-aware scheduling is higher. Note that these bounds are appropriate since only two benchmarks present degradations between 5% and 10%.

The degradation observed in this experiment motivates the design of a scheduling algorithm that considers the performance degradation of the processes in this scenario

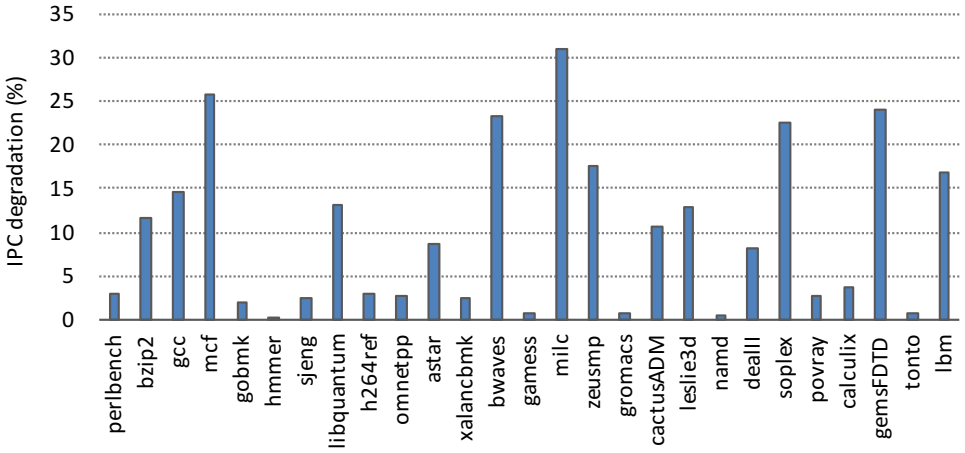


FIGURE 4.8: IPC degradation with total TR_{MM} of 30 transactions/microsecond when running with three co-runners.

to achieve higher performance. Such scheduling algorithm is proposed in Section 4.2.3, and uses the degradation measured in these experiments to execute the processes that suffer higher degradation in those execution periods with less main memory bandwidth requirements.

4.2 Memory-Hierarchy Bandwidth-Aware Scheduling

4.2.1 Baseline Main Memory Bandwidth-Aware Scheduler

Numerous schedulers have been proposed dealing with main memory bandwidth contention. Most proposals work as follows. First, they block the running processes, read performance counters, and update the bandwidth requirements of the processes for the next quantum from the counter values. Then, the scheduler selects which processes will be run concurrently during the next quantum according to their expected bandwidth utilization.

Typically, schedulers have pursued to keep full utilization of the available bandwidth, by selecting processes trying to match the peak memory bus bandwidth [14]. However, recent works proved that contention could exist before the bandwidth utilization reaches the peak bandwidth.

This chapter uses as baseline the scheduler proposed by Xu et al. [11]. This work defines the IABW using Equation 4.1, which quantifies the main memory bandwidth demand of a workload. The IABW is calculated as the sum of the number of main memory transactions performed by all the processes that compose the mix divided by its ideal execution time. This ideal time refers to the execution time of the mix assuming that there is no contention when the processes are concurrently executed (i.e,. assuming each process takes the same execution time as in stand-alone execution). Thus, to calculate the IABW the scheduler needs to know the stand-alone execution time and TR_{MM} of the benchmarks to be run.

$$IABW = \frac{\sum_{p=0}^{P} (TR_{MM}^{p}) * T^{p}}{\frac{\sum_{p=0}^{P} T^{p}}{\#cores}} \tag{4.1}$$

By scheduling jobs whose memory bandwidth requirements approach the IABW, performance degradation is reduced since bandwidth utilization is balanced along the workload execution time, so reducing contention.

4.2.2 Memory-Hierarchy Bandwidth-Aware Scheduler

The performance degradation analysis discussed in Section 4.1.3 and Section 4.1.4 claims for the necessity of a job scheduling policy that is aware of the available bandwidth in each potential contention point of the memory hierarchy, and not only of the main memory bandwidth (as stated in previous proposals). Therefore, the scheduler must monitor the transaction rates that each process experiences in any level of the memory hierarchy.

The proposed Memory-hierarchy bandwidth-aware scheduler addresses the target bandwidth at each contention point and schedules the processes in n steps (as many as levels with at least two shared caches in the hierarchy plus the main memory). The strategy follows a top-down approach, that is, in the first step processes are selected to match a target main memory bandwidth (upper contention point in Figure 1.1). Then, the LLC bandwidth is addressed by balancing the transactions of caches in the immediately higher level (closer to the cores). After that, contention points of the following levels of the cache hierarchy with at least two shared caches are addressed (if they exist). At the end, jobs are allocated to cores so that the bandwidth along the cache hierarchy is balanced. Notice that by using cache bandwidth to guide the scheduling strategy, the policy also takes into consideration cache space contention implicitly.

Algorithm 2 discusses the pseudocode of the Memory-hierarchy bandwidth-aware scheduler. The algorithm can be seen as logically divided in an initialization step and three phases. In the initialization step the scheduler calculates the IABW of the mix, for which the stand-alone execution time and TR_{MM} of the benchmarks of the mix must be provided to the scheduler.

In the first phase (lines 3 to 8), until all the processes have completed their execution, the scheduler repeats the following steps. First of all, the scheduler stops the processes running during the last quantum and updates their TR values. To calculate the TR values, the scheduler uses the performance counters to gather for every process running

Algorithm 2 Memory-hierarchy bandwidth-aware scheduler

Require: Benchmarks submitted with execution time and TR_{MM} in stand-alone execution

1: $IABW = \frac{\sum_{p=0}^{P}(TR_{MM}^{p})*T^{p}}{\frac{\sum_{p=0}^{P}T^{p}}{\#cores}}$
2: **while** there are unfinished jobs **do**
3: Block the executing processes and place them at the queue tail
4: **for** each process P executed in the last quantum **do**
5: **for** each cache level L **do**
6: Update TR for process P in cache level L
7: **end for**
8: **end for**
9: BW_{Remain} =IABW
10: Select the process P_head at the queue head
11: $\text{BW}_{Remain} -= TR_{MM}^{P\_head}$, CPU_{Remain} = #cores−1
12: **while** $\text{CPU}_{Remain} > 0$ **do**
13: select the process P that maximizes
14: FITNESS(p) = $\frac{1}{\left|\frac{BW_{Remain}}{CPU_{Remain}} - TR_{MM}^{P}\right|}$
15: $\text{BW}_{Remain} -= TR_{MM}^{P}$, $\text{CPU}_{Remain}--$
16: **end while**
17: **for** each level i in the cache-hierarchy with shared caches beginning from the LLC **do**
18: $\text{AVG\_TR}(\text{L}_i) = \frac{\sum TR_{L(i)}}{\#Caches\ \ at\ \ Li}$
19: **for** each cache in level L_i **do**
20: BW_{Remain} = AVG_TR(L_i), CPU_{Remain} = # cores sharing the cache
21: **while** $\text{CPU}_{Remain} > 0$ **do**
22: From the remaining processes selected to share the immediately lower memory level, select the process P that maximizes
23: FITNESS(p) = $\frac{1}{\left|\frac{BW_{Remain}}{CPU_{Remain}} - TR_{Li}^{P}\right|}$
24: $\text{BW}_{Remain} -= \text{TR}_{Li}^{P}$, $\text{CPU}_{Remain}--$
25: **end while**
26: **end for**
27: **end for**
28: Unblock the processes, and allocate them in the chosen core
29: Sleep during the quantum
30: **end while**

during the last quantum its number of main memory and cache misses as well as the number of executed cycles. The TR values reached during a given quantum are used by the scheduler as predicted TR requirements for the next quantum. In particular, in our experimental platform, for each process, TR_{MM} and TR_{L2} are updated with the gathered values to predict the bandwidth requirements of main memory and L2, respectively. Once the TRs are updated, the processes are inserted at the tail of the processes queue.

In the process selection phase (lines 9 to 16), the processes to be scheduled for the next quantum are selected attending to their main memory bandwidth requirements. In line

9, the scheduler initializes BW_{Remain} to the target IABW and CPU_{Remain} to the number of cores in the experimental platform. Then, the process located at the queue head is selected to avoid process starvation, while the remaining processes are selected according to the fitness function [13, 14] (lines 12 to 16). This function quantifies, for each process, the gap between the predicted TR of the process and the remaining bandwidth divided by the number of processes that still need to be selected (CPU_{Remain}). The process that maximizes the fitness function is selected to run during the next quantum, updating BW_{Remain} and CPU_{Remain} accordingly, until the number of selected processes reaches the number of cores. The result of this step is the list of the processes that will be executed during the next quantum.

Finally, in the process allocation phase (lines 17 to 27), the algorithm deals with balancing the contention for bandwidth at the shared levels of the cache hierarchy (e.g., L3 and L2 in Figure 1.1). To this end, for each level L_i, the required TR of all caches at L_{i-1} level is estimated and averaged considering the number of caches at L_i level (line 18). Then, for each cache, BW_{Remain} and CPU_{Remain} are set to the average TR and the number of cores sharing that cache, respectively (line 20). Next, processes are assigned to each cache structure at L_i level according to the fitness function, as done with the main memory bandwidth (lines 21 to 25). Notice that by using the proper inputs, the fitness function can be directly used at any cache level of the memory hierarchy. The loop ends when it reaches the highest level with shared caches (i.e., L2 in the experimental platform) where it selects two processes for each L2 cache, which can subsequently be allocated to any of the two cores sharing the L2 cache.

4.2.3 IPC-Degradation Memory-Hierarchy Bandwidth-Aware Scheduler

The analysis of the performance degradation under bandwidth-aware scheduling scenarios (Section 4.1.5) helps providing useful information to enhance scheduling and allocation decisions. As mentioned above, the Memory-hierarchy bandwidth-aware scheduler calculates the IABW of the mix, and then tries to schedule the processes for the next quantum to approach an overall bandwidth utilization as close as possible to the IABW. The IPC-degradation memory-hierarchy bandwidth-aware scheduler improves the scheduling decisions using the benchmark classification performed in Section 4.1.5.

This classification arranges the benchmarks as *sensitive* and *little sensitive*, depending on whether they suffer significant performance degradation or not when scheduled with other processes where the overall TR_{MM} is close to the IABW. As observed in Figure 4.8, the performance degradation experienced by processes in this situation widely differs among them, so a smart scheduler can use this information to increase the performance.

The key idea of the proposed technique consists in favoring the performance of *sensitive* benchmarks. To this end, when a *sensitive* benchmark is selected to run during the next quantum, the scheduler selects its co-runners to reach an estimated main memory bandwidth consumption below the IABW. To compensate this variation, *little sensitive* benchmarks will be scheduled to execute in situations where the total bandwidth utilization is above the IABW. Nevertheless, since the most sensitive processes are executed in favorable situations, a global performance increase is expected.

To incorporate this technique in the scheduling algorithm, a *penalty* coefficient is included. This coefficient is defined as a proportional part of the IPC degradation suffered by each benchmark. Different coefficient values were checked to maximize the performance (see Section 4.4.2), resulting the best penalty coefficient as a fifth of the process IPC degradation for *sensitive* benchmarks and zero for *little sensitive* benchmarks.

Algorithm 3 presents the proposed scheduler considering the performance degradation of the processes in bandwidth-contention aware scheduling scenarios. It extends the Memory-hierarchy bandwidth-aware scheduler presented before (Algorithm 2). The discussion focusses on the differences between both algorithms, which are highlighted in red color, and how they affect the scheduling that the new scheduler performs. Note that the first scheduling phase (lines 2 to 8), which gathers the performance counts, and the process allocation phase (lines 17 to 27) have not been modified.

In the initialization step, the scheduler calculates the *swelled IABW* (SIABW), which replaces the IABW as the target main memory bandwidth utilization to be achieved in each quantum. The SIABW is calculated in a similar way to the IABW, but adding the penalty coefficient of each process to its average TR_{MM}. By including the penalty coefficient of the processes, the calculated SIABW of a mix is higher than the IABW.

To select the processes that will be executed in the next quantum (lines 9 to 15), the algorithm starts setting BW_{Remain} to the SIABW (line 9). As Algorithm 2 does, the

Algorithm 3 IPC-degradation memory-hierarchy bandwidth-aware scheduler

Require: Benchmarks submitted with execution time and TR_{MM} in stand-alone execution, and penalty coefficients.

1: $SIABW = \frac{\sum_{p=0}^{P}(TR_{MM}^{p}+PenaltyCoef^{p})*T^{p}}{\frac{\sum_{p=0}^{P}T^{p}}{\#cores}}$

2: **while** there are unfinished jobs **do**
3: Block the executing processes and place them at the queue tail
4: **for** each process P executed in the last quantum **do**
5: **for** each cache level L **do**
6: Update TR for process P in cache level L
7: **end for**
8: **end for**
9: $\text{BW}_{Remain} = \text{SIABW}$
10: Select the process P_head at the queue head
11: $\text{BW}_{Remain} -= (TR_{MM}^{P\_head} + PenaltyCoef^{P\_head})$, $\text{CPU}_{Remain} = \#\text{cores}-1$
12: **while** $\text{CPU}_{Remain} > 0$ **do**
13: select the process P that maximizes
14: $\text{FITNESS(p)} = \frac{1}{\left|\frac{BW_{Remain}}{CPU_{Remain}} - (TR_{MM}^{P}+Penalty\ Coef^{p})\right|}$
15: $\text{BW}_{Remain} -= (\text{TR}_{MM}^{P}+\text{PenaltyCoef}^{P})$, $\text{CPU}_{Remain}--$
16: **end while**
17: **for** each level i in the cache-hierarchy with shared caches beginning from the LLC **do**
18: $\text{AVG\_TR}(\text{L}_i)=\frac{\sum TR_{L(i)}}{\#Caches\ \ at\ \ Li}$
19: **for** each cache in level L_i **do**
20: $\text{BW}_{Remain} = \text{AVG\_TR}(\text{L}_i)$, $\text{CPU}_{Remain} = \#$ cores sharing the cache
21: **while** $\text{CPU}_{Remain} > 0$ **do**
22: From the remaining processes selected to share the immediately lower memory level, select the process P that maximizes
23: $\text{FITNESS(p)} = \frac{1}{\left|\frac{BW_{Remain}}{CPU_{Remain}} - TR_{Li}^{P}\right|}$
24: $\text{BW}_{Remain} -= \text{TR}_{Li}^{P}$, $\text{CPU}_{Remain}--$
25: **end while**
26: **end for**
27: **end for**
28: Unblock the processes, and allocate them in the chosen cores
29: Sleep during the quantum
30: **end while**

process at the queue head is the first selected to run the next quantum to avoid process starvation (line 10). Note that after a process is selected, BW_{Remain} is updated subtracting the TR_{MM} of the process plus its penalty coefficient (line 11).

The penalty coefficient will be subtracted from the BW_{Remain} for every selected process (line 15) and allows the *sensitive* processes to *reserve* some additional bandwidth (equal to their penalty coefficient) that will not be effectively used. The effect of this action is that, when selecting *sensitive* benchmarks, the overall bandwidth contention

will be lower, thus favoring their execution. This also means that *little sensitive* benchmarks will run in scenarios with higher main memory bandwidth consumption, but their performance is less affected by this increased contention.

The remaining processes are selected, until the number of cores is reached, according to the new fitness function (line 14). The function has been modified to consider the penalty coefficient of the processes as an extra bandwidth requirement. Thus, a process should have a TR_{MM} plus its penalty coefficient as close as possible to the remaining bandwidth per remaining core to be selected. From a practical point of view, the change in the fitness functions restricts the selection of *sensitive* processes by increasing their bandwidth requirements to be selected.

A simple example can help to clarify how the changes affect scheduling. Let's assume that there is one available core and BW_{Remain} is 3 trans./usec. The two candidate processes are *libquantum* and *bwaves*, which present a TR_{MM} of 3 and 2 trans./usec., respectively. As can be observed in Figure 4.8, *libquantum* is a *sensitive* process and its penalty coefficient is by 2.5. Conversely, *bwaves* is *little sensitive* and thus its penalty coefficient is zero. Without considering penalty coefficients, such as in Algorithm 2, *libquantum* is a perfect fit since its TR_{MM} matches the BW_{Remain}. However, considering the penalty coefficients *bwaves* achieves higher fitness and would be the selected process. This is done because *libquantum* is sensitive and the algorithm tries to execute it when the bandwidth consumption is lower to favor its execution. In this case, *libquantum* would be a perfect fit when BW_{Remain} is by 5.5 trans./usec.

In summary, the IPC-degradation memory-hierarchy bandwidth-aware scheduler runs the *sensitive* benchmarks in execution periods and with co-runners where the main memory transaction rate is lower, favoring their performance. On the other hand, *little sensitive* processes run in scenarios with higher main memory transaction rate, but they do not suffer additional performance degradation for this situation.

4.3 Evaluation Setup

The evaluation of the proposed algorithms is performed in the Intel Xeon X3320 system (see Section 3.2.1). To ensure a fair comparison of different scheduling policies, all the

evaluated algorithms are implemented in the scheduling framework (see Section 3.1). The experimental evaluation is carried out following the process selection evaluation methodology, described in Section 3.3.1, which does not relaunch the applications after they complete their target number of instructions. In this chapter, the target number of instructions for each SPEC CPU2006 benchmark is set to the number of instructions each benchmark executes running alone for 120 seconds, and the scheduler quantum is fixed to 200 milliseconds. The performance evaluation is focussed on the turnaround time of the mixes, which measures the time required to complete the execution of the workloads (see Section 3.4 for further details about the metric).

4.3.1 Evaluated Algorithms

The experimental evaluation considers the following scheduling algorithms.

- **Linux:** the default Linux Completely Fair Scheduler (CFS).
- **Baseline Main memory bandwidth-aware (MMaS):** the scheduler proposed by Xu et al. [11].
- **Memory-hierarchy bandwidth-aware (MHaS):** our proposed scheduler that considers the bandwidth requirements along the memory hierarchy to reduce bandwidth contention.
- **IPC-degradation memory-hierarchy bandwidth-aware (IDaS):** our proposed scheduler that, in addition to mitigate bandwidth contention through the memory hierarchy, favors the execution of the *sensitive* applications.

4.3.2 Mix Design

To evaluate the effectiveness of the proposal we design a set of ten mixes. Mixes 1 to 7 contain a number of benchmarks twice as large as the number of cores, while mixes 8 to 10 triple this value. Table 4.1 presents the mixes and their associated IABW. The studied mixes present IABWs between 20 and 40 transactions/microsecond. Notice that this is the range where bandwidth-aware schedulers enable higher performance enhancements. There are benchmarks with poor memory requirements that present a limited number of

misses in both L1 and L2 caches. Hence, if the mix is built only using benchmarks with this behavior, the contention will be low, thus avoiding the necessity of the proposed scheduling policies. As opposite, if all the benchmarks in a mix have a high TR_{MM}, the schedulers will be forced to launch memory-bounded benchmarks together, leaving little room to improve performance. Therefore, a good mix should include a subset of memory-bounded benchmarks mingled with a subset of benchmarks with low memory requirements.

Mixes	Benchmarks	IABW
Mix 1	*GemsFDTD, H264ref, Hmmer, Lbm, Lbm, Mcf, Tonto, Xalancbmk*	34.45
Mix 2	*Astar, Calculix, GemsFDTD, H264ref, Hmmer, Lbm, Mcf, Tonto*	23.46
Mix 3	*Astar, GemsFDTD, Hmmer, Lbm, Lbm, Mcf, Tonto, Xalancbmk*	37.13
Mix 4	*Astar, CactusADM, GemsFDTD, Lbm, Lbm, Mcf, Tonto, Xalancbmk*	39.37
Mix 5	*Astar, Bwaves, CactusADM, Lbm, GemsFDTD, Mcf, Tonto, Xalancbmk*	26.37
Mix 6	*Astar, DealII, GemsFDTD, H264ref, Lbm, Mcf, Namd, Sjeng*	24.31
Mix 7	*CactusADM, GemsFDTD, Mcf, Milc, Lbm, Leslie3d, Tonto, ZeusMP*	26.32
Mix 8	*Astar, Bzip2, DealII, Gcc, GemsFDTD, H264ref, Lbm, Lbm, Mcf, Mcf, Namd, Sjeng*	29.45
Mix 9	*Astar, Bwaves, CactusADM, CactusADM, DealII, Lbm, Lbm, Mcf, Soplex, Tonto, Xalancbmk, ZeusMP*	31.14
Mix 10	*Astar, Bwaves, CactusADM, DealII, GemsFDTD, Lbm, Lbm, Mcf, Milc, Sjeng, Tonto, Xalancbmk*	29.81

TABLE 4.1: Mix composition and IABW of each mix.

4.4 Experimental Evaluation

4.4.1 Performance Evaluation

Figure 4.9 shows the speedup achieved by MMaS and both proposed schedulers, MHaS and IDaS, over the native Linux scheduler considered as baseline. As observed, regardless of the workload, the proposals always provide better performance than the main memory-bandwidth aware scheduler. For MMaS, the achieved speedup widely varies across mixes, ranging from 1.6% to 5.2%, with an average speedup of 3.6%, showing it can improve the performance of the studied mixes with respect to Linux, as stated in [11]. For MHaS, the achieved speedup ranges from 3.4% to 7.3%, averaging 5.4%. These results show that a scheduler considering the contention across the memory hierarchy can improve the performance of a scheduler that only considers the main memory contention. The achieved speedup is further improved by IDaS, whose speedup ranges from 3.7% to 9.6%, averaging 6.6%. The average speedup achieved by IDaS almost doubles the average speedup achieved by MMaS. Furthermore, in half of the mixes (2, 6, 8, 9, and 10), IDaS triples the speedup of MMaS. The main reason behind the performance of MHaS is that it balances the transactions across contention points along the cache hierarchy. Since the experimental platform has two shared L2 caches, the scheduler allocates jobs to cores taking into account that L1 misses must evenly access both L2 caches. In this way, the L2 bandwidth contention is reduced, which turns into performance enhancements.

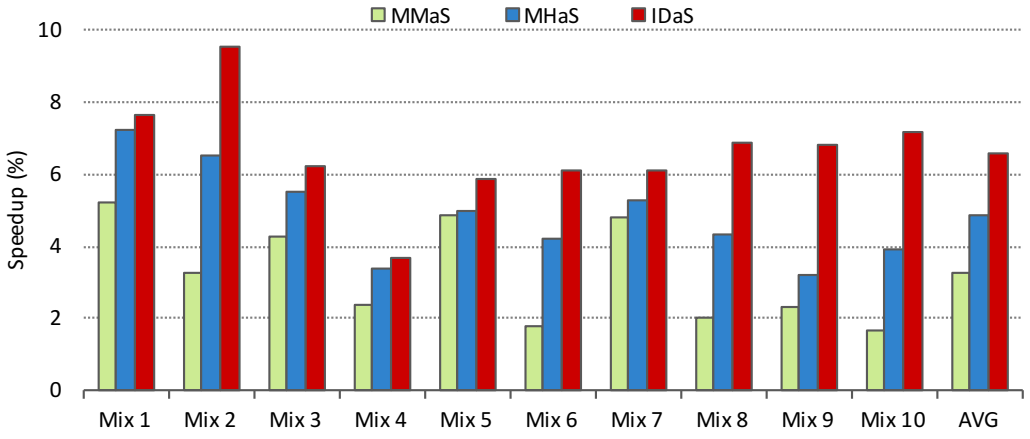


FIGURE 4.9: Speedup of the MMaS, MHaS, and IDaS schedulers over the native Linux scheduler.

To estimate how well this balancing works, we measured the TR_{L2} accessing both L2 caches and calculated their difference. Figure 4.10 presents the results. The histogram represents the frequency of the TR_{L2} difference between both L2 caches for MMaS and MHaS. Results are presented in intervals of 25 transactions per microsecond. The bigger the lower intervals (i.e., smaller difference), the better the accesses are balanced between L2 caches.

For example, if we compare MMaS bar versus MHaS bar in mix 1, we can observe that in MMaS, 40% of time (bottom bar) the TR_{L2} difference between both L2 caches is less than 25 transactions/microsecond. The immediately upper bar indicates that by 30% of times the difference falls in the range $[25, 50]$ and so on. In contrast, for MHaS, the $[0, 25]$ interval frequency increases up to 50% of time, resulting in better TR_{L2} distribution and better performance.

Results show a strong correlation between the frequency distribution and the speedup. For instance, mixes 2, 6, 8, and 10 present the widest distribution variation between both schedulers, which translates in the highest speedup variations. This can be appreciated in the lowest interval, that is $[0, 25]$, in mix 2, but also in the reduction of the intervals above 50 transactions/microsecond in mixes 6, 8, and 10.

To provide a sound understanding of why TR balancing improves the performance, let's look inside the dynamic execution of a mix. In particular, let's focus on mix 2 where MHaS improves by 50% the speedup achieved by MMaS. Figure 4.11 shows the TR_{L2}

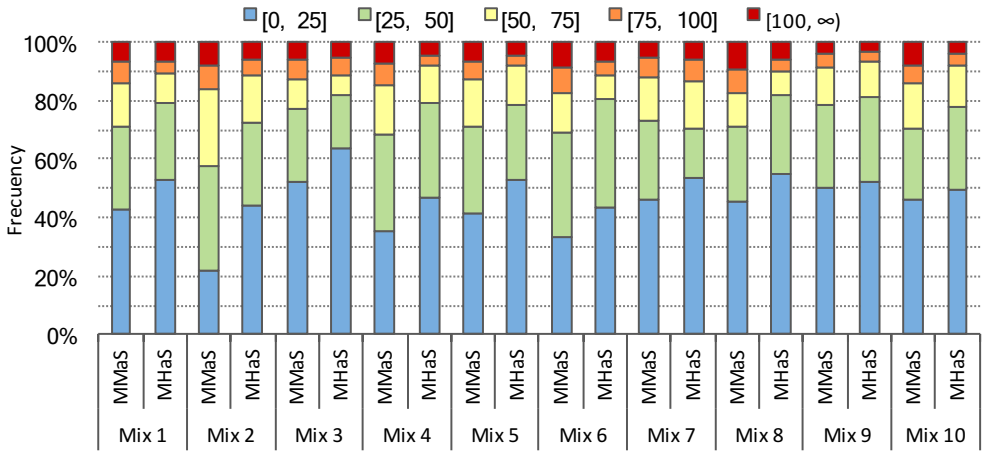


FIGURE 4.10: TR_{L2} differences between the MMaS and MHaS schedulers in the L2 shared caches.

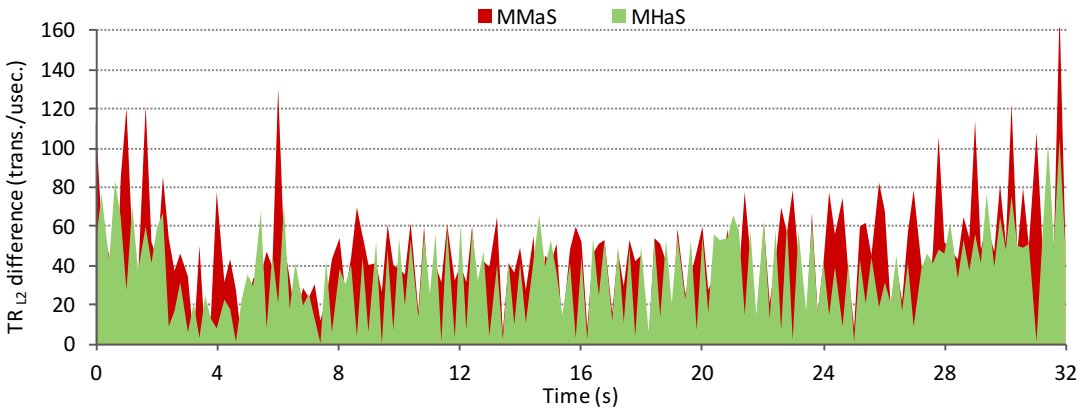


FIGURE 4.11: TR_{L2} differences between the MMaS and MHaS schedulers in the first 32 seconds of execution of mix 2.

difference of each quantum during the first 32 seconds of execution for both schedulers. The plot shows that the TR_{L2} difference for MMaS is usually higher than for MHaS. An even more important observation is that the peaks of this difference, which cause most of the contention are reduced by MHaS, both in number and size.

Figure 4.12 presents the dynamic TR_{L2} differences using the MMaS, MHaS, and IDaS schedulers during the first 275 seconds of execution of mix 2. TR_{L2} differences, which are mainly caused by the *mcf* benchmark appear before in MHaS than in MMaS. Notice that this speedup is not achieved at the expense of increasing the peak heights, since the heights are reduced too. This effect is improved by IDaS that places the peaks ahead of MHaS. Moreover, TR_{L2} differences among most peaks are also improved. Looking at the IDaS plot, it can be appreciated that in many intervals the TR_{L2} difference falls always below 50 transactions/microsecond. Notice that the difference usually falls above this value in MMaS.

Finally, to compare the benefits of IDaS against MHaS, we measured the percentage of benchmarks in each mix that reduce their execution time (speedup) and those that enlarge it (slowdown). Figure 4.13 shows the results. The first two intervals with negative values in the range refer to slowdown while the remaining ones (positive values) refer to benchmarks favored by IDaS. As observed, 9 of 10 mixes are benefited by the IDaS scheduler. Moreover, six mixes present by 60% of their benchmarks favored by the IDaS policy. The penalty coefficient included in IDaS cause the *sensitive* benchmarks to execute in scenarios with less contention. Thus, these benchmarks present speedup.

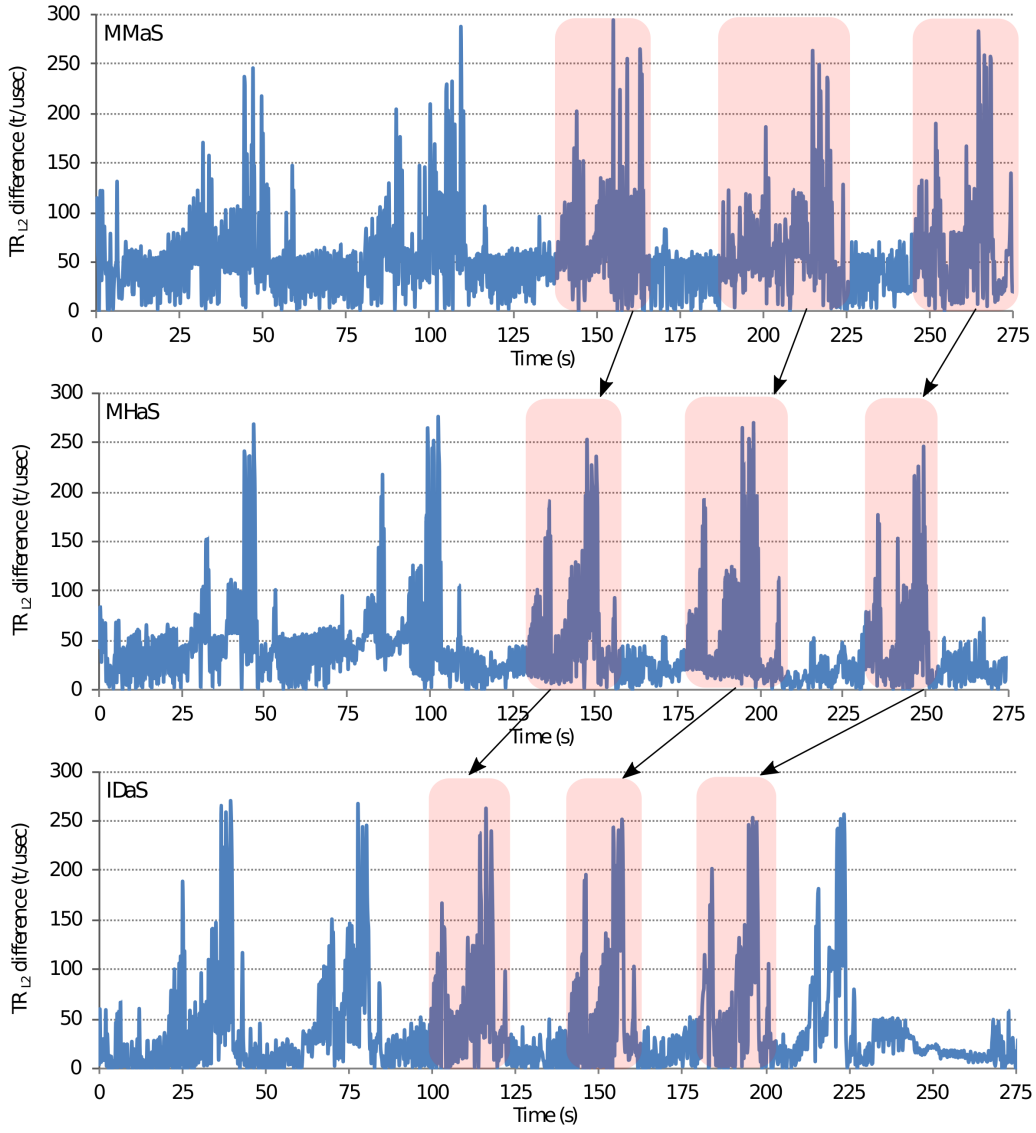


FIGURE 4.12: TR_{L2} difference evolution with time between the MMaS, MHaS, and IDaS schedulers.

On the other hand, *little sensitive* benchmarks are executed in scenarios with higher bandwidth contention, slowing down their execution but with a low impact on the overall performance.

Mix 4 is the only mix where the percentage of benchmarks with slowdown is higher than the percentage of benchmarks with speedup. Even in this mix, the execution time using the IDaS scheduler is better than using the MHaS scheduler. Notice that the individual speedups of the benchmarks do not take into account the fact that at the end of the mix execution, some benchmarks can be executed when the number of processes

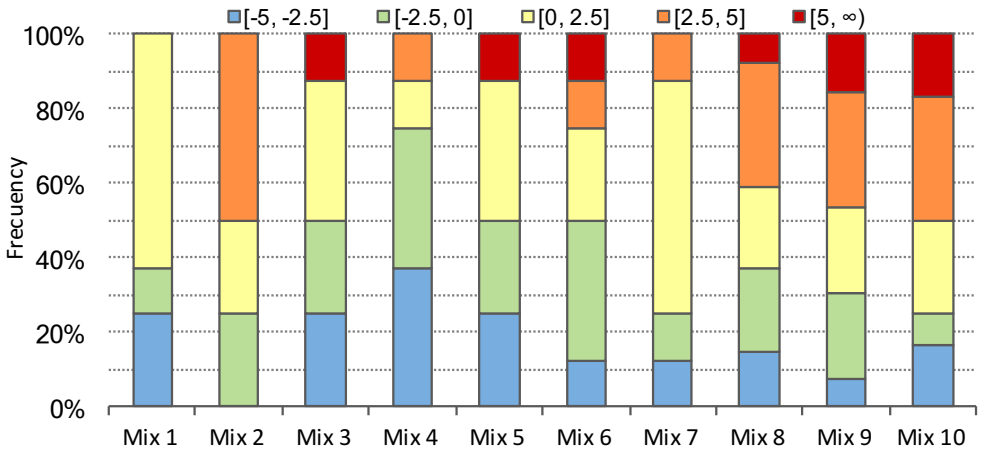


FIGURE 4.13: Speedup of the benchmarks of each mix with the IDaS scheduler against the MHaS scheduler.

is lower than the number of cores. When this situation is long enough, IPC of individual benchmarks is improved since there is less contention, but the mix execution time will not necessarily be improved.

4.4.2 Profiling the Penalty Coefficient

In Section 4.1.5, we discussed that each benchmark suffers different performance degradation when scheduled in the scenarios typically promoted by bandwidth-aware schedulers. To check this degradation, the IPC of each benchmark was measured when the overall main memory transaction rate was 30 transactions/microsecond, and the benchmarks were classified in two categories: *sensitive* and *little sensitive*, depending on whether they show high or low IPC degradation, respectively, when running in this scenario. Based on the observed results, a penalty coefficient with the aim of favoring the execution of the *sensitive* processes is defined (see Section 4.2.3).

The penalty coefficient is defined as directly proportional to the performance degradation of the benchmarks. To check where the highest performance is achieved, we profiled this coefficient for values falling in between 5% and 30% of the performance degradation. Figure 4.14 presents the speedups of the IDaS scheduler over Linux, with different values for the *penalty* coefficient. The figure shows that, on average, the maximum performance is obtained with a penalty coefficient of 20%. This coefficient is also the best one in five mixes (1, 3, 7, 8, and 10) and it is close to the best value in the remaining mixes. The

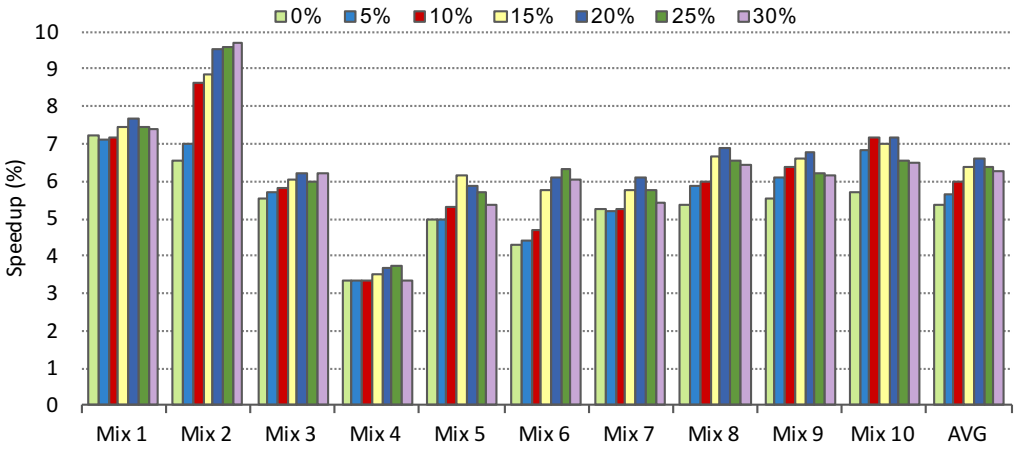


FIGURE 4.14: Speedups of the IDaS scheduler over the Linux scheduler varying the penalty coefficient.

largest difference with the maximum performance appears in mix 5 and it is only around 0.3%.

4.5 Summary

This chapter has addressed cache sharing contention in typical multicore processors, and has proven that the system performance can drop due to bandwidth contention located at different levels of the memory hierarchy. Since bandwidth contention in shared caches is expected to grow in future microprocessor generations with a higher number of cores, and wider and deeper memory hierarchies, we claim that process schedulers should be aware of the bandwidth contention through the cache hierarchy to prevent significant performance loses when running multiprogram workloads.

To deal with this performance concern, in this chapter we have proposed two scheduling algorithms for generic multicore processors. The Memory-hierarchy bandwidth-aware scheduler selects the processes taking into account the main memory bandwidth to reduce the global contention, and then follows a top-down multi-level approach that takes n steps (as many as cache levels with at least two shared caches) to plan a globally balanced schedule for the next quantum. The IPC-degradation memory-hierarchy bandwidth-aware scheduler enhances the first algorithm, favoring the execution of the

processes that are more sensitive to main memory bandwidth contention on schedules with lower main memory bandwidth utilization.

Experimental results show that, compared to the native Linux scheduler, the achieved speedups range from 3.4% to 7.3% and from 3.7% to 9.6%, for the former and latter scheduling algorithms, respectively. The average speedup for the IPC-degradation memory-hierarchy bandwidth-aware scheduler is by 6.6%, which almost doubles the speedup achieved by a state-of-the-art memory-contention aware scheduler.

The work discussed in this chapter has been published in [60], [61], and [12].

Chapter 5

Bandwidth-Aware Scheduling in SMT Multicores

Contention aware schedulers have been extensively used to mitigate the performance degradation caused by bandwidth interference on the memory hierarchy of multicore processors. However, since the L1 cache is implemented within the core pipeline and not shared with other cores, it has been left out of the scope of all these works. Nevertheless, simultaneous multithreading cores share the L1 bandwidth among the threads running on the same core, which turns L1 bandwidth into a potential contention point. This chapter analyzes the impact of L1 bandwidth contention on the performance and proposes a process allocation policy to deal with this contention point, and an entire scheduler for SMT multicores that addresses bandwidth contention at main memory and the L1 cache.

This chapter is organized as follows. First, the potential performance degradation due to L1 bandwidth contention is studied. Next, the proposed SMT bandwidth-aware scheduler, which consists of the Self-reliant main memory bandwidth aware process selection policy and the Dynamic L1 bandwidth-aware process allocation policy, is presented. Finally, the performance evaluation results of both proposed policies and the entire scheduler are discussed.

5.1 Performance Degradation Analysis

This chapter analyzes bandwidth contention on current SMT multicores, with particular emphasis on L1 bandwidth contention. The experiments of this chapter have been performed in a six-core dual-threaded Intel Xeon E5645 processor, as an example of a current SMT multicore. More details of the experimental platform can be found in Section 3.2.2.

5.1.1 Effects of L1 Bandwidth on Performance

Current microprocessors usually deploy a cache hierarchy organized in two or three levels of caches. The first-level cache, the closest one to the processor, is the most frequently accessed one. Consequently, L1 caches are critical for performance and thus, they are designed to provide fast access and high bandwidth.

This section analyzes the relation between L1 bandwidth consumption and processor performance. First, the dynamic behavior in stand-alone execution is analyzed. Then, we study how two co-runners[1] interact each other on their respective performance and L1 bandwidth consumption.

5.1.1.1 Stand-Alone Execution

As a first step to investigate the possible relation between the bandwidth utilization of the L1 cache and the overall processor performance, we measured the average L1 transaction rate (TR_{L1}) and the IPC achieved by each process. To avoid interferences of other applications each benchmark was run alone.

Figure 5.1 and Figure 5.2 depict both average IPC and TR_{L1} of the SPEC CPU2006 benchmarks. At a first glance, a certain correlation can be observed between both metrics since most benchmarks with high IPC also present high TR_{L1}, and conversely, benchmarks with low IPC also experience low TR_{L1}. However, benchmarks with similar IPCs can widely differ in their L1 transaction rates (e.g., *gobmk* and *hmmer*), and vice versa, benchmarks with close TR_{L1} can diverge in the achieved IPC (e.g., *dealII*

[1]The term co-runner is used in this chapter to refer to the processes that run simultaneously on the same core.

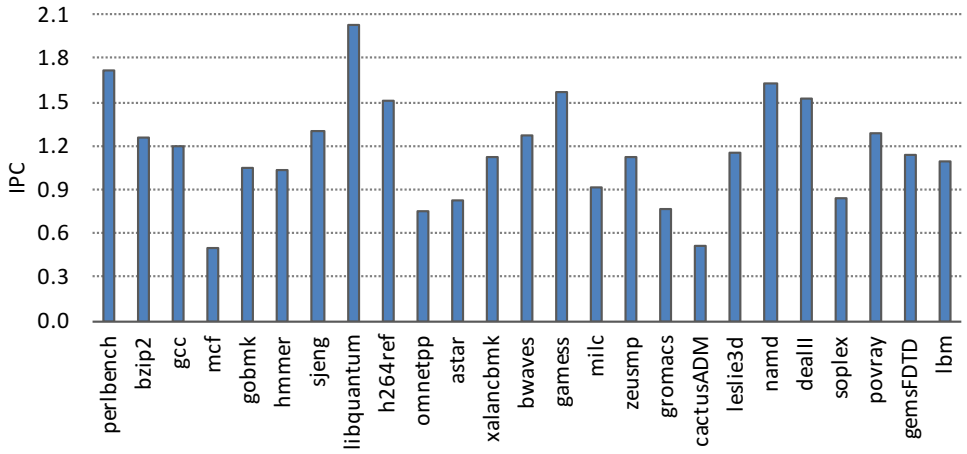


FIGURE 5.1: IPC for each SPEC CPU2006 benchmark.

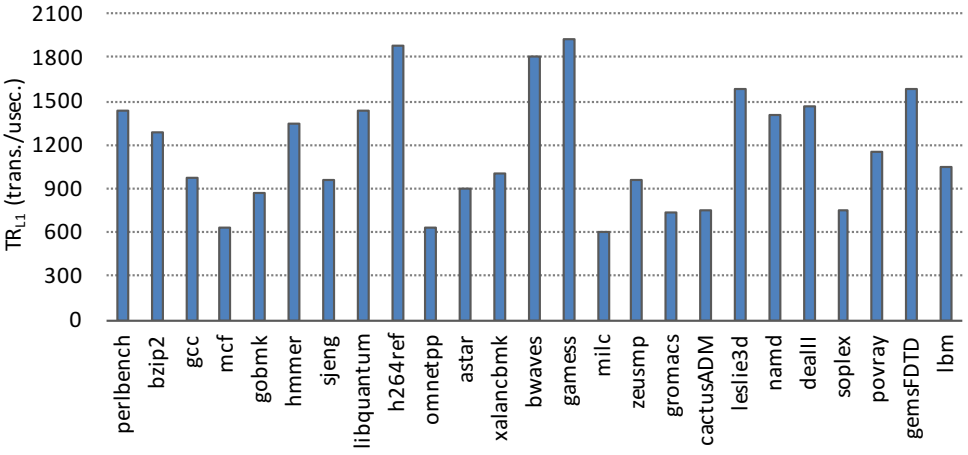


FIGURE 5.2: TR_{L1} for each SPEC CPU2006 benchmark.

and *gemsFDTD*). Thus, although certain similarities appear among both performance indicators, there is no clear evidence about the connection between them.

Nevertheless, it is well known that the benchmark behavior can widely vary over the execution time. Thus although some divergences can appear on the average values, one should look for further insights in the dynamic values of both metrics at run-time.

Figure 5.3 depicts the results of the first 200 seconds of the execution of a representative subset of benchmarks. Each plot presents, for a given benchmark, the IPC and the number of instructions that perform a L1 data cache read per cycle (RPC). Notice that the number of reads does not correspond with the number of loads in the x86 ISA. Some instructions (e.g., arithmetic) can access to the cache since an instruction operand

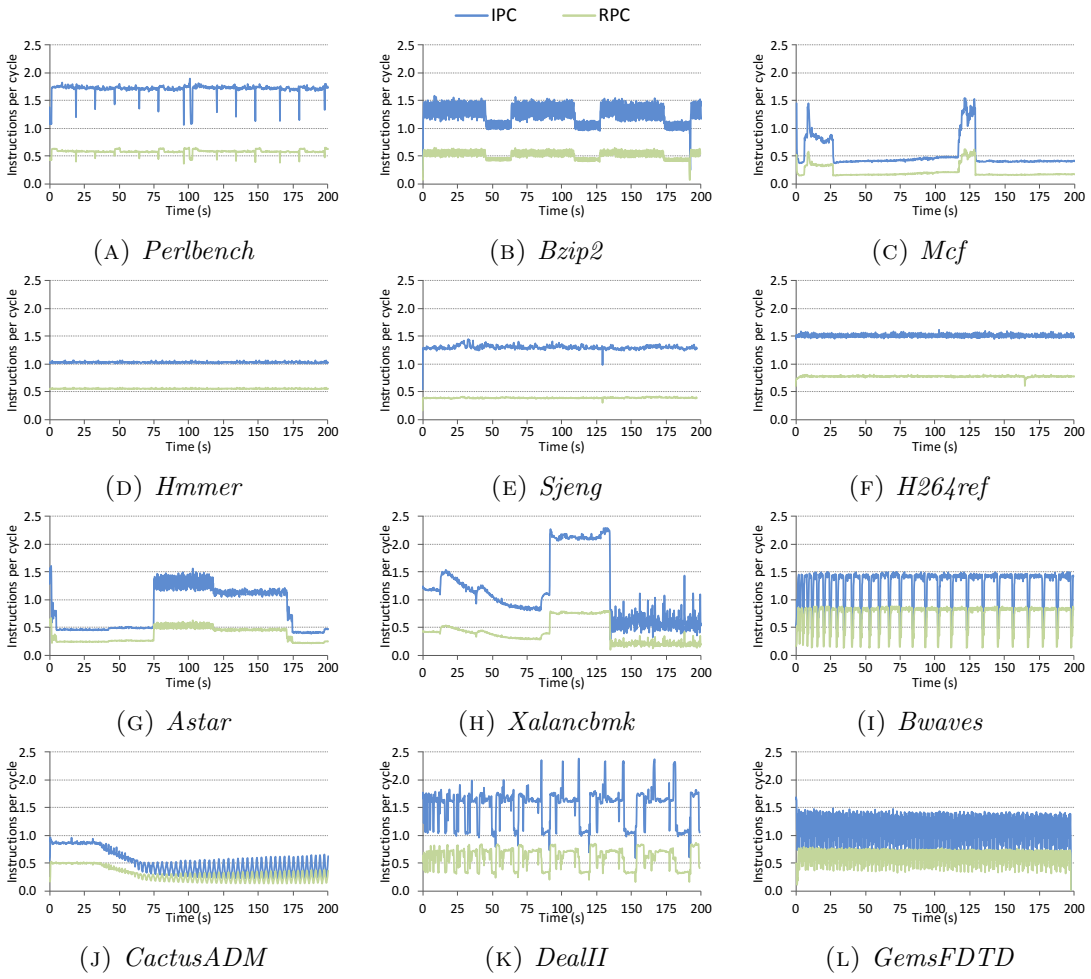


FIGURE 5.3: IPC and RPC evolution over time for a set of benchmarks.

(source or destination) can be a memory location. Note that the RPC of the processes is strongly related with their L1 bandwidth consumption.

The presented plots help detect the strong connection between RPC and IPC metrics. As observed, both metrics show an almost identical shape during the entire execution time across all the benchmarks. The metrics follow the same trend (rises and drops) in a synchronized way. This means that a high (or a low) IPC is typically correlated with high (or low) L1 bandwidth consumption. Note that, as soon as the L1 bandwidth starts to decrease (or increase), the performance of the process follows the same trend.

The finding that both IPC and RPC for a process follow a so synchronized and correlated trend has important connotations. It implies that when a process shows high performance during a running period, it will certainly show high L1 bandwidth consumption. And vice versa, if a process is consuming a small amount of L1 bandwidth then its IPC

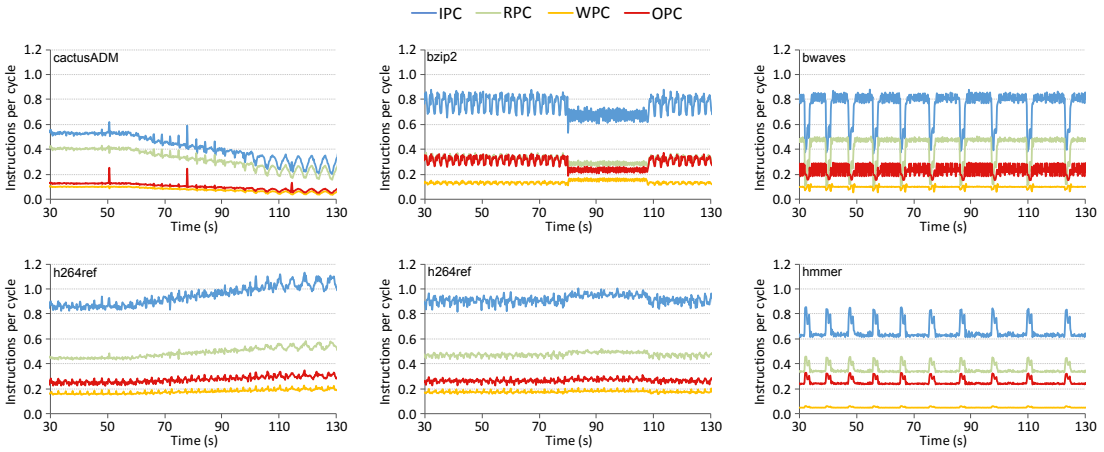
is expected to be low. Therefore, to allow processes achieve their best performance they must be run so that they can get the highest bandwidth consumption. To favor such scenarios, changes in the process allocation should be allowed dynamically at run-time, since some benchmarks present phases with widely different L1 bandwidth requirements.

5.1.1.2 Analyzing Interference between Co-Runners

In single-threaded cores, all the available L1 bandwidth is used by the same process. In contrast, in current SMT processors, those threads running concurrently on the same core compete for the available L1 bandwidth. Therefore, their performances suffer since, as shown above, the IPC of a process depends on the L1 bandwidth it uses.

This section analyzes how sharing the L1 bandwidth limits the application performance. To this end, multiple experiments running different couples of benchmarks on a single dual-threaded core were performed. Results show that whatever the pair of benchmarks launched to run concurrently is, the achieved IPC and L1 bandwidth are significantly lower for both co-runners than those obtained in stand-alone execution. These performance drops are caused, among others, by the L1 bandwidth constraints. Nevertheless, to clearly appreciate the impact of limited bandwidth on performance, the L1 bandwidth utilization of the benchmarks that run concurrently must fulfill two key characteristics. First, at least one benchmark with high L1 bandwidth requirements must be included to accentuate the impact of the contention on performance. Second, at least one of the co-runners must present a non-uniform bandwidth utilization along its execution time. Otherwise, no significant insights will be appreciated on the resultant plot.

Figure 5.4 presents the results of the described experiment for three pairs of benchmarks during the execution interval ranging from 30 to 130 seconds. For analysis purposes, each plot shows the dynamic evolution of the IPC of a given benchmark, and then differentiates between the RPC, the number of instructions that perform a L1 data cache write per cycle (WPC), and other instructions per cycle (OPC), which is calculated as the total number of instructions minus the number of instructions that perform a read or a write in the L1 cache. Each pair of benchmarks is presented by a figure on the top row of plots and the corresponding one in the bottom row. The pairs of processes that simultaneously run on the same core are *cactusADM* with *h264ref* (Figure 5.4a),



(A) *CactusADM* with *h264ref* (B) *Bzip2* with *h264ref* (C) *Bwaves* with *hmmer*

FIGURE 5.4: IPC, RPC, WPC, and OPC evolution over time when running a pair of benchmarks on the same SMT core.

bzip2 with *h264ref* (Figure 5.4b), and *bwaves* with *hmmer* (Figure 5.4c). Note that the benchmarks on the top row present non-uniform L1 bandwidth utilization in stand-alone execution (see Figure 5.3), while the ones in the bottom row show uniform L1 bandwidth utilization when running alone.

Several observations can be appreciated in this figure that can help design process allocation policies. First, when a pair of processes runs concurrently on the same core, its IPC and L1 bandwidth consumption significantly drop with respect to that achieved in stand-alone execution. Although such a drop was expected, it is interesting to notice that in some cases this drop is above 40% (e.g., *bwaves* or *cactusADM*, see Figure 5.3i and Figure 5.3j, respectively), which shows the importance of the L1 contention point. The second observation is that the IPC and RPC of each process are strongly related with that of its co-runner. In particular, when an applications experiences a drop in the IPC, a positive side effect occurs in the co-runner, which turns into an increase in its number of retired instructions.

A deeper look into the plots reveals more precisely how the co-runners affect each other. For instance, lets focus on the couple *cactusADM* and *h264ref*. The most interesting effect is the one caused by *cactusADM* on the behavior of *h264ref*. The decreasing trend in the IPC of *cactusADM*, in isolated execution, causes a synchronized increasing trend in the IPC of *h264ref* when they run concurrently on the same core. Note that in isolation, *h264ref* shows a uniform IPC. However, the key aspect lies in the RPC,

that is, the L1 bandwidth consumption. As the number of committed instructions in *cactusADM* is reduced, so does its RPC, which causes a reduction in the L1 bandwidth consumed by the process. In this way, there is more L1 bandwidth available to *h264ref*, which turns into an increase in its RPC. The IPC improvement is not exclusively caused by the increase in RPC since WPC and OPC also grow. Nonetheless, experimental results show that RPC is usually the component with highest weight on the overall IPC and presents the most similar shape to the IPC curve among the different studied components.

A similar behavior is observed with the other two pairs of benchmarks. The IPC of *h264ref* when running with *bzip2* grows synchronized with the IPC drop of *bzip2*. Although all the IPC components (RPC, WPC, and OPC) rise, RPC increase is that presenting the greatest magnitude. Similarly, in the last pair of benchmarks, *bwaves* and *hmmer*, the drops of the IPC, and particularly RPC, of *bwaves* leaves more L1 bandwidth available to *hmmer*, which takes advantage of this bandwidth to improve its IPC.

In summary, although multiple microprocessor components are shared in an SMT processor, L1 bandwidth contention can strongly drop the performance of the processes and become the major performance bottleneck. To reduce such a bottleneck, this chapter focuses on L1 bandwidth-aware process allocation policies.

5.1.2 Impact of Cache Space Contention on L1 Bandwidth Consumption

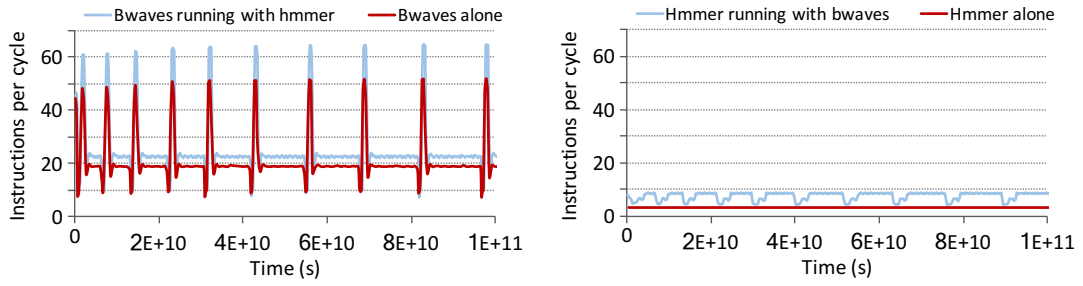
The impact of memory resource consumption (bandwidth and space) on shared caches has been addressed in previous work [39, 40], with the aim of estimating the performance of applications when the memory resources are being shared between different processes and thus, their availability is reduced with respect to stand-alone execution. Previous approaches rely on microbenchmarks, which are synthetic benchmarks that are run concurrently with the target application, but on distinct cores. This way makes performance interferences only to appear on the studied shared resource. Unfortunately, these approaches are not suitable to study space contention on L1 caches in SMT processors, since the microbenchmark and the application should be run on the same core

in order to share the same L1 cache; consequently, performance interferences other than L1 cache space rises.

Unlike previous work, this section tries to provide insights about how L1 cache space contention affects the cache performance of a given benchmark, which turns into a reduction of the L1 bandwidth consumption, without microbenchmarks. For this purpose, we analyze how the L1 misses per kilo instruction (L1 MPKI) of two processes running simultaneously on the same core increases over isolated execution. We use this metric because it is only affected by cache space. That is, neither pipeline resources contention nor cache bandwidth consumption significantly affect the L1 MPKI of a given process. As example, Figure 5.5 depicts the L1 MPKI corresponding to the co-runners of Figure 5.4c, both when they run simultaneously on the same core and in stand-alone execution. Notice that X-axis represents the number of committed instructions instead of time to match, in the figure, the stand-alone execution of each process with its concurrent execution.

Results show that the L1 MPKI of both processes rises when they run simultaneously due to space contention. As a result of the increase in the L1 MPKI, the out-of-order execution engine cannot hide most of the L1 miss penalty (i.e., latency of extra L2 cache accesses). This fact, jointly with SMT pipeline contention, slows down the execution time. Therefore, IPC and RPC, that is, L1 bandwidth consumption, decrease.

This conclusion can be confirmed by the fact that L1 MPKI rises and drops in Figure 5.5 are synchronized with reductions and increases, respectively, of the L1 bandwidth consumption in Figure 5.4c. In summary, bandwidth variation takes into account both



(A) *Bwaves* (B) *Hmmer*

FIGURE 5.5: L1 MPKI evolution over time when running a pair of benchmarks on the same SMT core.

L1 bandwidth and cache space contention; therefore, bandwidth utilization can serve as a good indicator of performance degradation due to L1 cache contention.

5.1.3 Performance Degradation due to Main Memory Bandwidth Contention

The goal of this section is not to perform an in-depth study of the performance degradation caused by main memory bandwidth contention, but to provide an overall overview of how this contention affects the performance of the processes in the current experimental platform. The performed analysis will motivate the use of a main memory bandwidth-aware process selection policy. A deeper study of the effects of bandwidth contention through the memory hierarchy on performance has been presented in Chapter 4.

To check the performance degradation caused by main memory bandwidth contention we configure the microbenchmark presented in Section 4.1.2 to achieve a main memory transaction rate (TR_{MM}) of 55 transactions/microsecond in stand-alone execution. This microbenchmark is designed to minimize cache space contention, distributing the occupied space among the cache sets, so that the measured performance degradation is caused by bandwidth contention. The performance degradation that each benchmark suffers has been analyzed when it runs concurrently with one and five instances of the microbenchmark, respectively. The former experiment evaluates a situation with only one microbenchmark, which emulates one memory-bounded application running on a different core. The latter experiment evaluates the scenario with highest main memory bandwidth contention. In this case, the system executes six processes (one on each core): the studied benchmark and five instances of the microbenchmark. Because of the high TR_{MM} of the designed microbenchmark, we guarantee that these five instances are enough to entirely consume the available main memory bandwidth.

Figure 5.6 shows the performance degradation of the benchmarks in the devised experiment. When running with only one memory-bounded instance of the microbenchmark, the highest performance degradation observed is around 45% in *xalancbmk*, but it is smaller than 10% in half of the benchmarks. However, when running with five instances, performance degradation increases dramatically to the extent that half of the

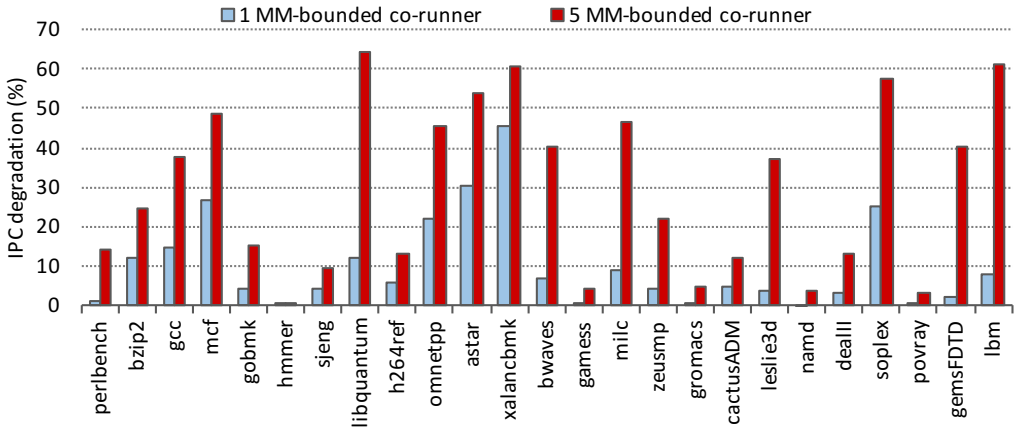


FIGURE 5.6: IPC degradation due to main memory bandwidth contention.

benchmarks suffer a degradation above 30% and five of them exceed 50%. Such degradations show the convenience of using a process selection based on the main memory bandwidth requirements of the processes.

5.2 SMT Bandwidth-Aware Scheduling

With multiprogram workloads and different levels of resource sharing, task scheduling is usually carried out in two phases. In the first phase, called process selection, the set of processes to be executed in the next quantum is selected. In the second phase, called process allocation, each selected process is mapped to a hardware context of the processor. In an SMT multicore, all the processes selected to be run in the next quantum will share main memory bandwidth but, as mentioned above, only the subset of processes assigned to a given core will share its L1 bandwidth. Thus, each scheduling phase is responsible for a resource sharing level.

Algorithm 4 SMT bandwidth-aware scheduler (BaS)

1: Update the bandwidth requirements for the next quantum of each process p executed in the previous quantum:
   - Gather consumed L1 bandwidth (TR^p_{L1})
   - Gather consumed main memory bandwidth (TR^p_{MM})
2: Process selection - Aware of main memory bandwidth requirements
3: Process allocation - Aware of L1 bandwidth requirements

Algorithm 4 presents the main steps of the proposed SMT bandwidth-aware scheduler (BaS). In the first step, performance counters are accessed to collect, for each individual process that was run during the last quantum, its number of L1 and main memory accesses, as well as its number of executed cycles. To this end, events *perf_count_hw_cache_l1d.access*, *off-core_response_0.any_data.local_DRAM*, and *unhalted_core_cycles* are measured. The collected values are used to calculate the transaction rates per microsecond on main memory (TR_{MM}) and L1 cache (TR_{L1}) performed by each process.

The bandwidth utilization of a given process during the last quantum is used as the predicted bandwidth utilization for its next execution quantum. Such a simple prediction has shown adequate accuracy. For example, the L1 bandwidth utilization during a given quantum differs on average, for all the SPEC CPU2006 benchmarks, about 5.5% from the utilization in the previous one.

Finally, the process selection and allocation steps, which are aware of the main memory and L1 bandwidth requirements of the processes, respectively, guide the scheduling decisions based on the predicted bandwidth utilization of the processes at their corresponding level of the memory hierarchy. In the proposed scheduler, the process selection phase follows the Self-reliant main memory bandwidth-aware process selection policy discussed in Section 5.2.1, while the Dynamic L1 bandwidth-aware process allocation policy presented in Section 5.2.2 is used to allocate the processes to the cores.

5.2.1 Self-Reliant Main Memory Bandwidth-Aware Process Selection

As discussed in Chapter 4, when running multiprogram workloads with significant memory requirements, main memory bandwidth contention causes an important performance degradation. Section 5.1.3 explored main memory bandwidth contention in the SMT multicore processor used as experimental platform in this chapter. The performed experiments showed that such a degradation can even exceed 50% the IPC of the processes, which illustrates the magnitude of this contention point. Therefore, it is interesting to design a process selection policy that is aware of the main memory bandwidth requirements of the processes to mitigate these performance drops.

The main goal of the devised process selection policy is to evenly distribute the amount of main memory accesses that all the processes of the workload perform throughout its

complete execution. By balancing the memory transactions along the execution time, the policy tries to minimize the contention in the main memory access, and prevents most of the memory transactions to be performed in a subset of the quanta suffering high contention, while the memory is much less stressed in other quanta. The proposed policy shares the key idea of distributing the memory accesses along the execution time with the scheduler proposed by Xu et al. [11]. The same idea was also followed in the Memory-hierarchy bandwidth-aware scheduler presented in Section 4.2.2. Nevertheless, while both proposals require prior knowledge of the main memory bandwidth requirements of the processes before running them, the policy we devise in this chapter does not require any prior information.

To balance main memory transactions across execution time, the proposed policy makes use of the Online Average Transaction Rate (OATR). The OATR is calculated as the average main memory bandwidth utilization (BW_{MM}) of the processes of the workload, multiplied by the number of hardware contexts (#CPUs) of the experimental platform (see line 2 of Algorithm 5). The OATR defines the overall main memory bandwidth that should be used at the next quantum in order to evenly distribute the transactions along the execution time. Hence, it is used as the target main memory bandwidth, such as the IABW proposed by Xu et al. [11] is used in Section 4.2.1.

The main difference of the OATR with the IABW is that while the latter is fixed before mix execution (it is calculated offline with prior information about the main memory requirements and execution time of the processes), the OATR changes dynamically during the workload execution based on the changes in the average main memory bandwidth utilization of the processes. As the execution progresses, the OATR reaches a value that is more realistic than the IABW since it is calculated using the bandwidth utilization gathered while the processes run concurrently. In contrast, the IABW is calculated from the bandwidth utilization measured in stand-alone execution. Note that Xu et al. actually correct this issue using a polynomial regression method, which is not required by the OATR.

Algorithm 5 presents the pseudocode of the Self-reliant main memory bandwidth-aware process selection policy proposed. The scheduling steps closely resemble the ones performed by Algorithm 2, with the main difference of using the OATR instead of the

Algorithm 5 Self-reliant main memory bandwidth-aware process selection policy

1: Update the BW_{MM}^{p} for each process p of the workload
2: Calculate the OATR: $OATR = \frac{\sum_{p=0}^{N} BW_{MM}^{p}}{N} * \#\text{CPUs}$
3: $\text{BW}_{Remain} = \text{OATR}$
4: Select the process P_head at the queue head
5: $\text{BW}_{Remain} -= TR_{MM}^{P\_head}$, $\text{CPU}_{Remain} = \#CPUs - 1$
6: **while** $\text{CPU}_{Remain} > 0$ **do**
7: select the process P that maximizes
8: $\text{FITNESS(p)} = \frac{1}{\left| \frac{BW_{Remain}}{CPU_{Remain}} - TR_{MM}^{P} \right|}$
9: $\text{BW}_{Remain} -= TR_{MM}^{P}$, $\text{CPU}_{Remain}--$
10: **end while**

IABW as the target main memory bandwidth utilization. The first step of the algorithm updates the average main memory bandwidth utilization of the processes that have run during the last quantum. Next, in line 2 the OATR for the next quantum is calculated using the BW_{MM} of all the processes of the workload and then, BW_{Remain} is set to the OATR (line 3).

The next scheduling steps match the ones performed by Algorithm 2 form line 10 to 16. In short, the algorithm selects as many processes as hardware contexts are available in the system. Processes are selected using the fitness function which quantifies, for each process, the gap between its predicted main memory transaction rate for the next quantum (TR_{MM}^{p}) and the average bandwidth remaining for each unallocated hardware contexts ($\text{BW}_{Remain}/\text{CPU}_{Remain}$). The process with the best fit is the one that maximizes the fitness function. After selecting a process, BW_{Remain} and CPU_{Remain} are update accordingly. This loop is repeated until no more hardware contexts are available. Please, refer to Section 4.2.2 for further details of the scheduling algorithm.

Due to the lack of previous information, the scheduler has to face a cold start the first quanta of the execution of a new workload, since it has no prior information about the processes. Besides the average main memory transaction rate of the processes can take a few quanta to reach a dependable value, which can increase the length of such cold start. To mitigate a possible negative impact on performance, we propose to let Linux drive the scheduling decisions during a few quanta at the beginning of the execution, while the proposed scheduler collects enough bandwidth utilization information of the processes. We found experimentally that a short period of about thirty quanta (over

executions that last more than five thousand quanta) is large enough to avoid significant performance losses.

5.2.2 L1 Bandwidth-Aware Process Allocation

5.2.2.1 Dynamic L1 Bandwidth-Aware Process Allocation Policy

The analysis presented in Section 5.1.1 illustrates that the high L1 bandwidth utilization of the processes can cause important bandwidth contention and performance degradation when two applications run simultaneously on the same core. Thereby, the allocation of the processes to the cores strongly impacts on the throughput the system can achieve. In addition, the L1 bandwidth requirements of the processes can widely vary over their execution. Thus, the thread to core allocation should be dynamically adapted to the changes fn the L1 bandwidth utilization to achieve the highest performance. To address these issues, this section proposes a dynamic process allocation policy that is aware of the L1 bandwidth requirements of the processes. Note that, despite guiding the allocation of processes to cores based on L1 bandwidth, the proposed policy addresses overall SMT contention.

The key idea of the process allocation policy consists in balancing the overall L1 bandwidth utilization of the running processes among all the processor cores. Hence, the policy tries to promote thread to core mappings that do not saturate the available L1 bandwidth of some cores while this bandwidth is underused in others. Notice that the process allocation policy assumes that the number of processes to be allocated matches the number of hardware contexts.

Algorithm 6 presents the pseudocode of the proposed Dynamic L1 bandwidth-aware process allocation policy. Since the experimental platform supports simultaneous execution of only two threads in each core, finding the thread to core assignment that achieves the optimal balance of L1 bandwidth consumption among cores is simplified. For instance, processes can be ordered according to their TR_{L1}[2] (line 1). Notice that the maximum number of processes that must be sorted each time the policy is executed is equal to

[2] The RPC used to study the effects of L1 bandwidth contention on performance could be used in this algorithm since it is basically the same metric expressed in different units. However, the algorithm uses TR_{L1} for consistency reasons.

Algorithm 6 Dynamic L1 bandwidth-aware process allocation policy

1: Sort the selected processes in ascending TR_{L1}
2: **while** there are unallocated processes **do**
3: Select the processes P_{head} and P_{tail} with maximum and minimum TR_{L1}
4: Allocate P_{head} and P_{tail} to the same core
5: **end while**

the number of hardware contexts, which limits the computational cost of sorting the processes. Such overhead has been measured experimentally and is negligible compared with the quantum length and the benefits provided by a good thread to core assignment. The threads with highest and lowest L1 bandwidth requirements are assigned to the same core (lines 3 and 4). This rule is iteratively applied to obtain the remaining pairs of co-runners.

If the SMT processor supports the execution of three or more threads, it is possible to balance the L1 bandwidth requirements following a similar approach to that explained in Algorithm 2 (Section 4.2.2) to distribute the selected processes among the shared caches of a given cache level. Thereby, the algorithm would calculate the cumulative TR_{L1} of all the processes that have been selected to run in the next quantum and would divide this value by the number of cores. Then, the processes could be properly allocated to the cores in order to balance TR_{L1} differences among L1 caches using the fitness function [13, 14], which is also described in Section 4.2.2.

Finally, to remark that the number of process migrations among cores is not limited by the proposed policy. In spite of an overhead is incurred when migrating the architectural state of the process and extra time is wasted warming up the L1 cache, we found that such overhead is negligible when working with long quanta like the ones used by modern operating systems [62].

5.2.2.2 Static L1 Bandwidth-Aware Process Allocation Policy

A static version of the process allocation policy can also be implemented. This version, referred to as Static L1 bandwidth-aware process allocation policy, follows the same algorithm as the dynamic policy. However, it uses the average L1 bandwidth utilization of the processes when running alone, instead of the dynamically updated TR_{L1} used in the dynamic policy. Therefore, it does not need to read performance counters at the

end of the quanta to update the TR_{L1} of the processes, but the average TR_{L1} of the processes in stand-alone execution must be provided as an input parameter.

A potential advantage that the static policy presents, is the fact that it uses the average TR_{L1} of the processes, measured in a profiling phase where the processes run alone in the system, thus avoiding interference from other co-runners. When the processes present a uniform L1 bandwidth shape, the average L1 bandwidth utilization obtained without interference can be a better estimate of the requirements of the processes than the TR_{L1} measured with co-runners interference.

Nevertheless, the dynamic policy presents two strong advantages with respect to the static policy. First, it should provide better L1 bandwidth balancing since it is able to react to non-uniform demands of L1 bandwidth. For instance, L1 bandwidth requirements of benchmarks like *astar* or *mcf* can be properly addressed. That is, the dynamic policy can allocate to the same core *astar*, when it presents low L1 bandwidth requirements, together with a process with high L1 bandwidth consumption. Later, when *astar* increases its bandwidth utilization, the policy can change its co-runner to a process with lower bandwidth requirements. Second, the dynamic policy is more practical than the static one since it does not require prior information of the processes.

5.3 Evaluation Setup

The experimental evaluation has been carried out in an Intel Xeon E5645 system (see Section 3.2.2). The proposed algorithms are implemented in the scheduling framework (see Section 3.1) to evaluate their effectiveness. Notice that this framework allows us to evaluate either the policies in an isolated way or combined. When evaluating the policies in isolation, the scheduling policy (process selection or process allocation) that is not being analyzed is set to the Linux policy.

We follow the process allocation evaluation methodology to study the process allocation policies and the process selection evaluation methodology to evaluate the process selection policies and the BaS scheduler. Please, refer to Section 3.3 for further details on the evaluation methodologies. The target number of instructions of the SPEC CPU2006

benchmarks is set to the number of instructions they complete running alone during 200 seconds, and the quantum length is set to 200 milliseconds.

A wide set of metrics has been analyzed for evaluation purposes. First, we use the average IPC of the threads as a pure performance metric, and the harmonic mean of the per-program IPC speedup to give a notion of fairness to the analysis. In addition to these IPC-related metrics, the turnaround time of the mixes has also been evaluated. See Section 3.4 for further details on these metrics.

5.3.1 Evaluated Algorithms

The experimental evaluation studies, first, the process allocation and process selection policies in isolation. Next, the Dynamic L1 bandwidth-aware process allocation and the Self-reliant main memory bandwidth-aware process selection policies are combined to build the proposed SMT bandwidth-aware scheduler. The multiple policies considered in the performed experiments are listed below.

Process selection policies:

- **Random**: a random process selection algorithm that selects a random subset of processes to be run each quantum.
- **Linux**: the policy used by the default Linux Completely Fair Scheduler (CFS). In short, the CFS scheduler tries to give all the processes the same CPU utilization.
- **Main memory bandwidth-aware (Memory_BW)**: the main memory bandwidth-aware scheduler proposed by Xu et al. [11], which has been discussed in Section 4.2.1.
- **Self-reliant main memory bandwidth-aware (Self-reliant_BW)**: the proposed process selection algorithm, described in Section 5.2.1. Unlike the Memory_BW, which needs to know the memory requests that each process is going to perform, the Self-reliant_BW policy does not require any preliminary information of the processes.

Process allocation policies:

- **Random**: a policy that randomly assigns the processes to the cores.
- **Linux**: the process allocation performed by the default Linux Completely Fair Scheduler (CFS). One of the actions that the CFS takes to maximize performance consists in avoiding constant process migrations keeping the affinity of the processes to the core where they are running.
- **Dynamic**: the proposed Dynamic L1 bandwidth-aware process allocation policy, presented in Section 5.2.2.1.
- **Static**: the proposed Static L1 bandwidth-aware process allocation policy, presented in Section 5.2.2.2.

5.3.2 Mix Design

To evaluate the process allocation policies we need a set of mixes where the number of processes matches the number of hardware contexts of the experimental platform. To design an interesting set of workloads, we classify the benchmarks in four groups according to their average L1 bandwidth requirements in stand-alone execution. Table 5.1 presents this classification. Benchmarks with higher L1 bandwidth utilization can potentially induce higher degradation in the co-runner and, at the same time, they can suffer a strong degradation due to L1 bandwidth constraints. Thus, it is critical to

Classification	Benchmarks
Extreme L1 bandwidth	h264ref, bwaves, gamess
High L1 bandwidth	perlbench, bzip2, hmmer, libquantum, leslie3d, namd, dealII, gemsFDTD
Medium L1 bandwidth	gcc, gobmk, sjeng, astar, xalancbmk, zeusMP, povray, lbm
Low L1 bandwidth	mcf, omnetpp, milc, gromacs, cactusADM, soplex

TABLE 5.1: Benchmark classification according to their L1 bandwidth requirements.

allocate them sharing the core with the appropriate co-runners to enhance performance. Otherwise, significant performance losses can appear.

Based on the benchmark classification, mixes are distinguished by the number of benchmarks with *extreme* L1 bandwidth requirements they include. The *balanced* mixes are formed with half the benchmarks belonging to the extreme L1 bandwidth category. These workloads can potentially offer higher benefits with a good process allocation

Mixes	Benchmarks
Mix 1	*gamess, h26ref, milc, omnetpp*
Mix 2	*bwaves, cactusADM, h26ref, soplex*
Mix 3	*bwaves, gamess, mcf, milc*
Mix 4	*bwaves, gamess, namd, soplex*
Mix 5	*bzip2, h26ref, lbm, xalancbmk*
Mix 6	*gemsFTDTD, gromacs, mcf, perlbench*
Mix 7	*bwaves, gamess, h26ref, mcf, milc, omnetpp*
Mix 8	*bwaves, cactusADM, gamess, h26ref, milc, soplex*
Mix 9	*bwaves x2, gromacs, h26ref, omnetpp, soplex*
Mix 10	*bzip2, gamess, gromacs, h26ref, mcf, soplex*
Mix 11	*astar, dealII, gamess, leslie3d, mcf, sjeng*
Mix 12	*gobmk, gromacs, libquantum, perlbench, xalancbmk, zeusMP*
Mix 13	*bwaves, gamess, gromacs, h26ref x2, mcf, milc, omnetpp*
Mix 14	*bwaves, gcc, gamess x2, h26ref, mcf, omnetpp, xalancbmk*
Mix 15	*bwaves x2, cactusADM, gamess, gromacs, h26ref, mcf, soplex*
Mix 16	*bwaves, bzip2, gamess, gromacs, h26ref, mcf, omnetpp, sjeng*
Mix 17	*bwaves, gobmk, h26ref, libquantum, mcf, omnetpp, perlbench, sjeng*
Mix 18	*astar, bzip2, gobmk, h26ref, namd, omnetpp, perlbench, sjeng*
Mix 19	*bwaves x2, gamess x2, gobmk, gromacs, h26ref x2, lbm, mcf, omnetpp, sjeng*
Mix 20	*astar, bwaves x2, cactusADM, gamess x2, h26ref x2, mcf, omnetpp, soplex, xalancbmk*
Mix 21	*bwaves x2, bzip2, gamess x2, gobmk, gromacs, h26ref x2, mcf, sjeng, zeusMP*
Mix 22	*dealII, gamess x2, gobmk, gromacs, h26ref x2, lbm, libquantum, omnetpp, soplex x2*
Mix 23	*bwaves, bzip2, gamess, gobmk, hmmer, h26ref, mcf, omnetpp, perlbench, sjeng, soplex, zeusMP*
Mix 24	*bzip2, cactusADM, gamess, gromacs, hmmer, h26ref, leslie3d, mcf, namd, omnetpp, sjeng, soplex*

TABLE 5.2: Mix composition designed to evaluate the process allocation policies.

since each benchmark with extreme L1 bandwidth demand can be allocated to a different core to run with a benchmark with lower L1 bandwidth requirements. Non-balanced mixes are formed with less extreme benchmarks than the number of cores. Since more applications can present intermediate bandwidth requirements, lower differences between allocations policies are expected.

We designed a wide variety of mixes consisting of up to twelve applications. In order to force that all the cores run two processes simultaneously, each mix is run on half the

Mixes	Benchmarks
Mix 1	*3 x Bwaves, 2 x CactusADM, DealII, 3 x Gamess, 2 x GemsFDTD, Hmmer, 2 x H264ref, 3 x Leslie3d, Lbm, 2 x Libquantum, Mcf, Milc, Povray, ZeusMP*
Mix 2	*3 x Bwaves, 3 x Gamess, GemsFDTD, Gromacs, Hmmer, 3 x H264ref, 2 x Leslie3d, 2 x Lbm, 2 x Libquantum, Mcf, Milc, Omnetpp, Perlbench, Xalancbmk, Povray*
Mix 3	*Bwaves, Bzip2, CactusADM, DealII, Gamess, Gcc, GemsFDTD, Gobmk, Gromacs, Hmmer, H264ref, Lbm, Libquantum, Leslie3d, Mcf, Milc, Namd, Perlbench, Povray, Omnetpp, Sjeng, Soplex, Xalancbmk, ZeusMP*
Mix 4	*Astar, 3 x Bwaves, 2 x CactusADM, 2 x Gamess, GemsFDTD, Gobmk, Gromacs, 2 x H264ref, Lbm, Leslie3d, 2 x Libquantum, Mcf, 2 x Milc, Omnetpp, Povray, Sjeng, ZeusMP*
Mix 5	*Bwaves, CactusADM, DealII, Gamess, Gcc, 3 x GemsFDTD, Gromacs, H264ref, 3 x Mcf, 2 x Milc, Namd, 3 x Lbm, Libquantum, 3 x Leslie3d, ZeusMP*
Mix 6	*2 x Astar, 2 x Bzip2, 2 x Gcc, 2 x Gobmk, 2 x Hmmer, 2 x H264ref, 2 x Libquantum, 2 x Mcf, 2 x Omnetpp, 2 x Perlbench, 2 x Sjeng, 2 x Xalancbmk*
Mix 7	*2 x Bwaves, 2 x CactusADM, 2 x DealII, 2 x Gamess, 2 x Gromacs, 2 x Lbm, 2 x Leslie3d, 2 x Milc, 2 x Namd, 2 x Povray, 2 x Soplex, 2 x ZeusMP*
Mix 8	*Astar, 2 x Bwaves, CactusADM, DealII, 2 x Gamess, Gcc, GemsFDTD, Gobmk, Gromacs, Hmmer, H264ref, Lbm, Libquantum, 2 x Mcf, Milc, 2 x Namd, Omnetpp, Perlbench, Povray, Soplex*
Mix 9	*3 x Bwaves, CactusADM, 3 x Gamess, Gcc, GemsFDTD, Gromacs, Hmmer, 3 x H264ref, Lbm, Libquantum, Mcf, 2 x Milc, Namd, Omnetpp, Sjeng, Soplex, ZeusMP*
Mix 10	*3 x Bwaves, Bzip2, DealII, Gamess, GemsFDTD, Gromacs, Hmmer, H264ref, 2 x Lbm, 2 x Leslie3d, 2 x Libquantum, Mcf, Milc, Namd, Omnetpp, 2 x Perlbench, Povray, Xalancbmk*
Mix 11	*3 x Bwaves, Bzip2, CactusADM, DealII, 2 x Gamess, GemsFDTD, Gobmk, Gromacs, Hmmer, 2 x H264ref, 2 x Leslie3d, Libquantum, Mcf, Milc, Namd, Omnetpp, Perlbench, Povray, Sjeng*
Mix 12	*3 x Bwaves, Bzip2, CactusADM, 3 x H264ref, 3 x Gamess, Gcc, GemsFDTD, Gobmk, Gromacs, 2 x Lbm, 2 x Leslie3d, 2 x Libquantum, Mcf, Milc, Namd,*

TABLE 5.3: Mix composition designed to evaluate the process selection policies and the entire schedulers.

number of cores that applications the mix contains. Table 5.2 presents the composition of the mixes used to evaluate the process allocation policies.

In the experiments where the process selection policies are considered (either isolated or as a part of an entire scheduler), we require a set of workloads whose the number of processes exceeds the number of hardware contexts. To this end, we design a set of twelve mixes, where each mix consists of twenty four benchmarks, that is, the number of processes doubles the available hardware contexts. The large number of processes of each workload, and the fact that we are attacking bandwidth contention at both L1 and main memory makes it difficult to classify the workloads in homogeneous groups. Therefore, the mixes have been built including a subset of benchmarks with high L1 bandwidth requirements, a subset of benchmarks with high main memory bandwidth requirements, and a bigger subset with benchmarks showing intermediate requirements of both bandwidths. The composition of each mix is presented in Table 5.3.

5.4 Experimental Evaluation

First, we analyze the performance benefits provided by both proposed Dynamic L1 bandwidth-aware process allocation policy and Self-reliant main memory bandwidth-aware process selection policy in isolation. Then, we study the performance of both policies together, that is the SMT bandwidth-aware scheduler, with respect to Linux. The plotted results in all the experiments correspond to the average values of twenty executions and 95% confidence intervals.

5.4.1 Evaluation of the Process Allocation Policies

The performance of the proposed dynamic and static L1 bandwidth-aware process allocation policies is evaluated and compared to Linux. A wide set of mixes has been evaluated for a different number of cores, ranging from two cores (four applications) to six cores (twelve applications). For each number of cores, we used both balanced and non-balanced mixes with different L1 bandwidth demands.

Figure 5.7 presents the speedup of the average IPC achieved by the proposed policies and Linux for each mix over the random process allocation policy, which has been used

as baseline. With XE we refer to a non-balanced mix with X *extreme* benchmarks; e.g., 1E means only one *extreme* benchmark.

Compared to Linux, the proposed policies achieve better performance across all the twenty-four evaluated mixes. While the dynamic and static process allocation policies provide speedups higher than 5% in seventeen and fifteen mixes, respectively, Linux only surpasses this value in four mixes. On the contrary, the speedup of Linux falls around or below 2% in six mixes, while this only occurs in one mix with the dynamic and static policies.

As observed, the dynamic policy performs better, on average, than the static one. Significant differences can be appreciated in some mixes like 2, 3, 8, 12, 16, and 24. The major differences appear when the mix includes benchmarks showing a non-uniform shape in their L1 bandwidth requirements. For instance, mix 2 includes *bwaves* and *cactusADM*, which present a non-uniform shape. On the contrary, mix 1 shows minor differences since all benchmarks present an almost uniform shape in their L1 bandwidth consumption. The only exception in which the static policy provides significant benefits over the dynamic one is in mix 6. The reason is that this mix includes *gemsFDTD*, whose L1 bandwidth utilization varies so fast (see Figure 5.3l) that the dynamic policy is not able to accurately predict the bandwidth requirement for the next quantum.

As expected, the policies offer higher performance when running balanced workloads. As the number of *extreme* threads drops in a mix, the achieved speedup is on average smaller

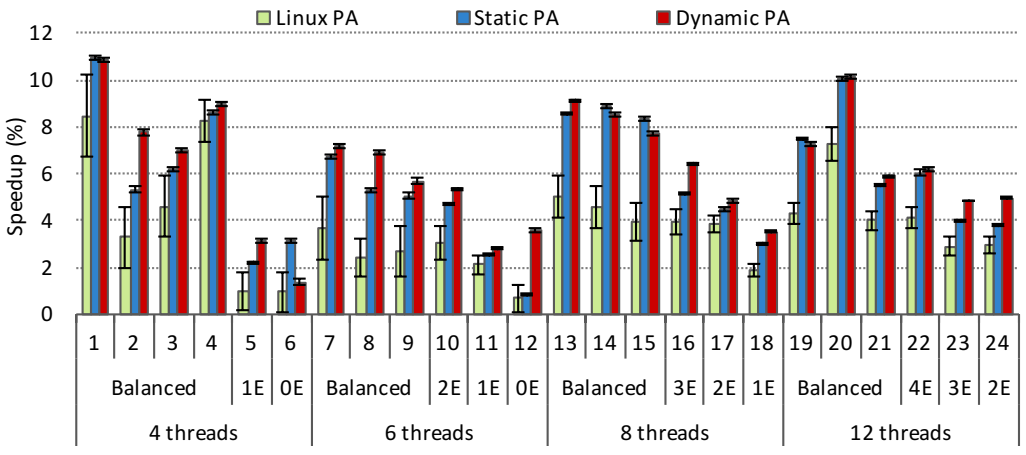


FIGURE 5.7: Speedup of the average IPC of the studied process allocation policies over the random policy.

since L1 bandwidth contention is reduced. Nonetheless, performance differences among mixes are also due to *non-extreme* benchmarks characteristics. For example, mix 20 includes one and five benchmarks with medium and low L1 bandwidth demand, respectively; while mix 21 includes one, three, and two benchmarks with high, medium, and low L1 bandwidth consumption, respectively. Since bandwidth differences among possible pairs can be higher in mix 20 than in mix 21, one should expect major performance benefits from appropriate process mappings in this mix. Thus, even in non-balanced workloads noticeable performance benefits can be achieved (e.g., 12, 16, 22, 23, and 24).

Notice too that confidence intervals of Linux are considerably larger than those of the dynamic and static policies. This is due to the fact that Linux does not consider L1 bandwidth to perform the allocation. Therefore, its thread to core mappings, and consequently their corresponding performance, greatly vary among different instances of the experiment. On the other hand, the confidence intervals for the devised policies are usually below 0.1%, ensuring that the achieved speedups are stable among executions.

Looking at Figure 5.8, which shows the speedups using the harmonic mean of the per-program IPC speedup, the same conclusions can be drawn. The speedup values are slightly reduced, however, differences between the speeupds of the dynamic and static policies are wider (e.g., mixes 3, 7, 17, and 24). Considering that this metric captures both performance and fairness, one can conclude that the Dynamic L1 bandwidth-aware process allocation policy is the best evaluated policy

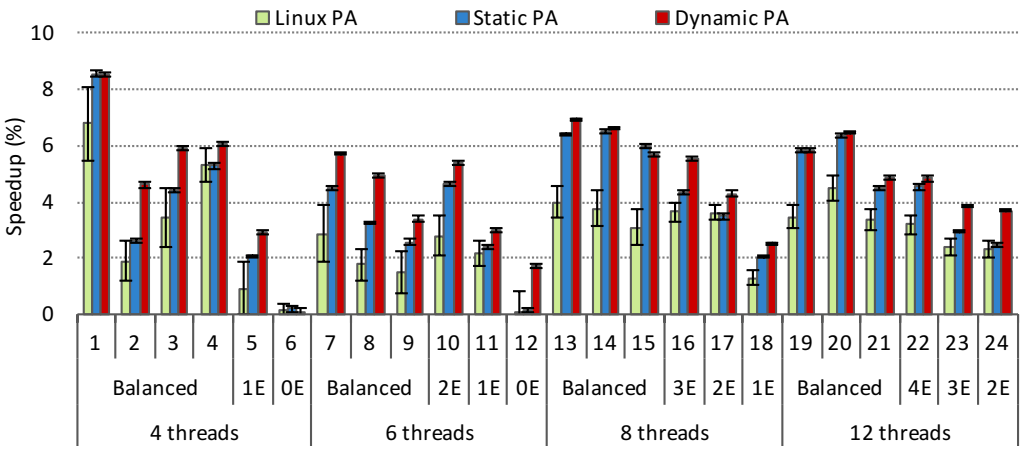
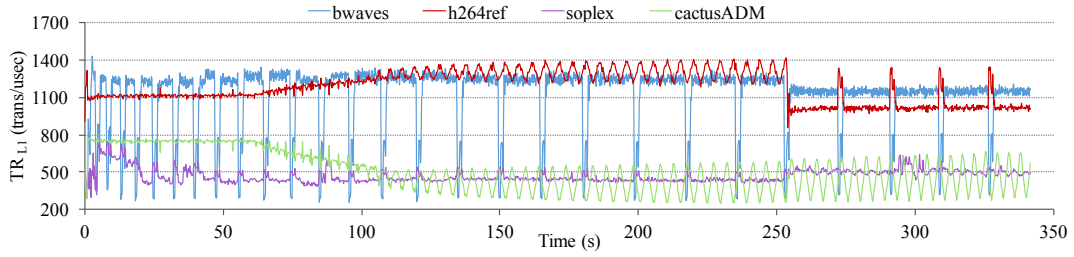


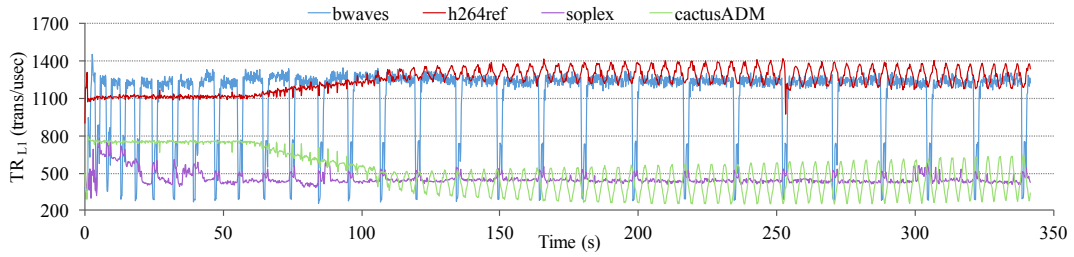
FIGURE 5.8: Speedup of the harmonic mean of the per-program IPC speedup of the studied process allocation policies over the random policy.

Average values, however, do not reflect what is happening over time. To provide insights and a sound understanding about how the different policies work with time, lets analyze the behavior of mix 2. In this mix, the static policy significantly improves the performance of Linux and, at the same time, the dynamic policy considerably improves the performance of the static one. Figure 5.9 shows the dynamic TR_{L1} of each benchmark during the complete execution of mix 2 under the studied allocation policies.

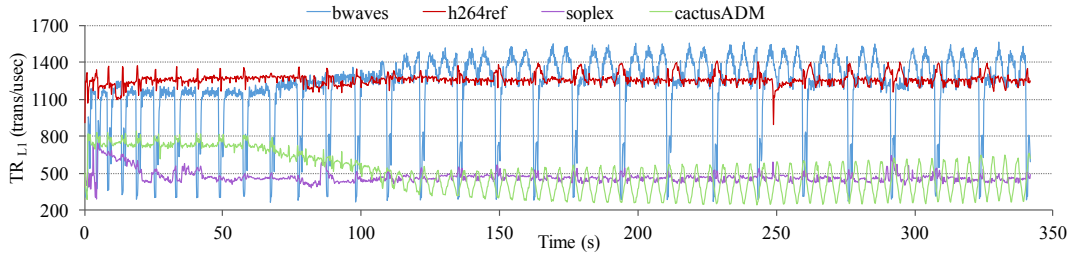
Notice that the plots of the Linux and static process allocation policies are quite similar during the first 250 seconds. According to the TR_{L1} curves, one can deduce that *h264ref* and *cactusADM* were running on one core and *bwaves* and *soplex* on the other one. Around second 250, Linux changes the process to core mapping and starts running together *h264ref* and *bwaves*. This can be deduced because the rises in the TR_{L1} curve of *h264ref* are synchronized with the drops of *bwaves*. However, notice that in spite of



(A) Linux process allocation



(B) Static L1 bandwidth-aware process allocation



(C) Dynamic L1 bandwidth-aware process allocation

FIGURE 5.9: TR_{L1} of benchmarks in mix 2 for the Linux, Static, and Dynamic process allocation policies.

this process to core mapping yields to lower performance, Linux keeps it until the end of the execution.

Unlike the previous policies, the dynamic process allocation policy usually selects as co-runners *bwaves* and *cactusADM*, which according to the observed TR_{L1} is the best choice. As observed, *bwaves* obtains regular peaks around 1500 transactions/microsecond, while the maximum TR_{L1} does not surpass 1400 transactions/microsecond in the other two process allocation policies. Finally, when *bwaves* experiences sharp drops in its TR_{L1} curve, the dynamic policy benefits *h264ref*, which at that point, is the process with higher L1 bandwidth utilization. Consequently, the L1 bandwidth consumption rises of *h264ref* that occur during drops of *bwaves* bandwidth consumption are higher than those obtained by *soplex* in the static policy, thus, increasing the overall performance.

Finally, to remark that the performance of the proposed policies scale well with the number of threads. Nevertheless, the number of accesses to main memory is expected to grow with the number of threads. Thus, it may happen that LLC and main memory contention grow and create new contention points. In such a case, the proposed allocation policies could be combined with main memory and LLC bandwidth-aware selection policies to tackle them.

5.4.2 Evaluation of the Process Selection Policies

In this section, the performance of the designed Self-reliant main memory bandwidth-aware process selection policy (Self-Reliant_BW) is compared to that achieved by both a main memory bandwidth-aware process selection policy based on Xu's scheduler [11] (Memory_BW) and the Linux policy implemented in the CFS scheduler. This comparison assumes as a baseline the performance of the random process selection policy.

Figure 5.10 and Figure 5.11 present the speedups achieved by Linux, Memory_BW, and Self-Reliant_BW relative to the random policy regarding IPC-based metrics. Results in terms of the the average IPC metric (Figure 5.10) show that Memory_BW and Self-reliant_BW improve the performance of Linux and the random policy. The speedups achieved by Memory_BW and Self-reliant_BW usually fall in between 3% and 5%, being

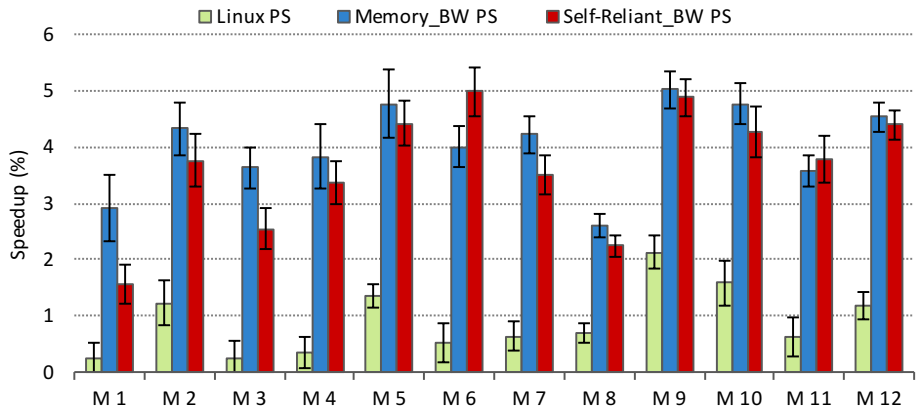


FIGURE 5.10: Speedup of the three process selection policies studied with respect to the random policy using the average IPC metric.

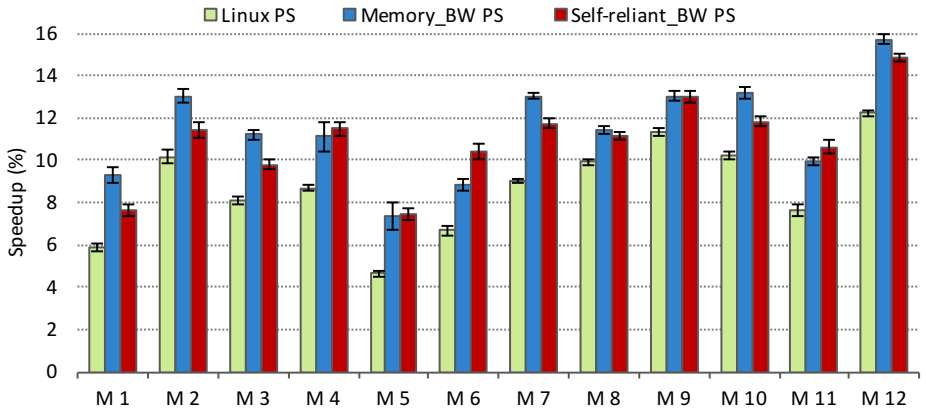


FIGURE 5.11: Speedup of the three process selection policies studied with respect to the random policy using the harmonic mean of the per-program IPC speedup.

higher for Memory_BW in all the mixes but two. With regard to Linux, it achieves much lower speedups and only mix 8 exceeds 2%.

Figure 5.11 depicts the speedups of the policies regarding the harmonic mean of the per-program IPC speedup. The benefits achieved with this metric are much higher for the three evaluated policies with respect to the random policy, which indicates that Linux, Memory_BW, and Self-reliant_BW perform a much fairer process selection. Memory_BW achieves the best results, showing the highest speedup in eight mixes and an average speedup of 11.4% across all the evaluated mixes. Close to this performance, Self-reliant_BW achieves the best speedup in four mixes, with an average speedup about 11%. Linux achieves the worst speedup with an average value of 8.7% over the random policy.

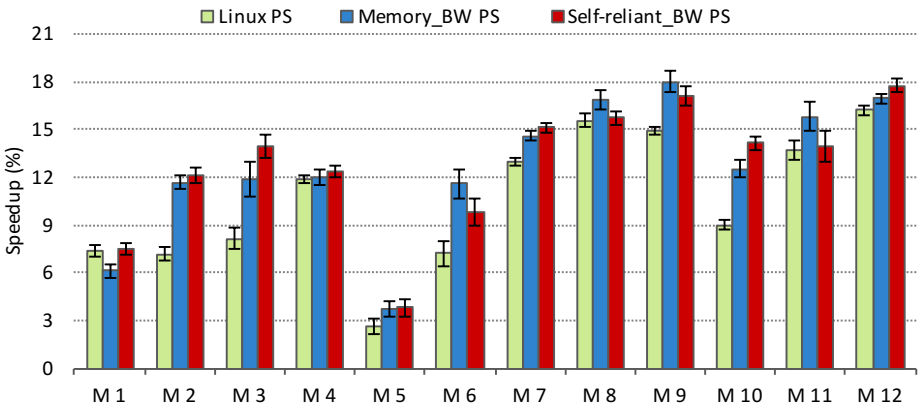


FIGURE 5.12: Speedup of the three studied process selection policies with respect to the random policy regarding turnaround time.

Finally, Figure 5.12 presents the speedups regarding the turnaround time of the mixes. Results show that all the process selection policies widely improve the performance of the random policy with speedups that usually exceed 12%. The reduction in the time required to complete the execution of the mixes shows the significance of the main memory bandwidth contention point and how smart policies can mitigate such contention and improve performance. Comparing the the evaluated policies, results suggest that Linux performs worse than Memory_BW and Self-reliant_BW, since it achieves significantly lower speedup in mixes like 2, 3, 6, or 10. Regarding Memory_BW and Self-reliant_BW, we can see that Self-reliant_BW achieves better performance than Memory_BW in eight mixes. In addition, the average speedup for the evaluated mixes is 12.6% and 12.8%, for Memory_BW and Self-reliant_BW, respectively, which shows that Self-reliant_BW performs slightly better in terms of turnaround time.

The achieved speedups regarding the turnaround time help explain the relatively low speedups observed with the average IPC metric. Notice that the random process selection policy significantly enlarges the execution time of the mixes, which causes the distribution of the overall main memory accesses in a longer interval, so reducing contention. In this way, the processes see their performance improved and the average IPC of the mix is enhanced, but it is not a desirable behavior since it is achieved at the expense of a higher turnaround time.

In summary, the three process selection policies evaluated significantly improve the performance of the random policy, with speedups that usually exceed 10% regarding the

harmonic mean of the per-program IPC speedup and turnaround time metrics. Among the policies, the best results are obtained with Memory_BW and Self-reliant_BW that perform better than Linux across all the evaluated mixes. However, notice that Self-reliant_BW is able to achieve performance comparable to (if not better than) that achieved by Memory_BW, despite the fact the latter policy uses bandwidth information obtained in prior executions of the processes to calculate the IABW. This can be explained by the fact that the bandwidth information used by Memory_BW is gathered in stand-alone execution and, despite being representative of the bandwidth requirements of the processes, it loses some accuracy when running with co-runners because it does not consider the interference that affect their bandwidth utilization.

5.4.3 Evaluation of the SMT Bandwidth-Aware Scheduler

This section analyzes the performance of the proposed SMT bandwidth-aware scheduler (BaS) with respect to Linux. Figure 5.13 and Figure 5.14 present the performance benefits reached using the IPC-based metrics. Figure 5.13 shows the speedup of the average IPC achieved in each mix and the geometric mean across all the studied mixes. BaS improves Linux in all mixes, with speedups ranging from above 3.0% to close to 7.0%, and with nine of twelve mixes achieving over 4.0% speedup and five exceeding 5.0%. Since average IPC is a performance focused metric, the results show that BaS

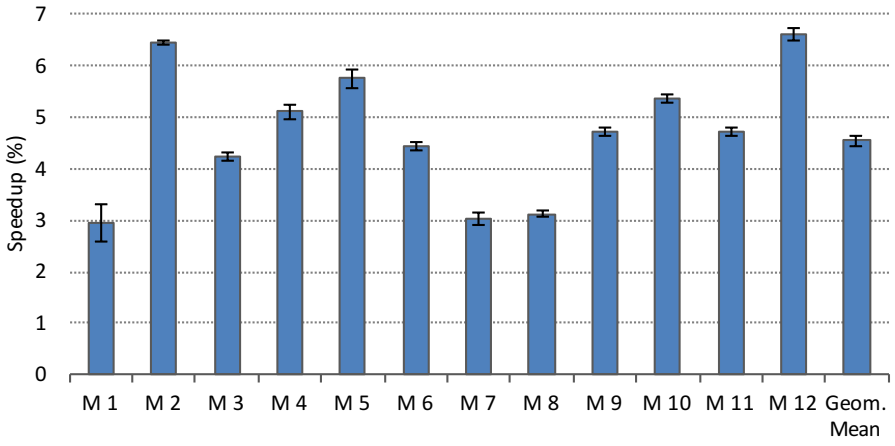


FIGURE 5.13: Speedup of the proposed BaS scheduler relative to the Linux scheduler using the average IPC metric.

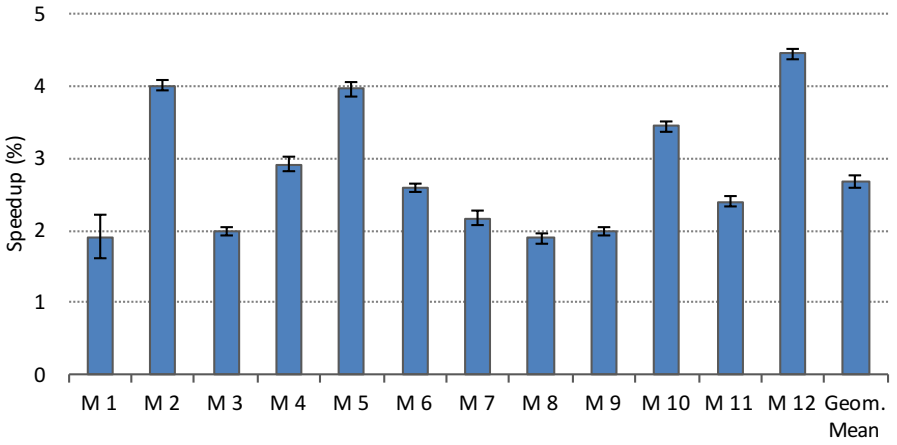


FIGURE 5.14: Speedup of the proposed BaS scheduler relative to the Linux scheduler using the harmonic mean of the per-program IPC speedup metric.

effectively addresses bandwidth contention at the L1 cache and main memory, which results in a significant performance increase.

Figure 5.14 compares the speedups with the harmonic mean of the per-program IPC speedup metric. BaS achieves speedups ranging from around 2.0% to 4.5% with respect to Linux. Although they are slightly reduced compared to those obtained with the average IPC metric, they show that, in addition to improve its performance, the proposed scheduler works fairer than Linux.

Figure 5.15 presents the speedup achieved by BaS regarding the turnaround time. The plot shows that BaS shortens the execution time of all the evaluated mixes with speedups over 2.0%, with the only exception of mix 9. Five mixes achieve a speedup between 3.0% and 4.0%. The wider confidence intervals in the turnaround time speedups are caused by the high variability of the turnaround time of the mixes in Linux. For instance, the typical deviation of the turnaround time of different executions of mix 2 in Linux triples the one obtained by BaS.

Notice that when dealing with bandwidth contention, the improvements achieved in throughput, as the average IPC speedups, do not directly correspond to reductions in the turnaround time of the mixes. In fact, when the turnaround time of the mix is shortened bandwidth contention rises, since the same number of memory or cache accesses are concentrated in a shorter period of time. This situation can cause some policies to achieve higher throughput but also longer execution time. Therefore, it is

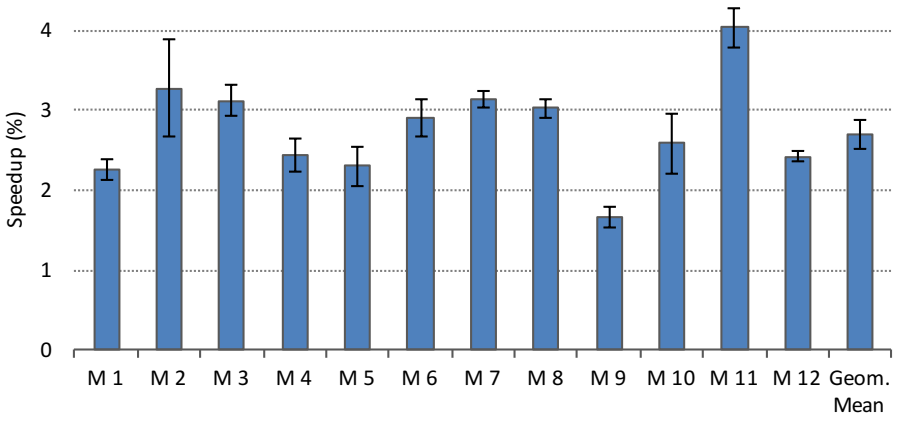


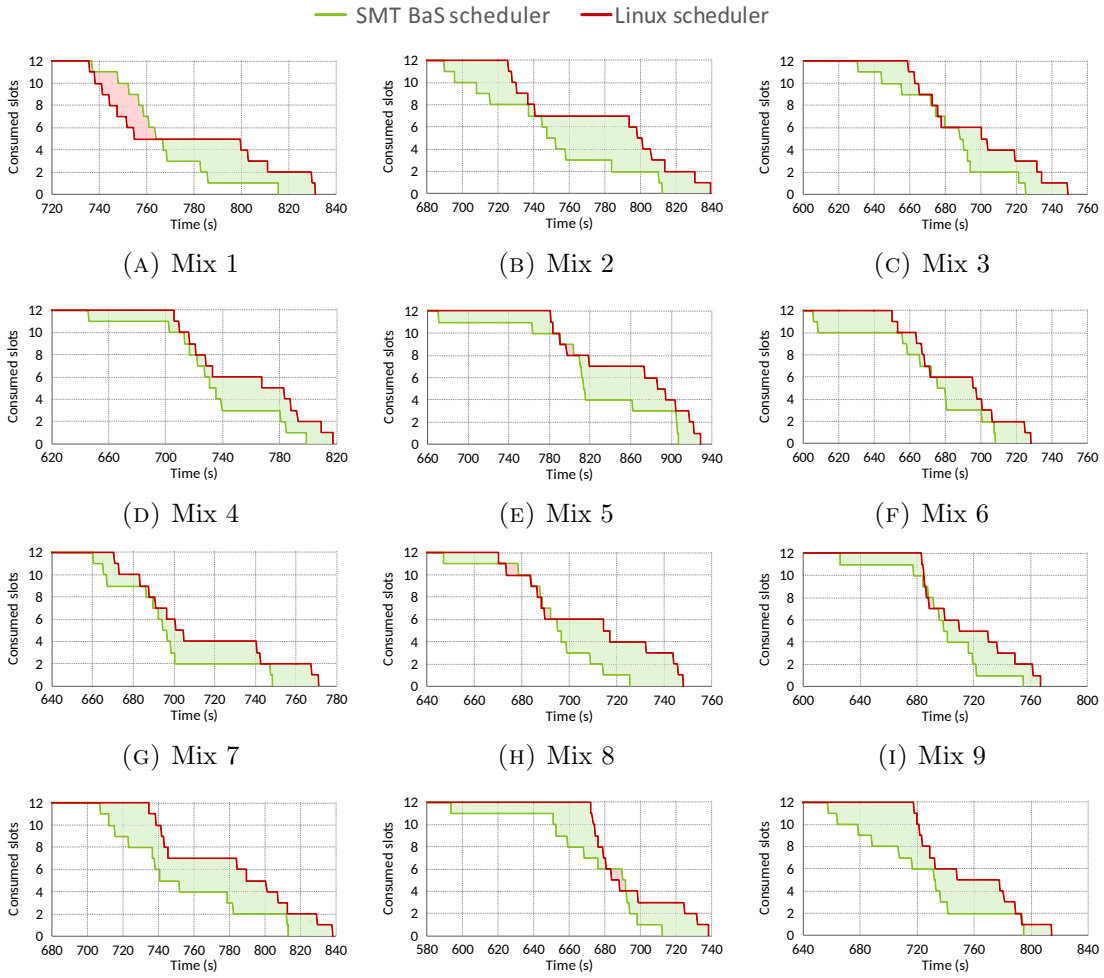
FIGURE 5.15: Speedup of the proposed BaS scheduler over the Linux scheduler using the turnaround time metric.

important to observe that BaS scheduling policy improves both metrics at the same time.

Unfortunately, the turnaround time does not take into account the fact that at the end of the mix execution the number of running processes will probably be lower than the number of hardware contexts of the processor, and these free hardware contexts could be used to run other workloads. To consider them in the evaluation, we measure the *consumed slots*, that is the accumulated number of hardware contexts used in each quantum required to complete the execution of a given workload. Notice that consumed slots could be more meaningful than turnaround time, since it gives lower weight to the quanta where the number of running processes is lower than the number of hardware contexts.

Figure 5.16 presents the evolution of the consumed slots during the execution of the studied mixes, which shows how the hardware contexts are released earlier with BaS than with Linux. The plot for each mix presents the number of consumed slots in the y-axis, that is, the number of threads running at each quantum. It ranges from twelve, the maximum number of threads that can run simultaneously in the experimental platform (six dual-threaded cores), to zero, which is the point where the workload execution finishes.

The plots show that the benefits provided by BaS, which are colored in green, go beyond the reduction in turnaround time. The proposed scheduling policy usually finishes the



(A) Mix 1 (B) Mix 2 (C) Mix 3

(D) Mix 4 (E) Mix 5 (F) Mix 6

(G) Mix 7 (H) Mix 8 (I) Mix 9

(J) Mix 10 (K) Mix 11 (L) Mix 12

FIGURE 5.16: Consumed slots in the workloads. The proposed scheduler saves slots in the green area, while Linux does it in the red area.

processes that form a workload earlier, allowing the scheduler to put some cores into a low power state or use them to run a different workload. Notice that an early completion of the processes can only be achieved without enlarging the execution time of the mixes if the bandwidth contention along the memory hierarchy is reduced, which is the main goal of the proposed scheduler.

For instance, Figure 5.16b, Figure 5.16k, and Figure 5.16l present the evolution of the consumed slots of mixes 2, 11, and 12, which showed the highest speedups with the previous metrics. As observed, BaS significantly reduces the number of slots required to complete the execution of the mixes. On the other hand, Figure 5.16a, Figure 5.16d, and Figure 5.16i present the consumed slots plots for some mixes that showed the lowest

speedups in the metrics previously studied. Even in these cases, BaS is able to bring forward the completion of the processes with respect to Linux, saving a noticeable amount of execution slots. Note that in mix 1 (Figure 5.16a), although Linux saves more execution slots from, approximately, second 740 to second 760 (bounded by the area shaded in red color), BaS saves a higher number of slots through the overall execution, which compensates this loss. With a lower magnitude, the same effect can also be observed in mix 9 and mix 11.

5.5 Summary

This chapter has addressed bandwidth contention on current SMT multicore processors, mainly focusing on how L1 bandwidth contention affects the performance of the processes running simultaneously on an SMT core. Two interesting findings have been made relative to L1 bandwidth contention: i) performance and L1 bandwidth consumption of a given process follow the same shape over the execution time regardless of the process runs in stand-alone execution or with co-runners, and ii) when two processes run simultaneously on an SMT core, its L1 bandwidth is insufficient to fit the requirements of both processes, and the implicit drops in the L1 bandwidth and IPC of a process trigger the opposite effect in the co-runner.

To deal with the observed L1 bandwidth contention, we propose a process allocation policy with the goal of balancing the L1 requests among the processor cores, which reduces contention and increases performance. The devised process allocation policy dynamically reads performance counters to update the L1 bandwidth requirements of the processes and adapts the process allocation according to the phase behavior of the applications.

Since main memory bandwidth contention can drop the performance up to 50% on the experimental platform, we combine the Dynamic L1 bandwidth-aware process allocation policy with a process selection algorithm that is aware of main memory bandwidth contention. The proposed process selection policy distributes the main memory accesses of the processes of a workload along its execution time by selecting the processes on each quantum that match a target main memory bandwidth utilization. Unlike previous

proposals, the target main memory bandwidth utilization to reach in each quantum is obtained at run-time without any preliminary information of the processes to be run.

Experimental evaluation with on Intel Xeon E5645 processor has shown that the Dynamic L1 bandwidth-aware process allocation policy significantly improves the performance with respect to the process allocation performed by Linux, which in many cases is unable to improve the performance of a random policy further than 1%. In contrast, the proposed policy achieves speedups as high as 10% over the random scheduler and doubles the speedups obtained by Linux in most evaluated mixes. Regarding the SMT bandwidth-aware scheduler, which addressed both main memory and L1 bandwidth contention, it achieves performance benefits up to 6.7%, with a geometric mean of speedups by 4.6% with respect to Linux.

The work discussed in this chapter has been published in [63], [64], and [65].

Chapter 6

Progress-Aware Scheduling to Address Fairness in SMT Multicores

Most scheduling algorithms are exclusively focused on performance, giving fairness a secondary or even inexistent role. This chapter concentrates on progress-aware schedulers to address system fairness. These schedulers estimate, at run-time, the progress made by the processes with respect to their isolated execution, which allows calculating the actual unfairness of a mix execution. Based on these estimates, processes with lower accumulated progress can be prioritized to improve system fairness.

This chapter is organized as follows. First, we discuss how progress can be estimated and identify the possible sources that cause inaccuracy when estimating it on SMT multicores. Next, two progress-aware scheduling algorithms are proposed. The first one is completely focused on maximizing fairness, while the second one simultaneously addresses both fairness and performance. Finally, the fairness and performance achieved with the proposed scheduling algorithms is evaluated.

6.1 Estimating Progress

Accurately estimating how a process progresses at run-time with respect to its isolated execution is the key point to provide fairness. Progress estimations are particularly challenging in SMT multicores due to the constant resource sharing among the processes running on the same SMT core. Such resource sharing triggers an interference that strongly and distinctly affects the performance of different processes. This interference is the main cause that lead the systems to be highly unfair. As an example of an SMT multicore, this chapter uses the system with the six-core dual-threaded Intel Xeon E5645 processor as experimental platform in the performed experiments.

To estimate the progress made by the processes, we use Equation 6.1, which accumulates, for the elapsed quanta, the ratio between the measured IPC that a process achieves when running concurrently with other processes ($IPC^{i}_{co-runners}$) and the estimated IPC that such a process would have achieved in isolation (IPC^{i}_{alone}) during the same quantum. The former is directly measured from the committed instructions and execution cycles gathered with performance counters. The difficulty lies in estimating isolated performance.

$$Progress = \sum_{i=0}^{Q} \frac{IPC^{i}_{co-runners}}{IPC^{i}_{alone}} \tag{6.1}$$

To estimate the stand-alone IPC of a process, we propose to arrange a low-contention schedule aimed at minimizing performance interference among the scheduled processes. The IPC of a target process is measured during the execution of the devised low-contention schedule and used as estimate of its stand-alone performance for the n following quanta in which the process is scheduled. During these quanta, a scheduling algorithm can increase system fairness by prioritizing processes with lower accumulated progress.

Two main reasons can cause deviations in the IPC estimates: i) the stand-alone IPC is assumed valid for a too long period (number of quanta), and ii) thread interference is higher than expected in the devised low-contention schedules. Below these two deviation sources are analyzed.

6.1.1 Period Length between IPC Estimates

Defining the period length between IPC estimates represents a trade-off between estimation accuracy and fairness. The longer the interval, the higher the number of quanta where a given IPC estimate is assumed valid, hence inaccuracy potentially rises. Conversely, the shorter the interval, the higher the number of quanta devoted to IPC estimations; thus, the fewer the quanta used to address fairness.

This section analyzes the accuracy of IPC estimates varying the period length between estimates. The study compares, for each benchmark, the average deviation (along its complete execution) of the IPC estimates with respect to the real IPC of each quantum. Figure 6.1 presents the average and maximum deviations across all the SPEC CPU2006 benchmarks when ranging the period length between IPC estimates from one to eight seconds. Green and red lines show the average and maximum deviations, respectively, across all the benchmarks. Average values are relatively low (below 2%) for periods shorter than eight seconds. Maximum deviation, however, grows faster as the period between IPC estimates is enlarged. Nonetheless, results show that reasonable accuracy can be achieved by estimating the stand-alone IPC of the benchmarks at relatively long periods of time.

To provide further insights in this claim, Figure 6.2 compares the dynamic IPC evolution of a subset of benchmarks measured at 200 milliseconds and 6 seconds periods. When the process presents uniform IPC, like *hmmer*, practically no difference is observed between

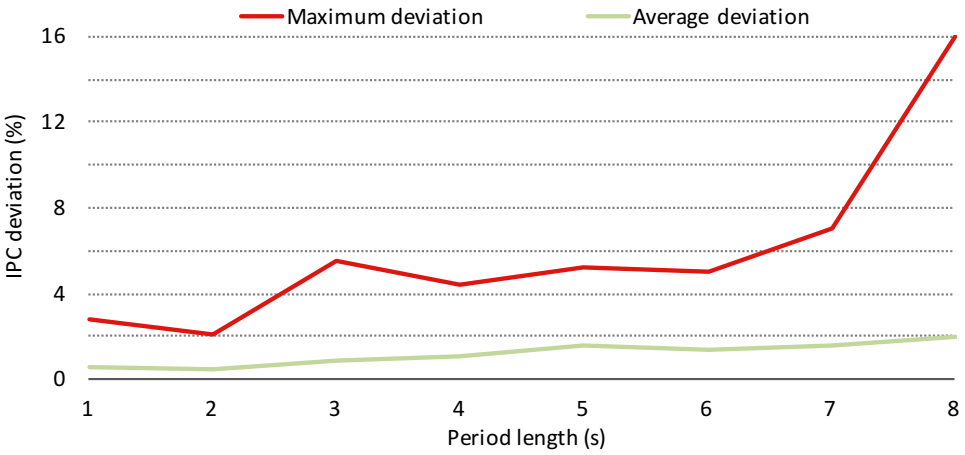
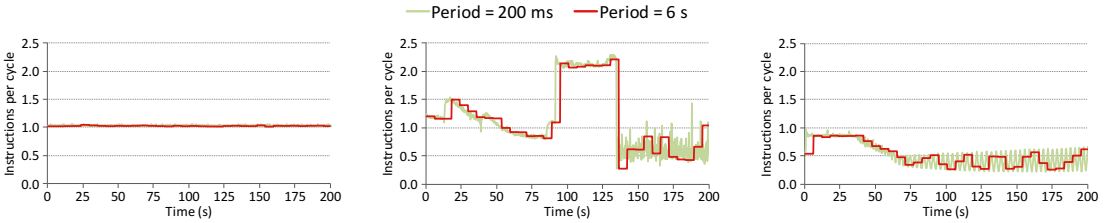


FIGURE 6.1: IPC deviation when increasing the period length between measures.



(A) *Hmmer* (B) *Xalancbmk* (C) *CactusADM*

FIGURE 6.2: Comparison between IPC measured each 200 ms and each 6 s.

both sampling periods (in spite that the 6 seconds period is 30× longer than the shorter one), while slight differences can be observed with processes with different phases of execution like *xalancbmk* or *cactusADM*.

Furthermore, these values were obtained running processes alone in the system, however, processes experience a slower (or much slower) progress running with a co-runner on the same SMT core. Therefore, longer periods might be considered in the devised scheduling algorithm (see Section 6.2.4).

6.1.2 Process Interference in Low-Contention Schedules

This section studies the performance interference that raises among processes in the shared resources. The analysis first considers only pairs of benchmarks running on an SMT core; then, the schedule is extended with more benchmarks (running on several cores) to analyze the impact of overall interference on individual performance. If the interference is prudent, then the stand-alone IPC may be estimated in schedules with multiple applications achieving reasonable accuracy.

As mentioned above, two levels of interference are distinguishable in an SMT multicore: intra- and inter-core. Intra-core interference is caused due to sharing critical core resources for performance like the L1 cache, the dispatch width, the instructions queues, or the execution units. This interference only appears among the processes that run in the same SMT core. In contrast, inter-core interference can be caused by any other process running in the multicore processor since they share the main memory and the cache hierarchy (the L3 cache in our target system).

Intra-core interference impacts more strongly on performance since a wider set of resources, including L1 caches and execution units, are shared among the processes running concurrently on the same core. Two processes that perform scarce use of inter-core resources can run concurrently without noticeable performance degradation. However, intra-core interference causes any two processes running simultaneously on the same SMT core to significantly reduce their performance. Therefore, to estimate the standalone IPC of a process it needs be scheduled alone on a core, avoiding intra-core interference. From now on, this section focuses on the performance interference that raises among processes running on different cores.

6.1.2.1 Interference between Pairs of Benchmarks

First, the analysis focuses on inter-core interference between pairs of benchmarks. To carry out this study, all possible couples of benchmarks are run, each benchmark on a distinct core, and their individual performance is compared to that achieved in isolation.

	perlbench	bzip2	gcc	mcf	gobmk	hmmer	sjeng	libquantum	h264ref	omnetpp	astar	xalancbmk	bwaves	gamess	milc	zeusMP	gromacs	cactusADM	leslie3d	namd	dealII	soplex	povray	gemsFDTD	lbm
perlbench	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	0%	1%	1%	0%	1%	0%	0%	1%	0%	0%	1%
bzip2	0%	0%	1%	6%	0%	0%	0%	8%	0%	4%	3%	3%	8%	0%	9%	3%	0%	2%	7%	0%	1%	6%	0%	8%	9%
gcc	0%	1%	3%	8%	1%	0%	1%	10%	1%	6%	4%	5%	11%	0%	11%	5%	1%	3%	9%	0%	1%	8%	0%	10%	10%
mcf	0%	0%	3%	24%	2%	2%	1%	28%	3%	15%	13%	17%	29%	0%	29%	4%	2%	6%	24%	0%	5%	13%	0%	28%	32%
gobmk	0%	0%	0%	1%	0%	0%	0%	4%	1%	1%	0%	2%	2%	0%	4%	2%	1%	2%	3%	1%	0%	3%	0%	4%	4%
hmmer	0%	0%	0%	1%	0%	0%	0%	2%	0%	1%	0%	0%	3%	0%	2%	1%	0%	0%	2%	0%	0%	1%	0%	1%	3%
sjeng	0%	0%	0%	1%	0%	0%	3%	6%	3%	1%	4%	4%	6%	3%	6%	4%	3%	4%	6%	3%	3%	5%	3%	6%	6%
libquantum	0%	0%	0%	0%	0%	0%	0%	1%	0%	2%	0%	0%	2%	0%	1%	1%	0%	0%	0%	0%	0%	2%	0%	4%	4%
h264ref	0%	0%	0%	4%	0%	0%	0%	6%	0%	2%	1%	3%	6%	0%	11%	2%	0%	1%	8%	0%	1%	6%	0%	10%	6%
omnetpp	1%	6%	7%	15%	3%	3%	3%	17%	5%	14%	10%	13%	17%	2%	18%	11%	4%	10%	16%	1%	4%	15%	0%	19%	19%
astar	1%	4%	5%	12%	2%	2%	3%	14%	3%	10%	17%	17%	14%	5%	22%	15%	7%	5%	21%	6%	3%	20%	1%	22%	23%
xalancbmk	0%	4%	7%	21%	2%	2%	2%	25%	3%	15%	14%	19%	28%	1%	28%	10%	2%	6%	23%	1%	4%	24%	0%	27%	30%
bwaves	0%	0%	0%	1%	0%	0%	0%	1%	0%	1%	1%	0%	9%	8%	9%	8%	8%	8%	9%	8%	8%	1%	8%	9%	9%
gamess	0%	0%	0%	1%	0%	1%	0%	1%	0%	0%	0%	1%	1%	0%	1%	1%	0%	0%	0%	0%	0%	1%	0%	1%	1%
milc	0%	0%	0%	1%	0%	0%	0%	2%	0%	1%	1%	1%	2%	0%	25%	24%	24%	24%	3%	24%	24%	24%	24%	25%	25%
zeusMP	1%	1%	1%	2%	0%	0%	1%	2%	0%	1%	1%	1%	2%	0%	2%	10%	8%	8%	9%	8%	8%	9%	0%	9%	9%
gromacs	0%	0%	2%	2%	0%	0%	1%	2%	0%	1%	2%	1%	2%	0%	2%	1%	1%	1%	3%	0%	1%	1%	0%	3%	3%
cactusADM	0%	1%	3%	8%	1%	0%	0%	9%	0%	6%	4%	5%	9%	0%	9%	5%	0%	4%	9%	0%	1%	8%	0%	10%	10%
leslie3d	0%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	1%	0%	0%	0%	20%	18%	18%	19%	18%	20%	20%
namd	0%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	1%	0%	0%	0%	1%	0%	0%	1%	0%	2%	1%
dealII	0%	0%	0%	1%	0%	0%	0%	2%	0%	1%	1%	0%	2%	0%	1%	0%	0%	0%	1%	0%	1%	1%	0%	2%	2%
soplex	2%	4%	6%	19%	3%	3%	3%	20%	5%	13%	11%	15%	21%	1%	19%	11%	2%	7%	17%	1%	4%	20%	0%	21%	24%
povray	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	1%	0%	0%	0%	0%	0%	0%	1%	1%	0%	1%
gemsFDTD	0%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	1%	0%	0%	0%	0%	0%	0%	1%	0%	2%	2%
lbm	0%	4%	23%	6%	0%	0%	0%	8%	0%	3%	2%	5%	11%	0%	9%	2%	0%	1%	7%	0%	1%	8%	0%	11%	36%

FIGURE 6.3: Performance degradation due to inter-core interference running pairs of benchmarks. Each row shows the degradation of a benchmark running with each co-runner on different cores.

Figure 6.3 presents the results. Each row shows the performance degradation of a benchmark caused by any possible co-runner. For instance, *bzip2* suffers a performance drop by 6% when running with *mcf*, while the performance of *mcf* is not reduced when it is executed with *bzip2*. Similarly, each column depicts the performance degradation a benchmark induces to each co-runner. For instance, among all the possible co-runners, *libquantum* causes the highest performance drop (by 28%) to *mcf*.

The performance degradation level is highlighted in the table with different colors. A cell (X,Y) colored in green, orange, or red, means that process Y affects the performance of process X less than 5%, between 5% and 10%, or more than 10%, respectively.

Depending on how benchmarks affect the performance of their co-runners, they can be classified in two main categories: *heavy-sharing* and *light-sharing*. The former category includes benchmarks that strongly affect the performance (i.e., above 10%) of a significant subset of the possible co-runners. Examples of benchmarks belonging to this category are *mcf*, *libquantum*, and *omnetpp*. The *light-sharing* category includes those benchmarks that scarcely affect the performance of the co-runners since they make a scarce use of the shared resources. This category includes benchmarks in columns that mostly show cells colored in green.

Note that for any target benchmark, a wide set of co-runners impacting its performance less than 5% can be found. For example, *perlbench* can be coupled to estimate its standalone IPC with any other benchmark since the maximum performance degradation it suffers is by 1%. Following the same rule, *astar* can be paired with *perlbench*, *bzip2*, *gcc* or *gobmk*, among others, but not with *mcf* or *omnetpp*.

A scheduler could use the above offline analysis to predict the performance interference. However, this way is unfeasible from a practical point of view. In contrast, our approach consists in classifying benchmarks as heavy-sharing or light-sharing at run-time, depending on their phase behavior. After a wide set of experiments analyzing distinct performance counters, we found that the bandwidth consumption of the uncore shared resources, that is, main memory and LLC, is appropriate to perform this classification.

At a first glance, it might be expected that processes with either high LLC or main memory bandwidth consumption fall on the heavy-sharing category since they are likely to interfere their co-runners performances. Conversely, processes that perform a scarce

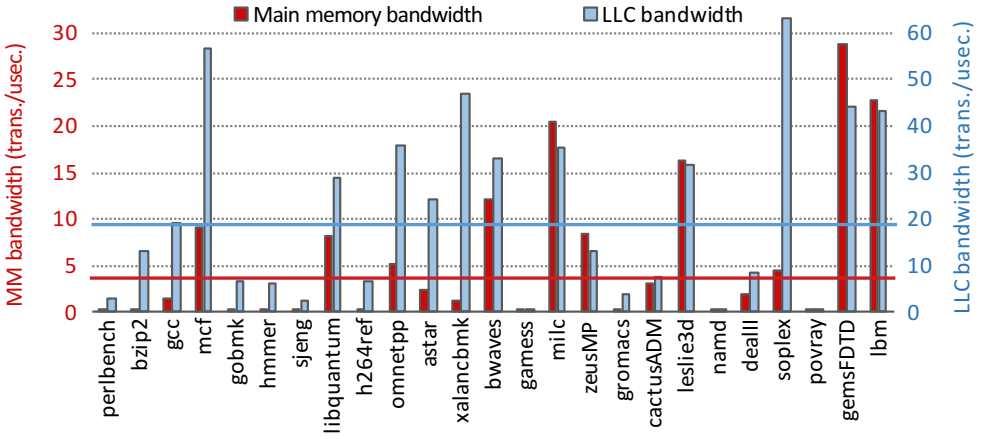


FIGURE 6.4: Average main memory and LLC bandwidth. The red and blue lines represent the thresholds devised on the main memory and LLC bandwidth to classify the benchmarks as heavy- or light- sharing.

use of these resources are unlikely to interfere with co-runners, so they could be classified as light-sharing.

Figure 6.4 depicts the average main memory and LLC bandwidth consumption of the benchmarks in stand-alone execution. As observed, all the benchmarks whose LLC bandwidth utilization is above 19.0 transactions/microsecond or whose main memory bandwidth utilization is above 3.5 transactions/microsecond belong to the heavy-sharing category. Otherwise, they fall in the light-sharing category. Notice that these thresholds are estimated in stand-alone execution, and the cache interference when the processes run concurrently can cause cache misses to grow. Thus, it is likely that processes with bandwidth utilizations close to the thresholds (e.g., *gcc* or *cactusADM*) end up exceeding them when running concurrently with other processes.

6.1.2.2 Cumulative Interference in Low-Contention Schedules

The previous section analyzed the interference in low-contention schedules composed only of a pair of co-runners. However, to avoid significant throughput losses when IPC estimates are required, the number of light-sharing benchmarks executing in a low-contention schedule should be as high as possible.

To analyze the cumulative interference in low-contention schedules consisting of more than two co-runners, the performance of all possible groups of three, four, five, and six

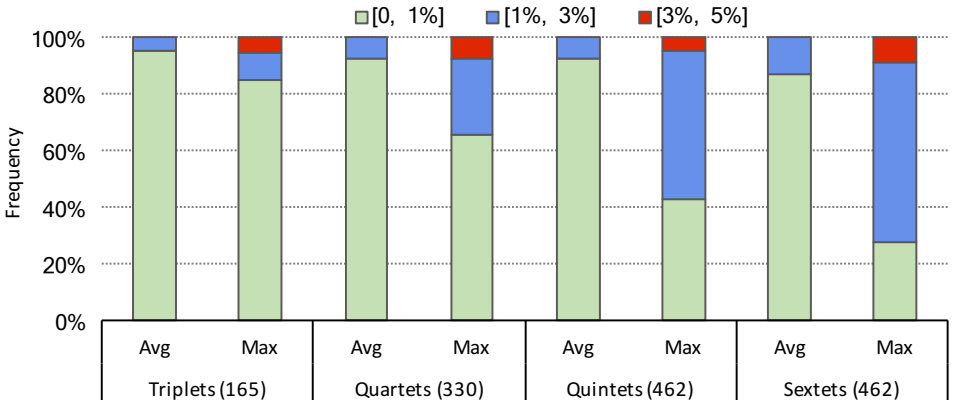


FIGURE 6.5: Histogram of the performance degradation on light-sharing schedules. In brackets, the total number of evaluated schedules.

light-sharing benchmarks has been explored. Figure 6.5 depicts the results. Looking at the average performance slowdown (Avg. bar), it can be observed that the interference is acceptable even in large groups. For instance, more than 85% of all the possible 6-process schedules (i.e., 462 sextets) present an average slowdown below 1%. The Max bar refers to the slowdown of the benchmark suffering the highest slowdown in the schedule. As expected, it grows as the number of processes of the schedules rises. However, only by 10% of the benchmarks in the 6-process schedules present a maximum performance degradation falling in between 3% and 5%.

To sum up, these results show that the low-contention schedules used to obtain IPC estimates are not restricted to a small number of processes, but good accuracy can be achieved even when the schedule disposes of at least one process per core. This finding is important because being forced to run schedules with a few processes when a new IPC estimate were required could strongly affect the system throughput.

6.2 Progress-Aware Fair Scheduling

The progress-aware Fair scheduling algorithm is designed to allow all the processes to achieve the same progress over the mix execution. Since the impact of interference on individual process performance widely differs across the studied processes, each process requires a distinct execution time to achieve the same progress. In other words, processes

with higher performance degradation induced by co-runners require more quanta of execution than processes with lower performance degradation to make the same progress.

In addition, as explained above, the scheduler needs to use some quanta periodically to estimate the isolated performance of each process, used to calculate their progress at run-time. These quanta can affect both fairness and performance due to scheduling constraints when creating low-contention schedules. For instance, in low-contention schedules light-sharing processes are prioritized over heavy-sharing processes regardless of their accumulated progress.

Hence, the algorithm implements two different process selection policies: IPC estimation-oriented and fairness-oriented. The former applies when any process needs to estimate its isolated IPC and a low-contention schedule is required. The latter guides the scheduling to improve fairness and applies when all the processes have valid IPC estimates. Algorithm 7 presents the pseudocode of the proposed scheduling algorithm, which differentiates between both process selection policies: IPC estimation-oriented (lines 1 to 9) and fairness-oriented (lines 10 to 16).

Performance counters play an essential role to implement the proposal and are used to dynamically compute the IPC and bandwidth utilization of the processes. The IPC of the processes is used to estimate their progress, while the main memory and LLC bandwidth utilization of the last executed quantum are used to determine at run-time if the process belongs to the light- or heavy- sharing category, as explained in Section 6.1. Finally, the L1 bandwidth utilization of the processes is used to guide the process allocation. For this purpose, the evens *instructions_retired*, *unhalted_core_cycles*, *offcore_response_0.any_data.local_dram* (main memory accesses), *offcore_response_0.any_data.local_cache* (LLC accesses), and *perf_count_hw_cache_l1d.access* (L1 accesses) are gathered.

The IPC estimate of each process is kept valid for a certain number n of quanta (see Section 6.2.4). A saturating counter is assigned to each process P to account the elapsed quanta and is updated each quantum the process is scheduled. When any counter saturates, the scheduler executes the IPC estimation-oriented process selection, and the counter is reset. Otherwise, the fairness-oriented process selection determines the schedules for the following quantum. These two policies and the process allocation

policy are described in the following section. Then, the implementation parameters are discussed.

The proposed algorithm could be extended to support user-defined priorities (i.e., Linux *nice* priorities) if required. Priorities can be established based on the progress made by each process similar to how the Linux CFS scheduler uses the *nice value* to weight the proportion of processor a process is to receive [66]. In this context, a process with higher priority should progress faster than a process with a lower priority. Thus, process priorities could be implemented by allowing a process to progress n% faster than others, where n depends on the nice value.

Algorithm 7 Progress-aware Fair scheduling algorithm

Update IPC and bandwidth utilization for each process P run in the last quantum.

PROCESS SELECTION

if the IPC estimation of any process P *has expired* **then**

IPC ESTIMATION-ORIENTED

  Reserve P to an entire core.
  if P is a light-sharing process **then**
    while IPC estimation of any light-sharing process P_{LS} is *close to expire* **and** there are free cores **do**
      Reserve P_{LS} to an entire core.
    end while
  end if
  Select as many light-sharing processes as available hardware threads, prioritizing those with lower progress.
else

FAIRNESS-ORIENTED

  Calculate the average progress of the processes of the mix.
  while a process P_{LP} is progressing *below the average* **do**
    Allocate P_{LP} to an entire core.
  end while
  Select as many processes as available hardware threads prioritizing those with lowest progress.
end if

PROCESS ALLOCATION

Allocate the threads that reserved an entire core to a core
Sort the remaining selected processes in ascending BW_{L1}
while there are unallocated processes **do**
  Allocate the processes P_{head} and P_{tail} to the same core
end while

6.2.1 IPC Estimation-Oriented Process Selection

The IPC estimation-oriented process selection (lines 2 to 10 of Algorithm 7) is triggered when a valid IPC estimate is required (line 1) for any process P. A low-contention scenario is scheduled to avoid intra-core and minimize inter-core interference. The former interference is removed by running P alone on an entire core. For now, an entire core is reserved for the process (line 2); the final core will be assigned by the process allocation policy. Inter-core interference is minimized by only including light-sharing processes in the schedule. If P itself is a light-sharing process (line 3), and there are other light-sharing processes whose IPC estimates are *close to expire* (see Section 6.2.4), then each of them also reserves an individual core (line 5). This way allows multiple IPC estimates to be obtained during the same quantum.

After that, the remaining cores are filled with light-sharing processes. In particular, as many light-sharing processes as available hardware contexts are scheduled (line 8). For the sake of fairness, the scheduler prioritizes the light-sharing processes that have experienced less accumulated progress. Moreover, if there are not enough light-sharing processes in the workload, the exceeding hardware contexts are left free, since selecting heavy-sharing processes could create bandwidth contention and affect the accuracy of the IPC estimates that are being performed during the quantum. The selected processes will be smartly allocated to cores in pairs to reduce the SMT intra-core interference in the process allocation step.

6.2.2 Fairness-Oriented Process Selection

As a rule of thumb, to improve fairness, the scheduling algorithm selects those processes with lowest accumulated progress to run the following quantum (line 14). In the Intel Xeon E5645 processor used as experimental platform, with six dual-threaded cores, the twelve processes with lowest progress are selected. Nonetheless, to maximize fairness, the process selection policy checks in a prior step if the progress of any process is falling behind the others. To this end, the scheduler computes the average progress of all the processes of the mix (line 10). Then, it is checked if the progress of any process is by 5% below the average (line 11). If there are processes in this situation, the scheduler reserves an entire core to each of them (line 12). Running them along on a core speedups

their individual progress since alone execution in the core is faster than SMT execution, where two processes are simultaneously run on the same core. After that, the algorithm proceeds selecting the remaining processes with lower accumulated progress.

Although it is not shown in the algorithm, note that even when working in the fairness-oriented process selection policy, unprompted scenarios can be leveraged to estimate isolated IPCs. For instance, if a schedule only includes light-sharing processes, the isolated performance can be estimated for those processes individually allocated to an entire core to boost their performance and compensate their lower accumulated progress, regardless of the deadline of their current IPC estimate.

6.2.3 Process Allocation

After the process selection has been performed by the IPC estimation-oriented process selection policy, some processes will require to run alone on a core. Thus, the first step of the process allocation assigns all these processes to entire cores (line 21). After that, the remaining processes are allocated using the Dynamic L1 bandwidth-aware process allocation policy (see Section 5.2.2), which reduces the interference between processes by allocating them to cores so that the L1 bandwidth is evenly distributed among the L1 caches. To this end, the processes are sorted in a list in ascending L1 bandwidth order (line 22). Then, the processes placed at the head and tail of the list are removed from it and allocated together to the same core. This action is performed iteratively until the list is empty (lines 23 to 25).

6.2.4 Implementation Considerations

The proposed algorithm relies on several parameters that must be tuned to provide the best results. Depending on the values of these parameters the schedulers can: i) maximize fairness with no performance consideration, ii) prioritize fairness over (but without compromising) performance. We focus our study on the second case because we do not want to improve fairness at the expense of performance. Different values for each parameter have been evaluated. This section presents and discusses the values used to evaluate the proposal, analyzing their advantages and disadvantages.

The maximum period between two standalone IPC estimates has been empirically set to 8 seconds, that is fourty 200 ms quanta ($n = 40$). Experimental results show that shorter periods can enhance fairness, but strongly affecting performance. Conversely, longer intervals negatively affect fairness without providing significant performance benefits. In the algorithm implementation, we also consider that an IPC estimation is *close to expire* when the number of quanta a process has been scheduled since its last estimation is half the maximum number of quanta between estimates ($n = 20$).

Main memory and LLC bandwidth thresholds to discern between light- and heavy-sharing processes are set to 3.5 and 19.0 transactions/microsecond, respectively, since these values offer a good trade-off between fairness and performance. Higher thresholds include more benchmarks classified as light-sharing even if they are not (i.e., they introduce considerable contention). As a consequence, more contention than expected can be generated, affecting the accuracy of the estimates and thus, system fairness can be compromised. On the contrary, lower thresholds classify more processes as heavy-sharing. However, in this case, performance may be affected since a higher number of heavy-sharing processes limits scheduling flexibility.

The last parameter used in the algorithm determines when a given process is unfairly *progressing slower than the others*. As explained before, when this situation occurs, the process progressing slower is allocated alone on a core to accelerate its progress, so avoiding inter-core interference. We consider that a process is progressing too slow when its progress differs above 5% from the average progress of the processes of the mix. Using a higher threshold would enlarge the accepted unfairness before taking scheduling decisions to reduce it. Conversely, a lower threshold would trigger the progress *correction* too frequently, affecting the system performance.

6.3 Progress-Aware Perf&Fair Scheduling

Unlike the progress-aware Fair proposal (from now on *Fair*), the progress-aware Perf&Fair scheduling algorithm confronts a twofold goal: reducing unfairness while enhancing performance. On the one hand, to lessen unfairness it estimates the progress made by each process and prioritizes the processes with lower accumulated progress, following the same idea of *Fair*. On the other hand, to improve performance, it minimizes main memory

and L1 cache bandwidth contention (see Chapter 5). The difficulty to accomplish both goals lies on finding the way to schedule processes so that both goals do not conflict on the scheduling decisions.

Perf&Fair follows a similar structure as *Fair*, implementing two distinct process selection policies referred to as IPC estimation-oriented and performance- & fairness- oriented. The former policy applies when any process needs to estimate its isolated IPC, and closely resembles the same process selection policy of *Fair*. The latter guides the scheduler to enhance performance and fairness, and applies when all the processes have valid IPC estimates. This process selection policy represents the main difference between both progress-aware scheduling algorithms. As discussed in Section 6.2.4, the IPC estimates for each process are kept valid for 40 quanta, because this interval offers a good trade-off between IPC estimation accuracy and overhead due to IPC estimations.

As discussed for *Fair*, Perf&Fair uses performance counters to i) measure the IPC of the processes and estimate their progress, ii) dynamically classify the processes as light- or heavy-sharing, and iii) guide scheduling decisions based on the bandwidth consumption of the processes at the different levels of the memory hierarchy. Unlike *Fair*, Perf&Fair also considers main memory bandwidth contention to select the processes to be run each quantum in the performance- & fairness- oriented process selection.

Algorithm 8 presents the pseudocode of the progress-aware Perf&Fair scheduler, which presents two process selection policies: IPC estimation-oriented (lines 3 to 9) and performance- & fairness- oriented (lines 11 to 26). The Dynamic L1 bandwidth-aware process allocation policy (discussed in Section 6.2.3) is used to allocate the processes to the cores (lines 27 to 31). Below, the two process selection policies are discussed.

6.3.1 IPC Estimation-Oriented Process Selection

The IPC estimation-oriented process selection policy in Perf&Fair has the same goal and works as the IPC estimation-oriented process selection policy of *Fair* (see Section 6.2.1). The main difference is that the estimation quantum length is set to 100 milliseconds. By halving the quantum length, the accuracy of the estimations is slightly reduced, which affects fairness, but more performance benefits can be achieved. Notice that even for a quantum length of 100 ms there is a minor scheduling overhead.

Algorithm 8 Progress-Aware Perf&Fair scheduler

Update IPC and bandwidth utilization for each process P run in the last quantum

PROCESS SELECTION

if the IPC estimation of any process P *has expired* **then**

IPC ESTIMATION-ORIENTED (Q_{length}=100 ms)

  Reserve an entire core to P
  if P is a light-sharing process **then**
    while IPC estimation of any light-sharing process P_{LS} is *close to expire* **and** there are free cores **do**
      Reserve an entire core to P_{LS}
    end while
  end if
  Select as many light-sharing processes as available hardware threads, prioritizing those with lower progress
else

PERFORMANCE- & FAIRNESS- ORIENTED (Q_{length}=200 ms)

  Calculate $OATR = \frac{\sum_{p=0}^{N} Avg\_BW_{MM}^{p}}{N} \times$ #CPUs
  Set $BW_{Remain} = OATR$, $CPU_{Remain} = \#CPUs$
  Set $MaxP =$ Maximum progress $\forall$ P_X $\exists$ Process queue
  while $CPU_{Remain} > 0$ **do**
    if $\exists$ processes with Progress(P_i) + 1 $< MaxP$ **then**
      $\forall$ P_i with Progress(P_i) + 1 $< MaxP$ **do**
      Select the process P that maximizes
      $\text{FITNESS(p)} = \frac{1}{\left|\frac{BW_{Remain}}{CPU_{Remain}} - BW_{MM}^{p}\right|}$
    else
      $\forall$ P_i **do**
      Select the process P that maximizes
      $\text{FITNESS(p)} = \frac{1}{\left|\frac{BW_{Remain}}{CPU_{Remain}} - BW_{MM}^{p}\right|}$
    end if
    Update $BW_{Remain} -= BW_{MM}^{p}$, $CPU_{Remain}--$
  end while
end if

PROCESS ALLOCATION

Allocate the threads that reserved an entire core to a core
Sort the remaining selected processes in ascending BW_{L1}
while there are unallocated processes **do**
  Allocate the processes P_{head} and P_{tail} to the same core
end while

6.3.2 Performance- & Fairness- Oriented Process Selection

In order to improve performance without sacrificing fairness, processes must be carefully selected. The main idea behind this process selection policy consists in selecting the processes following a performance approach but preventing, as much as possible, unfairness from growing.

Regarding performance, the algorithm calculates first the Online Average Transaction Rate (OATR) to select the processes for the following quantum (see Section 5.2.1). The OATR and the number of hardware contexts are initially assigned to the variables BW_{Remain} and CPU_{Remain}, respectively (line 12). As in previous scheduling algorithms, these variables are iteratively updated as processes are selected to be run in the next quantum (line 24).

To prevent unfairness from growing, the algorithm restricts the process selection (when it is possible) to the processes whose current progress is so low that if they were run in the next quantum (Q_{i+1}), their progress after Q_{i+1} should not exceed the current maximum progress (i.e., at the end of quantum Q_i) among all the processes of the mix. To do that, the algorithm determines the maximum progress $MaxP$ achieved among the running processes (line 13). Since the progress during a quantum is defined as $\text{IPC}_{co-runners}$ / IPC_{alone}, the maximum increase of progress that a process can experience in a quantum is 1. Based on this fact, only processes whose progress differ more than one unit from $MaxP$ are considered as schedulable at this point (line 16). Among the processes that fulfill the previous condition, the fitness function determines which ones are finally selected (lines 17 and 18) attending to their main memory bandwidth utilization to improve performance (see Section 4.2.2).

When the number of processes that fulfill the progress condition (line 16) is below the number of hardware contexts, all these processes are directly selected to run on the following quantum, updating the BW_{Remain} and CPU_{Remain} variables. The remaining processes, until the number of hardware contexts is reached, are selected by using the fitness function (as explained before), but considering all the remaining processes regardless of their accumulated progress (lines 20 to 22).

6.4 Flexible Progress-Aware Perf&Fair Scheduling: Trading Fairness for Performance

In spite of addressing fairness in addition to performance, Perf&Fair reaches noticeable performance compared to the Linux scheduler and the performance-oriented bandwidth-aware scheduling algoirthm proposed in Section 5.2 we will refer to it as *Perf*). With

low bandwidth requirements, Perf&Fair even improves *Perf*. However, when the average main memory bandwidth utilization of the workload is very high, *Perf* achieves higher performance. Thus, when running these workloads with Perf&Fair, it may be desirable to trade fairness for performance.

In part, *Perf* performs better than Perf&Fair due to the fact that *Perf* does not consider fairness at all and exclusively deals with bandwidth contention to improve performance. Conversely, Perf&Fair applies some constraints every quantum to prevent unfairness from growing, which reduces its ability to deal with bandwidth contention and achieve higher performance. In addition, to address fairness Perf&Fair devotes some quanta to estimate the performance that the processes achieve running in isolation, which are used to compute the progress that the processes make during the workload execution. Unfortunately, estimation quanta also affect negatively the performance since during these quanta the number of scheduled processes is lower than the number of available hardware contexts.

The straightforward solution to allow Perf&Fair to offer different levels of performance/-fairness is to modify the constraint that prevents processes that could exceed the maximum accumulated progress to be selected to run on the next quantum. This constraint ($\forall$ P_i *with Progress*(P_i) + $k < MaxP$) is checked in lines 15 and 16 of Algorithm 8, where $k = 1$. Notice that by lowering the k value (from 1 to 0), the process selection becomes less restrictive in terms of fairness and the algorithm is able to select among more processes. Under no progress constraints (i.e., $k < 0$), the scheduling algorithm addresses main memory bandwidth contention considering all the available processes, exactly as *Perf*. The main problem that this approach presents is that while the constraint is not removed (i.e., $k \geq 0$), the algorithm requires from performance estimates to account the progress of the processes. Because of estimation quanta achieve lower performance, this approach achieves less performance than *Perf*, which does not require from estimation quanta, thus making the above solution not practical.

Therefore, in order to let Perf&Fair to perform closer to *Perf*, it should not only select the processes similarly as *Perf*, but it should also reduce the number of estimation quanta. To achieve this behavior, we propose to combine bursts of quanta scheduled under the Perf&Fair scheduling algorithm (with the original constraint $k = 1$ and including its estimation quanta) with bursts of quanta scheduled under the *Perf* algorithm (neither

considering fairness nor scheduling estimation quanta). Note that before beginning a burst of quanta scheduled with the Perf&Fair algorithm, we reset the stored per-process progress information. The enhanced algorithm, which we call Flexible Perf&Fair, can trade fairness for performance by setting the relative length of the bursts scheduled under each algorithm.

6.5 Evaluation Setup

The experimental evaluation has been performed in the Intel Xeon E5645 system (see Section 3.2.2), and the proposed algorithms have been implemented in the scheduling framework described in Section 3.1.

We follow the process selection evaluation methodology (see Section 3.3.1), setting the target number of instructions for each SPEC CPU2006 benchmark to the number of instructions they execute running alone in the system during 100 seconds. Quantum length is set to 200 milliseconds, except for IPC estimation-oriented process selection quanta in Perf&Fair, where quantum length is set to 100 milliseconds. The overhead arising from the scheduling algorithms is negligible considering the quantum lengths at which scheduling is performed. Overall overhead, including process selection, process allocation and progress accounting, as well as processes and performance counters management, is by 0.1 milliseconds. Note that it is below 0.1% of the quantum length.

Since fairness can be achieved at the cost of performance, it should not be evaluated in isolation but performance metrics should also be considered. The turnaround time of the mixes has been used as performance indicator in this chapter, while the unfairness metric is used to estimate if performance benefits or losses are balanced across all the processes and do not concentrate only on a few of them. Please, refer to Section 3.4 for further details of the metrics and their calculation.

6.5.1 Evaluated Algorithms

The experimental evaluation takes into account the following scheduling algorithms.

- **Linux**: the default Linux Completely Fair Scheduler (CFS).

- **Performance-oriented bandwidth-aware (*Perf*)** the bandwidth-aware scheduling algorithm for SMT multicores, presented in Section 5.2.
- **Progress-aware Fair (*Fair*)**: the progress-aware scheduling algorithm proposed in Section 6.2, which exclusively tries to maximize fairness.
- **Progress-aware Perf&Fair (Perf&Fair)**: the progress-aware scheduling algorithm proposed in Section 6.3, which simultaneously addresses performance and fairness.
- **Flexible progress-aware Perf&Fair (Flexible Perf&Fair)**: the progress-aware scheduling algorithm proposed in Section 6.4, which can be configured to achieve different trade-offs between performance and fairness.
- **Oracle scheduler (Oracle):** the Perf&Fair scheduling algorithm enhanced with offline information. It uses stand-alone IPC traces to compute the progress of the processes and the IABW (see Section 4.2.1), which is used as target bandwidth utilization for each quantum.

6.5.2 Mix Design

A set of thirteen mixes composed of twenty-four SPEC CPU2006 benchmarks has been designed to evaluate the proposed algorithms. Each mix consists of a variety of *light-* and *heavy-sharing* benchmarks. The number of *heavy-sharing* processes in the workloads ranges from 9 to 17. Table 6.1 presents the mixes used in the experimental evaluation, sorted by their associated average main memory bandwidth consumption (BW_{MM}).

6.6 Experimental Evaluation

First, the experimental evaluation focuses on *Fair*. We compare the fairness that the proposed algorithm and Linux achieve, and analyze how the progress of the processes evolve over the workload execution time. We also study the accuracy of the IPC estimations. Next, we perform a wider evaluation where the fairness and performance achieved by Perf&Fair is compared to that of Linux, *Perf*, *Fair*, and Oracle. Finally,

we evaluate Flexible Perf&Fair to check how it can trade fairness for performance, and how this trade-off is particularly interesting in the workloads where the main memory bandwidth consumption is high.

Mixes	Benchmarks	BW_{MM}
Mix 1	*Astar x2, Bzip2 x2, Gcc x2, Gobmk x2, Hmmer x2, H264ref x2, Libquantum x2, Mcf x2, Omnetpp x2, Perlbench x2, Sjeng x2, Xalancbmk x2*	47.1
Mix 2	*Astar, Bwaves, CactusADM , DealII, Gamess x2, Gcc, Gobmk x2, Gromacs, Hmmer x2, H264ref x2, Libquantum, Mcf x2, Milc, Namd x2, Omnetpp, Perlbench, Povray, Sjeng*	53.8
Mix 3	*Bwaves, Bzip2 x2, Gamess x2, Gobmk x2, Gromacs x2, Hmmer x2, H264ref x2, Lbm, Leslie3d, Mcf, Namd x2, Milc, Omnetpp, Perlbench x2, Sjeng, Soplex*	58.1
Mix 4	*Bwaves , CactusADM, DealII , Gamess x2, GemsFDTD, Gobmk x2, Gromacs x2, Hmmer x2, H264ref x2, Lbm, Leslie3d, Libquantum , Mcf, Milc, Namd x2, Perlbench, Sjeng, Soplex*	75.6
Mix 5	*Astar, Bwaves x2, CactusADM , DealII, Gamess x2, Gobmk, Gromacs, Hmmer, H264ref, Lbm, Leslie3d, Libquantum, Mcf, Milc, Namd, Omnetpp, Perlbench, Sjeng, Soplex x2, Xalancbmk, ZeusMP*	89.8
Mix 6	*Bwaves, Bzip2, CactusADM, DealII, Gamess, Gcc, GemsFDTD, Gobmk, Gromacs, Hmmer, H264ref, Lbm, Leslie3d, Libquantum, Mcf, Milc, Namd, Omnetpp, Perlbench, Sjeng, Soplex x2, Xalancbmk, ZeusMP*	90.7
Mix 7	*Astar, Bwaves x2, CactusADM, DealII, Gamess x2, Gcc, GemsFDTD, Gobmk, Gromacs, Hmmer, H264ref, Libquantum, Mcf x2, Milc , Namd x2, Lbm, Omnetpp, Perlbench, Soplex x2*	96.9
Mix 8	*Bwaves x3, Bzip2, CactusADM x2, DealII, Gamess x2, GemsFDTD, Gobmk, Gromacs, Hmmer, H264ref x2, Leslie3d x2, Libquantum, Mcf, Milc, Namd, Omnetpp, Perlbench, Sjeng*	98.0
Mix 9	*Astar, Bwaves x2, CactusADM, Gcc, GemsFDTD, Gobmk x2, Gromacs x2, Lbm, Leslie3d, Libquantum x2, Mcf, Milc, Sjeng x2, Soplex x2, Xalancbmk x2, ZeusMP*	115.4
Mix 10	*Bwaves x3, CactusADM, Gamess x3, Gcc, GemsFDTD, Gromacs, Hmmer, H264ref x3, Lbm, Libquantum, Mcf, Milc x2, Namd, Omnetpp, Sjeng, Soplex, ZeusMP*	115.9
Mix 11	*Astar, Bwaves x2, Gamess x3, GemsFDTD, Gromacs, Hmmer, H264ref x3, Lbm x2, Leslie3d x3, Libquantum x2, Mcf, Milc, Perlbench, Soplex, Xalancbmk*	130.7
Mix 12	*Bwaves x2, CactusADM x2, DealII x2, Gamess x2, GemsFDTD x2, Gromacs x2, Lbm x2, Leslie3d, Milc x2, Namd x2, Soplex x2, ZeusMP x3*	131.1
Mix 13	*Bwaves x2, DealII x2, Gamess, GemsFDTD, Gromacs, H264ref, Lbm x2, Leslie3d x2, Libquantum x2, Mcf x2, Milc, Namd, Omnetpp, Perlbench x3, Povray, Xalancbmk*	131.8

TABLE 6.1: Mix composition and their average main memory bandwidth consumption.

6.6.1 Evaluation of the Progress-Aware Fair Scheduler

6.6.1.1 System Fairness Evaluation

This section evaluates the fairness of the progress-aware Fair scheduler in comparison with Linux. Figure 6.6 depicts the unfairness, in percentage, presented by both *Fair* and Linux across the evaluated mixes. For each mix, the figure presents the average values of twenty executions with both schedulers and 95% confidence intervals.

Fair performs fairer than Linux across all the studied mixes. Unfairness with *Fair* ranges in a relatively narrow interval, from 1.08 to 1.16, with an average by 1.12, using *Fair*. In contrast, Linux unfairness ranges in a much wider interval, from 1.19 to 1.44, with an average by 1.32. This means that under Linux, the slowest process progresses on average by 32% slower than the fastest one. These values seem high and inappropriate in many real systems. Compared to Linux, *Fair* reduces unfairness, on average, by a 3× factor under the studied mixes. In addition, the presented 95% confidence intervals show the steadiness of the unfairness values through multiple executions of each mix.

Taking into account that mixes have been sorted by the number of heavy-sharing processes they include, results suggest that, in general, Linux achieves lower unfairness when contention is low. On the contrary, the unfairness reached by *Fair* tends to be more uniform, regardless of the number of heavy-sharing processes included in a given

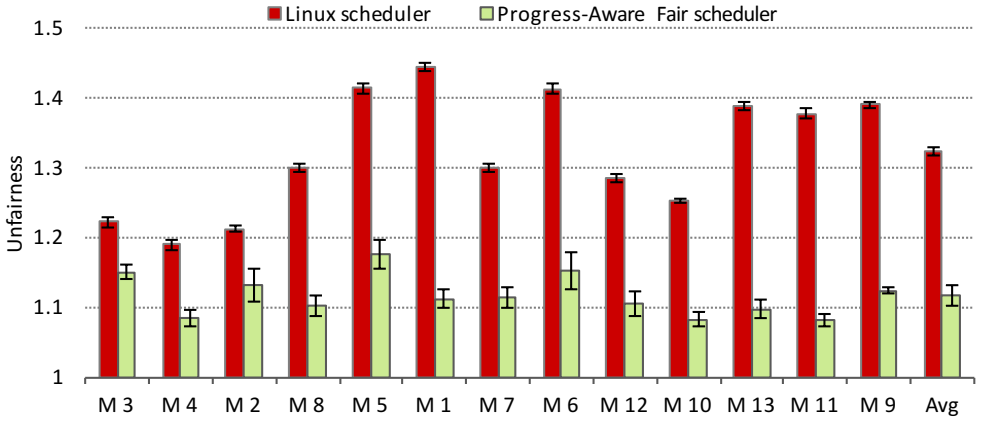


FIGURE 6.6: Unfairness of Linux and the progress-aware Fair scheduling algorithm. Unfairness is a lower-is-better metric.

mix. Therefore, we can conclude that the higher the contention, the higher the fairness benefits that *Fair* provides with respect to Linux.

Figure 6.7 focuses on mix 7 to illustrate how unfairness evolves over the mix execution. Average, maximum, and minimum progress achieved by the processes for Linux and *Fair* are plotted. Remark that in this figure, real progress is plotted since it is calculated and not estimated (only to show how the progress evolves, not to guide the scheduling decisions) for each process as the ratio between committed instructions and the target number of instructions to be committed.

Results depict how Linux unfairness grows with time. For instance, when the first process of the mix finishes at second 280 under Linux, the process with lowest progress has only completed by 40% of its execution. In contrast, *Fair* handles progress more uniformly across all the processes. At second 340, when the first process finishes, the process with lowest progress has committed about 75% of its instructions. Moreover, there is a bigger gap between the maximum and minimum progress with Linux for most of the execution time.

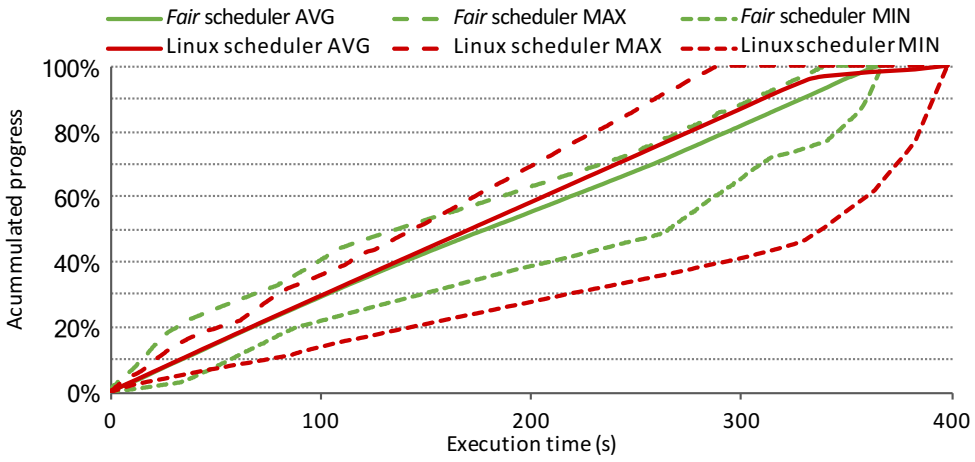


FIGURE 6.7: Dynamic progress of processes in mix M7 with the *Fair* and Linux schedulers.

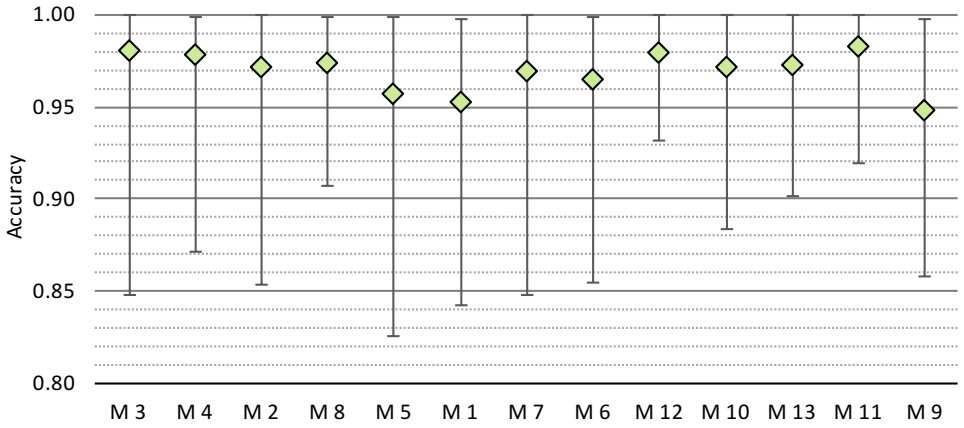


FIGURE 6.8: Average, maximum, and minimum accuracy of the isolated IPC estimations.

6.6.1.2 Accuracy of the Isolated IPC Estimations

Accurate IPC estimates are required to improve fairness. If these estimates are inaccurate, the computed progress will differ from the actual progress, which will yield unfairness to grow.

Figure 6.8 presents the average, maximum, and minimum IPC accuracy across the twenty four processes of each mix. Results show that average IPC accuracy ranges from 95% to 98%, which confirms that the proposed mechanism is able to correctly estimate isolated performance of the processes in schedules. Notice that by 100% accuracy is always achieved by at least one process of each mix. This is due to the fact that some processes present a uniform IPC across its execution time, which helps obtaining accurate estimates. Regarding maximum deviation from the real IPC, accuracy ranges from 82% to 93%.

6.6.2 Evaluation of the Progress-Aware Perf&Fair Scheduler

6.6.2.1 System Fairness Evaluation

Figure 6.9 depicts the unfairness presented by Linux, *Perf*, *Fair*, Perf&Fair, and Oracle across the studied mixes. *Perf* reaches extremely high levels of unfairness (geometric mean by 2.49), which means that the last process finishes its execution by 2.5× later

than the first process. Notice that the rule included in *Perf* to avoid process starvation is not able to keep unfairness at a reasonable level.

Linux is the second one with highest unfairness. Under Linux six mixes present an unfairness around 1.40 and a geometric mean by 1.32. Although much lower than that shown by *Perf*, this level of unfairness still seems high and might be inappropriate in some systems. In contrast, executions with *Fair* do not surpass an unfairness of 1.20, reaching the worst unfairness in mix 5, with a value of 1.17. *Fair* exhibits an average unfairness of 1.12, approximately 2.8× lower than the unfairness shown by Linux.

Perf&Fair shows similar unfairness as that of *Fair*. Unfairness only surpasses 1.20 in four mixes and the geometric mean is by 1.18%. The achieved unfairness makes a big difference with respect to that achieved by Linux, where all the mixes surpass an unfairness of 1.20 (except mix 4). In addition, the geometric mean of the unfairness is reduced from 1.32 to 1.18.

Finally, by using offline traces of the stand-alone performance of the processes, Oracle performs nearly completely fair, reaching an average unfairness by 1.03. The results of Oracle show that despite not being exclusively focused on improving fairness, Perf&Fair can perform nearly completely fair when the progress estimates are completely accurate. In other words, addressing performance additionally to fairness in the process selection does not affect the optimal unfairness that the progress-aware Perf&Fair scheduling algorithm is able to achieve.

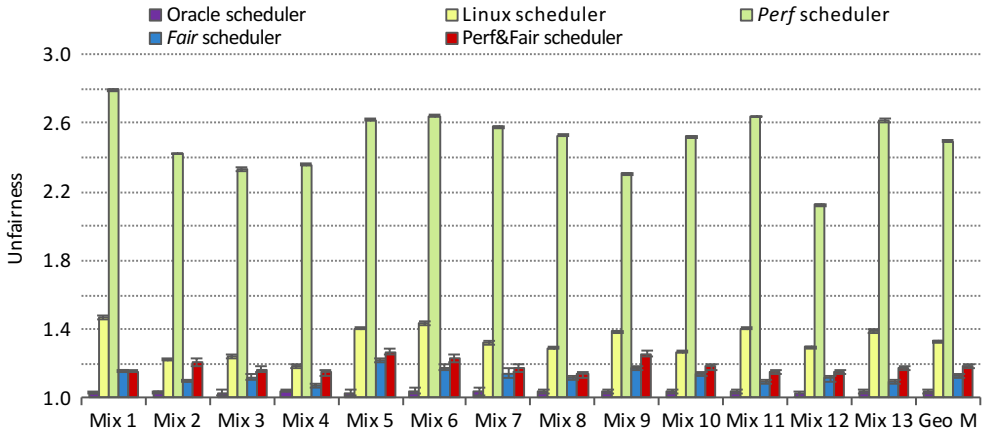


FIGURE 6.9: Unfairness achieved by the studied schedulers, including 95% confidence intervals. Unfairness is a lower-is-better metric.

6.6.2.2 Performance Evaluation

Figure 6.10 presents the speedup of the turnaround time achieved by *Perf*, *Fair*, Perf&Fair, and Oracle over Linux. Considering all the evaluated mixes, Perf&Fair reaches the highest geometric mean of speedup (5.6%), followed by *Perf* (5.0%), and *Fair* (2.2%). Two important observations can also be done at a first glance.

First, it is interesting to observe that, even though the main focus of *Fair* is on system fairness, it improves the turnaround time reached by Linux in all the studied mixes. This improvement is above 3.5% in five mixes, and by 2.2% on average, but shows that fairness can be improved without sacrificing performance. The reason that explains the performance benefits over Linux is that *Fair* allows all the processes to progress at similar rate, and consequently, reduces the fraction of time at the end of the execution, where the processor is running less processes than hardware contexts are available. In addition, it also uses the Dynamic L1 bandwidth-aware process allocation policy to allocate the processes to the cores which, as shown in Section 5.4.1, provides important performance benefits.

Second, it can be observed that, in spite of reaching an unfairness close to *Fair*, Perf&Fair achieves speedups closer to *Perf* than to *Fair*. In fact, it enhances the speedups achieved with the *Fair* in all the mixes.

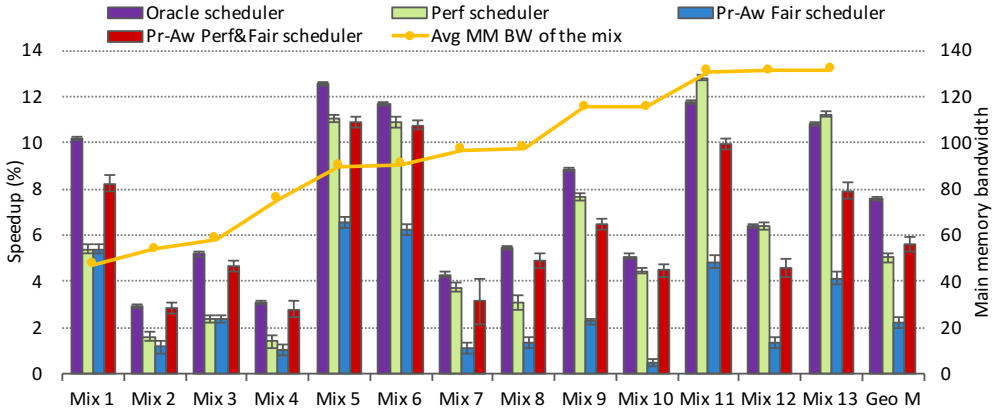


FIGURE 6.10: Speedup of the turnaround time achieved by the studied schedulers over Linux, including 95% confidence intervals. The line shows the average main memory bandwidth of the mixes.

The figure also plots the average BW_{MM} of the mixes (solid line and secondary y-axis), which helps understand the achieved results. Note that the studied mixes are sorted in increasing average BW_{MM} order. Mixes can be divided in three main groups that present different behavior according to their average BW_{MM}.

When the average BW_{MM} is relatively low (below 80 transactions/microsecond), though still significant, bandwidth contention and progress unbalancing similarly affect the turnaround time of the mixes. This is because, at the end of an unbalanced execution, the number of available processes is less than the number of hardware contexts during a significant number of quanta, which penalizes the turnaround time. In addition, since the bandwidth contention is not as high as in other workloads, the benefits of a better main memory bandwidth management decrease and can be canceled due to highly unbalanced executions. In these scenarios, a fairer scheduler, through a better progress balancing, can reduce the number of quanta where there are less available processes than hardware contexts and reduce the turnaround time. In fact, it can be observed that *Fair* reaches speedups that are very similar to that obtained with *Perf*, despite they schedule processes following a completely different strategy. Moreover, Perf&Fair effectively combines both performance and fairness approaches and, by concurrently mitigating bandwidth contention while keeping unfairness under control, it improves the speedups achieved by both *Perf* and *Fair*.

As the average memory bandwidth required by the mixes grows, bandwidth contention becomes a major performance limiter, which translates into larger turnaround times. When the bandwidth falls in between 80 and 120 transactions/microsecond, *Perf* benefits enough from the bandwidth contention to improve performance over *Fair*. In this scenario, Perf&Fair still reaches speedups closely resembling *Perf*, since it is still able to address bandwidth contention while keeping a good progress balancing among the processes.

In the most memory-bounded mixes studied, with above 120 transactions/microsecond, *Perf* greatly improves performance over *Fair*. In this case, Perf&Fair widely improves the results of both *Fair* and Linux, but it is not able to reach speedups as high as those achieved by *Perf* because it also deals with unfairness. Therefore, it cannot devote all the selected processes to maximize performance.

Regarding Oracle, by using offline traces it further enhances the performance of Perf&Fair, despite the benefits are not too large on some workloads (e.g., mix 2 and mix 3). The use of traces also allows Oracle to improve *Perf* in workloads with low and medium main memory bandwidth utilization because it achieves a better progress balancing. In the workloads with he highest main memory bandwidth utilization, Oracle reaches speedups very close to *Perf* but slightly below, since the former scheduler is partially constrained by fairness requirements.

Finally, it should be emphasized that the progress-aware Perf&Fair scheduling algorithm addresses both performance and fairness without requiring from offline traces. Thus, if both metrics are considered together, this scheduler is the one that behaves more satisfactorily. Moreover, the algorithm is flexible enough by design and can be tuned to provide different trade-offs between performance and fairness (see Section 6.6.3).

6.6.2.3 Process Completion in a Mix

To provide insights on the obtained turnaround time and unfairness results, we focus the analysis on mix 9, where the scheduling algorithms present widely different results. Figure 6.11 shows how the number of processes of mix 9 evolve over time when this mix is executed under the studied algorithms. The plot starts at second 150, where no process has yet finished, and shows how the execution of the processes is being completed. The time at which the last process of the 24-task mix finishes its execution determines its turnaround time. *Perf* shows the shortest turnaround time, closely followed by the Perf&Fair, then *Fair*, and finally Linux, which shows the largest turnaround time. On the other hand, unfairness is determined as the ratio between the time at which the first and the last processes of the mix complete their execution. As observed, *Fair* achieves the lowest unfairness, followed by Perf&Fair, Linux, and *Perf*.

The figure also illustrates the importance of fairness or progress balancing on the turnaround time of the mix. For instance, Linux is the scheduler that earliest completes the execution of the first twelve processes, but the last that completes the execution of the whole mix. This means that in this execution Linux leads the system to a low loaded state (i.e., with less applications than hardware contexts) earlier than the other scheduling algorithms (by second 325). However, this behavior is at the cost of system fairness,

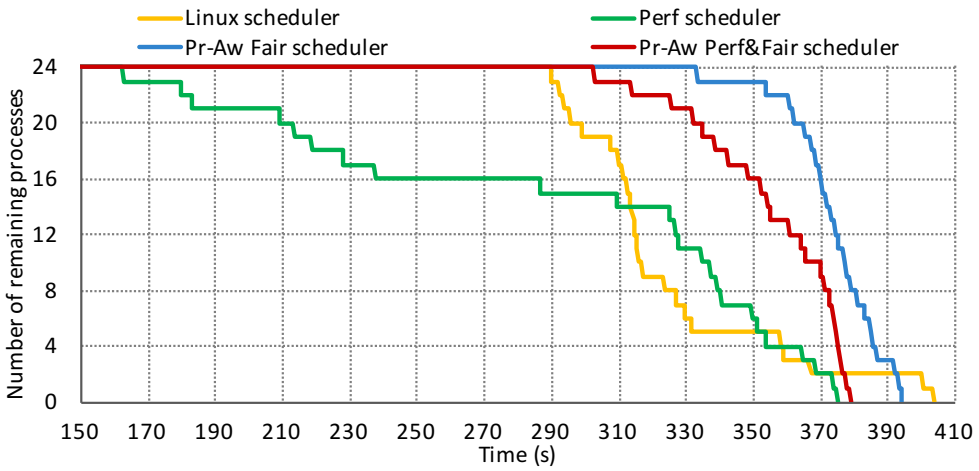


FIGURE 6.11: Number of remaining processes along the execution of mix 9 with the studied schedulers.

since Linux takes longer time to complete the last five processes of the workload, even if at this point each process is already running alone on a different core. Another observation is that *Perf* completes the first process as soon as second 162, which yields the system to the highest unfairness. Finally, regarding the Perf&Fair curve it is interesting to observe how, despite having a turnaround time close to *Perf*, it presents a process completion curve resembling that of *Fair*.

6.6.3 Evaluation of the Flexible Progress-Aware Perf&Fair Scheduler

Figure 6.12 presents the unfairness achieved by *Perf*, the original Perf&Fair scheduler, and Flexible Perf&Fair varying the relative length of the performance quanta bursts. We use the notation `X:Y` to refer to a Flexible Perf&Fair that schedules X-hundred cycles under the Perf&Fair scheduling algorithm, followed by Y-hundred cycles scheduled under the *Perf* algorithm.

As expected, the original Perf&Fair achieves the lowest unfairness across all the evaluated mixes, with a geometric mean by 1.18%. On the contrary, *Perf* shows the highest unfairness (geometric mean by 2.49). By increasing the length of the performance quanta bursts, Flexible Perf&Fair varies the achieved unfairness between the original Perf&Fair and *Perf*. For experimental purposes, we keep constant the number of quanta bursts (i.e., 100) devoted to the original Perf&Fair and vary the number of quanta devoted to performance (50, 150, and 300). With 50 performance quanta bursts (ratio `1:0.5`),

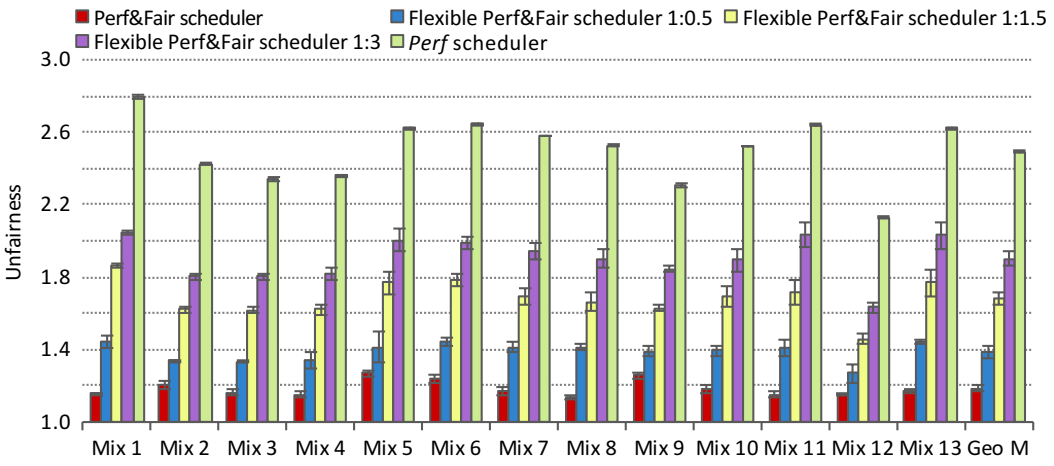


FIGURE 6.12: Unfairness achieved by Perf, Perf&Fair, and Flexible Perf&Fair with `1:0.5`, `1:1.5` and `1:3` ratios. Unfairness is a lower-is-better metric.

the geometric mean of the unfairness is by 1.37 and grows up to 1.68 and 1.87 when the length of the performance quanta bursts is increased to 150 (`1:1.5`) and 300 (`1:3`) performance quanta, respectively.

Figure 6.13 presents the speedup of the turnaround time achieved by *Perf*, the original Perf&Fair, and Flexible Perf&Fair with 1:0.5, 1:1.5 and 1:3 ratios. The figure also plots the average main memory bandwidth consumption (BW_{MM}) of the mixes (solid line and secondary y-axis). The studied mixes are sorted in increasing average BW_{MM} order. As explained above, including performance quanta bursts should improve performance when the average BW_{MM} is high, because in these mixes *Perf* performs better than the original Perf&Fair.

If we observe the mixes with higher main memory bandwidth utilization (mixes 11, 12, and 13), we can see how the inclusion of longer performance quanta bursts allows Flexible Perf&Fair to improve its performance and approach the performance achieved by *Perf*. When the average bandwidth utilization falls in between 80 and 120 transactions/microsecond (mixes 5 to 10), the original Perf&Fair already performs similarly to *Perf*. In this scenario, the inclusion of performance quanta bursts slightly affect the achieved performance. Nonetheless the trend that by using longer performance quanta bursts the achieved performance approaches to that of *Perf* is preserved. Finally, in the mixes with lower bandwidth consumption, the original Perf&Fair performs better than *Perf* since progress balancing is also an important factor to improve performance. In these mixes,

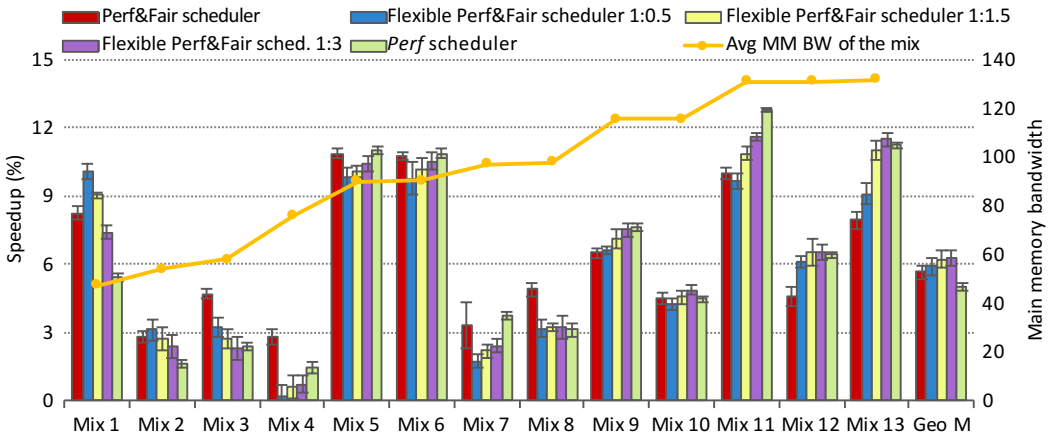


FIGURE 6.13: Speedup of the turnaround time chieved by Perf&Fair, *Perf*, and Flexible Perf&Fair, with `1:0.5`, `1:1.5` and `1:3` ratios, over Linux. The line shows the average main memory bandwidth of the mixes.

the inclusion of performance quanta bursts reduces the performance achieved with Flexible Perf&Fair, approaching to that of *Perf*. The only exception is mix 1, where the best performance is reached when performance bursts of 50 quanta are considered.

To sum up, by including (longer) performance quanta bursts, where estimation quanta are also avoided, the behavior of Flexible Perf&Fair gradually resembles that of *Perf*. This approach offers the users the chance of trading fairness for performance. Experimental results show that this trade-off is only beneficial when the workloads present high memory bandwidth consumption since this is the only scenario where *Perf* performs better than the progress-aware Perf&Fair scheduling algorithm.

6.7 Summary

Fairness-aware scheduling is gaining importance in multicore systems to guarantee correct management of process priorities, quality of service, and worst case execution times, among others. A simple approach, such as allocating the same execution time and resources to the running processes in a multiprogram workload does no longer provide fairness because of the unpredictable interference on the shared resources of current systems.

This chapter has presented two progress-aware scheduling algorithms for SMT multicores. The main challenge they present lies on dynamically and accurately estimate,

at run-time, the progress of each process over isolated execution. To accomplish it, the proposed algorithms periodically create low-contention schedules where the isolated performance of the processes can be estimated. First, we propose the progress-aware Fair scheduling algorithm, whose main target lies on maximizing system fairness. This goal is achieved by prioritizing the processes with lower accumulated progress. Second, we propose the progress-aware Perf&Fair scheduling algorithm, which simultaneously addresses performance and fairness. This is done by scheduling the processes to balance the bandwidth consumption among the available resources and over the execution time of the workload, but preventing unfairness from growing giving priority to the processes with lower accumulated progress. Furthermore, Perf&Fair can be tuned to provide different trade-offs between performance and fairness.

Experimental results obtained in an Intel Xeon E5645 system with six dual-threaded SMT cores show that both schedulers accomplish their goals. *Fair* reduces unfairness by a 3× factor with respect to Linux, while still achieving slight performance benefits over it (by 2.2%). Perf&Fair also meets its two-fold goal. Regarding performance, it achieves speedups of the turnaround time of the mixes that slightly enhance the performance of the SMT bandwidth-aware (performance-oriented) scheduler proposed in Section 5.2, with the only exception of workload with extreme main memory bandwidth requirements. Across the evaluated workloads, the speedup of Perf&Fair and this performance-oriented scheduler over Linux are on average by 5.6% and 5.0%, respectively. The key is that such a level of performance is achieved while unfairness is reduced from a geometric mean of 2.49 and 1.33 for the performance-oriented and Linux schedulers, respectively, to only 1.18 in the proposed progress-aware Perf&Fair scheduling algorithm. Finally, Flexible Perf&Fair can be used to achieve different trade-offs of performance and fairness, increasing the performance of Perf&Fair when the bandwidth contention is very high.

The work discussed in this chapter has been published in [67] and [68].

Chapter 7

Symbiotic Job Scheduling on the IBM POWER8

The number of hardware contexts quickly grows generation after generation in the prevalent architecture for high-end processors, which is a multicore processor consisting of SMT cores. The scheduler is a critical component of these systems, since there is often a combinatorial amount of different ways to schedule multiple applications, each one with different performance due to interference among applications.

This chapter addresses this scheduling problem and proposes a symbiotic job scheduler. This scheduler leverages the existing cycle accounting mechanism of modern processors to build a model that estimates the interference between applications. The use of this model allows the scheduler to evaluate the possible combinations of applications and select the optimal one.

This chapter is organized as follows. First, we present the SMT-interference model that estimates job symbiosis theoretically, and discuss their construction on a real system. Then, we present the symbiotic job scheduler, which uses the proposed SMT-interference model. Finally, we describe the evaluation setup and present the experimental evaluation results.

7.1 Predicting Job Symbiosis

The symbiotic job scheduler presented in this chapter for a multicore processor consisting of SMT cores is based on a model that estimates job symbiosis. The model predicts, for any combination of applications, how much slowdown each of the applications would experience if they were co-run on the same SMT core. It is fast, which enables the symbiotic scheduler to explore all possible combinations (or at least a very large subset of them), and only requires inputs that are readily obtainable using performance counters.

Our model implementation tackles the IBM POWER8 system because it implements an extensive performance counter architecture, including a built-in mechanism to measure CPI stacks, on which the proposed model is based. Besides, its high core count also makes the scheduling problem more challenging. Nonetheless, the proposed interference model could be adapted to other CMP architectures with SMT cores that provide a similar cycle accounting mechanism such as an Intel Xeon server [69].

7.1.1 SMT Interference Model

The model we present in this section is based on the model proposed by Eyerman and Eeckhout [7], which leverages CPI stacks to predict job symbiosis. A CPI stack (or breakdown) divides the execution cycles of an application on a processor into various components, quantifying how much time is spent or lost due to different events, see Figure 7.1 on the left. The base component reflects the ideal CPI in the absence of miss events and resource stalls. The other CPI components account for the *lost* cycles, where the processor is not able to commit instructions due to different resource stalls and miss events.

The SMT symbiosis model uses the CPI stack of an application when executed in single-threaded (ST) mode, and then predicts the slowdown by estimating the increase of the components due to interference, see Figure 7.1 on the right. Eyerman and Eeckhout [7] estimate interference by interpreting normalized CPI components as probabilities and calculating the probabilities of events that cause interference. For example, if an application spends half of its cycles fetching instructions, and another application one third of its execution time, there is a 1/6 probability that they want to fetch instructions at

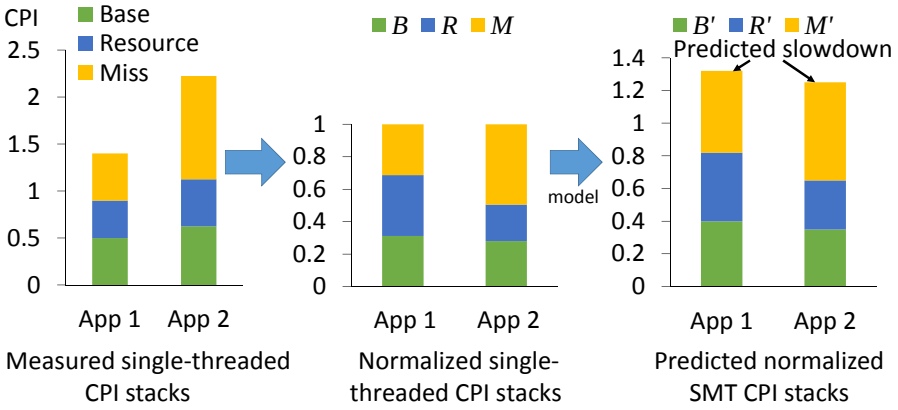


FIGURE 7.1: Overview of the model: first, measured CPI stacks are normalized to obtain probabilities; then, the model predicts the increase of the components and the resulting slowdown (1.32 for App 1 and 1.25 for App 2).

the same time, which incurs a delay because the fetch unit is shared. However, Eyerman and Eeckhout use novel hardware support [46] to measure the ST CPI stack components during multi-threaded execution, which is not available in current processors.

Interestingly, the IBM POWER8 has a built-in cycle accounting mechanism, which generates CPI stacks both in ST and SMT mode. However, this accounting mechanism is different from the cycle accounting mechanisms proposed by Eyerman et al. for SMT cores [46], which means that the model proposed in [7] cannot be used readily. Some of the components relate to each other to some extent (e.g., the number of cycles instructions are dispatched in [46] versus the number of cycles instructions are committed for the IBM POWER8), but provide different values. Other counters are not considered a penalty component in one accounting mechanism, while they are accounted for in the other mechanism, and vice versa. For example, following [46], a long-latency instruction only has a penalty if it is at the head of the reorder buffer (ROB) and the ROB gets completely filled (halting dispatch), while for the IBM POWER8 accounting mechanism, the penalty starts from the moment that the long-latency instruction inhibits committing instructions, which could be long before the ROB is full. On the other hand, the entire miss latency of an instruction cache miss is accounted as a penalty in [46], while for the IBM POWER8 accounting mechanism, the penalty is only accounted from the moment the ROB is completely drained (which means that the penalty could be zero if the miss latency is short and the ROB is almost full). Furthermore, some POWER8 CPI

components are not well documented, which makes it difficult to reason about which events they actually measure.

Because of these differences, we develop a new model for estimating the slowdown caused by co-running threads on an SMT core. The model uses regression, which is more empirical than the purely analytical model by Eyerman and Eeckhout [7], but its basic assumption is similar: we normalize the CPI stack by dividing each component by the total CPI, and interpret each component as a probability. We then calculate the probabilities that interfering events occur at the same time, which causes some delay that is added to the CPI stack as interference. The components are divided into three categories: the base component, the resource stall components, and the miss components. The model for each category is discussed in the following paragraphs. For now, let us assume that we have the ST CPI stacks at our disposal, measured offline using a single-threaded execution on the IBM POWER8 machine. This assumption will no longer be necessary in Section 7.1.3. The stack is normalized by dividing each component by the total CPI, see Figure 7.1. We denote B the normalized base component, R_i the component for stalls on resource i, and M_j the component for stalls due to miss event j (e.g., instruction cache miss, data cache miss, or branch misprediction). We seek to find the CPI stack when this application is co-run with other applications in SMT mode, for which the components are denoted with a prime (B', R'_i, M'_j).

7.1.1.1 Base Component

The base component in the IBM POWER8 cycle component stack is the number of cycles (or fraction of time after normalization) where instructions are committed. It reflects the fraction of time the core is not halted due to resource stalls or miss events. During SMT execution, the dispatch, execute, and commit bandwidth are shared between threads, meaning that even without miss events and resource stalls, threads interfere with each other and cause other threads to wait.

We find that the base component in the CPI stack increases when applications are executed in SMT mode compared to ST mode. This is because multiple threads can now commit instructions in the same cycle, so each thread commits fewer instructions per cycle, meaning that the number of cycles that a thread commits instructions increases.

The magnitude of this increase depends on the characteristics of the other threads. If the other threads are having a miss or resource stall, then the current thread can use the full commit bandwidth. If the other threads can also commit instructions, then there is interference in the base component. So, the increase in the base component of a thread depends on the base component fractions of the other threads: if the base components of the other threads are low, there is less chance that there is interference in this component, and vice versa.

We model the interference in the base component using Equation 7.1. For a given thread j, B_j represents its base component when running in ST mode (the ST base component), while B'_j identifies the SMT base component of the same thread.

$$B'_j = \alpha_B + \beta_B B_j + \gamma_B \sum_{k \neq j} B_k + \delta_B B_j \sum_{k \neq j} B_k \tag{7.1}$$

The parameters α_B through δ_B are determined using regression, see Section 7.1.2. α_B reflects a potential constant increase in the base component in SMT mode versus ST mode, e.g., through an extra pipeline stage. Because we do not know if such a penalty exists, we let the regression model find this out. The β_B term reflects the fact that the original ST base component of a thread remains in SMT execution. It would be intuitive to set β_B to one (i.e., the original ST component does not change), but the next terms, which model the interference, could already cover part of the original component, and this parameter then covers the remaining part. It can also occur that there is a constant relative increase in the base component, independently of the other applications. In that case, β_B is larger than 1. γ_B is the impact of the sum of the base components of the other threads. δ_B specifically models extra interactions that might occur when the current thread (thread j) and the other threads have big base components, similar to the probabilistic model of Eyerman et al. [7] (a multiplication of probabilities). Although not all parameters have a clear meaning, we keep the regression model fairly general to be able to accurately model all possible interactions.

7.1.1.2 Resource Stall Components

A resource stall causes the core to halt because a core resource (e.g., functional unit, issue queue, or load/store queue) is exhausted or busy. In the IBM POWER8 cycle accounting, a resource stall is counted if a thread cannot commit an instruction because it is still executing or waiting to execute on a core resource (i.e., not due to a miss event). By far, the largest component we see in this category is a stall on the floating-point unit, i.e., a floating-point instruction is still executing when it becomes the oldest instruction in the ROB. This can have multiple causes: the latency of the floating-point unit is relatively large, there are a limited number of floating-point units, or some of them are not pipelined. We expect a program that executes many floating-point instructions to present more stalls on the floating-point unit, which is confirmed by our experiments. Along the same line, we expect that when co-running multiple applications with a large floating-point unit stall component, the pressure on floating-point units will increase even more. Our experiments show that in this case, the floating-point stall component per application indeed increases. Therefore, we propose the following model to estimate the resource stall component in SMT mode ($R_{j,i}$ represents the ST stall component on resource i for thread j):

$$R'_{j,i} = \alpha_{Ri} + \beta_{Ri} R_{j,i} + \gamma_{Ri} \sum_{k \neq j} R_{k,i} + \delta_{Ri} R_{j,i} \sum_{k \neq j} R_{k,i} \tag{7.2}$$

Similar to the base component model, α indicates a constant offset that is added due to SMT execution (e.g., extra latency). β indicates the fraction of the single-threaded component that remains in SMT mode, while the term with γ models the fact that resource stalls of the other applications can cause resource stalls in the current application, even if the current application originally had none. The last term models the interaction: if the current application already has resource stalls, and one or more of the other applications too, there will be more contention and more stalls.

7.1.1.3 Miss Components

Miss components are caused by instruction and data cache misses at all levels, as well as by branch mispredictions. In contrast to resource stall components, a miss event

of a thread does not directly cause a stall for the other threads. For example, if one thread has an instruction cache miss or a branch misprediction, the other threads can still fetch instructions. Similarly, on a data cache miss for one thread, the other threads can continue executing instructions and access the data cache. One exception is that a long-latency load miss (e.g., a last-level cache miss) can fill up the ROB with instructions of the thread causing the miss, leaving fewer or no ROB entries for the other threads. As pointed out by Tullsen et al. [70], this is a situation that should be avoided, and we suspect that current SMT implementations (including the IBM POWER8) have mechanisms to prevent this from happening.

However, misses can interfere with each other in the branch predictor or cache itself. For example, a branch predictor entry that was updated by one thread can be overwritten by another thread's branch behavior, which can lead to higher or lower branch miss rates. Similarly, a cache element belonging to one thread can be evicted by another thread (negative interference) or a thread can put data in the cache that is later used by another thread if both share data (positive interference). Furthermore, cache misses of different threads can also contend in the lower cache levels and the memory system, causing longer miss latencies. Because we only evaluate multiprogram workloads consisting of single-threaded applications, which do not share data, we see no positive interference in the caches.

To model this interference, we propose a model similar to that of the previous two components:

$$M'_{j,i} = \alpha_{Mi} + \beta_{Mi} M_{j,i} + \gamma_{Mi} \sum_{k \neq j} M_{k,i} + \delta_{Mi} M_{j,i} \sum_{k \neq j} M_{k,i} \tag{7.3}$$

Although the model looks exactly the same, the underlying reasoning is slightly different. α again relates to fixed SMT effects (e.g., cache latency increase). The β term is the original miss component of that thread, while the γ term indicates that an application can get extra misses due to interference caused by misses of the other applications. We also add a δ interaction term: an application that already has a lot of misses will be more sensitive to extra interference misses and contention in the memory subsystem if it is combined with other applications that also have a lot of misses.

7.1.2 Model Construction and Slowdown Estimation

The model parameters are determined by linear regression based on experimental training data. This is a less rigorous approach than the model presented in [7], which is built almost completely analytically, but as explained before, this is due to the fact that the cycle accounting mechanism is different and partially unknown.

SMT processors share most of their internal resources, which are fully available for an application running alone in ST mode. This resource sharing can be implemented either by applying dynamic sharing or partitioning techniques. For instance, resources such as the ROB or the arithmetic units, among others, are dynamically shared in the IBM POWER8 while other internal resources, like instruction buffers, register renaming tables, or load/store buffers are partitioned [1]. If the resources are shared, interference among the threads can rise. On the contrary, if the resources are partitioned there is no interference among threads, but the performance gap between the ST and SMT modes can grow since a given thread cannot use the resources allocated to another thread. In addition, other characteristics such as instruction dispatch restrictions, prefetching, or branch prediction capabilities also differ among ST and the different SMT modes, further increasing the gap between ST and SMT performance.

Taking the previous rationale into account, and considering that the goal of the model is to estimate the interference between applications, we determine the SMT CPI stacks of the applications running in a schedule from their CPI stacks running in the same SMT mode and including the statically partitioned structures, but without interference on the shared resources caused by other co-running applications. These CPI stacks will be referred to as *throttled*-ST (tST) CPI stacks and, from a practical point of view, replace the ST CPI stacks used in Section 7.1.1 to discuss the interference model. Thus, with the methodology presented in this chapter, the SMT CPI stacks on a 2-application schedule (SMT2 CPI stacks) will be estimated from the tST CPI stacks of the applications in SMT2 mode. Similarly, SMT4 CPI stacks would be predicted from tST CPI stacks of applications executed in SMT4 mode.

The SMT modes are automatically set by the IBM POWER8 depending on the number of threads running on a core and therefore, the tST CPI stacks cannot be obtained when the applications are executed alone. To solve this problem, we implement a *nop-microbenchmark*, which is constantly performing *nop* operations. The nop-microbenchmark is intended to force the processor to work in the desired SMT mode while introducing negligible interference at the shared core resources. Thus, it is designed with the opposite goal of other microbenchmarks [48, 49], which are used to introduce contention in the shared resources. Note that obtaining tST CPI stacks is only required to determine the model parameter values, but does not affect how the model and the scheduler work during normal execution.

To train the model, we first run all benchmarks alone in each SMT mode (see Section 7.3 for the benchmarks we evaluate), and collect the tST CPI stacks every scheduler quantum (100 ms). To run an application in SMT2 or SMT4 modes, it is scheduled on a core with one or three instances of the *nop-microbenchmark*, respectively. We keep track of the instruction count per quantum to determine which part of the program is being executed in each quantum (we evaluate single-threaded programs with a bounded total instruction count). We also normalize each CPI stack to its total CPI.

Next, we execute all possible 2-benchmark mixes and a large and representative set of 4-benchmark mixes on a single core[1]. Notice that the number of possible 4-benchmark mixes exponentially grows with the number of benchmarks and evaluating all of them would take too much time. During these executions, we also collect per-thread CPI stacks and instruction counts for each quantum. Next, we normalize each SMT CPI stack to the previously collected tST CPI of the same instructions. We normalize to the tST CPI because we want to estimate the slowdown each application gets versus its execution alone in the same SMT mode, which equals the SMT CPI divided by the tST CPI (see the last graph in Figure 7.1). This is also in line with the methodology in [7]. Because the performance of an application differs between tST and SMT executions due to co-runner interference, and the quanta are fixed time periods, the instruction counts do not exactly match between both executions. To solve this problem, we interpolate the tST CPI stacks between two quanta to ensure that tST and SMT CPI stacks are

[1]The interference model used by the symbiotic scheduler are built with the data collected from all benchmarks. However, leave-p-out cross-validation is performed to evaluate their accuracy. See Section 7.4.1 for further details.

covering approximately the same instructions. Once the model has been constructed, we can use it to estimate the SMT CPI stacks from the tST CPI stacks for any combination of applications. We first calculate each of the individual components using Equations 7.1 to 7.3, and then add all of the components. The resulting number will be larger than one, and indicates the slowdown the application encounters when executed in that combination (see Figure 7.1 on the right). This information is used to select combinations with minimal slowdown (see Section 7.2).

7.1.3 Obtaining tST CPI stacks in SMT mode

Up to now, we assumed that we have the tST CPI stacks available. This is not a practical assumption, since it would require to keep all of the per-quantum tST CPI stacks in a profile. This is a large overhead for a realistic scheduler. An alternative approach is to periodically get the tST CPI stacks (sampling), and assume that the measured CPI stack is representative for the next quanta. Because programs exhibit varying phase behavior, it requires to resample at periodic intervals to capture this phase behavior. Sampling tST execution incurs performance overhead, because it has to temporarily stop other threads to obtain the tST CPI stacks, and it can also be inaccurate if the program exhibits fine-grained phase behavior. Moreover, this approach is not easily applicable since the model uses the isolated performance in the SMT modes, which will require to execute the nop-microbenchmark during sampling periods.

Instead, we propose to estimate the tST CPI stacks during SMT execution, similar to the cycle accounting technique described in [46]. However, the technique in [46] requires hardware support that is not available in current processors. To obtain the tST CPI stacks during SMT execution on a current processor, we propose to measure the SMT CPI stacks and 'invert' the model: estimating tST CPI stacks from SMT CPI stacks. Once these estimations are obtained, the scheduler applies the 'forward' model (i.e., the model described in the previous sections) on the estimated tST CPI stacks per application to estimate the potential slowdown for thread-to-core mappings that are different from the current one. By continuously rebuilding the tST CPI stacks from the current SMT CPI stacks, the scheduler can detect phase changes and adapt its schedule to improve performance. Note that, the proposed approach does not require any sampling phase and thus, it does not incur any performance overhead.

Inverting the model is not as trivial as it sounds. The 'forward' model calculates the normalized SMT CPI stacks from the normalized tST CPI stacks. As stated in Section 7.1.1, both stacks are normalized to the *single-threaded* CPI. However, without profiling, the tST CPI is unknown in SMT mode, which means that the SMT components normalized to the tST CPI (B', R'_i and M'_j in Equations 7.1 to 7.3) cannot be calculated. Nevertheless, we can calculate the SMT CPI components normalized to the *multi-threaded* CPI (see Figure 7.2b). By definition, the sum of these components equals one, which means that they are inaccurate estimates for the SMT components normalized to the tST CPI, because the latter add to the actual slowdown, which is higher than one (see the last graph in Figure 7.1).

Because we do not know the tST CPI, the model cannot be inverted in a mathematically rigorous way, which means we have to use an approximate approach. We observe that the SMT components normalized to SMT CPI are a rough estimate for the tST components

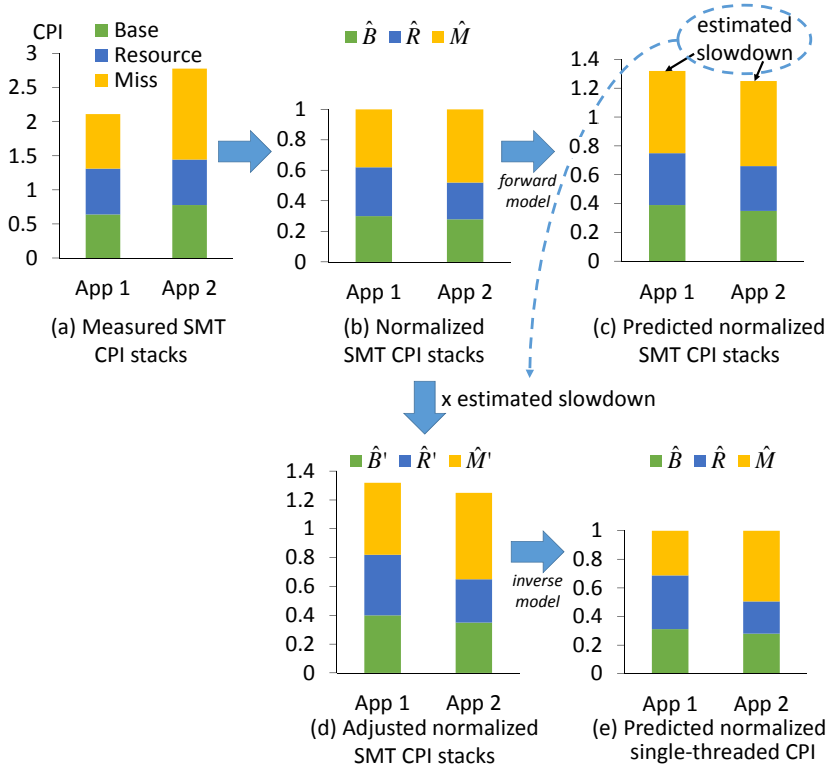


FIGURE 7.2: Estimating the single-threaded CPI stacks from the SMT CPI stacks. First, SMT CPI stacks (a) are normalized to the SMT CPI (b); next, the forward model is applied to get an estimate of the slowdown due to interference (c); then the SMT CPI stacks are adjusted using the estimated slowdown to obtain more accurate normalized SMT CPI stacks (d); lastly, the inverse model is applied to obtain the normalized single-threaded CPI stacks (e).

normalized to the tST CPI (B, R_i and M_j), for two reasons. First, both normalized CPI stacks add to one. Second, if all the components experience the same relative increase between the tST and SMT executions (e.g., all components are multiplied by 1.3), then the SMT CPI stack normalized to the SMT CPI would be exactly the same as the tST stack normalized to the tST CPI. Obviously, this is usually not the case, but intuitively, if a tST stack has a relatively large component, it is expected that this component will also be large in the SMT stack, so the relative fraction should be similar.

Therefore, a first-order estimation of the tST CPI stack is to take the SMT CPI stack normalized to the SMT CPI (see Figure 7.2b). The resulting tST CPI stack component estimations are however not accurate enough to be used in the scheduler. Nonetheless, by applying the 'forward' model to these first-order single-threaded CPI stack estimations (see Figure 7.2c), a good initial estimation of the slowdown each application has experienced in SMT mode can be provided. This slowdown estimation can be used to renormalize the *measured* SMT CPI stacks by multiplying them with the estimated slowdown (see Figure 7.2d). This gives new, more accurate estimates for the SMT CPI stacks normalized to the tST CPI (B', R'_i and M'_j).

Next, we mathematically invert the model to obtain new estimates for the tST CPI stacks (see Figure 7.2e). The mathematical inversion involves solving a set of equations. For two threads, we have two equations per component (one for each of the two threads), which both contain the two unknown single-threaded components, so a set of two equations with two unknowns must be solved (similar to four threads: four equations with four unknowns). Due to the multiplication of the single-threaded components in the δ term, the solution for two threads is in the form of the solution of a quadratic equation. For four threads, the inversion cannot be done analytically. We therefore decide to set δ to zero and train the model omitting this component of the model equation, which simplifies the formulas. This does not lead to a significant decrease in accuracy. The sum of the resulting estimates for the single-threaded normalized components (B, R_i and M_j) usually does not exactly equal one. Thus, the estimation can be further improved by renormalizing them to their sum.

7.2 SMT Interference-Aware Scheduler

In this section, we describe the implementation of the symbiotic scheduler that uses the proposed interference model to improve the processor throughput. The goal of the symbiotic scheduler is to divide n applications over c (homogeneous) cores, being $n > c$, in order to optimize the overall throughput. Each core supports at least $\lceil \frac{n}{c} \rceil$ thread contexts using SMT. Note that we do not consider the problem of selecting n applications out of a larger set of runnable applications, we assume that this selection has already been made or that the number of runnable applications is smaller than or equal to the number of available thread contexts. The scheduler implementation involves several steps which we discuss in the next sections.

7.2.1 Reduction of the Cycle Stack Components

The most detailed cycle stack that the performance monitoring unit (PMU) of the IBM POWER8 can provide involves the measurement of 45 events. However, the PMU only implements six thread-level counters. Four of these counters are programmable, and the remaining two measure the number of completed instructions and non-idle cycles. Furthermore, most of the events have structural conflicts with other events and cannot be measured together. As a result, 19 time slices or quanta are required to obtain the full cycle stack. Requiring 19 time slices to update the full cycle stack means that, at the time the last components are updated, other components contain old data (from up to 18 quanta ago). Since the scheduler uses 100 milliseconds quanta, this issue would make the symbiotic scheduler less reactive to phase changes in the best scenario, and completely meaningless in the worst case.

An interesting characteristic of the CPI breakdown model is that is built up hierarchically, starting from a top level consisting of five components and multiple lower levels where each component is split up into several more detailed components [71]. For example, the completion stalls event of the first level, which measures the completion stalls caused by different resources, is split in several sub-events in the second level, which measure, among others, the completion stalls due to the fixed-point unit, the vector-scalar unit, and the load-store unit. To improve the responsiveness of the scheduler and to reduce the complexity of calculating the model, we measure only the events that form

Counter	Explanation
PM_GRP_CMPL	Cycles where this thread committed instructions. *This is the base component in our model.*
PM_CMPLU_STALL	Cycles where a thread could not commit instructions because they were not finished. *This counter includes functional unit stalls, as well as data cache misses.*
PM_GCT_NOSLOT_CYC	Cycles where there are no instructions in the ROB for this thread, due to instruction cache misses or branch mispredictions.
PM_CMPLU_STALL_THRD	Following a completion stall (PM_CMPLU_STALL), the thread could not commit instructions because the commit port was being used by another thread. *This is a commit port resource stall.*
PM_NTCG_ALL_FIN	Cycles in which all instructions in the group have finished but completion is still pending. *The events behind this counter are not clear in [71], but it is non-negligible for some applications.*

TABLE 7.1: Overview of the measured IBM POWER8 performance counters to collect cycle stacks.

the top level of the cycle breakdown model. This reduces the number of time slices to measure the model inputs to only two. The measured events are indicated in Table 7.1. Note that the PM_CMPLU_STALL covers both resource stalls and some of the miss events. Because the underlying model for both is essentially the same, this is not a problem. Although the accuracy of the model could be improved by splitting up this component, our scheduler showed worse performance because of having to predict job symbiosis with old data for many of the components.

7.2.2 Selection of the Optimal Schedule

The scheduler uses the measured CPI stacks and the model to schedule the applications among cores. To simplify the scheduling decision, we make the following assumptions:

- The interference in the resources shared by all cores (shared LLC, memory controllers, memory banks, etc.) is mainly determined by the characteristics of all applications running on the processor, and not so much by the way these applications are scheduled onto the cores. This observation is also made by Radojković et al. [72]. As a result, with a fixed set of runnable applications, scheduling has no impact on the inter-core interference and the scheduler should not take them into account.

- The IBM POWER8 cores implement an issue queue divided in two symmetric halves. Some of the execution pipelines, such as the fixed-point, floating-point, vector, load, and load-store pipelines are similarly split into two sets. In the SMT

modes, the threads can only issue instructions to a single half of the issue queue [1]. Thus, two 4-application schedules such as ABCD and ACBD may reach different performance since in the first case, application A is sharing some of the execution pipelines with application B, and in the second case it shares these pipelines with application C. We have experimentally checked that the performance difference of these schedules is on average 0.9% across 50 application combinations. Therefore, we assume that they perform equally and they do not need to be evaluated individually.

Even with these simplifications, the number of possible schedules is usually too large to perform an exhaustive search. The number of schedules considering n applications and c cores equals $\frac{n!}{c!\left(\frac{n}{c}!\right)^c}$ (assuming n is a multiple of c). For scheduling 16 applications on 8 cores in SMT2 mode, there are already more than 2 million possible schedules. To efficiently cope with the large number of possible schedules, we use a technique proposed by Jiang et al. [73]. The technique models the scheduling problem for two applications per core as a minimum-weight perfect matching problem, which can be solved in polynomial time using the blossom algorithm [74].

When scheduling for higher SMT modes (e.g., SMT4), the number of possible combinations becomes prohibitive for even a relatively low number of cores. For example, to schedule 20 applications on 5 cores in SMT4, there are more than 2 billion possible combinations. In addition, the scheduling problem for more than two applications per core cannot be modeled as a minimum-weight perfect matching problem. In fact, Jiang et al. also prove that this problem becomes NP-complete as soon as $\frac{n}{c} > 2$.

To address this issue, we use the hierarchical technique also proposed by Jiang et al. [73]. Using this approach, the applications are first divided into pairs, and these pairs are then combined to quadruples, using the blossom algorithm at both levels. Next, a local optimization step rearranges applications in each pair of quadruples to obtain better performance. While this technique is not guaranteed to give the optimal solution, Jiang et al. [73] show it to perform well in a setup where applications need to be scheduled in a clustered architecture, where each cluster shares a cache.

In summary, the scheduler does the following steps at the beginning of each time slice to schedule the application in SMT4 mode. To schedule applications in SMT2 mode, steps 4 and 5 are ommited.

1. Collect the SMT CPI stacks for all applications over the previous time slice.
2. Use the inverted model to get an estimate of the tST CPI stacks for each application.
3. Use the SMT2 forward model to predict the performance of each 2-application combination, and use the blossom algorithm to find the optimal schedule.
4. Use the SMT4 forward model to predict the performance of each 4-application combination, combining the pairs of applications selected in the previous step, and use the blossom algorithm to find a close to optimal schedule.
5. Apply the local optimization to each pair of 4-application combinations selected in the previous step to further improve the selected schedule.
6. Run the best schedule for the next time slice.

7.2.3 Scheduler Implementation

Normally, workload execution and scheduling work (evaluating the performance of the possible schedules and selecting the best one) are performed in a serial way. In other words, the applications do not run while the process selection is being performed. However, depending on the number of possible schedules that need to be evaluated, this serialization could cause a considerable overhead. For instance, scheduling applications in SMT2 mode incurs a negligible overhead, which is clearly compensated by the speedup reached by selecting the optimal schedules. In contrast, when scheduling four or more applications per core, the explosion in the number of possible schedules can easily cause that the benefits achieved by running better schedules end up being canceled out by the overhead of evaluating and selecting these schedules.

To avoid this overhead, we let applications run in parallel with the scheduler while it evaluates the possible schedules and selects the one that will be executed in the next quantum. In order to avoid the workload to slow down the scheduling step, we choose

to devote one of the cores exclusively to the scheduler. This design decision implies that while the scheduler obtains the schedule for the next quantum, the number of runnable applications is higher than the number of available hardware contexts. During this period, we let Linux perform the task scheduling. As soon as the schedule for the next quantum is determined, the application are allocated on the cores accordingly and executed using the c cores. The (lower) throughput achieved during the fraction of the workload execution where the application run on $c - 1$ cores is included in the performance results presented for the symbiotic scheduler.

To sum up, with the proposed implementation the scheduler works as follows. When a quantum expires, the scheduler stops the processes and gathers the counts of the monitored events. Then, it launches the processes on $n - 1$ cores and runs on the remaining core to obtain the best schedule for the next quantum. When such a schedule is selected the scheduler allocates each process to its assigned core, sets the performance counters, and sleeps during the quantum length.

7.3 Evaluation Setup

The experiments are carried out in a 10-core IBM POWER8 machine. The detailed systems characteristics are described in Section 3.2.3. The IBM POWER8 can execute up to eight hardware threads simultaneously. Nonetheless, as we explain in Section 3.2.3, we focus the evaluation in the SMT2 and SMT4 modes. We target multiprogram workloads composed of single-threaded application and running on the SMT8 mode reduces the system throughput over lower SMT modes.

We follow the process allocation evaluation methodology described in Section 3.3.2. Following this methodology, we reduce the amount of variation in the benchmark execution times during the experiments, and ensure that we compare the same part of the execution of each application and that the workload is uniform during the full experiment. The target number of instructions for each benchmark is set to the number of instructions they run during 120 seconds in isolated execution, and the scheduler quantum to 100 milliseconds.

Our target metric is total system throughput (STP). Nonetheless, to provide a more solid performance evaluation, we also evaluate the average normalized turnaround time (ANTT) of the workloads, which provides insight into the per-application performance reached by each scheduler. See Section 3.4 for further details about the evaluated metrics.

7.3.1 Evaluated Algorithms

To evaluate the Symbiotic scheduler, we compare the following schedulers, implemented in our scheduling framework (see Section 3.1).

1. **Random**: applications are randomly distributed across the execution contexts. Each time slice, a new schedule is randomly determined.
2. **Linux**: the default Completely Fair Scheduler (CFS) in Linux. As discussed in Section 3.2.3.1, the CFS scheduler performs NUMA-aware scheduling in our experimental platform and allocates the applications with higher main memory requirements closer to the memory controller.
3. **Dynamic L1 bandwidth-aware**: this scheduler balances the L1 bandwidth requirements of the applications across the cores at run-time. To measure the L1 bandwidth utilization of the processes, the event *perf_count_hw_cache_l1d* is gathered. See Section 5.2.2 for further details.
4. **Symbiotic**: our proposed Symbiotic job scheduler, which uses the SMT interference model and considerers the processor as a uniform memory access (UMA) system. It can schedule the applications in the SMT2 and SMT4 modes.
5. **NUMA-aware Symbiotic**: a variant of the Symbiotic scheduler that is aware of the NUMA behavior of the system. The selected schedules are allocated to the cores considering the main memory bandwidth utilization. This is done by measuring the main memory requests of the applications at run-time using performance counters (event *pm_data_all_from_memory*), and allocating the pairs or quartets of applications with higher memory bandwidth utilization to the cores of the NUMA node 0, the one where the memory module is plugged in (see Section 3.2.3.1).

Notice that the aim of the NUMA-aware optimization is not to perform sophisticated NUMA-aware scheduling, but to provide a fair comparison with the (NUMA-aware) Linux scheduler. For this purpose, the basis of both NUMA-aware schedulers is the same: measure the memory accesses of the applications and allocate the most memory-intensive applications to the NUMA node where the physical memory is installed. Without the NUMA optimization, the benefits of selecting better schedules (the purpose of the symbiotic scheduler) can be hidden in case memory-intensive applications were allocated to cores of the NUMA node more distant to the DRAM module. This issue can become an important limitation of the symbiotic scheduler over Linux, as the experimental results will show.

7.3.2 Mix Design

We use all of the SPEC CPU2006 benchmarks that we were able to compile for the IBM POWER8 to evaluate our scheduler (21 out of 29). Benchmarks are run with the reference input set. We evaluate 100 random workloads overall. 50 workloads are devised to evaluate the SMT2 mode and their number of applications doubles the number of considered cores. Thus, they range from 12 applications when the study considers 6 cores to 20 applications when considering 10 cores. The remaining 50 workloads aim to evaluate the SMT4 mode, and include four application per core. Thus, the number of applications ranges from 24 to 40, when the experiments consider from 6 to 10 cores.

7.4 Experimental Evaluation

We now evaluate how well the scheduler performs compared to the default Linux scheduler and prior work. Before showing the scheduler results, we first evaluate the accuracy of the interference prediction models devised for the SMT2 and SMT4 modes. Then, the system throughput, the per-application performance, and the stability of the selected schedules are analyzed for the SMT2 and SMT4 modes.

7.4.1 Model Accuracy

To study the accuracy of the models, we analyze the error deviation of the predicted CPI stacks with respect to the measured CPI stacks. The evaluation needs to be done in two steps since it is not possible to measure both the tST and SMT CPI stacks together in the same quantum. In a preliminary step, we measure the per-quantum tST CPI stacks of the applications offline, keeping them in a profile with their instruction counts. These tST CPI stacks will be used to check the model accuracy. Next, we run the combinations of applications. The SMT CPI stacks of the applications when running the different combinations are predicted before each quantum starts from their profiled tST CPI stacks. When the quantum expires, the predicted SMT CPI stacks for the schedule are compared against the measured SMT CPI stacks. As done in the model construction, the tST CPI stacks of consecutive quanta are interpolated, if needed, to ensure that the profiled tST CPI stacks closely match the same instructions as the SMT CPI stacks. We explore all possible combinations of applications in SMT2 mode and a very large set of combinations in SMT4 mode, considering multiple time slices per combination to capture the phase behavior.

We use the leave-p-out cross-validation methodology to evaluate the accuracy of the proposed interference models. More precisely, leave-two-out cross validation and leave-four-out cross validation are used to measure the error of the SMT2 and SMT4 models, respectively. For each possible pair of applications, leave-two-out cross validation builds a model using the data from the remaining nineteen applications, and then evaluates the model error when predicting the SMT CPI stacks for the pair of applications left out to build the model. The average absolute error and error histograms are obtained combining the errors measured for each pair of applications with the model built leaving them out. The same steps are performed to evaluate the SMT4 model, but leaving out 4-application combinations. Notice that in this case, the training data set is significantly reduced with respect to the model built using all applications.

7.4.1.1 Regression Model Accuracy

Figure 7.3 and Figure 7.4 show the histograms of the errors of the interference prediction models (the 'forward' models) for the SMT2 and SMT4 modes, respectively. They show

the deviation committed when predicting the per-application slowdown from the tST CPI stacks of the applications to be co-run.

Since there are fewer applications interfering with each other on SMT2 schedules than on SMT4 schedules, it is to be expected that the SMT2 interference model is more accurate than the SMT4 model. On average, the deviation is by 7.6% and 11.5% for the SMT2 and SMT4 models, respectively. Note that for the SMT2 model, 45% of the deviations is within $[-5\%, 5\%]$ (29% for the SMT4 model).

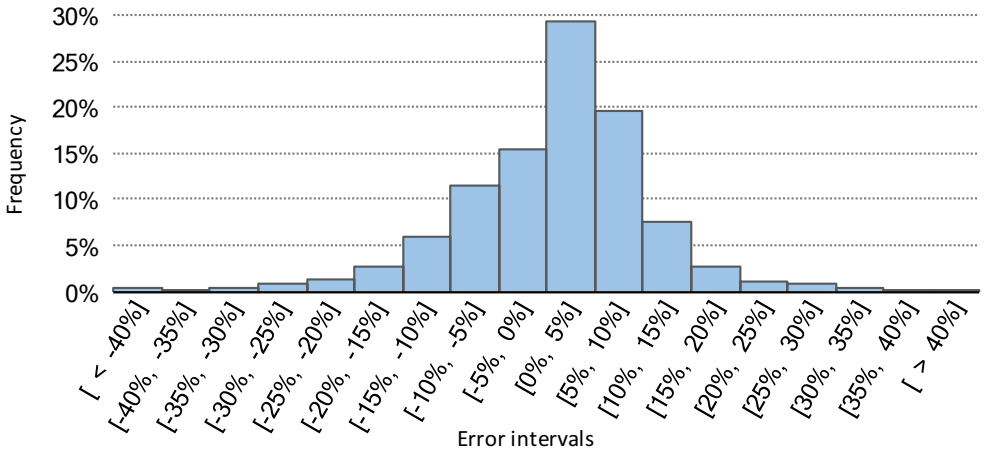


FIGURE 7.3: Forward SMT2 model error distribution.

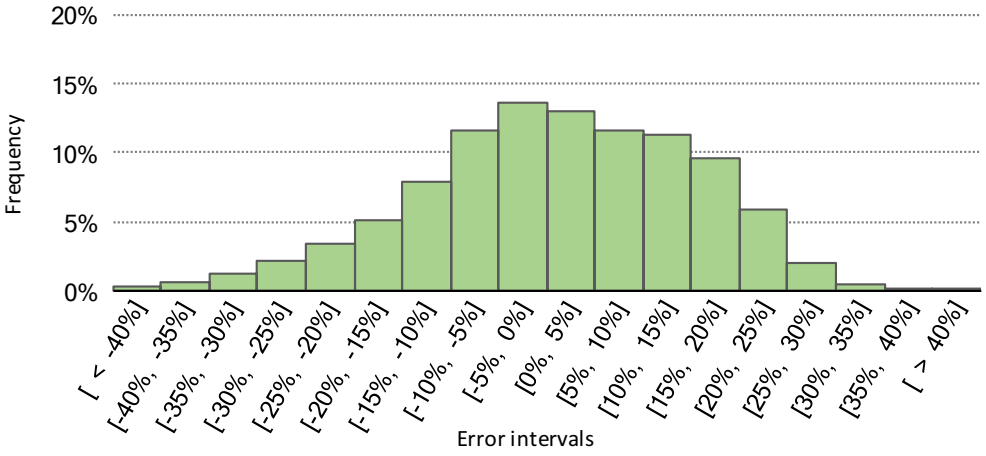


FIGURE 7.4: Forward SMT4 model error distribution.

7.4.1.2 Inverse Model Accuracy

The inverse models estimate the tST CPI stacks from the SMT CPI stacks of the applications when running concurrently on a schedule. By definition, the tST CPI stacks add to one. Since the last step of our model inversion approach is a normalization, the predicted stacks will also add to one. Thus, the accuracy of the inverse model cannot be measured by comparing the CPI stacks as a sum of their components.

Figure 7.5 and Figure 7.6 show the distribution of the error for the inverse models obtained when predicting the *completion stalls* component for the SMT2 and SMT4 modes, respectively. Completion stalls is the largest component and clearly dominates the CPI stack. It presents the highest average absolute error, which makes it a good estimate to evaluate the accuracy of the inverse model. The average absolute errors for the completion stalls component are 9.3% and 15.1%, in SMT2 and SMT4 modes, respectively. Notice that these average absolute error values do not highly differ from that obtained with the forward model. Finally, the frequency where the errors fall in the range $[-5\%, 5\%]$ also reaches similar frequencies to that of the forward model, being 47% and 24%, respectively, in SMT2 and SMT4 modes.

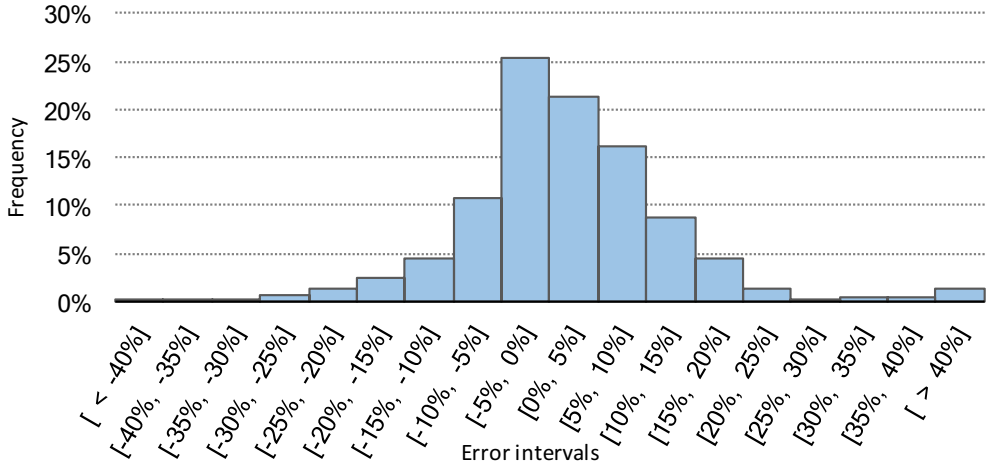


FIGURE 7.5: Inverse SMT2 model error distribution.

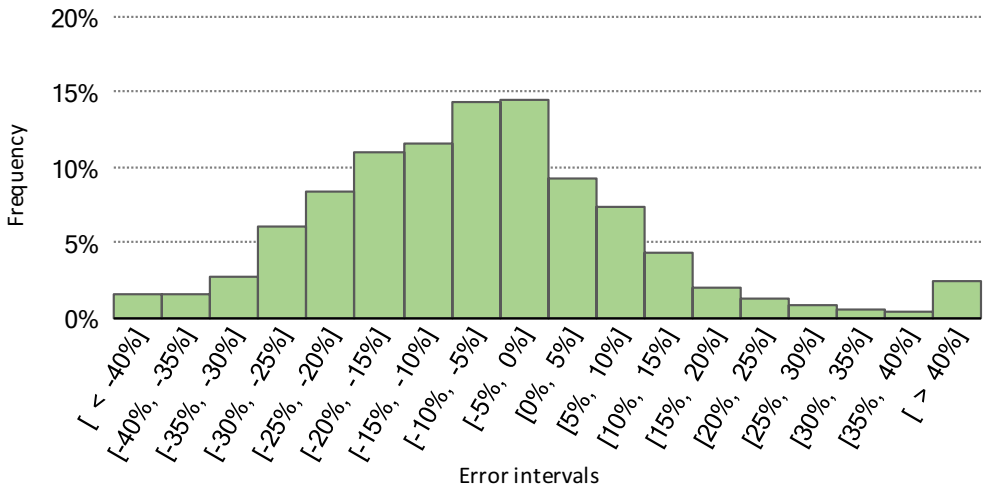


FIGURE 7.6: Inverse SMT4 model error distribution.

7.4.2 Symbiotic Scheduler Evaluation

Now that we have shown that the interference prediction models are accurate, we evaluate the performance of our proposed Symbiotic scheduler that uses the models to obtain better schedules. We also analyze the impact of symbiotic scheduling on per-application performance and study the stability of the selected schedules.

7.4.2.1 System Throughput

Figure 7.7 and Figure 7.8 present the system throughput increase achieved by the Symbiotic, NUMA-aware Symbiotic, Linux, and Dynamic L1 bandwidth-aware scheduling algorithms over the random scheduler when running in SMT2 and SMT4 modes, respectively. The speedups are averaged per core count, ranging from 6 to 10 cores. For each core count and scheduler, the bars represent the average speedup for a set of ten different workloads that are run 15 times, plotting 95% confidence intervals. As mentioned in Section 7.3, the number of applications of the SMT2 and SMT4 workloads doubles and quadruples, respectively, the number of cores considered in the experiment.

The results include the negligible overhead incurred by the symbiotic schedulers, mainly the time needed to gather the event counts from the performance counters and update the scheduling variables. As explained in Section 7.2.3, the applications are kept running

while the schedules for the next quantum are being obtained, which allows the scheduler to avoid the process selection overhead.

SMT2 mode. Figure 7.7 shows that the Symbiotic and NUMA-aware Symbiotic schedulers distinctly outperform all other schedulers. On average, across all core counts and workloads, the Symbiotic scheduler performs 12.1% better than the random scheduler, 5.2% better than the default Linux scheduler, and 4.6% better than the Dynamic L1 bandwidth-aware scheduler. The maximum benefits are achieved on the 7-core workloads, where the system throughput increase of the Symbiotic scheduler over the random and Linux schedulers is as high as 13.1% and 7.4%, respectively.

By taking into account the main memory accesses performed by each application to deal with the NUMA effects on our experimental platform, the NUMA-aware Symbiotic scheduler improves the performance achieved by the Symbiotic scheduler. On average, across the studied workloads, it performs 13.5% better than the random scheduler, 6.7% better than Linux, 5.9% better than the Dynamic L1 bandwidth-aware scheduler, and 1.3% better than the Symbiotic scheduler. With respect to Linux, it achieves a maximum average performance benefit of 11.0% on 6-core workloads. The comparison of the NUMA-aware Symbiotic scheduler against Linux is the best one to highlight the performance benefits provided by symbiotic scheduling since both schedulers implement similar NUMA-aware optimizations.

Regarding Linux, its performance benefits present an ascendant trend with the number of cores, but get somehow stabilized above 8 cores. The L1 bandwidth-aware scheduler

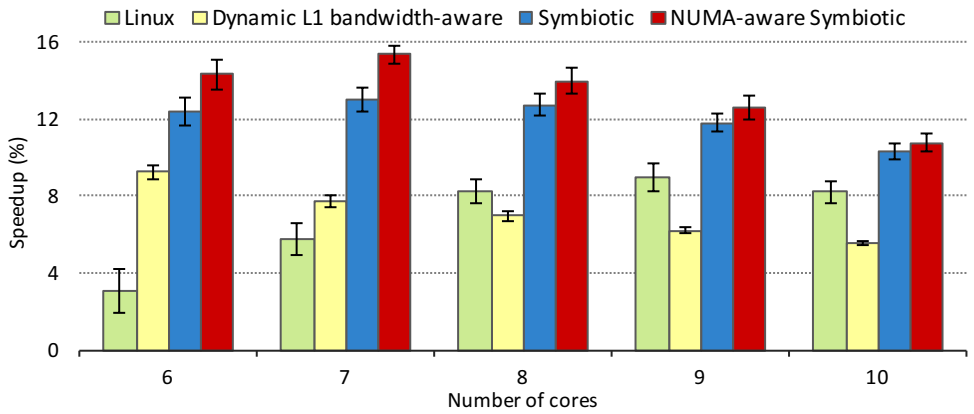


FIGURE 7.7: Average system throughput increase of the studied scheduler relative to the random scheduler when working in the SMT2 mode.

follows the opposite trend, and its performance benefits tend to decrease with the number of cores. These behaviors are clearly related with their scheduling algorithms. On the one hand, Linux monitors the memory behavior and tries to reduce memory contention, which is more beneficial when there are more applications and therefore more possible contention. In addition, Linux also performs NUMA-aware scheduling and tries to allocate the applications with higher memory requirements on the cores of the NUMA node 0 (see Section 3.2.3.1 for further details). In some cases, Linux even decides to pause threads, especially on the cores belonging to the NUMA node 1 (the farthest from the main memory modules) and when there are a lot of memory-intensive applications. On the other hand, the Dynamic L1 bandwidth-aware scheduler deals with L1 bandwidth contention, which plays a more important role when the number of applications is lower and main memory contention is not the main performance bottleneck. Anyway, both schedulers clearly perform worse than our proposed symbiotic schedulers.

SMT4 mode. Figure 7.8 depicts the STP increase of the studied schedulers in the SMT4 mode. In this SMT mode, the NUMA-aware Symbiotic scheduler greatly improves all other schedulers across all thread counts. The NUMA-aware Symbiotic scheduler performs, on average, 16.2% better than the random scheduler, 5.9% better than the Linux scheduler, and 5.3% better than the Symbiotic scheduler.

The achieved speedups grow, in general, with the number of cores as so do the speedups of the Symbiotic scheduler, since there is higher interference and more difference between the best and worst schedules. However, NUMA-aware scheduling extraordinarily

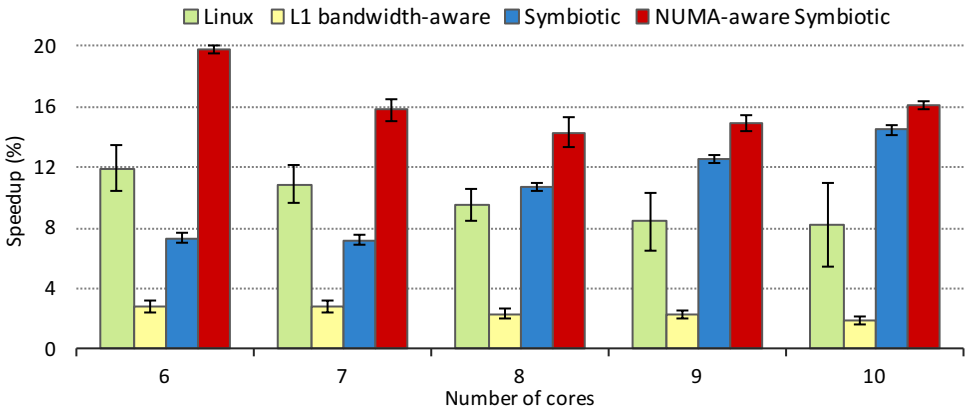


FIGURE 7.8: Average system throughput increase of the studied schedulers relative to the random scheduler when working in the SMT4 mode.

enhances the throughput for 6- and 7-core workloads, which somehow breaks the trend. The reasons that explains the big improvements for these workloads, is that with only one or two cores belonging to the NUMA node 1, the node with lower memory performance, a NUMA-aware scheduler is able to allocate all memory intensive applications on the cores of the NUMA node 0, the node with higher memory performance. In the workloads devised for higher number of cores, more cores from the NUMA node 1 are considered, and not all the memory intensive applications can be allocated to the NUMA node 0. Notice that same behavior is observed for Linux. An interesting observation is that NUMA-aware scheduling has a stronger impact on the performance of the SMT4 mode. This effect can be related with several issues. For instance, sharing the ROB with four threads can increase the penalty of a long-latency memory access. In addition, SMT4 workloads include more applications and thus, demand more memory bandwidth than SMT2 workloads.

Regarding the Symbiotic scheduler, it is really interesting to observe that its system throughput increase uniformly grows for all core counts from 6 to 10 cores. It performs better than Linux in 8-, 9-, and 10-core workloads, but worse in 6- and 7-core workloads where, as we have explained before, its NUMA unawareness strongly affects its performance. Meanwhile, the system throughput increase for the Linux scheduler follows a decreasing trend when the number of cores grows from 6 to 10. This is another indicative of the fact that NUMA-aware scheduling is critical for 6- and 7-core workloads, but reduces its importance as more cores are considered in the experiments. Finally, the Dynamic L1-bandwidth aware scheduler achieves the lowest performance benefits, and does not improve Linux. One of the reasons that explains its lower performance is that each process receives a smaller share of the resources when running in the SMT4 mode, which moves the main performance bottlenecks and the L1 bandwidth is no longer a critical resource. In addition, its performance for higher core counts is clearly constrained due to not performing NUMA-aware scheduling.

7.4.2.2 Per-Application Performance

Although the main goal of the proposed symbiotic scheduling is to maximize the system throughput, we also evaluate its impact on the average normalized turnaround time

metric. Figure 7.9 and Figure 7.10 depict the ANTT achieved by the Symbiotic, NUMA-aware Symbiotic, Linux, Dynamic L1 bandwidth-aware, and random schedulers when running the evaluated workloads for the SMT2 and SMT4 modes, respectively. The bars represent the average ANTT of the different schedulers across the ten workloads evaluated for each number of cores, plotting 95% confidence intervals. Notice that ANTT is a lower-is better-metric.

SMT2 results. Figure 7.9 shows that the Symbiotic and NUMA-aware Symbiotic schedulers clearly reach the lowest ANTT values across all evaluated workloads, followed by the Dynamic L1 bandwidth-aware scheduler and Linux. The symbiotic schedulers reach the highest differences over Linux and the random scheduler when the number of cores ranges from 6 to 8. For instance, the NUMA-aware Symbiotic scheduler achieves an ANTT 8.6% lower than Linux across these workloads. Between both symbiotic schedulers, the NUMA-aware version achieves the lowest ANTT across all core counts. Results show that, as a side effect, by reducing interference as much as possible to maximize system throughput, the symbiotic schedulers also reduce the average normalized turnaround time of the applications.

Regarding Linux and the Dynamic L1 bandwidth-aware scheduler, the same trends observed on the system throughput appear with the ANTT metric. The L1 bandwidth-aware scheduler reaches lower ANTT than Linux on workloads for 6 and 7 cores, and Linux improves the ANTT over the Dynamic L1 bandwidth-aware scheduler for larger

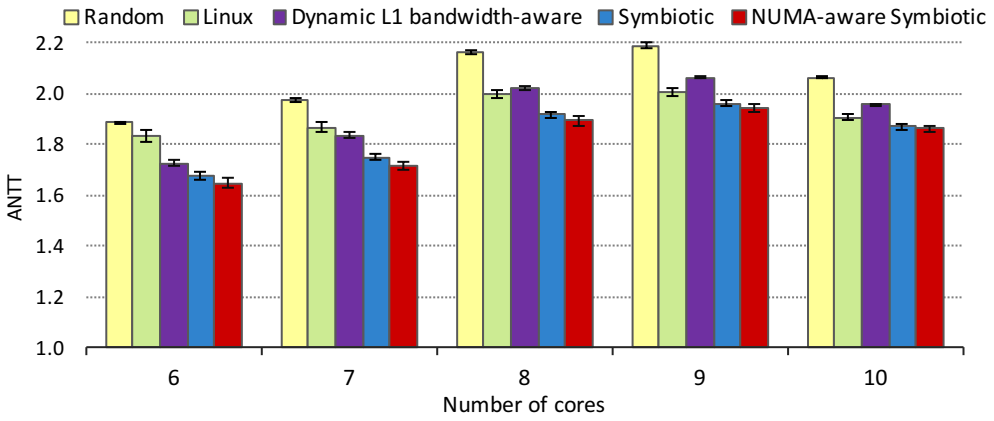


FIGURE 7.9: Average ANTT achieved by the Symbiotic, NUMA-aware Symbiotic, Dynamic L1 bandwidth-aware, Linux, and random schedulers when working in the SMT2 mode.

workloads. As explained before, this is due to the fact that Linux addresses memory bandwidth contention, which grows with the number of cores, and the Dynamic L1 bandwidth-aware scheduling algorithm deals with L1 bandwidth contention, which is more critical for lower core counts.

SMT4 results. Figure 7.10 shows that the NUMA-aware Symbiotic scheduling algorithm is the one that achieves the best per-application performance, according to the ANTT metric, in the SMT4 mode. The performance benefit is high over the random scheduler, specially as the workloads run on a higher number of cores. For instance, on the 10-core workloads, the NUMA-aware Symbiotic scheduler achieves an ANTT 14.7% lower than the random scheduler. The difference with Linux is negligible except on the 9- and 10-core workloads, where the NUMA-aware Symbiotic scheduler is 3.7% and 6.6% better, respectively.

Regarding the Symbiotic scheduler, it reaches high ANTT on workloads from 6 to 8 cores (only better than the random and Dynamic L1 bandwidth-aware schedulers). As discussed before this behavior is due to the fact that this scheduler is not aware of the NUMA effects on the IBM POWER8, which particularly affects the workloads for these numbers of cores.

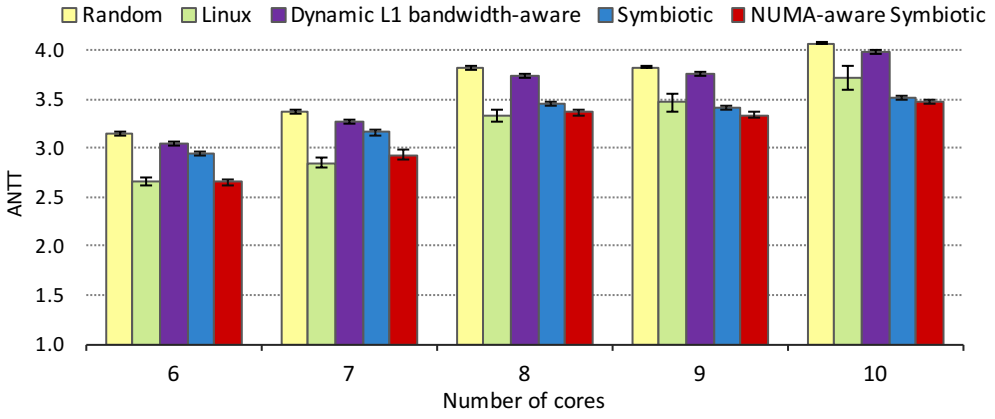


FIGURE 7.10: Average ANTT achieved by the Symbiotic, NUMA-aware Symbiotic, Dynamic L1 bandwidth-aware, Linux, and random schedulers when working in the SMT4 mode.

7.4.2.3 Symbiosis Patterns

The Symbiotic scheduler constantly re-evaluates the optimal schedule, which means that it adapts to phase behavior, updating the combinations of applications that are run together. If there is no phase change behavior, a static schedule would suffice, avoiding the overhead of recalculating the schedules. Figure 7.11 and Figure 7.12 present the frequency matrices of the schedules selected by the Symbiotic job scheduler for two 5-core workloads in SMT2 mode and a 5-core workload in SMT4 mode, respectively. The frequency matrices are symmetric matrices that represent the percentage of quanta where each combination of applications is scheduled on one core. The darker the color of the cell, the more frequently the associated pair of applications runs together on the same core.

SMT2 mode. The two matrices of Figure 7.11 represent two distinct behaviors that we have observed in the Symbiotic scheduler runs. The frequency matrix of workload 5_1 shows a workload where two couples of applications are scheduled on the same core very frequently (*h264ref* is scheduled with *libquantum* and *milc* with *bwaves*, in 66% and 70% of the time slices, respectively). This high frequency suggests that the applications present high symbiosis (e.g., a memory-bound application with a cpu-bound application) and a constant phase behavior. A different behavior is observed in the matrix of workload 5_2, where there is not a predominant pair of applications that is usually scheduled together, but all the applications are scheduled with multiple co-runners. This pattern occurs when the applications present phase behavior that changes the symbiosis of the applications, which makes it important to adapt the schedules to the current phase.

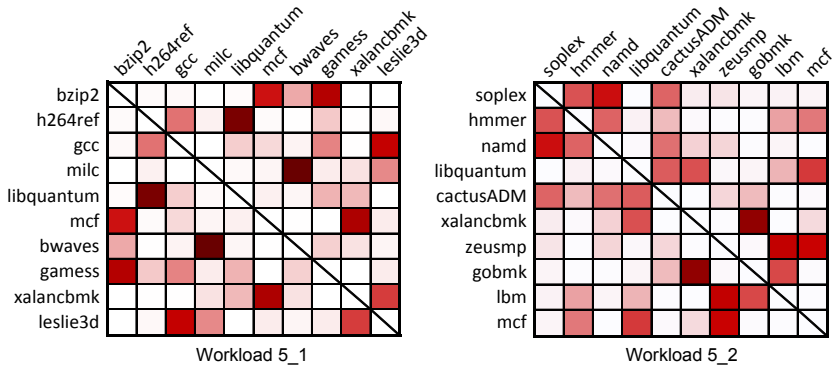


FIGURE 7.11: Frequency matrices for two 5-core workloads running in SMT2 mode.

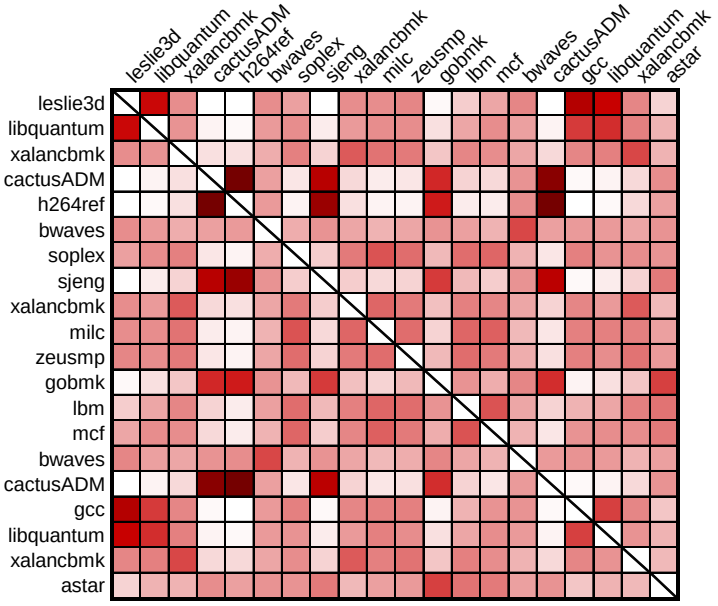


FIGURE 7.12: Frequency matrix for a 5-core workload running in SMT4 mode.

SMT4 mode. The frequency matrix of Figure 7.12 also shows the application behaviors described for the two SMT2 frequency matrices. For instance, from the group of applications *cactusADM* (two times), *h264ref*, *sjeng*, and *gobmk* four of them are usually scheduled together on the same SMT core. Another group of jobs formed by applications *leslie3d*, *libquantum* (two times), and *gcc* also tends to be scheduled on the same SMT core. This behavior is expected for applications that exhibit low phase behavior and high symbiosis among them. The other applications, either do not present high symbiosis with any application of the workload or they show a high phase behavior that makes them run on schedules with several applications through their execution.

7.5 Summary

This chapter has addressed the problem of scheduling multiprogram workloads on highly-threaded processors. This is a very hot and important problem, since scheduling has a considerable impact on these scenarios. First, because there are many possible ways of scheduling the applications and, second because each possible schedule can achieve very different performance due to the inter-application interference in the shared resources of the SMT cores. To solve these problems we propose a symbiotic scheduler.

The proposed scheduler is based on a model that estimates job symbiosis. The model predicts for any combination of applications, how much slowdown each of the applications would experience if they were concurrently run on an SMT core. It is based on CPI stacks and the parameters of the model are tailored to CPI components and not to particular applications. These parameters are obtained using regression in an offline training phase. Thus, assuming that the training set is diverse enough, there is no need to re-train the model to schedule new applications.

The symbiotic scheduler uses the interference model to quickly explore the space of possible schedules and selects the optimal schedule for the next quantum. Unfortunately, the number of possible schedules grows too fast with the number of hardware contexts. For instance, there are more than 2 billion of possible ways of scheduling 20 applications on 5 cores supporting four threads per core. To address this issue, the scheduling is modeled as a minimum-weight perfect matching graph problem that can be solved in polynomial time.

The experimental evaluation, carried out on an IBM POWER8 server, shows that the Symbiotic scheduler improves system throughput over the Linux scheduler. On average across 6- to 10-core workloads the throughput increase is by 6.7% and 5.9% in the SMT2 and SMT4 modes, respectively. Despite our current implementation is designed for the IBM POWER8, the symbiotic scheduler could be adapted to other CMP architectures with SMT cores that provide a similar cycle accounting mechanism, e.g., an Intel Xeon server [69]. This only requires a one-time training step. The scheduler can also support heterogeneous architectures, by creating different models for the various core types.

The work discussed in this chapter has been published in [75].

Chapter 8

Conclusions

This thesis has addressed the problem of scheduling multiprogram workloads on current single-threaded and SMT multicore processors. Experiments have considered three recent commercial processors and the proposed schedulers have been designed to adapt to the particular characteristics of each specific architecture. First, we have proposed multiple bandwidth-aware scheduling algorithms that tackle the bandwidth contention points of the memory hierarchy. Next, progress-aware schedulers have been devised to deal with the unfairness that resource sharing causes in SMT multicores. Finally, this dissertation has presented a symbiotic scheduler that uses SMT interference models based on CPI stacks to estimate the performance of the possible schedules with the goal of selecting the best one.

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

8.1 Contributions

In Chapter 4, a memory-hierarchy bandwidth-aware scheduling algorithm has been presented to deal with the bandwidth contention that arises at the different contention points of the memory hierarchy of current multicore single-threaded processors. The algorithm has been designed after finding that LLC bandwidth contention can even achieve a stronger impact on performance than main memory bandwidth contention on some scenarios. This situation is exacerbated by the industry trend of increasing the number of cores and multithreading capabilities, which will put even more pressure on the memory system of future multicore and manycore processors. The proposed scheduler pursues to evenly distribute the memory accesses over the workload execution time, and balances the accesses among the different caches of a given cache level when it implements shared caches. This is done by selecting the processes to achieve a certain main memory bandwidth utilization in the quantum and by allocating the selected processes to the proper cores depending on which cores share each cache, respectively. The algorithm is further improved to favor the execution of processes with higher performance degradation on less bandwidth-contentious scenarios. The experimental evaluation of the proposed schedulers shows that the turnaround time of multiprogram mixes can be reduced over Linux by 6.6% on average across the evaluated mixes.

In Chapter 5, we have proposed a bandwidth-aware scheduling algorithm for multicore processors consisting of SMT cores. Unlike single-threaded processors, SMT multicores implement L1 caches that are shared by the threads that simultaneously run on each core, creating a new potential contention point. The experimental analysis performed in this contention point illustrates its importance, since the performance and L1 bandwidth of the processes are strongly related at run-time. To address L1 bandwidth contention, we propose a Dynamic L1 bandwidth-aware process allocation policy that mitigates the contention by allocating the processes to the cores so that the L1 accesses are balanced among the L1 caches of the processor. Such a process allocation policy is then combined with a main memory bandwidth-aware process selection policy to build an entire scheduler that deals with bandwidth contention on SMT multicores. The experimental evaluation shows that the proposed SMT bandwidth-aware scheduler improves throughput over Linux by 4.6% on average across the studied mixes.

In Chapter 6, progress-aware schedulers have been introduced as an effective way to keep track of how the processes of a multiprogram workload progress at run-time. SMT multicores are able to concurrently run several applications, but this parallelism is reached sharing most of the processor resources among several processes. Depending on how the processes are scheduled, distinct processes can achieve widely different progresses at run-time, which can strongly affect the system fairness. Two progress-aware schedulers are proposed in this chapter. The progress-aware Fair scheduler exclusively addresses system fairness, prioritizing the processes with lower accumulated progress. The progress-aware Perf&Fair scheduler simultaneously deals with fairness and performance to provide fair executions trying to achieve the highest performance. The experimental evaluation shows that unfairness can be reduced to a third with respect to Linux when only system fairness is addressed. Furthermore, when simultaneously targeting fairness and performance, Perf&Fair reduces unfairness to a half while improving the turnaround time of the studied mixes by 5.6% on average with respect to Linux.

Finally, a symbiotic job scheduler is proposed in Chapter 7. This Symbiotic scheduler uses an SMT interference model to guide the scheduling decisions. The model is based on CPI stacks and estimates the performance of the possible schedules, considering the contention in all the shared resources of SMT cores. Due to the exponential number of possible schedules, even with our fast model, evaluating all possible schedules would cause a non-negligible overhead. To avoid this problem, the Symbiotic scheduler models the scheduling problem as a minimum-weight perfect matching graph problem that can be solved in polynomial time. The experimental evaluation shows that the symbiotic job scheduler is able to improve the system throughput, on average, by 6.7% and 5.9% over Linux in the SMT2 and SMT4 modes, respectively.

8.2 Future Directions

As for future work, we plan to extend our scheduling algorithms and propose new strategies to fit the requirements of a wider broad of systems. For example, parallel applications are gaining importance and weight on the high-performance workloads that small-scale servers typically run, and clearly require scheduling strategies that greatly differ from those strategies used to schedule sequential applications. Another example is

represented by mobile systems, where power consumption is a major concern and can be addressed with the appropriate scheduling algorithms. Lastly, scheduling is also useful to make a convenient use of some new features that recent processors implement such as cache partitioning.

A path we are starting to explore consists of scheduling parallel applications on current multicore multi-threaded processors, but also future manycore systems. Parallel programs divide their calculations into multiple threads that are executed concurrently, which speeds up their execution time significantly. However, in order to keep the shared data consistent and to guarantee the necessary dependencies, synchronization among threads is needed. Synchronization makes threads wait until other threads have finished some part of their execution. Accelerating a parallel program is therefore not straightforward, because speeding up a non-critical thread just makes it wait for longer and does not decrease the total program execution time. Similarly, performance variations between threads can make an originally balanced parallel program unbalanced, increasing the execution time of the program due to the slowest thread.

On the prevalent architecture nowadays, which is a multicore processor consisting of SMT cores, performance variation can be high due to interference in shared resources, but at the same time there is a lot of freedom in manipulating per-thread performance by changing the combinations of threads that execute together on a core. As such, the scheduler should try to reduce performance variations among threads in case of balanced parallel programs, as well as speeding up critical threads when there is unbalance on the work each threads performs. Therefore, a scheduling proposal tailoring parallel applications demands for distinct and specific strategies tailored to improve their performance and/or fairness.

Other systems that require from new scheduling strategies are mobile systems. These systems usually work powered by batteries, which means that they typically have to face heavy power constrains. Obviously, these power constrains influence the microprocessor design; hence, current mobile systems tend to implement heterogeneous processors, such as the big.LITTLE architecture [76]. This architecture implements two different kind of cores on the same chip. On the one hand, big cores offer high-performance but are power-hungry. As opposite, small cores present a much lower power consumption but their performance is also smaller. Since the set of applications to run and their characteristics

as well as the available power change dynamically, deciding which processes should run on the big and small cores at runtime is a task that perfectly fits a process scheduler.

An additional problem that arises in mobile systems is that some tasks demand for certain levels of quality of service (e.g., multimedia applications) and others should directly meet hard-real time constraints (e.g., network communications to manage phone calls). Optimizing quality of service for a certain power-budget, on scenarios with very different kinds of applications that present distinct temporary requirements (some of them provably needing to meet real-time constraints), and running on an heterogeneous architecture is a challenging and very relevant problem nowadays.

Finally, scheduling is also required to take advantage of advanced characteristics of current and future processors. Cache partitioning is a good example in current systems. The latest Intel processors offer the possibility to assign ways of the cache to the processes, limiting the cache space that each process can access. In this scenario, the scheduler can decide whether a process should receive more or less cache to maximize the performance and/or fairness. Looking at a close future, the dark silicon era will restrict the number of transistors that can be powered on at a given time to guarantee the thermal design power (TDP) constraints of future processors. Based on this prediction, some researches are exploring the design of processors implementing different execution pipelines (or hardware accelerators) to speedup the parts of the processes that suit the characteristics for which each pipeline is designed. In this scenario, schedulers can take control of which applications should run and which pipelines should be used at a given time to meet the performance, energy, and fairness requirements. We think that this kind of scheduling is going to be a hot topic in the coming years.

In summary, scheduling algorithms, tailored to the processor architecture, are required in current systems and will potentially gain more relevance in future processors to achieve the greatest performance, energy efficiency, and fairness, under different power constraints and running different kind of applications.

8.3 Publications

The following papers related with this dissertation were submitted and accepted for publication in different international journals and conferences with peer review.

Journals:

- J. Feliu, S. Petit, J. Sahuquillo, and J. Duato. Cache-Hierarchy Contention Aware Scheduling in CMPs. *IEEE Transactions on Parallel and Distributed Systems* (TPDS), volume 25, issue 3, pages 581-590, 2014.
- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Bandwidth-Aware On-Line Scheduling in SMT Multicores. *IEEE Transactions on Computers* (TC), volume 65, issue 2, pages 422-434, 2016.
- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Perf&Fair: a Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores. *IEEE Transactions on Computers* (TC), to appear in. DOI: 10.1109/TC.2016.2620977

Conferences:

- J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit. Symbiotic Job Scheduling on the IBM POWER8. In *Proceedings of the IEEE 22nd International Symposium on High Performance Computer Architecture* (HPCA), pages 669-680, Barcelona, Spain, 2016. This publication received a HiPEAC Paper Award.
- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. L1-Bandwidth Aware Thread Allocation in Multicore SMT Processors. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques* (PACT), pages 123-132, Edinburgh, Scotland, 2013.
- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler. In *Proceedings of the IEEE 29th International Parallel and Distributed Processing Symposium* (IPDPS), pages 187-196, Hyderabad, India, 2015.

- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Understanding Cache Hierarchy Contention in CMPs to Improve Job Scheduling. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium* (IPDPS), pages 508-519, Shanghai, China, 2012.
- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Addressing Bandwidth Contention in SMT Multicores Through Scheduling. In *Proceedings of the 28th International Conference on Supercomputing* (ICS), page 167, Munich, Germany, 2014.
- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Using Huge Pages and Performance Counters to Determine the LLC Architecture. In *Proceedings of the 2013 International Conference on Computational Science* (ICCS), pages 2557-2560, Barcelona, Spain, 2013.

In addition, other related papers have been published in international summer schools and domestic conferences:

- J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit. Improving Throughput on the IBM POWER8 with a Symbiotic Scheduler. In *Proceedings of the 12th International Summer School on Advanced Computer Architecture and Compilation for High-Performance and Embedded Systems* (ACACES), pages 201-204, Fiuggi, Italy, 2016.
- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Planificación Considerando el Ancho de Banda de la Jerarquía de Cache. In *Actas de las XXIII Jornadas de Paralelismo* (JP), pages 472-477, Elx, Spain, 2012.
- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Planificación Considerando Degradación de Prestaciones por Contención. In *Actas de las XXIV Jornadas de Paralelismo* (JP), pages 62-67, Madrid, Spain, 2013.
- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Ubicación de Procesos Considerando el Ancho de Banda de L1 en Procesadores Multinúcleo SMT. In *Actas de las XXV Jornadas de Paralelismo* (JP), pages 343-352, Valladolid, Spain, 2014.
- J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Planificación Orientada a Equidad Considerando el Progreso en Multinúcleos SMT. In *Actas de las XXVI Jornadas de Paralelismo* (JP), pages 118-126, Córdoba, Spain, 2015.

- J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit. Planificación Simbiótica de Procesos en el IBM POWER8. In *Actas de las XXVII Jornadas de Paralelismo* (JP), pages 315-324, Salamanca, Spain, 2016.

All works listed above are exclusively related with this thesis. The specific contributions of the Ph.D. candidate reside mostly in the design and implementation of the proposed algorithms, as well as the execution of the performed experiments, the analysis and discussion of the results, the writing of the paper drafts describing the work, and the presentation of the papers in the conferences. Along these processes, the co-authors have repeatedly provided useful hints and advices, which the Ph.D. candidate has then applied to make the work evolve into its final version.

References

[1] B. Sinharoy, J.A. Van Norstrand, R.J. Eickemeyer, H.Q. Le, J. Leenstra, D.Q. Nguyen, B. Konigsburg, K. Ward, M.D. Brown, J.E. Moreira, D. Levitan, S. Tung, D. Hrusecky, J.W. Bishop, M. Gschwind, M. Boersma, M. Kroener, M. Kaltenbach, T. Karkhanis, and K.M. Fernsler. IBM POWER8 Processor Core Microarchitecture. *IBM Journal of Research and Development*, 59(1):2:1–2:21, Jan 2015.

[2] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous Multithreading: Maximizing On-Chip Parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 392–403, 1995.

[3] J. Burns and J.-L. Gaudiot. SMT Layout Overhead and Scalability. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 13(2):142–155, Feb 2002.

[4] Y. Li, K. Skadron, D. Brooks, and Z. Hu. Performance, Energy, and Thermal Considerations for SMT and CMP Architectures. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 71–82, 2005.

[5] S. Eyerman and L. Eeckhout. The Benefit of SMT in the Multi-Core Era: Flexibility Towards Degrees of Thread-Level Parallelism. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 591–606, 2014.

[6] A. Snavely and D. M. Tullsen. Symbiotic Jobscheduling for Simultaneous Multithreaded Processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 234–244, 2000.

[7] S. Eyerman and L. Eeckhout. Probabilistic Job Symbiosis Modeling for SMT Processor Scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 91–102, 2010.

[8] O. Mutlu and T. Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 146–160, 2007.

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

[10] T. Moscibroda and O. Mutlu. Memory Performance Attacks: Denial of Memory Service in Multi-Core Systems. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, pages 18:1–18:18, 2007.

[11] D. Xu, C. Wu, and P.-C. Yew. On Mitigating Memory Bandwidth Contention through Bandwidth-Aware Scheduling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 237–248, 2010.

[12] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato. Cache-hierarchy contention aware scheduling in CMPs. *IEEE Transactions on Parallel and Distributed Systems*, 25 (3):581–590, March 2014.

[13] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Scheduling Algorithms with Bus Bandwidth Considerations for SMPs. In *Proceedings of the 32nd International Conference on Parallel Processing (ICPP)*, pages 547 –554, 2003.

[14] C. D. Antonopoulos, D. S. Nikolopoulos, and T. S. Papatheodorou. Realistic Workload Scheduling Policies for Taming the Memory Bandwidth Bottleneck of SMPs. In *Proceedings of the 11th International Conference on High Performance Computing (HiPC)*, pages 286–296, 2004.

[15] E. Koukis and N. Koziris. Memory and Network Bandwidth Aware Scheduling of Multiprogrammed Workloads on Clusters of SMPs. In *Proceedings of the12th*

International Conference on Parallel and Distributed Systems (ICPADS), pages 345–354, 2006.

[16] F. Pinel, J. E. Pecero, P. Bouvry, and S. U. Khan. Memory-Aware Green Scheduling on Multi-core Processors. In *Proceedings of the 39th International Conference on Parallel Processing Workshops*, pages 485–488, 2010.

[17] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing Memory Systems. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 208–222, 2006.

[18] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via Source Throttling: a Configurable and High-Performance Fairness Substrate for Multi-Core Memory Systems. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 335–346, 2010.

[19] D. Xu, C. Wu, P.-C. Yew, J. Li, and Z. 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*, pages 295–306, 2012.

[20] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems. In *19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 639–650, 2013.

[21] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn. Using OS Observations to Improve Performance in Multicore Systems. *IEEE Micro*, 28(3):54–66, may 2008.

[22] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 129–142, 2010.

[23] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M.-L. Soffa. The impact of memory subsystem resource sharing on datacenter applications. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[24] M. Sato, I. Kotera, R. Egawa, H. Takizawa, and H. Kobayashi. A Cache-Aware Thread Scheduling Policy for Multi-Core Processors. In *Proceedings of the 6th International Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 109 –114, 2009.

[25] M. K. Qureshi, D. N. Lynch, O. Mutlu, and Y. N. Patt. A Case for MLP-Aware Cache Replacement. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, pages 167–178, 2006.

[26] D. Kaseridis, J. Stuecheli, Jian Chen, and L. K. John. A bandwidth-aware memory-subsystem resource management using non-invasive resource profilers for large CMP systems. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1 –11, 2010.

[27] A. Fedorova, M. Seltzer, and M. D. Smith. Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler. In *Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 25–38, 2007.

[28] A. Fedorova, S. Blagodurov, and S. Zhuravlev. Managing contention for shared resources on multicore processors. *Commun. ACM*, 53(2):49–57, feb 2010.

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

[30] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. Transactions on High-Performance Embedded Architectures and Compilers III. chapter Dynamic cache partitioning based on the MLP of cache misses, pages 3–23. 2011.

[31] S. Srikantaiah, M. Kandemir, and Q. Wang. SHARP Control: Controlled Shared Cache Management in Chip Multiprocessors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 517–528, 2009.

[32] R. Manikantan, K. Rajan, and R. Govindarajan. Probabilistic Shared Cache Management (PriSM). In *Proceedings of the 39rd Annual International Symposium on Computer Architecture (ISCA)*, pages 428–439, 2012.

[33] G.E. Suh, S. Devadas, and L. Rudolph. A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA)*, pages 117–128, 2002.

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

[35] J. Chang and G. S. Sohi. Cooperative cache partitioning for chip multiprocessors. In *Proceedings of the 21st Annual International Conference on Supercomputing (ICS)*, pages 242–252, 2007.

[36] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual Private Caches. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, pages 57–68, 2007.

[37] J. A. Colmenares, G. Eads, S. Hofmeyr, S. Bird, M. Moreto, D. Chou, B. Gluzman, E. Roman, D. B. Bartolini, N. Mor, K. Asanovic, and J. D. Kubiatowicz. Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous Adaptation. In *Proceedings of the 50th ACM/EDAC/IEEE International Conference on Design Automation Conference (DAC)*, pages 1–10, 2013.

[38] S. Eyerman, L. Eeckhout, T. Karkhanis, and J.E. Smith. A Performance Counter Architecture for Computing Accurate CPI Components. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 175–184, 2006.

[39] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache Pirating: Measuring the Curse of the Shared Cache. In *Proceedings of the 40th International Conference on Parallel Processing (ICPP)*, pages 165–175, 2011.

[40] M. Casas and G. Bronevetsky. Active Measurement of Memory Resource Consumption. In *Proceedings of the IEEE 25th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 995–1004, 2014.

[41] S. Hily and A. Seznec. Contention on 2nd Level Cache May Limit the Effectiveness of Simultaneous Multithreading. In *[Research Report] RR-3115, INRIA*, 1997.

[42] V. Čakarević, P. Radojković, J. Verdú, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Characterizing the Resource-Sharing Levels in the UltraSPARC T2 Processor. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 481–492, 2009.

[43] C. Acosta, F. J. Cazorla, A. Ramirez, and M. Valero. Thread to Core Assignment in SMT On-Chip Multiprocessors. In *Proceedings of the 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 67–74, 2009.

[44] T. Moseley, J.L. Kihm, D.A. Connors, and D. Grunwald. Methods for Modeling Resource Contention on Simultaneous Multithreading Processors. In *Proceedings of the 23th International Conference on Computer Design (ICCD)*, pages 373–380, 2005.

[45] A. Settle, J. Kihm, A. Janiszewski, and D. Connors. Architectural Support for Enhanced SMT Job Scheduling. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 63–73, 2004.

[46] S. Eyerman and L. Eeckhout. Per-Thread Cycle Accounting in SMT Processors. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–144, 2009.

[47] L. Porter, M. A. Laurenzano, A. Tiwari, A. Jundt, W. A. Ward, Jr., R. Campbell, and L. Carrington. Making the Most of SMT in HPC: System- and Application-Level Perspectives. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):59:1–59:26, jan 2015.

[48] J. Mars, L. Tang, R. Hundt, K. Skadron, and Mary L. Soffa. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In

Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 248–259, 2011.

[49] Y. Zhang, M. A. Laurenzano, J. Mars, and L. Tang. SMiTe: Precise QoS Prediction on Real-System SMT Processors to Improve Utilization in Warehouse Scale Computers. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 406–418, 2014.

[50] K. Luo, J. Gummaraju, and M. Franklin. Balancing Throughput and Fairness in SMT Processors. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 164–171, 2001.

[51] S. Eyerman and L. Ecckhout. A Memory-Level Parallelism Aware Fetch Policy for SMT Processors. In *Proceedings of the 13th International Symposium on High Performance Computer Architecture (HPCA)*, pages 240–249, 2007.

[52] G. Varghese, J. Sanjeev, T. Chao, S. Ken, D. Satish, S. Scott, N. Ves, K. Tanveer, S. Sanjib, and S. Puneet. Penryn: 45-nm Next Generation Intel Core 2 Processor. In *IEEE Asian Solid-State Circuits Conference*, pages 14–17, 2007.

[53] A. B. Caldeira, V. Haug, M. E. Kahle, C. D. Maciel, M. Sanchez, and S. Y. Sung. *IBM Power Systems S812L and S822L Technical Overview and Introduction.* IBM Redbooks, 2014.

[54] L. W. McVoy and C. Staelin. lmbench: Portable Tools for Performance Analysis. In *Proceedings of 5th USENIX Security Symposium on USENIX Security Symposium*, pages 279–294, 1996.

[55] J. McCalpin. STREAM benchmark. *Link: www.cs.virginia.edu/stream/ref.html*, 1995.

[56] J. Corbet. NUMA Scheduling Progress. *Link: https://lwn.net/Articles/568870/*, 2013.

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

[58] R. Das, R. Ausavarungnirun, O. Mutlu, A. Kumar, and M. Azimi. Application-to-Core Mapping Policies to Reduce Memory Interference in Multi-Core Systems. In

Proceedings of the 119h International Symposium on High-Performance Computer Architecture (HPCA), pages 107–118, 2013.

[59] K. Yotov, K. Pingali, and P. Stodghill. Automatic Measurement of Memory Hierarchy Parameters. In *Proceedings of the 5th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, pages 181–192, 2005.

[60] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Understanding cache hierarchy contention in CMPs to improve job scheduling. In *Proceedings of the IEEE 23th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 508–519, 2012.

[61] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Using Huge Pages and Performance Counters to Determine the LLC Architecture. In *International Conference on Computational Science (ICCS)*, pages 2557 – 2560, 2013.

[62] A. Lukefahr, S. Padmanabha, R. Das, F. M. Sleiman, R. Dreslinski, T. F. Wenisch, and S. Mahlke. Composite Cores: Pushing Heterogeneity Into a Core. In *Proceedings of the 42th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 317–328, 2012.

[63] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. L1-Bandwidth Aware Thread Allocation in Multicore SMT Processors. In *Proceedings of the 22nt International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 123–132, 2013.

[64] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Addressing Bandwidth Contention in SMT Multicores Through Scheduling. In *Proceedings of the 28th Annual International Conference on Supercomputing (ICS)*, pages 167–167, 2014.

[65] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Bandwidth-Aware On-Line Scheduling in SMT Multicores. *IEEE Transactions on Computers*, 65(2):422–434, February 2016.

[66] R. Love. *Linux kernel development*. Pearson Education, 2010.

[67] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Addressing Fairness in SMT Multicores with a Progress-Aware Scheduler. In *Proceedings of the IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS)*, pages 187–196, 2015.

[68] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato. Perf&Fair: a Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores. *To appear in IEEE Transactions on Computers*, 2017. doi: 10.1109/TC.2016.2620977.

[69] A. Nowak, D. Levinthal, and W. Zwaenepoel. Hierarchical Cycle Accounting: a New Method for Application Performance Tuning. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 112–123, 2015.

[70] D. M. Tullsen and J. A. Brown. Handling Long-latency Loads in a Simultaneous Multithreading Processor. In *Proceedings of the 34th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO))*, pages 318–327, 2001.

[71] IBM Knowledge Center, *Analyzing application performance on Power Systems servers*, 2015. URL `https://www-01.ibm.com/support/knowledgecenter/linuxonibm/liaal/iplsdkusetools.htm`.

[72] P. Radojkovic, V. Cakarevic, J. Verdu, A. Pajuelo, F. J. Cazorla, M. Nemirovsky, and M. Valero. Thread assignment of multithreaded network applications in multicore/multithreaded processors. *IEEE Transactions on Parallel and Distributed Systems*, 24(12):2513–2525, Dec 2013.

[73] Y. Jiang, X. Shen, J. Chen, and R. Tripathi. Analysis and Approximation of Optimal Co-scheduling on Chip Multiprocessors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 220–229, 2008.

[74] J. Edmonds. Maximum Matching and a Polyhedron with 0, l-Vertices. *J. Res. Nat. Bur. Standards B*, 69(1965):125–130, 1965.

[75] J. Feliu, S. Eyerman, J. Sahuquillo, and S. Petit. Symbiotic Job Scheduling on the IBM POWER8. In *Proceedings of the 22nd International Symposium on High-Performance Computer Architecture (HPCA)*, pages 669–680, 2016.

[76] Samsung Electronics. Samsung Primes Exynos 5 Octa for ARM big.LITTLE Technology with Heterogeneous Multi-Processing Capability. *Press release*, 2013.