UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

TESIS DOCTORAL

Software de sistema de nueva generación para gestión de la contención de recursos, optimización del uso de CPU, y explotación de asimetría en sistemas multinúcleo

Novel system software for addressing resource contention, maximizing CPU usage, and harnessing performance asymmetry on multicore systems

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Carlos Bilbao Muñoz

Director

Juan Carlos Sáez Alcaide

Software de sistema de nueva generación para gestión de la contención de recursos, optimización del uso de CPU, y explotación de asimetría en sistemas multinúcleo

Novel system software for addressing resource contention, maximizing CPU usage, and harnessing performance asymmetry on multicore systems

UNIVERSIDAD COMPLUTENSE DE MADRID

Facultad de Informática

Tesis Doctoral

MEMORIA PARA OPTAR AL GRADO DE DOCTOR PRESENTADA POR

Carlos Bilbao Muñoz

Dirigida por: Juan Carlos Sáez Alcaide

Programa: Doctorado en Ingeniería Informática (RD 99/2011)

Agradecimientos

Quisiera primero dar las gracias a mis padres, M. Paz y Eusebio. Sin su paciencia y amor incondicional no habría logrado nada.

Me considero inmensamente afortunado de haber contado con el Dr. Juan Carlos Sáez como director de mi tesis. Gracias a él, conseguí mi primer trabajo, ya que pude realizar mis prácticas de empresa con una Beca de Colaboración en el Departamento de Arquitectura de Computadores y Automática de la Facultad de Informática. Fue entonces cuando empecé a cogerle el gusto a la investigación. Seis años después, afortunadamente, hemos podido mantenernos en contacto, a pesar de vivir en continentes distintos. Durante el desarrollo de este doctorado, Juan Carlos ha sido paciente y comprensivo cuando no he logrado compaginar mi rol profesional con la tesis. Él siempre ha encontrado la manera de apoyarme y, al mismo tiempo, continuar con su exigente labor como profesor, investigador, y joven padre. Esta experiencia me ha demostrado que Juan Carlos es, en definitiva, un modelo absoluto de dedicación en todas esas áreas. Quisiera sinceramente darle las gracias por haber tenido un impacto tan positivo en mi vida y espero poder seguir siendo su alumno en el futuro.

Doy las gracias a los doctores Dario Suárez Gracia y Salvador Petit por el esfuerzo de revisar mi tesis y hacer valiosas recomendaciones de mejora.

Agradezco la colaboración de Javier Rubio en la contribución de la tesis sobre elasticidad. Su Trabajo de Fin de Grado sobre esta temática sentó las bases del gestor de elasticidad propuesto en esta tesis doctoral.

También quisiera agradecer al Dr. Manuel Prieto-Matias por su ayuda y, en general, al grupo de investigación ArTeCS de la Universidad Complutense de Madrid por darme acceso a todos los recursos computacionales sin los cuales esta tesis no habría sido posible.

Finalmente, quiero dar las gracias a Elyssa y Pipa, por su apoyo y cariño durante el desarrollo de la tesis.

Contents

List of Figures

List of Tables

Abstract

The steady reduction in transistor size was for a long time considered the norm in computer systems manufacturing, until mounting obstacles – collectively referred to as the *Power Wall* – increasingly began to crop up. In response to this issue (and others, like the *Memory Wall*), the hardware industry shifted its focus to Chip Multicore Processors (CMPs), which integrate multiple cores on a single chip. CMPs have now established themselves as the de facto general-purpose architecture across a broad range of commercial platforms, from embedded and mobile devices to high-end server systems.

This thesis proposes a number of scheduling and resource management strategies for multicore processors that have been implemented in the operating system (OS) kernel and runtime system levels. The main objective of these proposals is to take on three key challenges that multicore systems pose to the system software: (i) addressing shared-resource contention effects, (ii) maximizing CPU utilization, and (iii) dealing with performance asymmetry under the presence of different core types in the platform.

Shared-resource contention in multicores arises as a result of the natural competition among several co-running applications for key resources shared among cores, including the last-level cache (LLC) or the DRAM controller. Unfortunately, this source of contention leads to uneven and unpredictable performance degradation of the co-running workloads, which negatively impacts system fairness, causing numerous undesirable effects. Contention-induced performance disparities are exacerbated in NUMA multicores due to the high latency of remote memory accesses and the impact of thread-to-socket placement. To address these shortcomings, this thesis proposes Divide&Content (DC), a fairness-aware resource management technique that combines partitioning of the LLC with dynamic thread placement on NUMA multicores. The smart combination of these two contention-mitigation techniques allows DC to substantially outperform state-of-the art strategies in terms of fairness, while delivering benefits automatically to unmodified applications.

Maximizing CPU utilization is critical for cloud providers to minimize costs and maximize profitability. To complement existing mechanisms for increasing resource usage, this thesis proposes a novel strategy that harnesses the elastic execution of HPC/scientific workloads. These workloads often exhibit a high degree of inherent parallelism, enabling them to efficiently utilize multiple cores during significant portions of their execution. Leveraging this characteristic, our proposal exploits brief idle core cycles during workload execution to opportunistically run additional

threads from elastic workloads. This helps increase CPU utilization and throughput, with minimal impact on the performance of applications without elasticity support. Our approach accomplishes this without requiring modifications to the applications; instead, it relies on OS-level support and explicit cooperation between the OS scheduler and the runtime system.

This thesis also focuses on *the design of scheduling techniques to address performance asymmetry in asymmetric multicore processors (AMPs)*. Recently adopted in commercial desktop products, AMPs combine high-performance big cores with energy-efficient small cores, all sharing a common ISA that ensures compatibility with legacy software. To cater to AMPs' unique set of challenges, this thesis proposes novel throughput-oriented scheduling support implemented across different system software layers. Crucially, our proposal combines asymmetry-aware loop-scheduling with OS-runtime interaction. This thesis also conducts a comprehensive experimental evaluation of the Intel Thread Director Technology, a set of hardware facilities integrated into recent AMPs from Intel to aid the OS scheduler in making asymmetry-aware thread placement decisions at runtime.

Lastly, a transversal objective of this thesis was to simplify the development of scheduling and resource management extensions in the OS kernel, resulting in the creation of the open-source PMCSched framework. The exploitation of PMCSched in every research line of this thesis reflects the crucial role of the system software in addressing the aforementioned three challenges of multicore processors.

Keywords: Linux kernel, multicore processors, resource management, runtime systems, cache partitioning, operating systems, elasticity, malleability, asymmetric multicores.

Resumen

Durante mucho tiempo, la constante reducción del tamaño de los transistores se daba por hecho en la fabricación de sistemas de cómputo, hasta que se hizo evidente el fenómeno conocido como *Power Wall*. Para adaptarse a este problema (y otros, como el *Memory Wall*), la industria del hardware centró su atención en los procesadores multinúcleo (CMP, por sus siglas en inglés), que integran múltiples núcleos (cores) en un solo chip. Los CMP son actualmente la arquitectura de propósito general de facto en la mayoría de plataformas comerciales, desde dispositivos integrados y móviles hasta servidores de alto rendimiento.

Esta tesis propone una serie de estrategias de planificación y gestión de recursos para procesadores multinúcleo que se han implementado a nivel de kernel del sistema operativo (SO) y del *runtime system*. El objetivo principal de estas propuestas es abordar tres retos clave que los sistemas multinúcleo plantean al software del sistema: (i) abordar los efectos de la contención de recursos compartidos, (ii) maximizar la utilización de CPU, y (iii) gestionar la asimetría de rendimiento bajo la presencia de diferentes tipos de núcleos en la plataforma.

En los sistemas multinúcleo, *la contención por recursos compartidos en el sistema* surge de forma natural como resultado de la competición por estos recursos producida al ejecutar varias aplicaciones simultáneamente. Los núcleos en un CMP comparten ciertos recursos clave, como la caché de último nivel (LLC) o el controlador de DRAM. Lamentablemente, esta fuente de contención provoca una degradación del rendimiento desigual e impredecible de las aplicaciones, lo que impacta negativamente en la justicia del sistema, causando numerosos efectos nocivos. Tal disparidad en el rendimiento inducida por contención se agrava en los sistemas NUMA multinúcleo debido a (i) la alta latencia de los accesos a memoria remota y (ii) al impacto de la asignación de los hilos a los distintos sockets o nodos NUMA. Para superar estos problemas, esta tesis propone Divide&Content (DC), una estrategia de gestión de recursos consciente de la justicia que combina el particionado de la LLC con la asignación dinámica de hilos en sistemas NUMA. La combinación estratégica de estas dos técnicas para mitigar la contención permite a DC mejorar sustancialmente la justicia sobre otras estrategias existentes, extrayendo beneficios automáticamente sin requerir modificaciones en las aplicaciones.

Maximizar la utilización de CPU, y de los recursos del sistema en general, es crucial para los proveedores de cloud, que buscan minimizar costes y maximizar la rentabilidad. Complementando los mecanismos existentes para incrementar el uso de recursos, esta tesis propone una nueva estrategia que explota la ejecución elástica

de cargas de trabajo HPC y científicas. Estas cargas de trabajo suelen exhibir un alto grado de paralelismo inherente, lo que les permite utilizar eficientemente múltiples núcleos durante fases significativas de su ejecución. Aprovechando esta característica, nuestra propuesta explota ciclos breves de inactividad durante la ejecución de una carga de trabajo para lanzar de forma oportunista hilos adicionales de aplicaciones elásticas. Esto permite aumentar el grado de utilización de la CPU y la productividad del sistema, con un impacto mínimo en el rendimiento de las aplicaciones que no explotan elasticidad. Nuestro enfoque consigue hacer esto sin modificar las aplicaciones. Para ello, explota soporte de planificación a nivel de SO, así como la cooperación explícita entre el planificador del SO y el *runtime system*.

Esta tesis también se centra en el *diseño de técnicas de planificación para lidiar con la asimetría de rendimiento en procesadores multinúcleo asimétricos* (AMPs, por sus siglas en inglés). Adoptados recientemente en productos comerciales de escritorio, los AMPs combinan núcleos de alto rendimiento, con núcleos más simples y energéticamente eficientes, todos ellos compartiendo un repertorio de instrucciones común que garantiza la compatibilidad con software ya desarrollado (*legacy*). Para abordar los retos específicos asociados a la explotación de los AMPs, esta tesis propone un novedoso soporte de planificación para maximizar el rendimiento, que se implementa en diferentes niveles de la pila de software del sistema. Cabe también destacar que nuestra propuesta combina la planificación de bucles consciente de la asimetría con la interacción entre el SO y el *runtime system*. En esta tesis se realiza también una evaluación experimental exhaustiva de la tecnología Intel Thread Director, que introduce soporte hardware específico en procesadores recientes de Intel para ayudar al planificador del SO a decidir cómo asignar los hilos de la carga a los distintos núcleos disponibles.

Finalmente, un objetivo transversal de esta tesis ha sido simplificar el desarrollo de extensiones de planificación y gestión de recursos en el kernel del SO, lo que ha propiciado la creación del framework de código abierto PMCSched. El uso de PMCSched en cada línea de investigación de esta tesis refleja el papel crucial que el software del sistema juega al abordar los tres mencionados retos de los procesadores multinúcleo.

Palabras clave: kernel Linux, procesadores multinúcleo, planificación, gestión de recursos, runtime systems, particionado de caché, sistemas operativos, elasticidad, maleabilidad, procesadores multinúcleo asimétricos.

Chapter 1

Introduction

Imagine you start an airline and buy your first airplane. Given that most manufacturers, like Boeing and Airbus, design airplanes with flexible layouts in mind, it is up to you to decide how many seats your airplane will have. Other decisions, such as how many flight attendants to hire or meals to carry onboard, can be made accordingly. Once you establish the seating capacity, there is clearly no need to worry over hypothetical scenarios in which the number of passengers changes mid-flight. In contrast, the thought process behind the design of general-purpose computing systems is fundamentally reversed: machines are designed first, and then efforts are made to serve as many users (passengers) as possible through creative distribution of the available hardware resources. This thesis aims to make progress in this direction, seeking to improve our understanding of efficiently managing a set of finite computing resources in current and emerging general-purpose architectures.

This chapter discusses the motivation behind this thesis, presents the state of the art associated with its main research lines, and provides a brief description of the main thesis contributions. The last section of this chapter describes the overall structure of the thesis.

1.1 Motivation and state of the art

The challenge of distributing limited CPU time among applications is as old as operating systems themselves. This problem is addressed through scheduling algorithms, which allocate slots of processing time for the applications executing on the machine. In the last two decades, the discourse around CPU schedulers has seen a revival in both academia and industry [38,53,58,103,107,142,143,146,150,158]. The question is: Why now? To put context to this seemingly sudden surge in interest, we must journey back to the 1980s. Robert H. Dennard, an electrical engineer from Texas, described in a celebrated article [43] a trend observed in the semiconductor industry for many years: as transistors and other components of integrated circuits were scaled down in size, their performance would improve, while power consumption and density remained relatively constant. This meant that as transistors be-

came smaller, their speed and efficiency would increase, but the power required to operate them would stay the same or decrease per unit area. For a long time, this prediction held true, paving the way for a striking technological evolution of consumer devices, such as laptops or smartphones, and of many other general-purpose computing systems. However, around the mid-2000s, Dennard's scaling rule began to break down [47, 68]. The traditional benefits of scaling down transistor's size faced new challenges, related to power consumption and heat dissipation at smaller scales. Specifically, as transistors became smaller and operated at higher frequencies, they generated more heat in a smaller space, giving rise to the *Power Wall* [47,68]. Even more famous than Dennard's prediction was Moore's Law, which stated that the number of transistors on a chip would double approximately every two years. However, Moore's Law also hit a wall [47, 68] due to the physical limitations of CMOS technology. In the last decade, the ability to integrate a greater number of transistors within a given space deviated significantly from the original forecasts of the 80s. Intel's ambitious Tick-Tock model [1], which promised to release a new family of chips every 18 months featuring either a new microarchitecture design or smaller size, had to be abandoned in 2016 [41].

In addition to the aforementioned technological walls, the increasing gap between CPU processing speeds and main-memory (DRAM) access times – referred to as the *Memory Wall* – turned memory-related pipeline stalls into a major issue in effectively exploiting increasingly advanced CPU architectures (i.e., those with high clock speeds and aggressive superscalar and speculative designs), especially when running single-threaded programs [61]. Together, these challenges drove the hardware industry toward adopting multiple, but simpler, CPU cores per system, leading to the emergence of Chip Multicore Processors (CMPs). The number of cores in CMPs has grown alongside their popularity across all sectors, making them the preferred architecture for cloud datacenters [34, 92, 154] and a key building block of HPC systems [36, 167]. Some commercial CMP processors already reach hundreds of cores; take, for instance, the AMD EPYC 9965 processor, which is part of the fifth generation of the AMD EPYC server processor family, featuring 192 cores and handling 384 hardware threads in total. [1]

An increase in the number of cores is also taking place in CMPs for mobile and desktop environments. Interestingly, while current multicore server CPUs are made up of identical cores, asymmetric multicore processors (AMPs) are being gradually adopted in commercial products within the desktop market segment [14, 15, 72]. AMPs – also referred to as heterogeneous single-ISA (Instruction Set Architecture) multicores – combine big high-performance cores with smaller, power-efficient ones, all exposing a shared ISA to software [82]. The common ISA, in conjunction with the general-purpose nature of the AMP cores, allows the execution of legacy (unmodified) software. Due to their promising energy-efficiency benefits with respect to their symmetric counterparts [82], asymmetric multicores began to gain commercial traction initially in the embedded market, with the ARM big.LITTLE asymmetric processor [125] being one of its greatest exponents. The striking potential of AMPs

[1] `https://www.amd.com/en/products/processors/server/epyc/9005-series/amd-epyc-9965.html`

for improved energy efficiency has more recently drawn the attention of major hardware players, giving rise to desktop-oriented products such as the Intel Alder Lake and Raptor Lake processor families [72], and the Apple M1 to M4 system on chips (SoCs) [14, 15].

Besides CMPs, other types of heterogeneous systems and accelerators have gained substantial notoriety in the last decade. GPUs deserve a special mention in the new golden age of artificial intelligence (AI) we are currently experiencing [54]. Although GPUs have existed for decades, nowadays there is a much greater variety of accelerators [68], including established ones like Neural Processing Units (NPUs) and Tensor Processing Units (TPUs) [100]. Furthermore, exploiting them effectively has become easier. Previously, a substantial level of expertise on the architecture of specific GPU models was required to fully tap into their potential (e.g., CUDA programming [111]). Now, the numerous software libraries available make the benefits of such hardware accessible for anyone to use in popular application domains such as cryptocurrency mining, machine learning, scientific simulations, and more [100]. AI-tailored accelerators are specially capable of training and running neural networks like Large Language Models (LLMs) [28, 164], such as ChatGPT [91], due to their ability to efficiently execute matrix operations and other intensive computations in parallel. Nevertheless, this thesis does not exploit these hardware architectures; instead, it focuses solely on CMPs, aiming to mitigate three of their key major challenges: (1) addressing shared-resource contention effects, (2) maximizing CPU utilization, and (3) improving asymmetry-aware scheduling in AMPs.

1.1.1 Addressing shared-resource contention effects

Making the most out of the available cores, and more generally, maximizing resource utilization in cloud datacenters, is paramount to increase revenue and reduce utility costs of cloud providers [6]. Accomplishing this is no minor challenge, provided that typical cloud workloads – most of them latency-critical (LC), such as search engines or AI-based services – usually exhibit highly variable loads caused by irregular diurnal patterns [34, 92, 175]. Even in the HPC arena, fully utilizing all available cores is not always feasible using a single HPC application. Load imbalances among worker processes/threads or the presence of other scalability bottlenecks may lead to serious core infrautilization [36, 137, 167].

Colocating multiple applications on the same physical server (also known as *multi-tenancy*) is now a popular practice for minimizing resource idleness in both HPC and the cloud [34,167]. Unfortunately, co-running multiple applications, virtual machines (VMs), or containers on the same server naturally leads to *shared resource contention* in CMPs [112, 133], which causes a number of undesirable effects that substantially impact the performance of individual applications. This kind of contention stems from the fact that cores in current CMP systems typically share critical resources with the neighboring cores, such as the last-level cache (LLC), memory channels, and DRAM controllers. As an illustrative example, Figure 1.1 shows the various cores and the cache hierarchy of the Intel Xeon Gold 6138 pro-

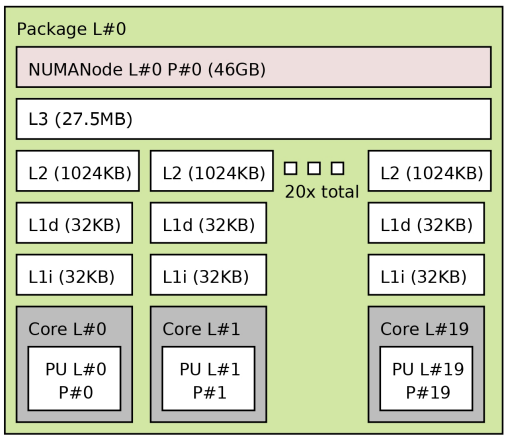


Figure 1.1: Cache hierarchy of the Intel Xeon Gold 6138 "Skylake" processor.

cessor, one of the multicore CPUs used for evaluation in this thesis. Applications running simultaneously and using separate cores of this CPU enjoy per-core private L1 and L2 caches, but they have to compete for space in the shared LLC (L3 cache). On top of this, applications running in a multicore server featuring just one instance of this CPU model (UMA platform) also naturally compete for the available memory bandwidth [103] and may be affected by DRAM-controller contention [104, 174]. The fact that the fraction of various shared resources allotted by the hardware to applications greatly depend on their rate of demand makes the actual resource distribution inherently unfair [136, 171].

In general, shared-resource contention may lead to uneven and hard-to-predict performance degradation across applications, making it challenging to enforce system-wide fairness [136, 152, 171] and QoS (Quality of Service) constraints [34, 112, 154] in current CMP systems. Particularly, an application's completion time or its tail latency may largely depend on the application's co-runners [34, 178]. Moreover, the slowdown (i.e., relative performance degradation compared to an isolated execution) of equal-priority applications that run simultaneously tends to differ significantly across runs under contention [112, 136, 158, 171]. Clearly, such performance disparities are unacceptable from the user satisfaction (*à propos* the airplane metaphor, ensuring legroom) and fairness standpoints [133], also making it difficult to prioritize critical applications [34], offer performance guarantees [112, 174], or ensure correct billings in commercial cloud-like computing services [53]. Notably, uneven thread progress caused by contention may also greatly limit the scalability of parallel applications [58, 167].

One of the main objectives of this thesis has been to deliver fairness (i.e., to even out per-application slowdowns) in NUMA multicores by mitigating shared-resource contention. NUMA CMPs comprise multiple memory nodes, each featuring a set of cores typically sharing an on-chip DRAM controller. In these platforms, local memory accesses happen via the local DRAM controller, whereas remote ones occur via a cross-chip interconnection network [95].

As we demonstrate in this thesis, contention-induced performance disparities are exacerbated in NUMA multicores, because an application's degree of contention is

largely affected by its co-runners, which, in turn, depend on the particular thread-to-core mappings imposed by the user or the system software. To illustrate this issue, we conducted 10 runs of a multi-application workload on a 40-core NUMA system (refer to Section 2.1 for details on this platform). Figure 1.2 shows the resulting distribution across runs of the average slowdowns for each program when scheduled by the default Linux scheduler. The results reveal substantial variability in average slowdown resulting from disparities in the execution time across runs. This is caused by the random thread mappings that the Linux scheduler performs, placing each application with a different set of socket-level co-runners every time the workload is launched. This produces a varying degree of contention across executions. In fact, this variability issue becomes apparent in all CMP platforms featuring multiple LLCs, each shared by different groups of cores; this platform category also includes specific UMA multicore topologies with multiple LLCs in the same chip, like those considered in previous work [171, 176].

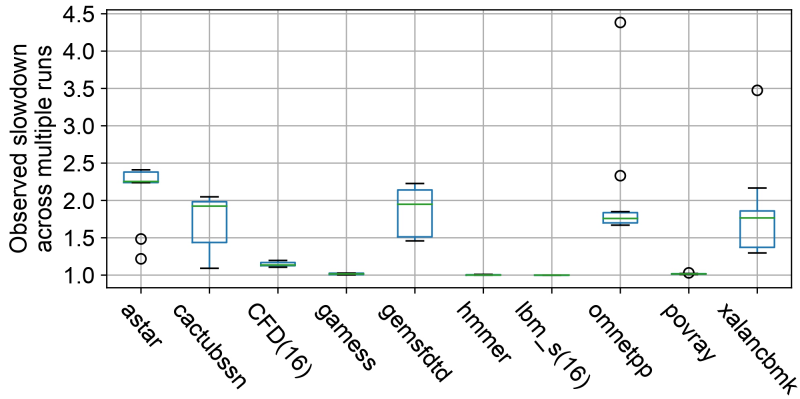


Figure 1.2: Variability of the average observed slowdown across 10 runs of a multi-application workload. Two benchmarks run with multiple threads, their count is listed in parentheses along with the names. The default Linux scheduler was used.

Previous work has aimed to mitigate shared-resource contention effects in CMPs, mostly by exploiting two main types of control mechanisms: (1) contention-aware thread placements, and (2) LLC-partitioning. Proposals leveraging the first mechanism [26, 95, 171, 178] involve mapping threads to groups of cores – sharing a LLC and/or a DRAM controller – so as to ensure a balanced degree of contention across core groups. For these techniques to be effective, the system software has to be cognizant of the underlying hardware and the memory -and LLC- related behavior of the various applications, which may dynamically vary with program phases [178]. Partitioning the LLC enables to impose a certain degree of isolation among applications/VMs [136, 154]. While proposals on cache-partitioning techniques exist since the advent of multicore processors [101, 127], there has been a growing interest in the design of LLC-partitioning policies in recent years [34, 46, 122, 136]. This interest has been strongly motivated by the fairly recent adoption of hardware partitioning extensions in commodity processors [92]. Specifically, current server CPUs from Intel and AMD feature LLC-partitioning support as part of the set of hardware extensions commercially known as Intel Resource Director Technology [2] and the AMD64 Technology Platform Quality of Service Extensions [11], respectively.

Recent implementations of this set of hardware extensions also include support enabling the system software to limit the memory bandwidth consumption of an application, VM, or container, as a means to arbitrate bandwidth competition. Nevertheless, this thesis does not investigate the exploitation of these bandwidth extensions, as they were not implemented in the experimental platforms available to us at the time.

Crucially, prior efforts on shared-resource contention fail to effectively combine contention-aware thread placement and LLC-partitioning. In particular, some techniques only exploit thread-placement strategies [26, 171] but do not partition the LLC. Conversely, most recent partitioning strategies were specifically designed for UMA systems with a single LLC [112,122,133,136], or they perform LLC-partitioning independently within each NUMA socket, regardless of the thread-to-core mappings [34].

To fill this gap, this thesis proposes an OS-level technique that simultaneously leverages dynamic contention-aware thread placement and resource partitioning to enhance system-wide fairness in NUMA multicores. Our research also reveals that addressing thread placement and LLC-partitioning as separate and orthogonal optimization problems results in suboptimal solutions in terms of fairness. This demonstrates that the combination of multiple existing techniques, each exploiting one of these control mechanisms separately, is clearly not the right path when designing an all-encompassing fairness-optimized resource manager.

1.1.2 Maximizing resource utilization in CMP systems

As discussed earlier, colocating multiple applications/VMs on the same multicore server is crucial to improve resource utilization. Moreover, to further reduce resource idleness, cloud providers typically combine colocation with the opportunistic execution of in-house production workloads on the same set of servers devoted to running customers' workloads [33, 92, 116]. Notably, when managing the various server resources dynamically, these in-house workloads are generally treated as best-effort (BE) jobs whose execution can be deferred if necessary [175]. Most state-of-the-art resource-management strategies for cloud workloads are designed to increase resource utilization when possible (through the execution of BE jobs) while avoiding violations of Service Level Agreements (SLAs) associated with cloud services [33, 34, 92, 116, 175].

While existing cloud-oriented resource managers focus on enforcing SLAs for latency-critical (LC) services, they lack specific support to efficiently run HPC and scientific workloads together with cloud services. In general, running HPC workloads efficiently on the public cloud poses significant challenges due to critical divergences amongst cloud services. Particularly, HPC workloads are typically executed in batches rather than as 24/7 services, are not usually subject to strict SLAs [109], and tend to consume more CPU and memory resources, often causing severe contention on shared servers [24, 136]. Moreover, as we discuss next, HPC workloads are not amenable to current cloud mechanisms for *elasticity*, that is, the ability to

dynamically leverage a varying amount of resources (CPUs, VMs, etc.) across one or multiple servers.

Today, elasticity is crucial to dynamically assign on-demand cloud resources to an application in a cost-effective way, without resource over-provisioning [10]. Elasticity can be exploited horizontally by increasing the number of instances of the application using multiple containers/VMs (the horizontal arrow in Figure 1.3), or vertically by dynamically increasing the associated CPU and memory resources (the vertical arrow in Figure 1.3). Current cloud elasticity mechanisms are typically engaged upon substantial fluctuations in the application's load, namely the amount of external requests that an application receives [6]. Examples of this include Amazon's Elastic Compute Cloud (Amazon EC2 [2]), which provides on-demand infrastructure, or the DigitalOcean Elasticity tool [3], which monitors the server's CPU usage and will create a new *droplet* (a VM instance) if needed. Previous research has highlighted that, while this is suitable for client-server applications, it does not meet the needs of workloads that do not depend on external requests, such as scientific applications [56]. Another limitation of current cloud dynamic elasticity mechanisms is that they are generally applied at coarse granularities [6], thus making it very difficult to effectively utilize cores that go idle just for a few seconds.

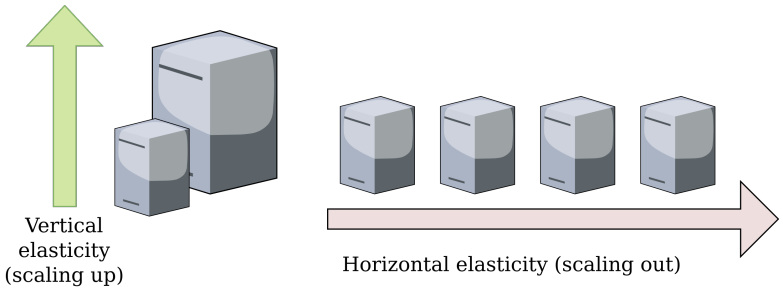


Figure 1.3: Vertical and horizontal elasticity scaling of resources.

In this thesis, we explore the design of a fine-grained elasticity mechanism that targets HPC and scientific workloads. We show that these workloads harness ample potential to opportunistically use idle cores thanks to their inherent amount of parallelism. Specifically, our proposed elasticity mechanism leverages dynamic *malleability*, which is the capability to utilize a varying number of processes/threads at runtime to adapt to fluctuations in resource availability [10, 16]. A key aspect of our approach to elasticity is its reliance on the synergistic interaction between the runtime system and the OS scheduler to increase resource utilization without requiring changes in the applications.

1.1.3 Making effective use of asymmetric multicore systems

Asymmetric multicore systems are also subject to the aforementioned issues of conventional homogeneous multicores, but introduce an additional and unique set

[2] `https://aws.amazon.com/ec2/`
[3] `https://www.digitalocean.com/community/tools/digitalocean-elasticity`

of challenges that arise from the combination of multiple core types (big and small) in the same platform. An important issue is the fact that the relative performance (or *speedup*) that each application or individual thread derives from running on a big core versus a small one can substantially differ among applications and even vary across program phases within the same application [58,79]. For instance, in the Intel Alder Lake processor used in this thesis, we observed program phases where using big cores provided nearly no performance gains over small cores, while other program phases derived speedups higher than 6x. By leveraging information from extensive offline analyses, some applications can be hand-tuned (or built from scratch) to take these relative performance disparities into consideration to maximize performance or optimize other key objectives, such as energy consumption. However, in this thesis, we explored ways to automatically deliver the benefits of AMPs to unmodified applications, which is widely acknowledged as one of the main challenges to be addressed by the system software [62, 102].

To accomplish our goals for scheduling in AMPs, the various system software layers – traditionally designed assuming that the platform comprises identical cores – require profound changes [88, 128]. Previous research has incorporated asymmetry awareness separately into different system-software levels, particularly within the OS scheduler [58,88,143], at the runtime system level [37,138], or in the VMM monitor [77,84]. Previous work has demonstrated that one of the most critical factors in making smart scheduling decisions on AMPs is to take into account the big-to-small speedup derived by each thread or virtual CPU on the system. This value, also referred to as the *speedup factor* (SF), is crucial when it comes to maximizing system throughput and fairness [88, 147]. For example, in workloads consisting of diverse single-threaded applications, the system throughput is maximized by devoting big cores to running those threads in the workload that exhibit higher SF values, while relegating low-SF applications to small cores [21, 82, 139]. Similarly, awareness of threads' SFs is critical to ensure that each thread in the workload makes an even amount of progress when scheduled on a set of asymmetric cores. Previous research shows that guaranteeing even thread progress is vital to deliver system-wide fairness on AMPs [58, 143, 165]. In the context of runtime-system level scheduling, the SF also plays a key role when balancing the load across worker threads, which is crucial to achieve good scalability and performance in parallel programs [77, 138, 140].

Despite the importance of thread SFs in maximizing the potential of asymmetric multicores, obtaining accurate SF values at runtime remains a challenging task for system software [146]. The fact that SFs vary across program phases further complicates this endeavor. Therefore, equipping the system software with a robust mechanism to approximate thread SF is still today a critical challenge [146]. Existing techniques to determine thread SFs at runtime can be categorized into three groups: direct measurement, utilization of prediction models based on performance monitoring counters (PMCs), and reliance on special hardware extensions specifically designed for SF estimation [146].

The direct method, known as *IPC sampling* [82, 165], involves measuring the instructions per cycle (IPC) completed by a thread on both core types using PMCs. The SF can then be approximated with the IPC ratio, factoring in the potentially

different frequencies of the various cores. Although this approach seems straightforward, it generally incurs higher overhead than other techniques due to the thread migrations specifically required to measure the IPC on the various core types. More importantly, the fact that phase-changes may occur while measuring IPC values on different core types [148], coupled with the effects of shared-resource contention [58] may lead to inaccurate SF predictions.

In the quest for SF-estimation techniques that do not require thread migrations for performance measurement, other researchers have proposed the use of SF *estimation models* based on performance metrics gathered with PMCs, such as the LLC miss rate, percentage of stall cycles, etc. In exploiting these techniques, two separate platform-specific models must be built: one for predicting SF when running on the big core and another for prediction from the small core. Early strategies relying on PMC-based prediction models leveraged a subset of PMC metrics that exhibit a strong correlation with the SF on the various core types [58, 79, 125]. Extensive offline analyses were required to identify key metrics for building the model. Recently, more elaborate estimation models have emerged, leveraging machine-learning techniques [17, 30, 107, 139, 143, 144].

To the best of our knowledge, Van Craeynest et al. [166] were the first to propose the utilization of specific hardware extensions to feed the thread scheduler with SF predictions and produce estimations. Although this particular hardware support has not yet been adopted in commercial processors, the idea of building specialized hardware for SF prediction has crystallized in the Intel Thread Director Technology [71, 131]. This set of hardware extensions has been designed by Intel to provide the OS with SF estimates at runtime. Thread Director was first introduced in the 12th generation of Intel Core processors, codenamed as "Alder Lake", which integrates big and small cores [102].

In this thesis, we have introduced the necessary support in the Linux kernel to carry out a comprehensive evaluation of Intel Thread Director as a tool to guide thread scheduling. We should highlight that even at the time of this writing (December 2024), the Linux kernel lacks official support in *mainline* to leverage this hardware extension from the process scheduler. As a consequence, the Linux scheduler does not use Thread Director in any capacity. More importantly, our research encompasses the first set of publicly available scientific contributions evaluating the effectiveness of Intel Thread Director at the OS scheduler level [23, 25, 131].

This thesis also addresses an important aspect not explicitly covered by previous work on asymmetry-aware scheduling: the exploitation of the synergistic interaction between different system-software layers for improved scheduling on asymmetric multicores. While a large body of scheduling techniques has been proposed for AMPs, they are adopted in a single layer of the system software, such as inside the operating system kernel [58, 79, 87, 142], as part of a user-space scheduling prototype [117, 118] (as a shortcut to evaluate an OS-level scheduler), or within the runtime system [38, 138, 163] (i.e., user-space library). Specifically, this thesis proposes a mechanism that leverages explicit interaction between the runtime system and the OS scheduler to maximize system throughput on AMPs. Our proposed

runtime-level extensions perform asymmetry-aware loop scheduling leveraging OS-level hints, making it possible to substantially improve the performance of regular loop-based OpenMP programs in single- and multi- application scenarios.

1.2 Thesis contributions

A common aspect of many of the proposals in this thesis is their reliance on hardware extensions for system monitoring, resource management, and scheduling, which are present in off-the-shelf processors. This includes performance monitoring counters [145], hardware support for software-controlled cache-partitioning [2, 11], and novel technologies like Intel Thread Director [23, 131]. All these facilities are controlled through special registers that can only be accessed directly in privileged processor modes, where the operating system (OS) kernel or virtual machine monitors typically operate. Thus, the OS stands out as the ideal playground for implementing mechanisms reliant on hardware extensions, specifically for scheduling and resource management in multi-application scenarios.

Aware of the difficulties of developing within the OS level, many researchers turn to user-space prototypes for scheduling and resource management [26, 53, 150, 177]. However, this approach is not without its significant drawbacks. These include: (i) the additional context-switch overhead from continuously gathering monitoring data from the OS, which limits scalability [136], (ii) the inability to observe and quickly respond to low-level scheduling events, such as thread transitions between runnable and non-runnable (e.g., blocking) states [129, 134], and (iii) the lack of access to new hardware extensions that are not yet exposed through the OS kernel [23]. The choice then becomes accepting the limitations of user-space (which may be impractical for resource management in production environments) or facing the difficulty of kernel development. In this thesis, all aforementioned drawbacks from user-space became apparent while developing the proposed scheduling and resource management strategies. Therefore, we opted to implement most of them in the OS kernel.

In this thesis, we employed the Linux kernel – the most common OS kernel baseline for system software research – as a vehicle for showcasing the effectiveness of our different scheduling and resource management proposals. Working at this level introduces additional development complexity compared to user-space. For example, an important barrier is that adding new scheduling algorithms in the Linux kernel requires modifying the core kernel, recompiling it, and rebooting the system for each change to take effect. Such workflow is highly time-consuming [146]. Moreover, OS developers historically lack the layered abstractions and safety nets available in user-space, which exposes them to the risk of minor errors leading to system-wide crashes. To navigate these complexities and deliver programming productivity in this context, we decided to find ways to ease kernel development. The result of these efforts was the creation of the PMCSched framework, which is introduced below.

To summarize, the primary contributions of this thesis are as follows:

1. **Ease development of scheduling-related kernel support**. We created PMCSched, an open-source framework that enables rapid development of scheduling algorithms and custom resource managers in the Linux kernel. It also features built-in support for seamless cooperation with user-space runtime systems. In PMCSched, new scheduling and resource management features can be implemented as a *plugin* within a kernel module, that can be loaded into unmodified (vanilla) versions of the kernel. Although developed as part of this thesis, PMCSched is based on the PMCTrack tool (introduced in Chapter 2), and it leverages PMCTrack's kernel-level APIs for seamless access to PMCs [145] and hardware cache-partitioning extensions [136]. For each research use case (the remaining contributions), PMCSched was leveraged, adapted, and improved to meet the specific needs of the use case. PMCSched will be introduced in Chapter 3 and utilized extensively throughout this thesis.

2. **Mitigate contention in CMPs**. We propose a fairness-aware OS-level resource manager, referred to as Divide&Content (DC), that combines the two main contention-mitigation techniques: LLC partitioning and thread-to-core assignments. DC's design was possible thanks to the insights of a comprehensive simulation study, which we conducted to gain a deep understanding on how to best combine the two aforementioned techniques. Notably, DC is the first NUMA-aware contention-conscious resource manager to simultaneously exploit LLC-partitioning and dynamic thread placement, which (as demonstrated in this thesis) cannot be treated as orthogonal problems. DC's effectiveness was evaluated on a real dual-socket NUMA system, using a wide range of workloads consisting of long-running applications. The results show that DC reduces unfairness by more than 17% on average compared to both Linux and a state-of-the-art NUMA-aware resource manager.

3. **Apply elasticity to improve CPU utilization**. This thesis introduces a novel vertical elasticity strategy to improve CPU utilization by leveraging the execution of HPC and scientific workloads. Our approach opportunistically executes additional threads from elastic applications during brief idle core cycles left by *regular* applications (without elasticity support). This increases CPU utilization and throughput, and delivers performance gains to elastic applications with minimal impact on the performance of non-elastic applications. This proposal represents the first attempt, to our knowledge, to leverage OS-runtime cooperation to exploit vertical elasticity as a means to improve CPU utilization under mixes of unmodified elastic and non-elastic applications. As a proof of concept to bring elasticity support to unmodified programs, we also proposed a set of malleability extensions in the OpenMP runtime system that target regular loop-based OpenMP programs.

4. **Design of novel scheduling support to improve performance on AMPs**. Regarding asymmetry-aware scheduling, we (i) comprehensively evaluated the potential of the Intel Thread Director (TD) technology in guiding scheduling in AMP systems, (ii) revisited asymmetry-aware loop scheduling in OpenMP

applications for improved performance portability on AMPs, and (iii) devised novel mechanisms to further improve performance via explicit interaction between the runtime system and the OS scheduler. Notably, to make our evaluation of the TD technology possible, we addressed the lack of TD support in the Linux kernel by developing an open-source PMCSched extension, which constitutes another contribution of this thesis. To evaluate our asymmetry-aware scheduling proposals, we performed extensive evaluations on an Intel Alder Lake processor.

1.3 Thesis structure

The remainder of this thesis is organized as follows:

- Chapter 2 introduces the experimental multicore platforms, tools, metrics, and experimental methodology used throughout the thesis.
- Chapter 3 provides an extensive summary of the thesis's main contributions, which address the aforementioned key challenges of CMPs: shared-resource contention, maximizing CPU utilization, and asymmetry-aware scheduling. It also details the purpose and architecture of PMCSched.
- Chapter 4 includes the journal article entitled "*Divide&Content: A Fair OS-Level Resource Manager for Contention Balancing on NUMA Multicores*".
- Chapter 5 contains the conference paper entitled "*Exploiting Elasticity via OS-runtime Cooperation to Improve CPU Utilization in Multicore Systems*".
- Chapter 6 includes the journal article entitled "*Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake*".
- Chapter 7 concludes the thesis, presents its main takeaways, and discusses promising future research directions on scheduling and resource management.
- Appendix A includes the workshop paper entitled "*Rapid development of OS support with PMCSched for scheduling on asymmetric multicore systems*". Note that the journal article found in Chapter 6 is an extended version (special issue of journal) of this earlier workshop paper.
- Appendix B includes the workshop paper research article entitled "*Revisiting Fairness and Throughput Metrics for Cache-Partitioning Policy Assessment: Insights and Recommendations*". This article constitutes a secondary contribution of the thesis discussed in detail in Section 2.3.1.
- Appendix C enumerates all the thesis publications.

Chapter 2

Experimental Platforms, Tools, and Evaluation Methodology

In this chapter, we first present the various platforms employed for the experimental evaluation of the different thesis proposals. Next, we describe the primary tools and technologies that were routinely leveraged throughout this thesis, many of which were augmented to meet the specific needs of our research objectives. Particularly, this chapter offers an overview of PMCTrack, PBBCache, Intel CAT, LFOC+, and Het-Harness. The chapter concludes with an overall discussion of our evaluation methodology and with an introduction of the metrics used for the assessment of the effectiveness of the different proposals.

2.1 Experimental platforms

We primarily utilized four experimental platforms during the evaluation of the proposals in this thesis, which we label for easy reference in the subsequent sections and chapters. For clarity, the name of each platform includes the processor microarchitecture and the platform's core count. The platforms we leveraged throughout this thesis are as follows:

1. `Skylake-40`. It is a dual-socket NUMA machine with 96 GB of DRAM, integrating two 20-core Intel Xeon Gold 6138 ("Skylake") processors, where cores run at 2 GHz. The topology of one of its NUMA nodes is shown in Figure 2.1. Each processor features an 11-way 27.5 MB LLC (L3), and supports the Intel Resource Director Technology (RDT) [2], which enables LLC-partitioning. All cores have two private cache levels (a 64 KB L1 and a 1 MB L2).

2. `CascadeLake-16`. This is a UMA system with 64 GB of DRAM which features a 16-core Intel Xeon Gold 5218 ("Cascade Lake") processor, where cores run at 2.3 GHz. Its topology is presented in Figure 2.2. All cores feature two private cache levels (64 KB L1 + 1 MB L2) and share a 22 MB last-level (L3) cache.

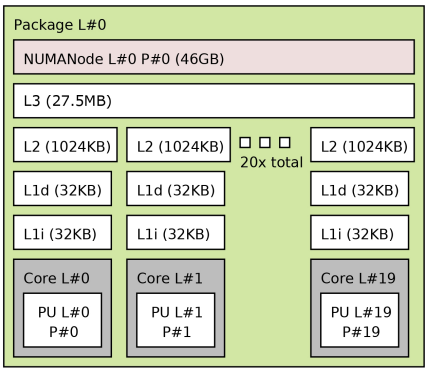


Figure 2.1: Topology of one of the two NUMA nodes of `Skylake-40`.

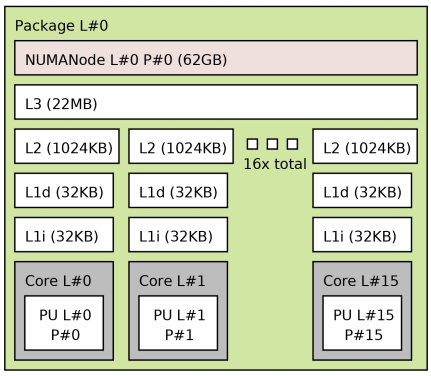


Figure 2.2: Topology of the `CascadeLake-16` platform.

3. `AlderLake-16`. This platform – with 32 GB of DRAM – includes an Intel Core i9-12900K ("Alder Lake") processor. Unlike the processors present in the other platforms employed in the thesis, Intel Core i9-12900K is an asymmetric multicore processor (AMP), which combines big and small cores. Particularly, it integrates 8 "Golden Cove" big cores, and 8 "Gracemont" small cores. Intel's nomenclature refers to big cores as P-cores (for performance) and to small cores as E-cores (for efficiency). P-cores are more powerful – from the performance and power consumption standpoints –, and are designed to handle more demanding, single-threaded tasks. Conversely, E-cores are optimized for energy efficiency, and are better suited to multithreaded workloads. The processor's topology is depicted in Figure 2.3. E-cores (cores 8-15) are grouped into two 4-core clusters, where each group shares a 2 MiB L2 cache. By contrast, each P-core (cores 0-7) has a private 1.25MiB L2 cache. All platform cores integrate a private L1 cache but share a 30 MB L3 (LLC) with the remaining E and P cores.

4. `Zen2-128`. This 128-core NUMA multicore server, equipped with 512 GB of DRAM, integrates two AMD EPYC 7742 processors (Zen2 microarchitecture), each one featuring 64 cores. The cache and memory hierarchy of this processor differs substantially from that of the other processors used in this thesis. Figure 2.4 depicts the multi-chip design of the AMD EPYC 7742 processor, which consists of nine interconnected dies. One of these dies, referred to as the I/O Die (IOD), provides access to main memory and I/O. The 8 remaining dies include the various cores and caches; in AMD's terminology, these dies are known as *Core Complex Dies* (CCDs). Each CCD in this processor model is further divided into two independent parts, each comprising 4 cores and a shared 16 MB L3 cache. Each 4-core segment of the CCD is referred to as a *Core CompleX* (CCX). `Zen2-128` allows the L3 cache of each CCX to be partitioned separately and supports memory-bandwidth monitoring and allocation. These features are enabled by hardware resource-management facilities, commercially known as AMD64 Technology Platform Quality of Service Extensions [11].

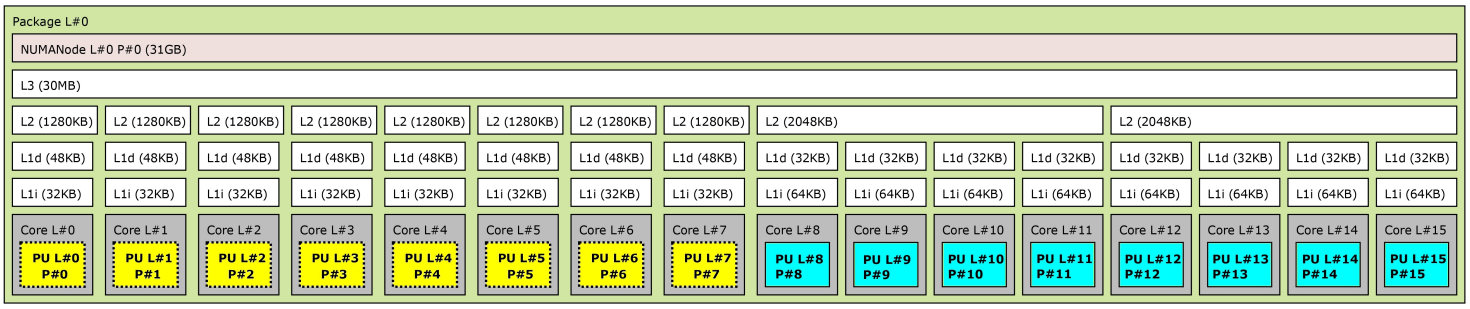


Figure 2.3: Topology of the `AlderLake-16` platform.

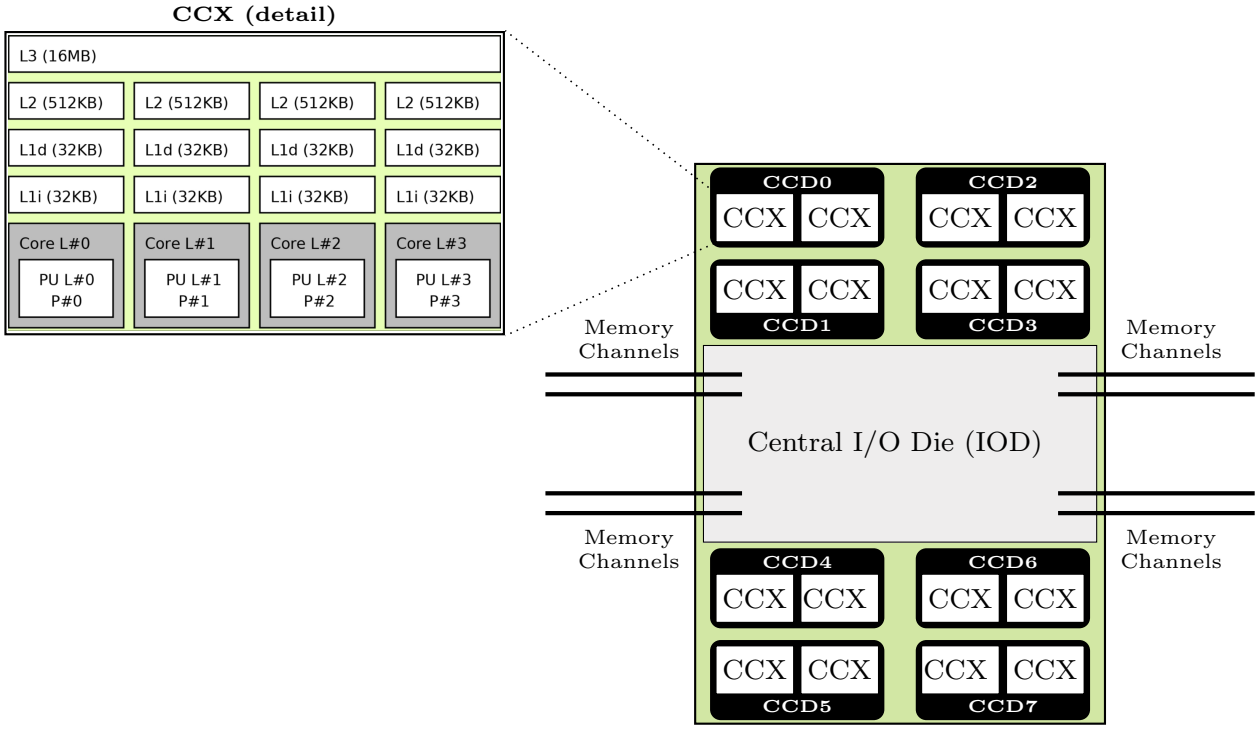


Figure 2.4: Topology of one socket of the `Zen2-128` platform.

The processor models present in all platforms mentioned above support a form of Simultaneous Multithreading (SMT), a hardware feature enabling us to leverage up to two logical processors per core. However, we disabled SMT in the experiments conducted in this thesis, as it is known to substantially impact the performance of co-running applications, particularly when allowing the Linux scheduler to automatically assign threads to logical processors [51, 53, 93, 141]. Disabling SMT or simply spawning a number of threads in the workload that does not exceed the physical core count (which has a similar effect to disabling SMT) is a standard practice in recent research evaluating strategies similar to those this thesis focuses on, including cache-partitioning policies [46, 112, 121, 133, 136, 152], schedulers for asymmetric multicores [19, 58, 137, 146, 147], and elasticity and co-execution methods [36, 159, 179]. Hence, to eliminate SMT contention and the associated performance variability when running CPU-bound workloads, only one logical processor per core is exposed to the operating system in our setup.

2.2 Technologies and Tools

2.2.1 PMCTrack

PMCTrack [135] is a versatile open-source performance monitoring tool for GNU/Linux, actively utilized in a variety of academic research on systems software for more than a decade [98,135,136,143,157,160]. It was mainly built to assist kernel developers in performing OS-level optimizations, mostly related to scheduling and resource management. The most distinguishing feature of PMCTrack is that it provides seamless access to hardware performance monitoring counters (PMCs) for on-the-fly optimizations, offering an interface that hides the underlying architecture. This feature makes performance monitoring and optimization, both for sequential and parallel applications, more approachable across all layers of the software stack when aided by PMCTrack.

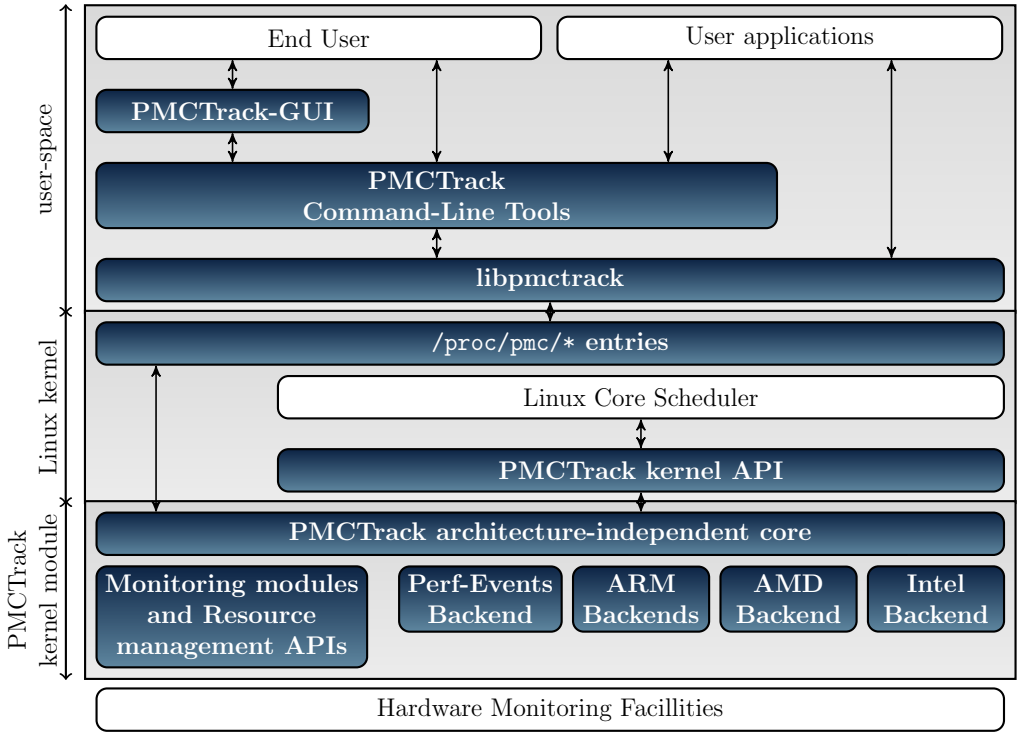


Figure 2.5: Architecture and main components of the PMCTrack tool (before the adoption of the PMCSched framework).

The effectiveness of hardware PMCs to characterize application performance has been well-established; extensive research has demonstrated the operating system's ability to perform effective runtime optimizations in multicore systems by leveraging PMC data, with a particular focus on optimizations within the OS scheduler [57, 78, 99, 118, 136]. In this direction, PMCTrack allows the OS scheduler to access per-thread PMC data with an architecture-agnostic in-kernel API, enabling seamless support for a wide range of processor families. Additionally, PMCTrack offers access to other hardware-oriented metrics available in recent commercial platforms that go beyond typical PMC events, such as cache occupancy,

memory-bandwidth consumption, and energy consumption. This additional monitoring information is exposed to user-space as a set of *virtual counters*.

PMCTrack's architecture is depicted in Figure 2.5. This tool consists of both user-space and kernel-space components, being the vast majority of its functionality implemented in a loadable kernel module. User-space components make it possible to retrieve both PMC-based and virtual counter information from the command line (via the `pmctrack` tool), from any instrumented C/C++ program – by leveraging the `libpmctrack` library or via a graphical frontend, referred to as *PMCTrack-GUI*. More information on the various use cases of PMCTrack from user-space can be found on its official website [120]. PMCTrack's kernel module consists of three major building blocks: (1) a set of *PMC backends*, (2) a collection of *monitoring modules*, and (3) an architecture independent-layer, enabling both the OS scheduler and user-space components to periodically access HW monitoring information with a per-thread or system-wide granularity. Each PMC backend bundles the necessary low-level support to access PMCs directly on a given processor architecture. By contrast, monitoring modules are in charge of managing other hardware monitoring information, which is typically exposed to the end user via the virtual counters abstraction.

Monitoring modules play a critical role in PMCTrack's extension capabilities. Apart from enabling the creation of necessary support to interact with new hardware monitoring facilities, the *monitoring module* abstraction decouples scheduling algorithms from the architecture- and platform- specific gathering of monitoring data. To do so, PMCTrack's architecture-independent layer exports a set of high-level metrics (each with an integer ID) and functions that allow a scheduling algorithm implemented in the kernel to seamlessly retrieve the required metric by simply referencing its ID. This allows the construction of architecture-independent schedulers on Linux that leverage hardware monitoring information; by using PMCTrack's API, the scheduler's code remains agnostic to the underlying platform-specific method required to collect that information. Crucially, PMCTrack's API allows to access PMCs from *interrupt context*, where key scheduler functions run, ranging from tick processing to the selection of the next process to run. To make this possible, this API avoids the use of blocking synchronization mechanisms, unlike other alternatives to access PMCs in the kernel, like *Perf Events*, which cannot be used directly in scheduler code.

The ability to extend PMCTrack by developing new monitoring modules opened the door to the creation of the PMCSched framework [23], a key contribution of this thesis described in detail in Chapter 3. PMCSched leverages PMCTrack's API as well as the scheduling-related callbacks that make up the interface of operations associated with a monitoring module. Nevertheless, the design efforts behind PMCSched, which is now part of PMCTrack, have crystallized in substantial improvements in the latter, starting from version 3.1. Chapter 3 delves into PMCSched's design and discusses further benefits stemming from the adoption of this framework in PMCTrack.

2.2.2 Intel CAT

In this thesis, we have leveraged the hardware cache-partitioning extensions available in the `Skylake-40` platform (presented in Section 2.1). These extensions are commercially known as Intel Cache Allocation Technology (CAT) [110], and are part of the set of technologies that comprise Intel RDT [1]. A CAT-enabled processor allows the system software to gain control over the allocation (partitioning) of the cache for different cores or applications. CAT allows to partition the LLC (L3) or the L2, depending on the specific processor model. In this thesis, we focus on LLC-partitioning.

Intel CAT allows to partition the LLC with way-partitioning, enabling the system software to impose a certain degree of isolation among applications/VMs [136, 154]. This is achieved by assigning applications/VMs to Classes of Service (CLOSs). In processors supporting LLC-partitioning, each CLOS is associated with an LLC partition, which can be configured via a bit mask. This mask is referred to as a Capacity BitMask (CBM); if the i-th bit in the CBM is enabled, the i-th way of the LLC belongs to the associated cache partition, and so it can be used to accommodate new cache lines upon an LLC miss.

Intel CAT-enabled processors expose a number of Model Specific Registers (MSR), allowing the operating system or VMM to partition the LLC. Specifically, for each LLC on the system, a set of registers (as many as the maximum number of supported CLOSs) is available to specify the CBM associated with each CLOS. Moreover, a per-core register is reserved to indicate the CLOS ID of the currently running thread/vCPU. When a context switch takes place, the OS/VMM is responsible for updating the value of the associated CLOS register in that core.

To illustrate how CAT works, let us consider the LLC-partitioning scenario in Figure 2.6, where eight single-threaded applications (`a0-a7`) run on an 8-core system with a partitioned 11-way LLC (like the one on `Skylake-40`). This constitutes an example of *strict* cache-partitioning, where each program has a separate LLC partition, denoted in a different color. All programs are assigned a 1-way partition, except from `a7`, which is allotted a 4-way partition. Table 2.1 shows how to configure the CBMs associated with each CLOS ID to arrive at the partitioning scenario depicted in Figure 2.6 with Intel CAT.

Notably, Intel CAT makes it possible to create overlapping cache partitions – by enabling the same bits in multiple CBMs – and allows multiple programs/VMs to be assigned to the same LLC partition. This opens the door to the implementation of cache-clustering policies [46, 57, 152]. This kind of policies, also referred to as *partition sharing* strategies, extend the concept of strict cache-partitioning by allowing groups of applications (aka *clusters*) to share cache partitions rather than assigning each application its own dedicated partition. Prior research has shown that in systems that allow only a very limited number of partitions, or where only

[1] Intel RDT hardware facilities also allow control over the bandwidth allocation through the Memory Bandwidth Allocation (MBA) technology. However, the processor in our `Skylake-40` platform does not implement this specific hardware feature.

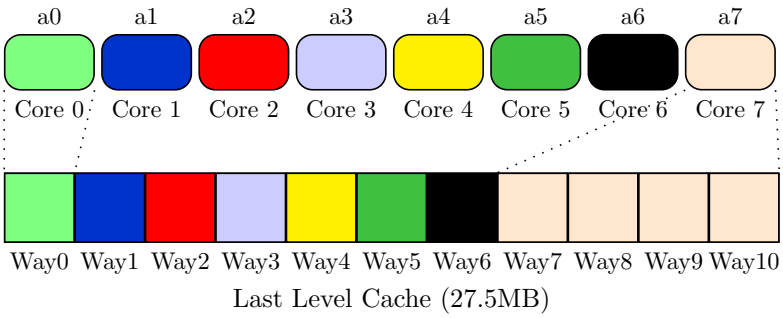


Figure 2.6: Example of strict cache-partitioning, with 8 applications (`a0-a7`) each running on a separate core and LLC partition.

	Capacity Mask's Bits										
Class of Service	0	1	2	3	4	5	6	7	8	9	10
CLOS 0 (a0)	**1**	0	0	0	0	0	0	0	0	0	0
CLOS 1 (a1)	0	**1**	0	0	0	0	0	0	0	0	0
CLOS 2 (a2)	0	0	**1**	0	0	0	0	0	0	0	0
CLOS 3 (a3)	0	0	0	**1**	0	0	0	0	0	0	0
CLOS 4 (a4)	0	0	0	0	**1**	0	0	0	0	0	0
CLOS 5 (a5)	0	0	0	0	0	**1**	0	0	0	0	0
CLOS 6 (a6)	0	0	0	0	0	0	**1**	0	0	0	0
CLOS 7 (a7)	0	0	0	0	0	0	0	**1**	**1**	**1**	**1**

Table 2.1: Cache partitioning of 11 ways among 8 applications (`a0-a7`) using classes of service (CLOSs), with no two applications sharing CLOS.

large, coarse-grained partitions (on the order of megabytes) can be created, cache clustering is more effective than strict partitioning as the number of applications increases [57,59,152]. This is because clustering enables a finer distribution of cache resources by allowing multiple applications to share cache ways, leading to more efficient use of the cache.

To illustrate a partition-sharing scenario, let us consider an example with eight applications (`a0`–`a7`), grouped based on certain shared characteristics such as their cache-access patterns. In this example, shown in Figure 2.7, `a0`–`a4` receive just one shared cache way, while applications `a5` and `a6` share three cache ways. Finally, `a7` is assigned the seven remaining cache ways. This distribution of the LLC space can be imposed with CAT through the CLOS assignments detailed in Table 2.2: `a0`–`a4` share CLOS 0, `a5` and `a6` share CLOS 1, and `a7` is exclusively assigned to CLOS 2.

In this thesis, we leverage Intel CAT together with hardware PMCs. PMCs offer

	Capacity Mask's Bits										
Class of Service	0	1	2	3	4	5	6	7	8	9	10
CLOS 0 (a0 - a4)	**1**	0	0	0	0	0	0	0	0	0	0
CLOS 1 (a5, a6)	0	**1**	**1**	**1**	0	0	0	0	0	0	0
CLOS 2 (a7)	0	0	0	0	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Table 2.2: Cache partitioning among 8 applications (`a0-a7`) using classes of service (CLOSs), in which `a0-a4` share `CLOS 0`, `a5` and `a6` share `CLOS 1`, and `a7` is the only assignment to `CLOS 2`.

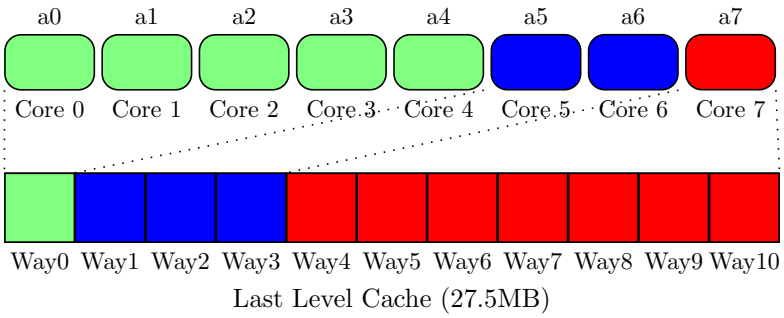


Figure 2.7: Example of partition sharing, where four applications (`a0-a4`) share `Way 0`, the next two share three ways, and `a7` has exclusive use of the remaining seven cache ways.

valuable insights into the cache-access patterns of applications, enabling dynamic, on-line guidance for cache-partitioning algorithms [57, 152]. In particular, the PMC-Track tool is equipped with in-kernel APIs to access both kind of hardware facilities from a loadable kernel module. This greatly simplifies the development of dynamic PMC-aided cache-partitioning policies in the Linux kernel.

2.2.3 The PBBCache simulation tool

PBBCache [59] is an open-source tool for rapid prototyping of LLC-partitioning policies. For a given multi-program workload and partitioning strategy, PBBCache allows obtaining the degree of throughput and fairness under different LLC and system configurations. To achieve this, it relies on offline-collected applications' performance data – such as instructions per cycle, LLCMPKI, etc. – obtained beforehand for different LLC sizes on a target platform. This information, the LLC-space distribution enforced by the partitioning strategy, and other system features (e.g., maximum memory bandwidth) are used by PBBCache to approximate the slowdown an application suffers as a result of both LLC sharing and competition for memory bandwidth [59]. In particular, to compute the slowdown, PBBCache considers both the number of assigned ways and the degree of bandwidth contention, applying a variant of the probabilistic model proposed by Morad et al. [103]. Subsequently, it determines different system-wide metrics to assess the effectiveness of the partitioning strategy. Moreover, PBBCache implements a parallel algorithm to find the optimal partitioning/cache-clustering solution for diverse optimization objectives. PBBCache served as a valuable resource during this thesis for devising and evaluating LLC partitioning policies, with the ultimate goal of addressing contention in CMPs.

Figure 2.8 illustrates the usage of PBBCache with an example of its inputs and outputs. As a command-line tool, PBBCache takes two inputs: a workload file and a metrics file. The workload file (`workloads.csv` in Figure 2.8) specifies the benchmarks for each workload, listing the applications to be considered. The metrics file (`metrics.csv` in Figure 2.8) contains runtime metrics collected offline for each benchmark, such as cache miss rates, memory bandwidth consumption, and their respective IPCs for different LLC sizes. Using these inputs, the simulator outputs

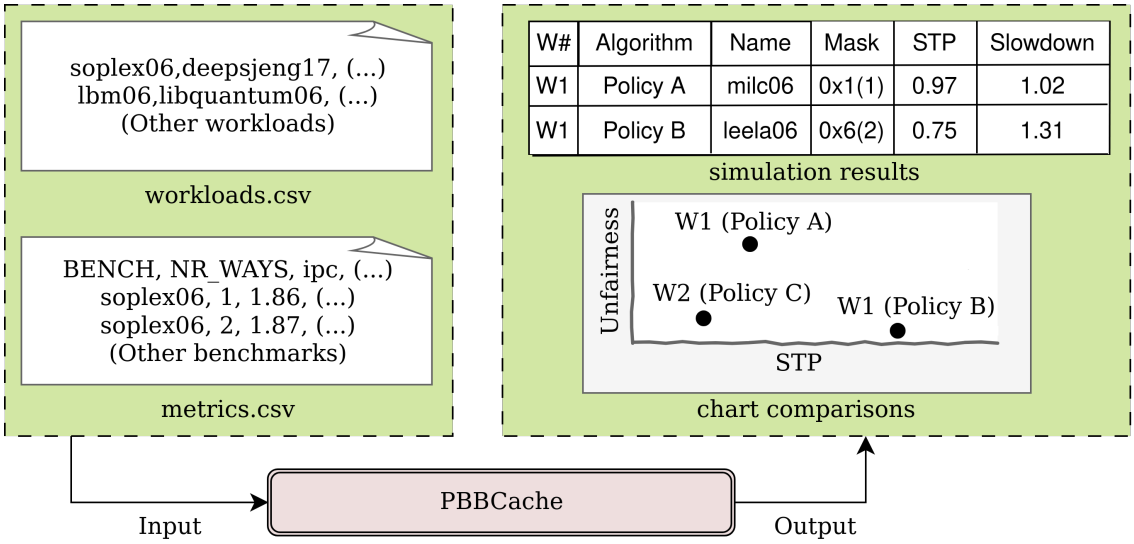


Figure 2.8: Input files for PBBCache (left side, `workloads.csv` and `metrics.csv`) and generated output (right side, simulation results and graphics).

the number of cache ways allocated for each partitioning algorithm considered, as well as the applications' slowdown.

Initially designed to simulate systems with a single LLC, we extended PBBCache to meet the requirements of this thesis. Our modifications included: (1) extending the slowdown prediction model to enable slowdown estimation on multi-core systems with separate LLCs (such as `Zen2-128` or `Skylake-40`), (2) enabling the evaluation of strategies that combine two different contention-mitigation techniques (this is explained in Section 3.2.2), and (3) developing a parallel optimizer to determine the optimal usage of those techniques.

2.2.4 LFOC+

During this thesis, we also leveraged LFOC+ [136], an OS-level cache-partitioning strategy built on top of PMCTrack. LFOC+ is a near-optimal fairness-aware strategy that aims to minimize system unfairness by guaranteeing even slowdown across applications while delivering acceptable system throughput.

Prior to describing how LFOC+ functions, let us introduce the three application categories it relies on: *cache-sensitive*, *streaming*, and *light sharing*. Cache-sensitive programs are those particularly susceptible to LLC contention, and their performance degradation increases substantially as their LLC-way allotment is reduced. The streaming class encompasses bandwidth-intensive programs that incur a high number of LLC misses per 1K cycles (LLCMPKC), and exhibit both a low LLC-reuse rate and a low performance penalty for nearly all way allocations when in isolation; these programs often degrade the performance of cache-sensitive applications co-located on the same LLC partition. Light-sharing programs are neither cache sensitive nor contentious to others, as most of their working set fits within the

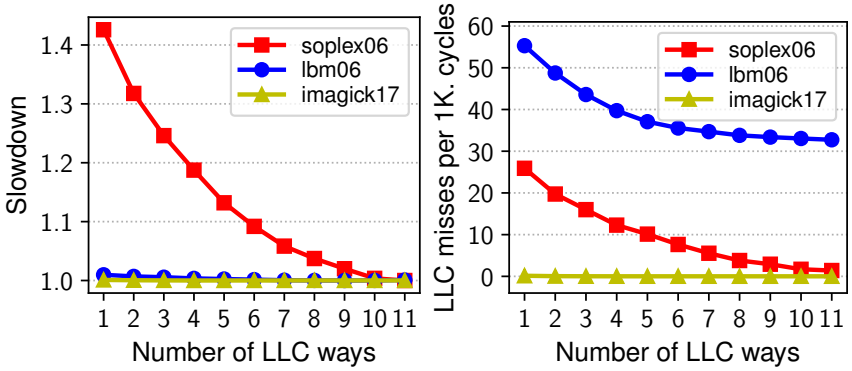


Figure 2.9: Slowdown (left side) and LLCMPKC (right side) for three different applications and varying numbers of cache ways allocated to them. Each application belongs to one of the classes distinguished by LFOC+: cache-sensitive (`soplex`, in red) streaming (`lbm`, in blue), and light-sharing (`imagick`, in green).

core's private cache levels. To illustrate the differences among these application categories, Figure 2.9 shows how the application slowdown (left) and the LLCMPKC (right) vary with the number of assigned LLC ways for a cache-sensitive program (`soplex`), a streaming application (`lbm`), and a light-sharing one (`imagick`) on the `Skylake-40` platform. The slowdown of `soplex` is significantly reduced by the increase in cache ways, as this decreases LLCMPKC. On the other hand, `lbm` reaches an "asymptote" at approximately 30 LLCMPKC, beyond which further reduction in misses does not occur as a result of higher cache-way allocations. Finally, neither the slowdown nor the LLCMPKC of `imagick` is affected by changes in the allocated cache ways.

As applications run under LFOC+, the OS continuously gathers the value of several runtime metrics via PMCs, including the LLCMPKC and IPC, among others [136]. Based on these metrics, applications are then classified into the three aforementioned classes. LFOC+ operates in two modes: *fairness* and *sampling*. In fairness mode, per-thread statistics are gathered using PMCs, and LFOC+'s partitioning algorithm is applied periodically at a configurable period. In particular, the partitioning strategy constitutes a cache-clustering (also known as partition-sharing) approach [46, 59], which maps a set of applications to the same cache partition. When cache-sensitive programs are present in the workload, LFOC+ takes care of confining streaming programs (if there are any) in up to two small LLC partitions. [2] Subsequently, LFOC+ distributes the remaining LLC space among cache-sensitive programs. To determine the number and size of LLC partitions for cache-sensitive applications, LFOC+ relies on a lightweight algorithm called *pair clustering*, which is described in detail in prior work [136]. This algorithm aims to minimize unfairness by assigning each cache-sensitive application to a private LLC partition or to a partition shared with another cache-sensitive program. Lastly, light-sharing applications are distributed across the various partitions by first populating partitions with streaming applications.

[2] In our experimental platform, we use 2-way partitions (5 MiB) to confine streaming programs. Using smaller partitions (1-way) leads to severe bandwidth-contention scenarios, as demonstrated in LFOC+'s original work [136].

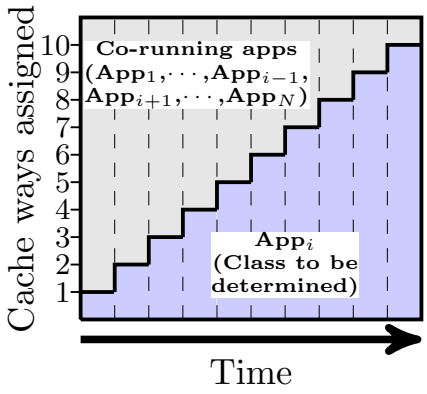


Figure 2.10: Distribution of the LLC space over time during the LFOC+'s sampling mode for App_i. Figure reproduced from LFOC+'s literature [136].

LFOC+'s *sampling mode* is used to determine an application's class and is engaged in two specific cases: when an application enters the system (after a warm-up period) and when a class transition is detected. A class transition occurs when an application suddenly exhibits a performance profile that does not match its current class, for example, a supposedly streaming application begins to exhibit a low LLCMPKC. When the sampling mode is triggered, the application that initiated the transition into this mode is isolated from the rest in an LLC partition, referred to as the *sampling partition*, whose size is then gradually increased (one LLC way each time). Meanwhile, the remaining applications are confined in a complementary LLC partition that shrinks over time, covering the remaining LLC space. Figure 2.10 illustrates this sampling process for an 11-way LLC, such as that of `Skylake-40`. As shown, the partition size gradually increases for App_i. During the sampling mode, LFOC+ observes the application performance (IPC) and the LLCMPKC for different way counts. By doing so, (1) the OS can quickly identify the application's class without exploring all possible LLC sizes (different heuristics are used to achieve this [136]), and (2) slowdown and miss-rate curves are built just for cache-sensitive programs. These curves represent normalized performance (reduction in IPC) and LLCMPKC, respectively, for different number of LLC ways, and are required for the *pair clustering* algorithm. LFOC+ estimates the slowdown of cache-sensitive programs for different ways, considering the IPC observed for a specific number of ways and the actual IPC achieved by the application when the *sampling partition* reaches its maximum size (an estimate for IPC in isolation).

Unlike other partition-sharing strategies [46, 59, 121, 122, 152] that rely on application classification but only support single-threaded applications, LFOC+ also has the ability to efficiently handle regular data-parallel multithreaded applications, like some of the ones we experimented with in this thesis. For these applications, where all threads exhibit an almost identical PMC-related profile (as they do the same kind of work on different data), LFOC+ employs a lightweight classification method. Essentially, this method relies on tracking the PMC metrics of a selected *reference* thread in the application to guide its online classification. At the same time, LFOC+ always ensures that threads in the multithreaded process are consistently mapped to the same LLC partition in either of its operating modes. When

a change in the application's LLC partition is in order, LFOC+ effectively carries out the partition update by synchronously adjusting the partition-related per-core registers associated with all the application's threads [136].

2.2.5 Het-Harness

Het-Harness is an in-house tool developed by members of the ArTeCS research group [3] at the Complutense University of Madrid (UCM). It chiefly makes it possible to automate workloads execution and simplifies the configuration and management of diverse test scenarios in system software experiments on real platforms. To do so, it provides the user with a number of flexible abstractions at different levels: for individual benchmarks, workloads, and scheduler configurations. Particularly, it provides built-in support for launching a large range of benchmarks widely used in computer-architecture research, such as SPEC CPU, PARSEC, or NAS Parallel Benchmarks (NPB). Moreover, it allows the user to seamlessly define sets of workloads by referring to the available per-benchmark scripts or by defining custom benchmark scripts. More importantly, Het-Harness is equipped with a number of command-line tools and Python libraries that bridge the gap between the experiment launching process and the subsequent data processing, analysis, and visualization. Thus, it offers an all-encompassing experimental evaluation workflow to the user.

In the case of this thesis, we leveraged Het-Harness to conduct the vast majority of the experiments on the various multicore platforms used in our evaluations. To this end, the functionality of Het-Harness had to be augmented in various ways. For example, for a proper integration with PMCSched (see Section 3.1), we added a number of helper commands to make it easier to switch between different plugins and configuration options. This was crucial to ease the experimentation with different resource management schemes under PMCSched. Furthermore, we extended Het-Harness with support to conduct static cache-partitioning driven by the PBBCache simulator. We also created a workflow enabling to feed the PBBCache simulation tool with information of a particular workload and then automatically apply static cache-partitioning when launching the workload by leveraging the output provided by the simulator. This feature was crucial to validate the accuracy of the simulations discussed in Section 3.2.2 and Chapter 4.

2.3 Evaluation Methodology

2.3.1 Metrics

This thesis employs a variety of system-level throughput and fairness metrics for multi-program workloads. In our context, a multi-program workload W includes N

[3] `https://artecs.dacya.ucm.es/`

co-running applications: $W = \{a_1, a_2, \cdots, a_N\}$. All the metrics used in this thesis are dependent on the performance of each application a_i, which we measured via its *completion time* (CT).

Importantly, most system-level metrics considered in this thesis are defined in terms on the observed *Slowdown* of each app a_i, which indicates its performance degradation when run on the system together with the other applications in the workload compared to running alone. The slowdown is defined as follows:

$$Slowdown_{a_i} = \frac{CT_{policy,a_i}}{CT_{alone,a_i}} \tag{2.1}$$

Here, CT_{policy,a_i} represents the completion time of application a_i when executed in the workload W under a given scheduling or resource-management policy. Additionally, CT_{alone,a_i} indicates the time of a_i when run on the system in isolation.

Most available metrics that quantify throughput are indicators where higher values are desirable (higher-is-better) [50]. Among the available metrics, in this thesis we primarily leverage the *System Throughput* (STP) [24, 48, 53, 136, 152], defined as follows:

$$STP = \sum_{i=1}^{N} \left(\frac{CT_{alone,a_i}}{CT_{policy,a_i}} \right) = \sum_{i=1}^{N} \left(\frac{1}{Slowdown_{a_i}} \right) \tag{2.2}$$

For our experimental evaluations on asymmetric multicore systems (AMPs), we quantified a workload's degree of throughput under a given policy using the *Aggregate Speedup* (ASP) metric, widely used by previous research on asymmetry-aware scheduling [58, 60, 123, 139, 143, 144]. This metric is defined as follows:

$$ASP = \sum_{i=1}^{n} \left(\frac{CT_{small,a_i}}{CT_{policy,a_i}} - 1 \right) \tag{2.3}$$

Here, CT_{small,a_i} denotes the completion time of application a_i when running alone on the system, using only the platform's small cores. Unlike the STP, the Aggregate Speedup metric captures the total benefit that the workload as a whole extracts from the high-performance big cores available in the AMP system [60]. Particularly, a positive AMP value indicates that the workload derives some performance benefit from big cores; the higher the value, the greater the benefit. By contrast, negative (or zero) values suggest a poor exploitation of big cores' resources; this could be caused by a number of factors, including severe shared-resource contention [58], poor thread-to-core mapping decisions [143], or inherently inefficient utilization of the big cores' microarchitectural resources by the applications present in the workload [79, 137, 139].

For the purposes of performance evaluation during this thesis, we opted not to use throughput metrics that depend solely on the *raw IPC* achieved by each application in the multi-program workload [113, 121, 170]. We avoid such metrics since they can be misleading in workload scenarios including multithreaded programs [9, 49], such

as those considered in most of the experiments conducted in the thesis. Essentially, in certain HPC programs, threads may busy-wait for a certain amount of time while waiting pn synchronization primitives; a busy-waiting thread usually achieves a high IPC without making any actual progress, thus making an inappropriate performance metric in this context.

System throughput is often reported together with fairness figures. In this thesis, we adopt a notion of fairness commonly used in previous work [136, 152, 171], which considers that a policy is fair if it ensures that equal-priority applications in a workload are subjected to the same slowdown as a result of sharing the system. To employ this notion of fairness, we use the widely popular (lower-is-better) *Unfairness* metric [53, 58, 136, 171], defined below:

$$Unfairness = \frac{MAX(Slowdown_{a_1}, \ldots, Slowdown_{a_N})}{MIN(Slowdown_{a_1}, \ldots, Slowdown_{a_N})} \tag{2.4}$$

In addition, we employ the *FairSlowdown* metric [170], also known as *Average Normalized Turnaround Time* (ANTT) [24,152,170]. This complementary fairness metric captures the average performance degradation of applications relative to their performance when running alone:

$$ANTT = \frac{1}{N} \sum_{i=1}^{N} Slowdown_{a_i} \tag{2.5}$$

Another alternative metric we relied upon to assess the degree of unfairness is the $UnfairnessCoV$ metric [113,152], based on the Coefficient of Variation (CoV) across per-application slowdowns:

$$Unfairness_{\text{CoV}} = \frac{\sigma_{Slowdown}}{\mu_{Slowdown}} \tag{2.6}$$

where σ_{Slowdown} and μ_{Slowdown} represent the standard deviation and mean slowdown observed across applications, respectively.

While quantifying the effectiveness of the various proposals, we also considered other fairness and throughput metrics but opted to narrow down the focus to the set of metrics defined above. For this purpose, we performed a simulation-based study examining the interrelationships between the various metrics in the context of cache-partitioning policy evaluation. This study, which constitutes an additional contribution of this thesis, can be found in Appendix B.

2.3.2 Applications and benchmark suites

In this thesis, we employed both single-threaded and multi-threaded applications for the experimental evaluation of the various proposals. Particularly, we used

single-threaded programs from the SPEC CPU2006 and SPEC CPU2017 benchmark suites, which are standard for performance evaluation in computer architecture research. These benchmarks have proven successful in covering a wide range of application types for the evaluation of the strategies proposed in this thesis. Specifically, in Chapter 4, we show that SPEC CPU applications comprise a very diverse set of programs in terms of cache hierarchy utilization efficiency, making them ideal for evaluating the potential of LLC-partitioning policies. Similarly, as we show in Chapter 6, the diverse utilization of various CPU and memory-related resources across SPEC CPU programs results in a wide range of relative performance values when executed on the various cores of an asymmetric multicore system. Such diversity makes these programs perfect candidates for constructing workloads that span an ample spectrum of behaviors related to the efficient use of asymmetric cores, thereby ensuring their suitability for evaluating asymmetry-aware schedulers targeting CPU-bound workloads, such as those proposed in this thesis.

Parallel (multithreaded) applications also play a crucial role in the experimental evaluations conducted throughout this thesis, where the diversity in the degree of thread-level parallelism, scalability, and the impact of user-level runtime systems becomes paramount. To cover a broad range of scenarios concerning these aspects, we experimented with OpenMP applications as well as programs based on POSIX threads. In particular, we used OpenMP programs from popular HPC benchmark suites, such as Rodinia [31], NAS Parallel Benchmarks [18], PARSEC [22], and SPEC CPU2017 (which includes OpenMP programs beyond single-threaded applications [136]). These benchmarks were complemented by `RNASeq` [35], an OpenMP-based RNA sequencing application. For POSIX-threaded applications, we leveraged programs from the PARSEC benchmark suite, along with two other scientific applications: `BLAST` [4], a bio-informatics application, and `FFTW3D` [12], a scientific benchmark performing the fast Fourier transform. Classifications of these programs based on diverse aspects such as cache sensitivity, scalability, and efficiency in using asymmetric cores can be found in Chapters 4-6.

2.3.3 Setting up, launching workloads, and performance measurement

In our experiments, we used both single-application workloads, consisting of a multithreaded program, and multi-application workloads, comprising mixes of single-threaded and multi-threaded programs. Given that all applications considered are CPU bound (i.e., they do not spend a significant portion of their execution on I/O operations), the total thread count of each workload was set to match the number of cores on the platforms used for experiments. This allowed us to conveniently stress the platform's available hardware resources when running CPU-bound programs.

The workloads constructed for the experimental evaluations throughout this thesis were specifically designed to cater to the specific needs of each evaluation scenario. While gathering a diverse set of workloads was a common goal across most experiments, the distinct nature of the various thesis proposals inevitably lead us to

employ a different interpretation of *workload diversity* for each evaluation scenario. For example, employing program mixes that exhibit a varying spectrum of big-to-small speedups is usually a key requirement to comprehensively evaluate schedulers for asymmetric multicore systems [30, 58, 143, 146]. In contrast, when assessing the potential of contention-aware approaches (particularly LLC-partitioning strategies), more emphasis should be placed on capturing a wide range of application behaviors regarding the degree of cache sensitivity, memory intensity, and contentiousness [34, 46, 122, 136, 161, 178]. To meet the varying requirements of each scenario, we adhered to a general experimental methodology characterized by the following aspects:

- We start by defining a set of application classes for the experimental scenario in question and then conduct a performance characterization of the various applications to identify the class they belong to based on their dominant behavior [4]. In particular, for the evaluation of contention-aware approaches, we used the application categories described in Section 2.2.4: cache-sensitive, streaming, and light-sharing. When building application classes for our experiments on AMP systems (see Chapter 6), we primarily examined the application's speedup factor, as well as the portion of its execution that runs with a single runnable thread. Lastly, in evaluating our elasticity approach for maximizing CPU utilization, each application was classified based on its scalability features and its degree of CPU usage over time.

- Once the applications are categorized, we proceed to construct a set of multi-application workloads. To this end, we select a number of applications within each category and mix applications from various classes. To ensure comprehensive coverage, we randomly combine applications of different types in varying proportions, covering the full range of workload characteristics. Details on workload composition can be found in Chapters 4- 6.

- For most workloads (single- and multi-application), we conduct a full execution of each application and run each program multiple times. To measure the performance of each application, we calculate the geometric mean of completion times across the various runs. This per-application geometric mean is then used to determine the value of the set of system-wide metrics introduced in Section 2.3.1.

- When launching multi-application workloads, where applications may have significantly different completion times, we adopt a strategy inspired by prior work [27, 51, 58, 136, 155], designed to avoid scenarios where the execution time is dominated by the slowest program(s) in the workload. This is, all applications start executing at the same time, and when one finishes, it is restarted until the longest-running application has completed a given number of full executions.

[4] Applications may go through different program phases over time, so an application's class could vary across phases. While the application's overall class (dominant behavior) is considered when constructing workloads, most of the proposed approaches in this thesis are phase-aware, making runtime decisions based on the application's current class.

Chapter 3

Overview of Contributions

This chapter includes a detailed overview of the principal contributions of the thesis. As explained in the Introduction, our interest was addressing several key challenges of Chip Multicore Processors (CMPs): the adverse impacts of shared-resource contention, maximizing CPU utilization, and scheduling on asymmetric multicore processors. Furthermore, simplifying operating system development was of great importance to achieving our research goals in a timely fashion.

This chapter is structured as follows: Section 3.1 presents our PMCSched framework. Section 3.2 introduces Divide&Content, a fair contention-aware resource manager for NUMA multicores. Section 3.3 outlines our efforts to increase CMP utilization via vertical elasticity. Finally, Section 3.4 discusses our contributions on asymmetry-aware scheduling.

We should highlight that Sections 3.2 to 3.4 constitute compact summaries of the articles found in Chapters 6-5. Many details from these articles have been omitted in the summary of the corresponding sections, including the pseudo-code listings of the proposed algorithms, in-depth related-work discussions, and comprehensive explanations of the results obtained in all the experimental analyses conducted. By contrast, additional content has been included in these sections to better contextualize the specific research and ease a high-level understanding of the various proposals via intuitive examples.

3.1 PMCSched framework

This section begins by presenting the motivation behind the development of PMCSched and justifying its reliance on the PMCTrack tool. Next, we describe PMCSched's architecture, its main abstractions, and the major challenges we had to be overcome to make it a reality. Lastly, a brief tutorial is provided to illustrate the steps required to implement an OS-level scheduling-related extension with PMCSched.

3.1.1 Motivation

One of the main goals of this thesis was to design a number of OS-level scheduling policies and resource management strategies for better exploitation of current and emerging multicore architectures. Implementing these OS extensions requires close cooperation with the process scheduler, as making the necessary decisions requires to be aware of key scheduling events – such as context switches or process creation – which can only be managed directly from the scheduler's code. In the Linux kernel, the scheduler design is modular, allowing kernel developers to implement custom scheduling algorithms as separate *scheduling classes*. However, testing scheduler modifications can be a tediously slow process, in part due to Linux's monolithic design [146]. All scheduler changes require compiling and reinstalling the kernel, followed by rebooting the machine for the changes to take effect. Testing an individual change in this way can take a full coffee break, depending on the features and resources of the target platform and the development host.

This greatly hinders developer's productivity, thus posing a major burden to accomplish the goals of this thesis. More generally, the slow testing process for scheduler modifications constitutes a substantial barrier to system software research that relies on kernel changes for prototyping and evaluating novel policies. With this in mind, we explored ways to enable rapid prototyping of such policies in Linux that would not require a system reboot.

Currently, Linux kernel modules allow developers to add OS extensions at runtime. These modules can also be removed from the kernel when not in use; this allows buggy code to be unloaded from the kernel and reinserted after applying the necessary bug fixes. Therefore, using kernel modules seemed to be the most appropriate mechanism to enable efficient prototyping of our proposed policies.

However, two main issues became apparent in using the out-of-the-box kernel module support to pursue our objectives. Firstly, the kernel API for modules (at the time of writing, kernel version v6.10) does not allow the creation of custom schedulers or resource managers from a loadable kernel module. As described earlier, this kind of OS extensions require changes in the core kernel, necessitating a system reboot for the associated testing. Secondly, we found that leveraging performance monitoring counters (PMCs) from key scheduler critical points – an essential aspect of most of our proposals – was impractical using the current Linux kernel API for accessing PMCs, provided by the *Perf Events* subsystem. Essentially, our proposals required reading PMC values and resetting PMCs from routines invoked upon key scheduler events (e.g., tick processing, context switch, etc.). To this end, Perf event's functions for reading and configuring PMCs must be used. Unfortunately, these functions leverage blocking synchronization primitives, which cannot be used from interrupt context, where the aforementioned scheduler routines execute. This limitation renders Perf events useless in this context; in fact, using Perf's API from the scheduler code causes the entire OS to crash.

To build our own scheduling framework (PMCSched [23]), while retaining the ability to dynamically load kernel code via modules, we chose the PMCTrack tool as

its main foundation. As introduced in Chapter 2, PMCTrack [145] is a performance monitoring tool for Linux. Selecting PMCTrack as the basis for the design of PMCSched was a deliberate and well-founded decision for several reasons. First, it makes it possible to implement custom OS extensions, referred to as *monitoring modules*, bundled within a loadable kernel module. Developing a monitoring module comes down to implementing an interface of operations, consisting of callbacks invoked upon main scheduling events. Second, PMCTrack provides a comprehensive architecture-independent API to access PMCs and other hardware monitoring and resource-management facilities (e.g., for cache-partitioning). This was appealing for this thesis as leveraging those existing kernel APIs enabled rapid development of PMC-based cache-partitioning and scheduling policies. Lastly, unlike Perf events, PMCTrack's functionality can be augmented by making changes in an extensible loadable kernel module, where PMCs can be easily managed from scheduling-related callbacks, most of which run in interrupt context.

One of the main goals of PMCSched was to allow researchers to create custom scheduling-related OS-level extensions in a loadable kernel module which could be inserted in unmodified (unpatched) kernels. This was crucial to allow these extensions to be potentially adopted in production systems, where patching the kernel is not always possible. Note that most previously proposed scheduling-related kernel frameworks need kernel patches to function or constitute substantial forks of the kernel [19,29,85,94,105]. This approach makes it challenging to use these frameworks in production systems [105], and more importantly, it substantially complicates their maintenance across kernel versions, especially considering the frenetic pace of Linux kernel development and its continuously evolving nature. Apart from allowing users to work on top of *vanilla* kernels, PMCSched has been also designed to allow the implementation of strategies that leverage scheduling decisions across various system software layers (e.g., by exploiting the synergistic cooperation between between the runtime system and the OS scheduler [58,137,140]). All in all, its design makes it a versatile tool for scheduling and resource management, easing development by reducing kernel-level programming efforts.

3.1.2 PMCSched architecture and main abstractions

To accomplish the main objectives of our PMCSched framework while retaining PMCTrack as its foundation, we had to overcome two main challenges. The first one was removing the dependency on the patched kernel required by the version of PMCTrack available at that time. PMCTrack's patch extends the task structure to allow the tool to point to thread-specific data. It also introduces minor changes in the Linux scheduler implementation by issuing a set of notifications (aka PMCTrack's kernel API) to make PMCTrack's kernel module aware of key scheduling events, ranging from thread creation/destruction, to context switches or periodic scheduler activations. Awareness on the occurrence of these scheduling events is crucial to save and restore the PMC-related context in a per-thread fashion and to manage other hardware monitoring features. The second challenge we had to take on was to equip PMCSched-enabled scheduling extensions with the ability to

customize the behavior of the Linux load balancer.

To remove the dependency on PMCTrack's kernel patch, we leveraged two key mechanisms: installing scheduling hooks via kernel tracing technologies, and creating a *dummy* event from the Perf Events subsystem to hold PMCTrack's thread-specific data. Specifically, we rely on two modern tracing facilities of the Linux kernel: *dynamic ftrace* [130] and *tracepoints* [89], which support dynamic and static kernel instrumentation, respectively. Noticeably, both have full support for a wide range of processor architectures in recent kernels (v5.8.y and later), and can be found enabled by default on most popular Linux distributions. Unlike other kernel instrumentation facilities (like Kprobes), these technologies make it possible for a module to be notified when a kernel function is invoked or when a static tracepoint is reached with virtually no overhead [89,130]. We altered PMCTrack's kernel module so that it installs a number of callback (via tracing *hooks*) associated with every per-thread scheduling event that PMCTrack needs.

The second and final step to eliminate PMCTrack's dependency on the kernel patch was to envision a way to hold PMCTrack thread-specific data within the task structure without modifying the kernel. To this end, we modified PMCTrack's kernel module to create a per-thread *dummy* (fake) software event from the Perf Events subsystem. We found that the structure that represents this dummy event in the kernel (`struct perf_event`) contains an unused void pointer field (`pmu_private`) which can be utilized to point to any other structure. We employ this void pointer to reference PMCTrack's per-thread structure (`pmon_prof_t`). The dummy event structure is allocated and properly initialized by the scheduling hook invoked upon process and thread creation (i.e., when `fork()` or `clone()` are invoked). It is then inserted into the event linked list present in Linux's task structure (`perf_event_list` field), which allows us to retrieve the event in all per-thread scheduling-related hooks. This provides access to PMCTrack's per-thread structure, which is critical for its internal operation.

To make it possible to implement custom load balancing policies, PMCSched introduces the *core group* abstraction. Essentially, cores in the system are organized into different sets (or *core groups*) based on their type (for AMP systems) and/or their hierarchical relationship in the platform's topology (e.g., cores sharing a last-level cache, or part of the same NUMA node). PMCSched automatically divides cores into different core groups based on system topology, but considering a configurable granularity (LLC, socket, or NUMA domain). To implement custom and scalable OS-level load balancing policies or perform specific thread-to-core mappings, a scheduler or resource manager implemented in PMCSched must assign threads to specific core groups by using affinity masks. In using this approach, enforcing load balancing across cores within the same group is up to the Linux load balancer, which respects affinity masks.

The PMCSched framework is now part of the open-source PMCTrack project; particularly, its source code was first released along with PMCTrack v3.0. Figure 3.1 depicts the architecture of this new version of PMCTrack, which no longer needs a kernel patch to function thanks to the development efforts behind PMCSched. The

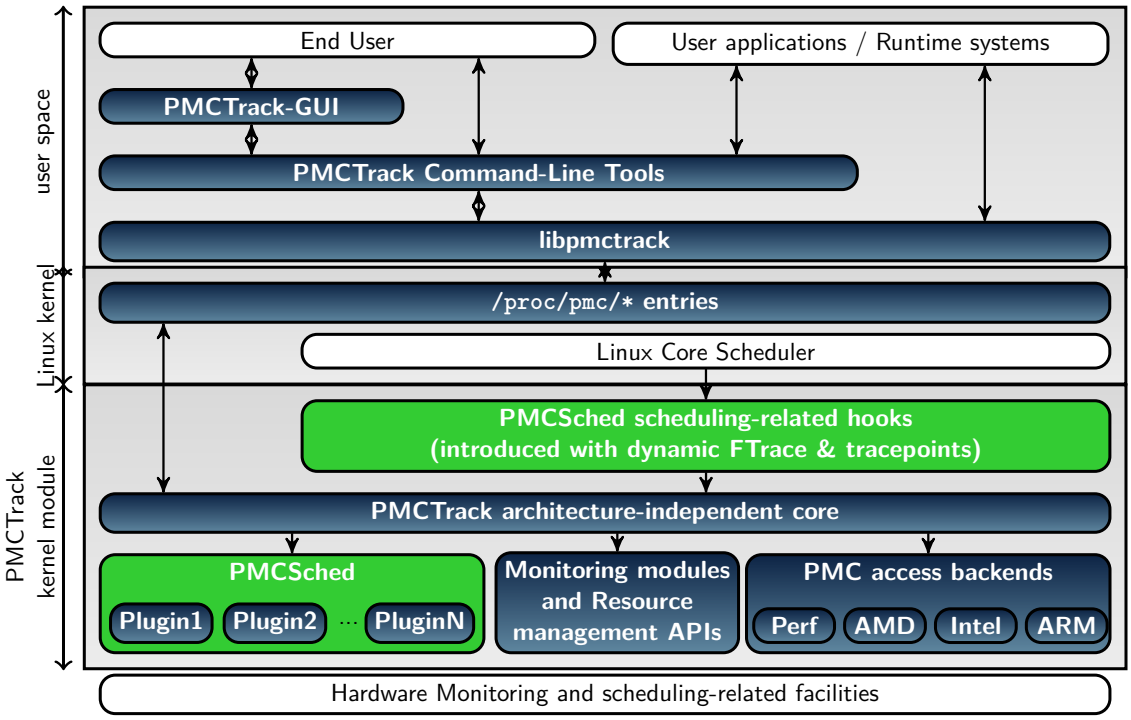


Figure 3.1: Addition of PMCSched within the larger PMCTrack system.

main architectural differences with respect to PMCTrack v2.x include the removal of PMCTrack's kernel API (tracing-based scheduling hooks are used instead) and the adoption of the PMCSched subsystem. As illustrated in Figure 3.1, this subsystem – implemented as an extensible PMCTrack monitoring module – consists of a set of *scheduling plugins*, each implementing a custom scheduling or resource management algorithm.

In addition to leveraging the direct exploitation of the rich set of PMCTrack APIs to manage PMCs and hardware cache-partitioning extensions, PMCSched's plugin developers have access to a set of extensible data types that represent common scheduling-related abstractions, like threads or multithreaded processes. Particularly, threads are represented in our framework with the `pmcsched_thread_data_t` data type. A pointer to this structure is stored within PMCTrack's per-thread structure (`pmon_prof_t`), which in turn can be retrieved from the Linux task descriptor (`struct task_struct`). To further simplify plugin development, the framework also provides developers with the `group_app_t` and `sched_app_t` structures enabling to keep track of statistics and runnable threads of the same application currently assigned to a certain *core group* or globally present in the system, respectively. Lastly, PMCSched maintains a data structure for each *core group* in the system, `sched_thread_group_t`. This structure features a number of linked lists and other control fields allowing plugin developers to perform a coordinated management of all active threads and applications currently running on a given *core group*. This per-core group scheduling structure is a crucial abstraction of our framework as it allows to make scheduling and resource-management decisions independently for threads assigned to different core groups, which favors scalable implementations.

Furthermore, in the context of this thesis, the `sched_thread_group_t` abstraction played a critical role in facilitating the implementation of cache-partitioning policies, as the hardware extensions available in commercial x86 processors are directly managed by the OS/VMM on a per-core group basis. Take, for instance, Intel's support for hardware-aided cache-partitioning (referred to as Intel CAT), which – on our `Skylake-40` platform – exposes a separate set of privileged registers for each LLC on the system, each enabling to configure cache partitions for applications running on cores sharing that LLC.

PMCSched also includes built-in support to facilitate efficient communication between applications and the OS kernel. To this end, shared memory is leveraged to exchange scheduling-relevant information between both spaces, opening the door to promising optimizations bundled in PMCSched plugins. To enable any process/thread to access this kind of shared memory region, PMCSched exports a special file through the procfs pseudo-filesystem, `/proc/pmc/schedctl`. Specifically, to retrieve a pointer to the region, the thread must open that special file and then invoke `mmap()`, passing the file descriptor returned by `open()` as a parameter. We should highlight that this PMCSched feature was inspired by the `schedctl()` system call present in the Solaris operating system [97]. The call returns a pointer to a scheduling-related data structure, allowing efficient bidirectional communication between user and kernel spaces without the need for additional system calls. With PMCSched, we achieve a similar feature on the Linux kernel, but making use of existing POSIX system calls (`schedctl()` is not available on Linux) and allowing to implement scheduling-related optimizations in a loadable kernel module. This support is widely exploited in this thesis, as two of the main proposals rely on it to leverage interaction between the runtime system and the OS kernel. We elaborate on this aspect in Sections 3.4.2.2 and 3.3.1.

Building a PMCSched plugin boils down to instantiating an interface of scheduling operations (`sched_ops_t`) and implementing the corresponding interface functions in a separate '.c' file. This way, all plugins adhere to a "contract" – this is, a predefined set of requirements, which is a common practice in kernel-module development. This interface is represented by the `sched_ops_t` structure, defined as follows:

```
1  typedef struct sched_ops
2  {
3    char* description;
4    sched_policy_mm_t policy;
5    unsigned long flags;
6    struct list_head link_schedulers;
7    pmcsched_counter_config_t* counter_config;
8
9    /* Callbacks */
10   int (*on_fork_thread)       (pmcsched_thread_data_t* t,
11                                unsigned char is_new_app);
12   void (*on_exec_thread)      (pmon_prof_t* prof);
13   void (*on_active_thread)    (pmcsched_thread_data_t* t);
14   void (*on_inactive_thread)  (pmcsched_thread_data_t* t);
```

```
15   void (*on_exit_thread)      (pmcsched_thread_data_t* t);
16   void (*on_free_thread)      (pmcsched_thread_data_t* t,
17                               unsigned char is_last_thread);
18   void (*on_migrate_thread)   (pmcsched_thread_data_t* t,
19                               int prev_cpu, int new_cpu);
20   int (*on_read_plugin)       (char *aux);
21   int (*on_write_plugin)      (char *line);
22   void (*on_migrate_thread)   (pmcsched_thread_data_t* t,
23                               int prev_cpu, int new_cpu);
24   void (*sched_timer_periodic)(void);
25   void (*sched_kthread_periodic)
26                               (sized_list_t* migration_list);
27   int (*on_new_sample)        (pmon_prof_t* prof, int cpu,
28                               pmc_sample_t* sample,int flags,
                                  void* data);
29 } sched_ops_t;
```

The structure consists of a set of fields and callbacks. The purpose of these callbacks [1] is as follows:

- `on_fork_thread`: This function is called when a thread invokes the `fork()` system call. In addition to the `pmcsched_thread_data_t` parameter with the thread information itself, this callback receives an additional boolean parameter, `is_new_app`, to distinguish between newly created processes and threads of already existing processes. Since this is the first function invoked in the life cycle of a new thread, it serves as an ideal location for plugins to initialize their per-thread metrics.
- `on_exec_thread`: This function is invoked when a thread calls the `exec()` system call.
- `on_active_thread`: This function executes when a thread becomes runnable and receives the thread's PMCSched descriptor as a parameter. Similarly, `on_inactive_thread()` is invoked when the thread blocks, sleeps or terminates.
- `on_exit_thread`: This callback is invoked when the thread terminates its execution.
- `on_free_thread`: This function executes when the kernel frees up the memory associated with a thread's task structure. This is the right moment for plugins to also release any memory reserved for keeping track of thread-specific data. Notably, when the current thread is the last thread of the process whose descriptor is being freed up, the `is_last_thread` parameter is set to one. Plugins that dynamically allocate memory for process-wide data structures should consider this flag, as these structures must be freed up when `is_last_thread` is set to one.

[1] Note that it is up to the plugin developer to decide which callbacks to define, depending on the requirements of the scheduler or resource manager the plugin implements.

- `on_migrate_thread`: This callback is called when a thread is migrated from one core to another. It enables tracking of the thread's movement across CPUs and or core groups for load balancing. Plugins can implement this callback to ensure that a previously triggered migration completes successfully.

- `on_write_plugin`: PMCSched exposes the `/proc/pmc/sched` special file for framework configuration and tracking. This callback allows the plugin to expose configurable parameters to the user, who can alter them by writing to said file. This functionality makes it possible for users to update plugin-specific settings, thus facilitating dynamic reconfiguration of the behavior of the plugin's resource manager or scheduler.

- `on_read_plugin`: This operation is called when the `read()` system call is invoked on `/proc/pmc/sched`. This callback is provided to allow plugins to include relevant information in the output of that file, thus enabling to expose any kind of plugin-specific information, such as the current value of configurable parameters.

- `sched_timer_periodic()`: This callback allows the plugin to perform periodic operations in a per core-group fashion, such as recalculating metrics relevant to the scheduler and other housekeeping logic. For potentially blocking operations – such as triggering a thread migration, or delivering a signal to a process –, plugins should use `sched_kthread_periodic()` instead.

- `sched_kthread_periodic`: This function is called periodically by a kernel thread (*kthread*) that runs in process context, allowing it to invoke blocking operations. It can be used by plugins, for example, to perform a thread migration, which is a blocking operation in the kernel.

- `on_new_sample`: This callback must be defined by plugins that leverage PMCs to gather scheduling-relevant statistics in a per-thread fashion. In particular, it is invoked when PMCTrack has collected new PMC samples for a given thread, whose descriptor is passed as a parameter. Typical operations performed inside this callback include the computation of high-level metrics based on the low-level hardware events provided by PMCs, or storing the necessary per-thread statistics or metrics in the thread's descriptor, so that they are accessible from other plugin's callbacks. In some of the plugins developed for this thesis, we leverage the `on_new_sample` callback to classify threads based on their cache-related behavior. By exploiting PMC values, we also estimate a thread's big-to-small speedup while it executes on an asymmetric multicore processor.

The remaining fields in `sched_ops_t` serve the following purposes:

- `policy`: This field stores a constant enumerate value that uniquely identifies the scheduling policy.

- `description`: It includes a human-readable description of the scheduling policy, displayed when reading the `/proc/pmc/sched` file.

- `flags`: This field allows developers to decide whether to rely on PMCSched to handle the locking synchronization logic in the callback, as in the above code example of per-group locking (`PMCSCHED_CPUGROUP_LOCK`), or to manage the locks themselves. Plugins that choose the latter (flag `PMCSCHED_CUSTOM_LOCK`) are responsible for their own synchronization – such as managing lists of active or inactive applications/threads and avoiding race conditions – but gain greater control and locking granularity.

- `link_schedulers`: This field allows the plugin's descriptor to be inserted in the doubly-linked list of plugins that PMCSched maintains. Note that many data structures in the Linux kernel include one or several `list_head` fields so that the structure can be inserted in the generic doubly-linked list data structure that the kernel implements.

- `counter_config`: This field describes the hardware performance counter configuration of the various PMC events that the plugin wants to gather on a per-thread fashion. This configuration is specified using PMCTrack's raw event format, described in prior work [145]. The `counter_config` field also enumerates the high-level metrics that the plugin needs to calculate based on the gathered PMC event values – PMCTrack provides an API to make metric calculation automatic –, as well as the set of virtual counters that it exposes to the various PMCTrack's components. In the event that the plugin does not leverage performance monitoring counters, `counter_config` must be set to `NULL`.

This thesis extensively tested the effectiveness and flexibility of PMCSched through a variety of experimental case studies, which involved different CMP configurations, including symmetric and asymmetric CMPs, and NUMA multicores. Indeed, major contributions of this thesis were made possible by PMCSched, including the Divide&Content resource manager (presented in Section 3.2), our elasticity manager (Section 3.3), and extensions for asymmetric multicores through coordination between the runtime system and the OS (Section 3.4). For each of these cases, PMCSched had to be adapted and improved, ultimately becoming a more comprehensive framework as a result of addressing the limitations we encountered along the way.

3.1.3 Tutorial: Creating a new PMCSched plugin

To illustrate the process of plugin creation, let us walk through a simple example and its associated code. First, we must implement the pre-defined `sched_ops_t` interface (described in the previous section) in a separate '.c' source file. We begin by developing our new plugin by creating a new file, named `example_thesis.c`. As explained, its functions will handle particular scheduling events. The minimal required callbacks, along with a policy ID, optional flags, and string description, are defined as follows:

```
sched_ops_t thesis_plugin =
{
    .policy                  = SCHED_THESIS,
    .description             = "Example plugin",
    .flags                   = PMCSCHED_CPUGROUP_LOCK,
    .sched_timer_periodic    = sched_timer_periodic_thesis,
    .sched_kthread_periodic  = sched_kthread_thesis,
    .on_exec_thread          = on_exec_thread_thesis,
    .on_active_thread        = on_active_thread_thesis,
    .on_inactive_thread      = on_inactive_thread_thesis,
    .on_fork_thread          = on_fork_thread_thesis,
    .on_exit_thread          = on_exit_thread_thesis,
    .on_migrate_thread       = on_migrate_thread_thesis,
};
```

Registering our new plugin in PMCSched requires declaring the plugin's descriptor in the framework's main header file, `pmcsched.h`. Since we implement the functions in a separate file (`example_thesis.c`), the descriptor must be declared as `extern`:

```
extern struct sched_ops thesis_plugin;
```

Two additional changes in the header file are necessary to finish registering our `SCHED_THESIS` plugin: first, in the enumeration of scheduling policies, and second, in the array of available schedulers. We start by adding the plugin's ID to the enumeration of available plugins:

```
/* Supported scheduling policies */
typedef enum
{
    /* Examples of previous scheduling plugins */
    SCHED_DUMMY_MM=0,
    SCHED_GROUP_MM,
    SCHED_BUSYBCS_MM,
    /* New plugin */
    SCHED_THESIS,
    NUM_SCHEDULERS
}
sched_policy_mm_t;
```

We now include the plugin's descriptor in the array of available plugins:

```
static __attribute__ ((unused)) struct sched_ops*
    available_schedulers[NUM_SCHEDULERS] =
    {
        /* Examples of previous scheduling plugins */
        &dummy_plugin,
```

```
        &group_plugin,
        &busybcs_plugin,
        /* Our new plugin */
        &thesis_plugin,
    };
```

The last step is to include the source file of our new plugin in the list of ".c" files to be compiled, which is found in the architecture-specific Makefile of PMCTrack's kernel module. For instance, to target Intel processors, the example plugin's object file (`example_plugin.o`) is added as shown below:

```
MODULE_NAME=mchw_intel_core
obj-m += $(MODULE_NAME).o
PMCSCHED-objs= pmcsched.o dummy_plugin.o group_plugin.o
   busy_plugin.o example_plugin.o
```

Once the PMCTrack kernel module is loaded into the system (see PMCTrack's official documentation [120]), the scheduling plugins can be selected and configured by writing to special files in Linux's `procfs`. To do so, we first activate PMCSched via the `/proc/pmc/mm_manager` file, managed by PMCTrack:

```
# Check PMCSched's monitoring module ID (platform specific)
$ cat /proc/pmc/mm_manager
[*] 0 - This is just a proof of concept
[ ] 1 - IPC sampling SF estimation module
[ ] 2 - PMCSched
[ ] 3 - AMD QoS extensions (monitoring and allocation)

## Activate PMCSched
$ echo 'activate 2' > /proc/pmc/mm_manager

## Make sure that PMCSched has been activated
$ cat /proc/pmc/mm_manager
[ ] 0 - This is just a proof of concept
[ ] 1 - IPC sampling SF estimation module
[*] 2 - PMCSched
[ ] 3 - AMD QoS extensions (monitoring and allocation)
```

Reading from `/proc/pmc/sched` allows us to find out which PMCSched's plugin is currently active, as well as to retrieve the ID of our new plugin:

```
$ cat /proc/pmc/sched
The developed schedulers in PMCSched are:
[*] 0 - Dummy default plugin (Proof of concept)
[ ] 1 - Group Scheduling Plugin (Proof of concept)
[ ] 2 - Busy scheduler
```

```
[ ] 3 - Example scheduler
- To change the active scheduler echo 'scheduler <number>'
---
(Plugin specific output)
```

Once this ID is known – referred to as `[scheduler_id]` –, the active plugin can be changed with the following command: `echo [scheduler_id] > /proc/pmc/sched`.

3.2 A Fair OS-level resource manager for contention balancing on NUMA multicores

As discussed in the thesis introduction, co-running multiple applications/VMs on a multicore system naturally leads to competition for shared platform resources, such as the LLC and available memory bandwidth [178]. When severe, this competition may substantially impact the performance of individual applications and cause unfairness [45, 136, 152] – that is, uneven performance degradation across applications that may arise when sharing a computing system.

One of the main goals of this thesis was to devise means to reduce unfairness in NUMA multicores. Due to their decentralized and scalable nature, this kind of architecture has become widespread both in cloud datacenters [92, 112, 154] and HPC platforms [26, 65]. A large body of proposals have looked at optimizing memory placement and to devise NUMA-specific page migration policies [42, 44, 65]. However, the exploration of solutions that make coordinated scheduling and resource-management decisions to specifically address the contention issues of NUMA systems has received little attention [24].

Specifically, previous work addressing resource contention on multicores fails to effectively combine two popular mitigation techniques that are crucial in NUMA systems: dynamic thread-to-core mappings [26] and LLC-partitioning [24]. Particularly, existing policies that exploit dynamic thread-to-core assignments to reduce shared-resource contention on NUMA multicores [26] do not leverage LLC-partitioning. Conversely, most LLC-partitioning policies were designed for UMA systems, where cores share a single LLC [46, 113, 121, 136, 152]. These policies do not dynamically adjust thread-to-core assignments, relying instead on fixed mappings. Notably, we found that solely applying fairness-aware LLC-partitioning independently within each NUMA socket does not optimize system-wide fairness on NUMA multicores. Therefore, a trivial NUMA extension of any existing fairness-aware LLC-partitioning policy (via independent per-socket partitioning) constitutes a suboptimal solution.

To understand how to best combine LLC-partitioning and dynamic thread placement, we carried out a comprehensive simulation study. In this study, we measured the degree of throughput and fairness achieved by different theoretical and heuristic-based partitioning policies. The overarching conclusion of our study is that treating

thread placement and LLC-partitioning as separate and orthogonal optimization problems results in suboptimal solutions in terms of fairness. More specifically, the effectiveness of a policy in enforcing fairness is significantly influenced by thread placement, as inappropriate thread assignments can intensify contention for different types of shared resources.

The findings of our simulation study informed the design of our Divide&Content (DC) proposal, an OS-level contention-aware resource manager for NUMA multicores. DC constitutes a key contribution of this thesis. Our extensive experimental evaluation demonstrated that it significantly improves fairness compared to Linux and a NUMA-aware contention-conscious scheduling strategy [26], while maintaining consistent performance and throughput. While DC was evaluated on a NUMA multicore system (the `Skylake-40` platform, presented in Section 2.1), its reliance on the PMCSched's core group abstraction allows it to be exploited on any kind of multi-LLC multicore system; this is, whether UMA or NUMA platforms, where cores are divided into separate groups, each group sharing an independent LLC. Take for instance any single-socket system consisting of an AMD Zen2 multicore processor from the EPYC 7002 series – like AMD EPYC 7742 (see Section 2.1) – which integrates several Core Complex (CCX) dies, with each one featuring 4-cores sharing an L3 cache.

The remainder of this chapter is organized as follows: Section 3.2.1 delves into the motivation behind these research efforts. Section 3.2.2 summarizes the insights from our simulation study. Section 3.2.3 discusses the design and implementation of DC. Finally, the experimental evaluation of DC is presented in Section 3.2.4.

3.2.1 Motivation

To illustrate how much a partitioning policy is influenced by thread placement, we considered a multi-application workload consisting of 10 programs, which we ran on the `Skylake-40` platform (a dual-socket 40-core NUMA platform, described in Section 2.1). The workload includes applications from the three categories introduced in Section 2.2.4: streaming, cache sensitive, and light sharing. In our experiment, we applied six different fixed thread-to-socket assignments, labeled as Map1 through Map6. To show the impact that a particular assignment has on the effectiveness of a cache-partitioning policy, we created a direct extension of the LFOC+ strategy for NUMA systems. Recall that the original LFOC+ partitioning policy [136] is described in Section 2.2.4. The variant of LFOC+ we built for this experiment applies the associated partitioning algorithm separately to the set of threads mapped to each socket. Therefore, it does not trigger thread migrations.

Figure 3.2 shows the per-application slowdown as well as the value of the unfairness and STP metrics for the different mappings, where the LLC of each socket is partitioned using LFOC+'s partitioning algorithm. As it is evident, the degree of fairness delivered by LFOC+ in this context greatly depends on the mapping. In particular, the worst fairness-wise mapping (Map2) increases unfairness by 28% compared to

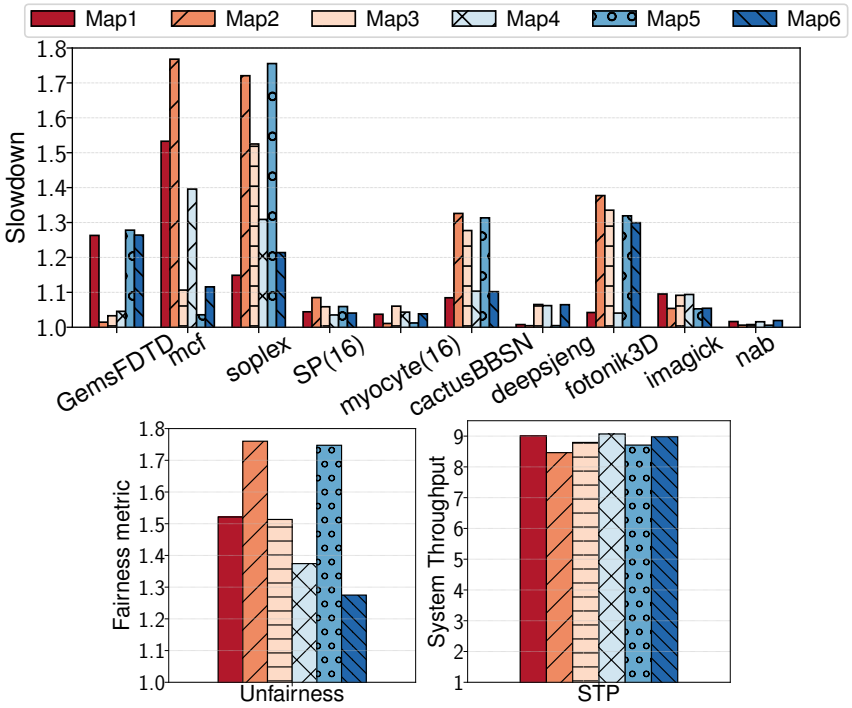


Figure 3.2: Per-application slowdown (top) and unfairness and throughput (bottom), for 6 different thread-to-LLC mappings of a 10-application workload under the LFOC+ LLC-partitioning policy. The numbers in parentheses by `SP` and `myocyte` (parallel) indicate their thread count.

the best one (Map6), which also delivers a higher degree of throughput, with a 6.1% improvement.

To understand why these divergences in unfairness occur, we focus on the slowdown associated with the `soplex` and `mcf` programs. Both applications are cache sensitive, and demand a substantial amount of dedicated space in the LLC to match the performance they deliver in isolation. Specifically, to reduce the slowdown of any of these applications below 1.05 (5%) under a scenario with no memory-bandwidth contention, each application requires a devoted LLC partition of over 55% of the total LLC size (i.e., at least 6 cache ways out of 11 for `soplex` [2]). Therefore, in mappings where both programs are assigned to the same NUMA node, such as Map2, it is simply unfeasible to simultaneously fulfill the LLC-space requirements of both programs, regardless of the underlying partitioning algorithm. By contrast, under Map6, where said programs are assigned to different NUMA nodes, and memory-bandwidth consumption is balanced across nodes, LFOC+ can grant a bigger LLC share to both applications (assigned to different LLCs), thus reducing their slowdown and improving unfairness. This observation indicates that when the demand for a particular shared resource (such as the LLC) is uneven across NUMA nodes, it becomes impossible to fulfill the requirements of all the programs. This uneven demand inherently leads to unfairness, thus limiting the potential of the underlying partitioning policy.

The results enable us to draw two insightful conclusions. First, optimizing system-

[2]The slowdown curve for program `soplex` can be found in Figure 2.9 (Chapter 2).

wide fairness in NUMA multicores (or, more broadly, in multi-LLC systems) is not possible solely by minimizing unfairness at each socket (or core group) separately, namely, without explicit control of thread-to-socket placements. Hence, previous fairness-aware cache-clustering policies [57,136,152] designed for UMA systems cannot automatically optimize system-wide fairness on NUMA multicores, as they make no decisions on thread-to-socket placements. Second, an LLC-partitioning policy that operates independently in different NUMA nodes cannot guarantee by itself repeatable performance and fairness across runs. Therefore, to optimize system-wide fairness and deliver repeatable results, a fairness-conscious LLC-partitioning policy must be complemented with a consistent strategy to map threads to sockets, ensuring a balanced demand for the memory-related resources shared among cores in each NUMA node. Notably, leaving thread placement decisions to the end user is burdensome, as it requires extensive application profiling, which is often impractical or unfeasible in multi-application scenarios, especially in general-purpose or cloud environments where unknown applications may be present.

Acknowledged the importance of consciously combining LLC-partitioning and thread-to-group assignments, this analysis raises a number of questions:

1. *Does the optimal fairness-wise mapping alone, without partitioning the LLC, constitute a good starting point to leverage fair LLC-partitioning later?*
2. *Should optimal thread-mapping and optimal LLC- partitioning (with partition sharing) be treated as separate optimization problems to be handled in sequence, or should they be addressed in a coordinated way to optimize system-wide fairness?*
3. *Is LFOC+ still effective in NUMA systems, so that it can serve as an appropriate building block of a fairness-aware resource manager?*
4. *How can we efficiently determine a good thread-to-socket mapping without extensively exploring all available mapping choices?*

3.2.2 Simulation study

To answer the questions posed in the previous section, we carried out a simulation study on optimizing the combination of LLC-partitioning and thread placement. In particular, we carried out a comprehensive exploration of theoretical solutions for specific strategies across various workloads, with specific thread-to-group mappings and fairness-oriented LLC-partitioning strategies. Finding the cache-partitioning or cache-clustering solutions that optimize a given optimization objective (e.g., fairness or throughput) is an NP-hard problem [59, 83, 136, 178]; the search space grows exponentially with the number of applications and LLC ways. This complexity presents a significant challenge in determining the optimal solution.

We tackle this challenge using for our analysis the PBBCache simulation tool [59], presented in Section 2.2.3. After including additional functionalities to PBBCache

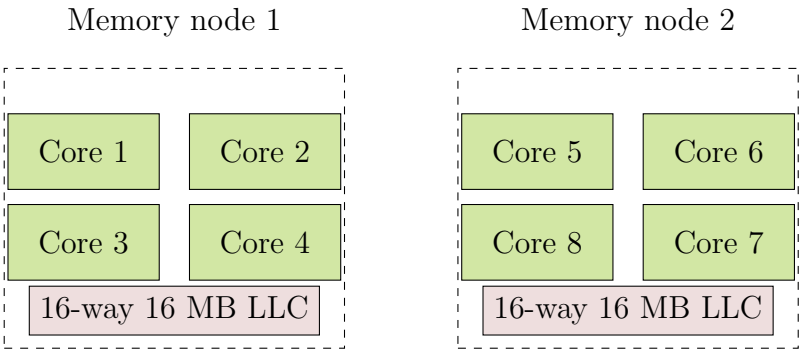


Figure 3.3: Platform A with 8 cores, organized into two four-core memory nodes.

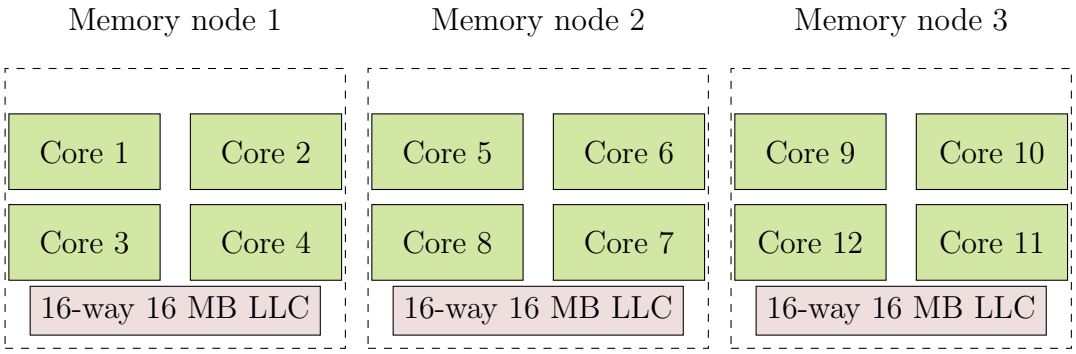


Figure 3.4: Platform B, consisting of 12 cores, is organized into three memory nodes.

(also outlined in Section 2.2.3), we could assess the effectiveness of strategies that combine thread-to-group placement with per-group LLC-partitioning in multi-core systems with separate LLCs (of which NUMA systems are a specific case). We recreated in the simulations two small-sized NUMA systems, Platform A and Platform B, whose hardware configurations are depicted in Figures 3.3 and 3.4, respectively. For both platforms, each node features a 4-core group with the same specifications. In particular, each node includes four cores sharing a 16-way 16 MB LLC, resembling the Core Complex (CCX) of the AMD EPYC Rome 7742 processor, introduced in Section 2.1.

For the study, we built 240 workloads by randomly mixing SPEC CPU2006 and CPU2017 programs, matching the number of applications (8 or 12) to the number of cores on each platform. The workloads were designed by mixing benchmarks from the three aforementioned application categories described by LFOC+ (light sharing, cache sensitive, and streaming). We focused on single-threaded programs and fixed application-to-group/partition mappings throughout the execution, for simplicity. We leveraged the PMCTrack tool to gather the necessary PMC metrics as input for PBBCache (refer to Section 2.2.3). Specifically, we collected the necessary hardware performance counter values during the execution of all workloads' applications, with varying number of available LLC cache ways, on the `Zen2-128` platform.

For each workload, we obtained four theoretical solutions:

1. **OptMap** explored how far we could go in terms of fairness without partitioning the cache. This is, it considered all possible thread-to-group mappings (without partitioning the LLC) and selected the mapping that provided the optimal (minimal) unfairness.

2. **OptMap+OptPart** treated the cache-partitioning and mapping problems as orthogonal: it would start with OptMap's mapping solution, and, on top of that, it would find the best way to partition the LLC (via cache-clustering) so that unfairness was minimized.

3. **Optimal** provided the all-encompassing optimal fairness solution; it combined optimal application-to-core mapping and optimal cache-clustering for the workloads, by exploring all possible mappings and finding the optimal cache-clustering solution for each mapping that minimizes unfairness.

4. **BestMapLFOC+** was the best solution, among all possible mappings, that minimized unfairness and where the LLC in each group was partitioned with LFOC+'s cache-clustering algorithm.

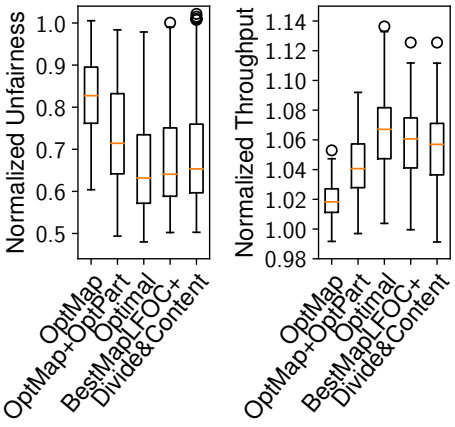


Figure 3.5: Unfairness and throughput distribution for the normalized simulation results, with 240 workloads across four theoretical solutions.

Figure 3.5 depicts the overall throughput and unfairness numbers obtained for the workloads by each theoretical approach, in addition to the mapping and partition implemented by our DC proposal, described in subsequent sections. After a thorough analysis (described in detail in Chapter 4), we were able to reach the following conclusions:

a. A huge gap exists (more than a 20% average reduction in unfairness) between Optimal and OptMap. Moreover, applying cache-clustering on top of OptMap (i.e, OptMap+OptPart) is still far from Optimal. Our results demonstrate that *the mapping of threads to core groups, optimized for fairness, does not uniformly translate into an optimal setup for implementing fairness-aware LLC-partitioning later*. Such mappings often result in imbalanced pressure on various shared resources across core groups, which limits the effectiveness of LLC-partitioning.

b. Our study underscores the need for a fully coordinated approach in executing LLC-partitioning and thread-to-group mappings to optimize system-wide fairness. Specifically, the assignment of threads to core groups must be tailored to facilitate optimal LLC-partitioning, thereby achieving the desired degree of fairness. In other words, our findings suggested that treating LLC-partitioning and thread mapping as orthogonal optimization problems leads to suboptimal solutions. Hence, *achieving effective fair resource partitioning requires thread-to-group assignments that balance memory-related resource pressure across NUMA nodes*. We consider this insight crucial for the design of NUMA-aware resource managers.

c. We confirm the suitability of the LFOC+ partitioning algorithm, known for its near-optimal fairness-oriented cache-clustering in single-LLC systems [136], as a helpful building block for developing fair NUMA-aware resource managers. This algorithm, in conjunction with an effective thread-to-group mapping strategy, demonstrates the capability to achieve fairness levels nearing those of the best-performing solutions among all potential cache-clustering choices.

d. Lastly, our analysis of the global fairness-wise optimal solution underscores *the importance of a well-balanced thread-to-group mapping in ensuring equitable competition for memory bandwidth and LLC space across core groups*. This is a principle that our OS-level proposal, DC, exploits to accelerate the mapping selection without exhaustive exploration.

3.2.3 Design and implementation of Divide&Content

Leveraging insights from our simulation study, we designed and implemented DC in the Linux kernel v5.10.114, as a plugin of PMCSched. DC dynamically maps threads to core groups (i.e., NUMA nodes in our platform) and allocates LLC space to the various applications at runtime via cache-partitioning. Our proposal adapts to program phases without requiring prior application knowledge or profiling.

At a high level, DC works as follows. When a thread enters the system, it is assigned to one of the core groups, which defines the set of cores the thread can run on. DC relies on the Linux scheduler's default functionality, which factors in the system's load across cores. As applications run, the operating system continuously classifies them online into the three cache-related categories considered by LFOC+: light sharing, streaming, and cache sensitive (see Section 2.2.4 for details). Periodically, and on a per-core-group basis, DC checks whether the degree of contention of the local group differs significantly from that of any of the remaining core groups. We consider the overall pressure on the different shared resources within each core group to determine its degree of contention based on a set of indicators that will be introduced later. If a remote core group is found such as there is a clear contention-wise imbalance between groups, a balancing algorithm is executed to address this imbalance. Thread migrations are triggered as requested by the contention-balancing algorithm, and once completed in the local core group, DC applies LFOC+'s partitioning algorithm (explained in detail in prior work [136]) to improve fairness. This

partitioning algorithm is also re-applied when threads mapped to the core group transition into a different application class.

LFOC+ is one of the key building blocks of DC. Nevertheless, because LFOC+ was originally designed for UMA systems with a single LLC, substantial changes were required in its original implementation to allow the independent utilization of the partitioning strategy within each core group. Chapter 4 describes these changes in detail. Apart from the necessary changes allowing to apply LFOC+ independently within each core group, additional modifications were necessary for its full integration with DC, where thread migrations are triggered across core groups. Specifically, our LFOC+ variant reacts to thread migrations to transfer per-application statistics (e.g., slowdown curves) to another group when needed (i.e., the first thread of an application is migrated onto a different group). Moreover, it exposes LFOC+'s current operating mode in each group so that DC inhibits contention-related migrations in the group when LFOC+'s sampling mode is enabled.

DC's contention-balancing algorithm – executed periodically – is explained in detail in Chapter 4. Here, we provide a high-level view of its inner workings and discuss an illustrative example. The overall idea is to map applications to the various core groups on the system to achieve a balanced competition for memory bandwidth and LLC space. To this end, the algorithm leverages two per-application indicators: bandwidth load (L_{bw}) and LLC load (L_{LLC}). An application's bandwidth load is defined as its total memory-bandwidth consumption, as reported by the hardware. The *LLC load* of an application matches its *LLC critical point*, which denotes the number of LLC ways at which the application's slowdown due to cache sharing falls below 5%. The higher the *LLC load* or critical point, the more LLC space the application requires to reduce its slowdown. The LFOC+ partitioning scheme – which DC relies on – automatically estimates the application's critical point when a new slowdown curve is obtained, as described in previous work [136].

DC's application placement decisions are based on the aggregated L_{LLC} and aggregated L_{bw} of each group, denoted as AG_{bw} and AG_{LLC}, respectively. Specifically, AG_{bw} and AG_{LLC} represent the sum of the associated contention indicators for all the applications currently assigned to a group. Essentially, DC strives to balance the degree of contention for shared resources by evening out the AG_{bw} and AG_{LLC} across core groups. To this end, it triggers thread migrations when necessary to redistribute contention and partitions the LLC to improve fairness via LFOC+.

Figure 3.6 illustrates how DC's contention balancing algorithm works using a hypothetical scenario where 8 single-threaded applications (labeled from `A` to `H`) run on a dual-socket 8-core system (two core groups). DC determines the application categories online via continuous PMC-aided profiling. In this case, four applications are classified as streaming (denoted in red in the figure), two are light sharing (in light green), and the remaining 2 are cache sensitive (in yellow). With the initial application-to-group assignment, DC identifies contention imbalance. On the left core group, high aggregate bandwidth consumption is detected due to the bandwidth demand of three streaming programs (`A`, `B`, and `C`); conversely, LLC-contention is not observed since no cache-sensitive programs are mapped to this group. This

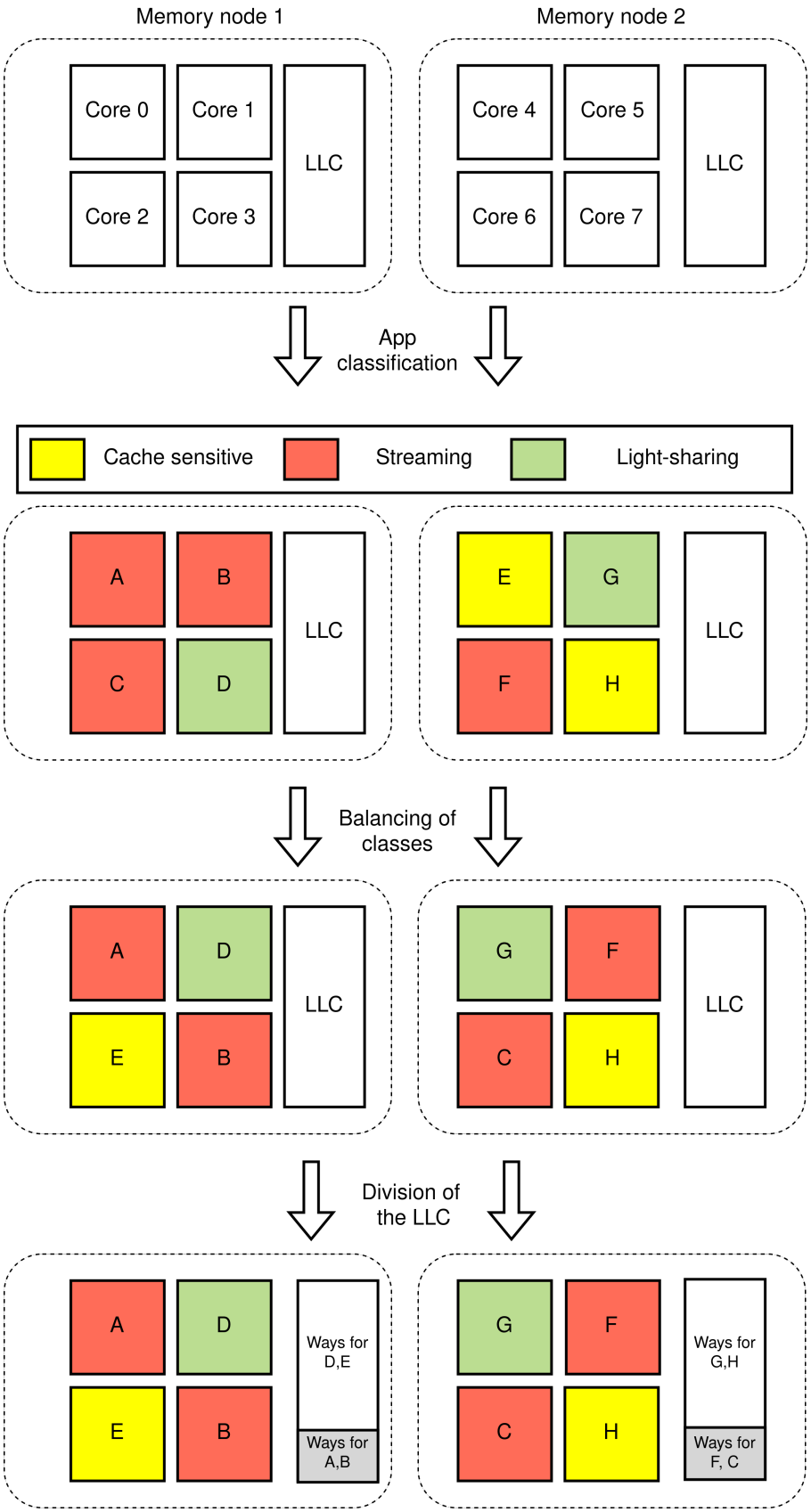


Figure 3.6: High-level overview of the contention balancing algorithm of Divide&Content.

contrasts with the right core group where LLC contention is the dominant factor, because it accommodates the only two cache-sensitive applications (`E` and `H`) in the workload. Catering to the AG_{bw} and AG_{LLC} indicators of both groups, DC alters the application-to-group assignment to reduce contention. Particularly, to minimize competition for LLC space, DC maps each cache-sensitive program to a different group, thereby balancing AG_{LLC} across groups. Moreover, it distributes streaming applications across groups to even out per-group AG_{bw} indicators. Finally, DC performs cache-partitioning with our variant of LFOC+. In this setup, DC allots the vast majority of LLC space to cache-sensitive programs so as to reduce their slowdown – which is key for minimizing unfairness.

Crucially, this example simply depicts a hypothetical situation at a given point in time, when applications go through a program phase exhibiting the features of a certain application category. Nevertheless, to cater to the time-changing application behavior, the contention balancing algorithm must be executed periodically. This, however, could potentially introduce overhead, as a result of the thread migrations triggered to balance contention across groups. Notably, DC implements a carefully crafted migration control mechanism to reduce migration overhead and work in synergy with Linux's automatic NUMA balancing feature [115]. This feature, enabled by default in recent kernel versions, exploits lightweight detection of page reuse, and triggers automatic page migrations when beneficial. Specifically, when an application is migrated to another memory node, NUMA balancing automatically migrates referenced remote pages to the current NUMA node.

To work in cooperation with Linux automatic NUMA balancing, DC exports a configurable parameter, `min_migration_period`, that dictates the minimum time a thread must remain in its assigned core group before being migrated to another. Thus, DC ensures that applications that have been recently migrated between core groups remain pinned to their core group for some time. This keeps page migrations automatically triggered by NUMA balancing under control, while permitting this kernel feature to migrate a substantial amount of the application's reused pages, which improves locality and reduces both effective memory latency and interconnect contention [26, 42].

3.2.4 Experimental evaluation

We performed an exhaustive assessment of the effectiveness of DC in comparison to three strategies:

1. **Stock-Linux**: the default scheduler in the unmodified Linux kernel v5.10.114, which does not partition the LLC.
2. **DINO** [26]: a state-of-the-art NUMA-aware contention-aware policy. DINO does not leverage LLC-partitioning; instead, it combines dynamic thread-to-core mappings with automatic page migration. In addition, unlike DC, DINO does not partition the LLC. To decide on thread placement, DINO classifies threads at runtime based on its current LLC misses per 1K instructions

Strategy	mean	max
DINO	1.34%	17.39%
DINO/LFOC+	5.41%	43.23%
Divide&Content	18.67%	44.35%

(a) Unfairness reductions

Strategy	mean	max
DINO	0.56%	5.00%
DINO/LFOC+	-2.91%	1.89%
Divide&Content	0.12%	5.36%

(b) Throughput gains

Strategy	mean	max
DINO	0.63%	9.36%
DINO/LFOC+	-2.31%	16.65%
Divide&Content	1.94%	17.95%

(c) Reductions in ANTT

Strategy	mean	max
DINO	-1.07%	28.28%
DINO/LFOC+	2.28%	53.14%
Divide&Content	33.24%	59.65%

(d) Reductions in UnfairnessCoV

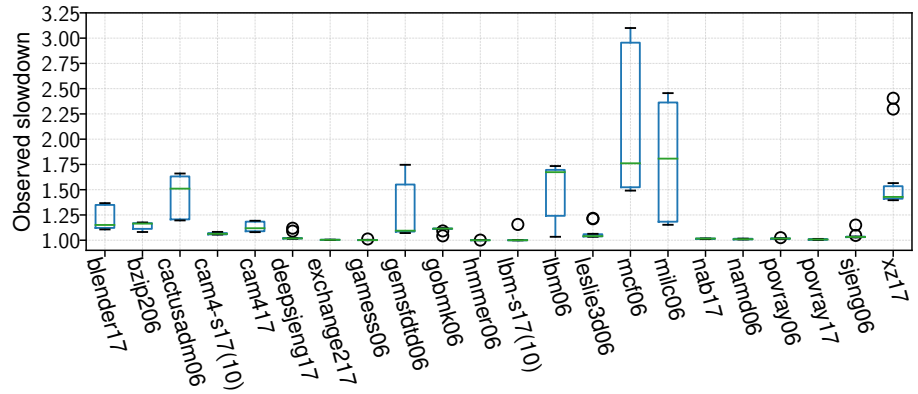
Table 3.1: Average and maximum gains/reductions for various metrics with respect to Stock-Linux.

(LLCMPKI) into three classes: *turtles* (low LLCMPKI), *devils* (medium LLCMPKI), and *superdevils* (high LLCMPKI). It spreads threads of the various classes across core groups so as to even out the aggregated LLCMPKI among them. To reduce the number of migrations, DINO updates threads classes and readjusts thread-to-core mappings with a coarse granularity.

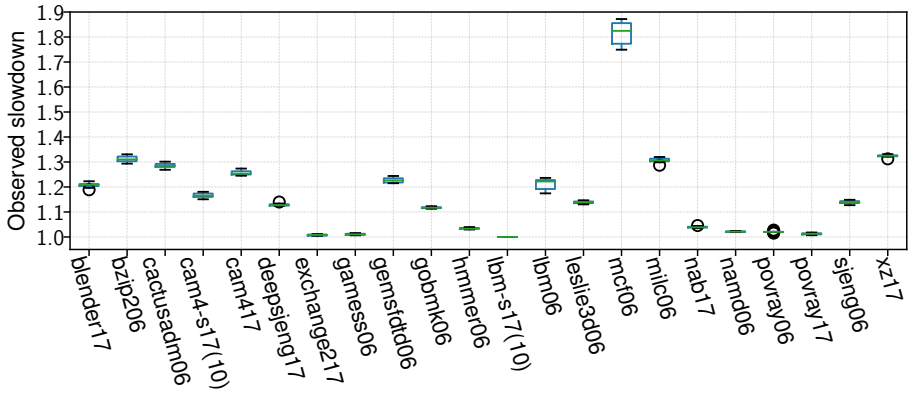
To perform an experimental comparison of our OS-level approach against DINO, we created an OS-level implementation of the latter. This implementation, like DC's, uses Linux's automatic NUMA balancing feature [115]. Notably, this kernel feature was unavailable when DINO was proposed.

3. **DINO/LFOC+**: a variant of DINO that partitions the LLC with DC's LFOC+ version. We created DINO/LFOC+ to analyze the impact of combining two existing complementary approaches [26,136] that make uncoordinated contention-aware decisions, namely, where thread placement is done without knowing that the LLC is going to be partitioned afterward.

We performed an exhaustive experimental evaluation of DC on the `Skylake-40` platform (a 40-core dual-socket NUMA system described in Section 2.1). We followed the standard practice when assessing the effectiveness of LLC-partitioning strategies for compute-intensive workloads [46, 112, 113, 121, 122, 136, 152]: considering randomly generated mixes comprising long-running programs from standard benchmark suites. Specifically, we used workloads that combine single-threaded and multithreaded applications and utilize 61 different programs from 6 benchmark suites: SPEC CPU2006, CPU2017, PARSEC3, SPEC OMP2012, NAS Parallel Benchmarks and Rodinia. Our experiments included 30 randomly generated program mixes that combined varying amounts of streaming, cache-sensitive, and light-sharing applications. In all workloads, the total thread count was set to match the number of cores on our platform (40). Particularly, all workloads included 2 multithreaded programs, and a varying number of single-threaded applications.



(a) Stock-Linux



(b) Divide&Content

Figure 3.7: Distribution of the per-application slowdown for one of the tested workloads under Stock-Linux (a) and Divide&Content (b).

The detailed composition of the various workloads considered, as well as the full discussion of our experimental results, can be found in Chapter 4. Here, we summarize the main insights of our experimental evaluation:

1. DC outperformed other contention-aware strategies in terms of fairness for most program mixes, reducing unfairness by up to 44.35% compared to Linux. Against DINO and DINO/LFOC+, DC improved fairness by 17% and 13.3% on average, respectively. Specifically, Table 3.1 provides the average and maximum reductions of unfairness and throughput delivered by the contention strategies DINO, DINO/LFOC+, and DC. Although the throughput differences among the strategies were minor, generally within a 4% range, DC managed to achieve significant fairness gains without a notable impact on average throughput.

2. For completeness, Table 3.1(c) and 3.1(d) report two additional fairness metrics – Average Normalized Turnaround Time (ANTT) and UnfairnessCoV – along with the other metrics, all defined in Section 3.1. DC reduced the ANTT (as shown in Table 3.1(c)) by nearly 2% on average and achieved a 33.24% UnfairnessCoV reduction, which demonstrates its effectiveness in improving system-wide fairness.

3. The default Linux kernel, in contrast to DC, exhibits substantial variability in performance metrics across different runs of the same workload. This is due to the Linux scheduler's random initial thread-to-core-group mappings and its strategy of minimizing thread migrations, particularly across NUMA nodes. To illustrate this fact, Figure 3.7(a) shows the per-application slowdown values observed across 10 independent runs of one of the multi-program workloads we tested with. The different application-to-group mappings that Linux performs across runs does not enable it to deliver balanced and consistent degree of LLC and memory-bandwidth contention from run to run. This leads to significant and highly-variable slowdowns, especially for cache-sensitive and bandwidth-intensive programs. In contrast, DC is not subjected to this variability issue. As Figure 3.7(b) reveals, DC achieves consistent slowdowns across runs for the aforementioned workload.

4. DINO distributes high-LLCMPKI applications across core groups, helping balance the LLC miss rate across NUMA nodes and thereby reducing bandwidth contention. This approach leads to modest throughput improvements over Linux. However, DINO's fairness often lags DC. This is also due to DINO's reliance on LLCMPKI, which can be misleading as it does not accurately reflect an application's sensitivity to resource contention. As a result, DINO fails to differentiate between cache-sensitive and streaming programs, often placing them together, which degrades the performance of cache-sensitive programs.

5. DINO/LFOC+'s generated mixed results across workloads. Though LFOC+'s partitioning on top of DINO brings significant fairness improvements in some cases, DINO/LFOC+ underperforms DINO in many program mixes. This is chiefly due to two reasons. First, DINO's thread-to-group mappings often lead to an uneven distribution of cache-sensitive programs, resulting in high LLC-space competition within some groups – making it difficult for LFOC+ to provide substantial benefits. Second, DINO/LFOC+ sometimes increases cross-group migrations compared to DINO, leading to additional overhead that negates LFOC+'s benefits. These frequent migrations are caused by variations in application classes due to changes in LLC occupancy, causing fluctuating application classes and extra migrations. These issues did not affect DC, which satisfactory leverages LFOC+ by balancing LLC space competition across core groups.

6. Regarding overhead, an analysis was conducted revealing that DINO and DINO/LFOC+ have a cross-group thread migration rate approximately 2.5 times higher than DC. This lower rate in DC is due to its migration avoidance mechanisms and LFOC+'s stable classification method. Additionally, DINO's contention-aware placement algorithm takes nearly twice as long to execute on average compared to DC's contention balancing algorithm; albeit both take a couple tens of microseconds in one core per second, which does not constitute a substantial system overhead.

3.3 An elasticity manager to improve CPU utilization on CMP systems

As stated in Section 1.1.2, maximizing resource utilization in multicore servers is paramount for increasing the benefits on cloud environments. The vast majority of proposals in this context attempt to reduce resource idleness by leveraging the execution of in-house workloads of the cloud provider during periods of resource underutilization in datacenter servers [33, 34, 92, 116, 175]. By contrast, this thesis explores a complementary approach to maximizing resource usage that relies on the execution of elastic HPC/scientific workloads.

Particularly, we exploit the parallelism inherently available in many HPC/scientific workloads to effectively utilize cores that may go temporarily idle in a shared server, even for short time periods. To this end, our resource-management proposal dynamically adjusts the number of active worker processes/threads in an HPC workload, which successfully boosts the performance of elastic HPC applications while increasing CPU utilization and system throughput. Notably, it does so with minimal impact on the performance of the remaining applications. To accomplish this goal, we exploit OS-level monitoring for efficient detection of idle cycles, and leverage explicit interaction between the OS and elastic HPC applications.

To benefit from the elasticity features of our proposed framework, HPC applications must support dynamic *malleability*, this is, the ability to utilize a varying number of threads/cores at runtime. In HPC, delivering malleability support to applications is simpler than in other scenarios, as HPC applications often make use of runtime systems; implementing malleability support at the runtime system level may require little or no changes to the application's source code. Despite the fact that applications can be adapted to be malleable by changing its implementation, in this thesis we explored ways to bring malleability to applications transparently via explicit runtime system support and OS-runtime interaction. This way, the application code does not need to be modified to adopt malleability. As a proof of concept for this approach, we augmented the GNU OpenMP runtime system with malleability support in *parallel for* constructs. We show that this approach brings malleability to a wide range of well-known OpenMP benchmarks, thus allowing us to quantify the benefits that come from using unmodified applications.

Figure 3.8 depicts the intended use scenario of our proposed elasticity framework and the interactions between its various components. We assume that the workload combines a mix of applications, where each application may run either directly on top of the operating system or inside a separate container. Some applications run unmodified programs – potentially, legacy software – in which threads block when not doing useful work. This is typical in most cloud-like applications, ranging from database workloads to web servers. These applications can be single- or multi-threaded, and they may or may not use a runtime system. In case a runtime is used, it does not interact with the OS or exploit malleability. Henceforth, we refer to these programs as *non-malleable* applications. Other programs leverage malleability through a cooperative runtime system that runs in user-space and interacts with

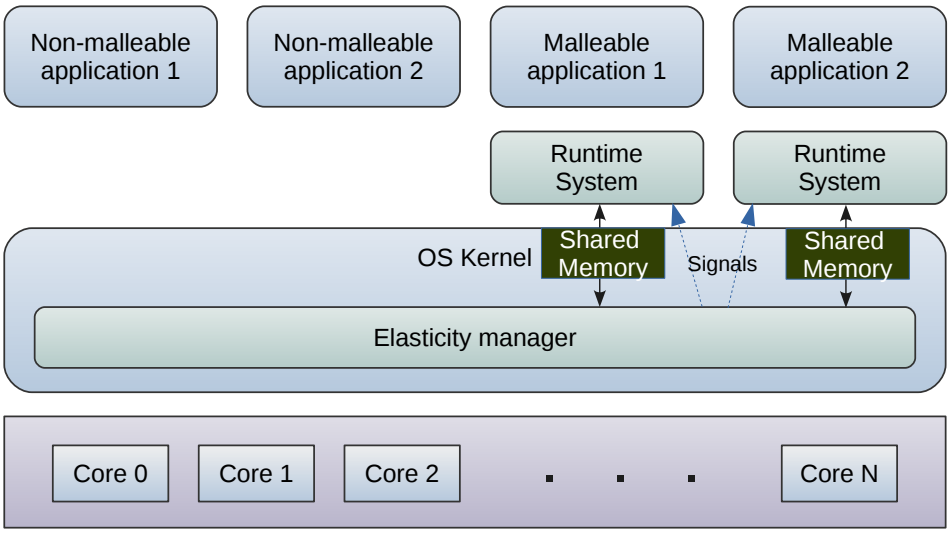


Figure 3.8: Diagram of the elasticity manager, including the shared memory region for communication between the runtime and the elasticity manager.

our OS-level elasticity manager.

At a high level, our proposed system works as follows. An OS-level elasticity manager (EM) continuously monitors the CPU utilization of non-malleable applications. To this end, it employs a carefully crafted mechanism that relies on tracking the amount of time threads spend in runnable and non-runnable states. When at least one non-malleable application consistently leaves cores idle, the kernel reassigns them to malleable programs. To do so, it communicates the target active thread count to the application through a shared memory region, where the OS notifies changes via a signal delivered to the associated process. The application or malleabillity-enabled runtime system is ultimately responsible for enforcing the desired active thread count. At the same time, EM exploits thread packing to confine malleable and non-malleable programs in different core groups, so as to minimize interference among applications.

Our proposed mechanism complements vertical elasticity in the cloud and could also enhance CPU utilization in general-purpose OSs on desktops. To the best of our knowledge, this was the first proposal to use elastic OS-runtime resource management for both malleable and non-malleable applications without requiring changes to their source code (we provide the necessary application support at the runtime system level). Previous techniques for improving CPU utilization depend on customized runtime systems and are incompatible with legacy software [36,67,167]. On the contrary, our approach leverages explicit OS-runtime interactions, making it suitable for a wider spectrum of applications.

The remainder of this section is structured as follows. Section 3.3.1 describes the interface provided by the OS-level elasticity manager for direct communication with malleable applications. Section 3.3.2 provides a high-level description of the inner workings of our OS-level elasticity manager. Section 3.3.3 discusses our malleability extensions to GNU's OpenMP runtime system (aka *libgomp*), which targets appli-

cations based on parallel loops. Lastly, Section 3.3.4 summarizes the most relevant insights drawn from the experimental evaluation of our proposal.

3.3.1 OS-runtime interface

To share information between the OS kernel and applications, we leveraged PMC-Sched's built-in support specifically designed for that purpose (refer to Section 3.1.2). In particular, the communication between the OS kernel and the malleable application (or the malleability-enabled runtime system) takes place through a per-application shared memory region. In our implementation, this region holds a simple two-field data structure. The first field, called `is_malleable`, is a flag that the runtime system sets to inform the OS that the application supports malleability. The second field (`target_threads`) is exclusively modified by the OS-level elasticity manager, and exists to inform the runtime system of the maximum allowed number of active threads for the application. Every time this integer field is altered, the elasticity manager sends a signal to the application process. We opted to deliver the signal SIGUSR1 as it is reserved for user-defined purposes in the POSIX standard. The application or malleability-enabled runtime system must install the associated signal handler and carry out the necessary implementation-specific actions upon signal reception to enforce that maximum target thread count.

3.3.2 OS-level Elasticity Manager (EM)

We implemented our elasticity manager (EM) as an OS scheduler extension in the Linux kernel v5.16.20. Specifically, this extension is bundled as a plugin of the PMCSched framework. The EM operates on top of the default Linux scheduler, by monitoring the typical number of unused cores left by regular (non-malleable) applications, and by dynamically assigning available cores to malleable applications. While the EM monitors the number of idle cores on the system in a continuous fashion, dynamic core redistribution occurs periodically at a configurable rate specified by the `em_period` parameter.

Accurately determining the number of idle cores left by regular applications is a crucial aspect of our OS-level proposal. During initial design stages, we found that relying on an application's average percentage of CPU utilization – obtained from builtin statistics already gathered by the Linux kernel – does not enable us to accurately determine the typical number of cores used by an application during a certain period. Particularly, this value tends to underpredict the application's effective CPU utilization; in fact, the observed utilization may be greater than the average for a substantial amount of time during a given monitoring interval. Therefore, the average percentage of CPU utilization is not a practical estimate for the number of cores that the application utilizes.

To accurately track effective CPU usage, the EM continuously monitors thread transitions between runnable and non-runnable states – for example, when a thread goes

to sleep due to a pending I/O, terminates, or wakes up from a blocking synchronization primitive. In reacting to these state transitions, the EM tracks the amount of time that each application spends with different runnable thread counts. This information is stored in a per-application *activity vector* [3], where each i-th entry stores the amount of time (Ti) that the application spends with i runnable threads. To predict an application's effective CPU utilization, the EM employs the Effective Thread Count (ETC) metric, which EM calculates on its periodic activation by leveraging the application's activity vector. Specifically, an application's ETC is defined as the highest runnable thread count k such that $\sum_{i=k}^{N} T_i \geq usage$, where *usage* denotes a configurable time, defined as a small percentage of the EM activation period (`em_period`). Intuitively, an application's ETC captures the highest active thread count k observed such that it spends most of the activation period with a runnable thread count smaller than k. The number of idle cores the application leaves during that period is calculated by subtracting its ETC from the number of cores assigned to it.

During its periodic activation, the EM leverages a core-allocation algorithm described in detail in Chapter 5. To illustrate how the EM dynamically distributes idle cores among malleable applications, let us consider the step-by-step example depicted in Figure 3.9. In this hypothetical scenario, two applications run on an 8-core system. One of the programs is malleable and the other is not. For simplicity, we will refer to the malleable application as program *M* and to the non-malleable one as program *N*. In the example, the number of active threads from program N (in red) vary over time, hence leaving idle cores sometimes. The EM opportunistically exploits those idle cores to run threads from program M (in blue), as follows:

Step 1 In this initial scenario, both applications run using half of the cores: threads from program N are assigned to cores 0-3, whereas threads from program M are mapped to cores 4-7. Moreover, program M spawns 4 additional worker threads that are blocked at the beginning. Some (or all) of these worker threads will be awakened by the malleability-enabled runtime system when the EM increases the core allocation for program M. This initial setup – with the thread count of program M totaling the core count – reflects our experimental choice to set the maximum number of threads to the system's core count, a standard approach for exploiting application malleability [36, 162].

Step 2 At a certain point, two threads from program N become inactive, as shown in Figure 3.9b. The EM effectively detects this situation by continuously monitoring the active thread count of program M and calculating the number of idle cores it leaves. The application's ETC is used for that purpose, as explained earlier.

Step 3 Since application N consistently leaves two cores idle (cores 2 and 3, in the example), the EM allots two extra cores to application M, by changing

[3] Maintaining this vector has negligible overhead. In particular, the EM makes changes in a process's activity vector only when one of its threads becomes inactive (blocks or terminates) or runnable again; these events can be captured by leveraging specific callbacks provided by PMCSched. Hence, when the system is idle, these callbacks are not invoked.

the affinity mask of its threads. Notably, for this application to utilize those extra cores, the EM explicitly interacts with it. Particularly, the EM alters the `target_threads` field in the memory region shared between the OS kernel and the runtime system, then delivers the `SIGUSR1` signal to the corresponding process. Upon signal reception, the runtime system increases the worker thread count in accordance with the current number of cores allotted.

Step 4 Eventually, the newly activated worker threads will populate all available cores (See Figure 3.9d). Note that the Linux load balancer is responsible for this, reacting to the changes in the affinity mask of the various threads of program M.

Step 5 The EM is also responsible for reclaiming cores temporarily "stolen" from non-malleable applications when the CPU utilization of these applications increases. This is reflected in Step 5 (See Figure 3.9e), where the effective thread count (ETC) of program N increases to 4.

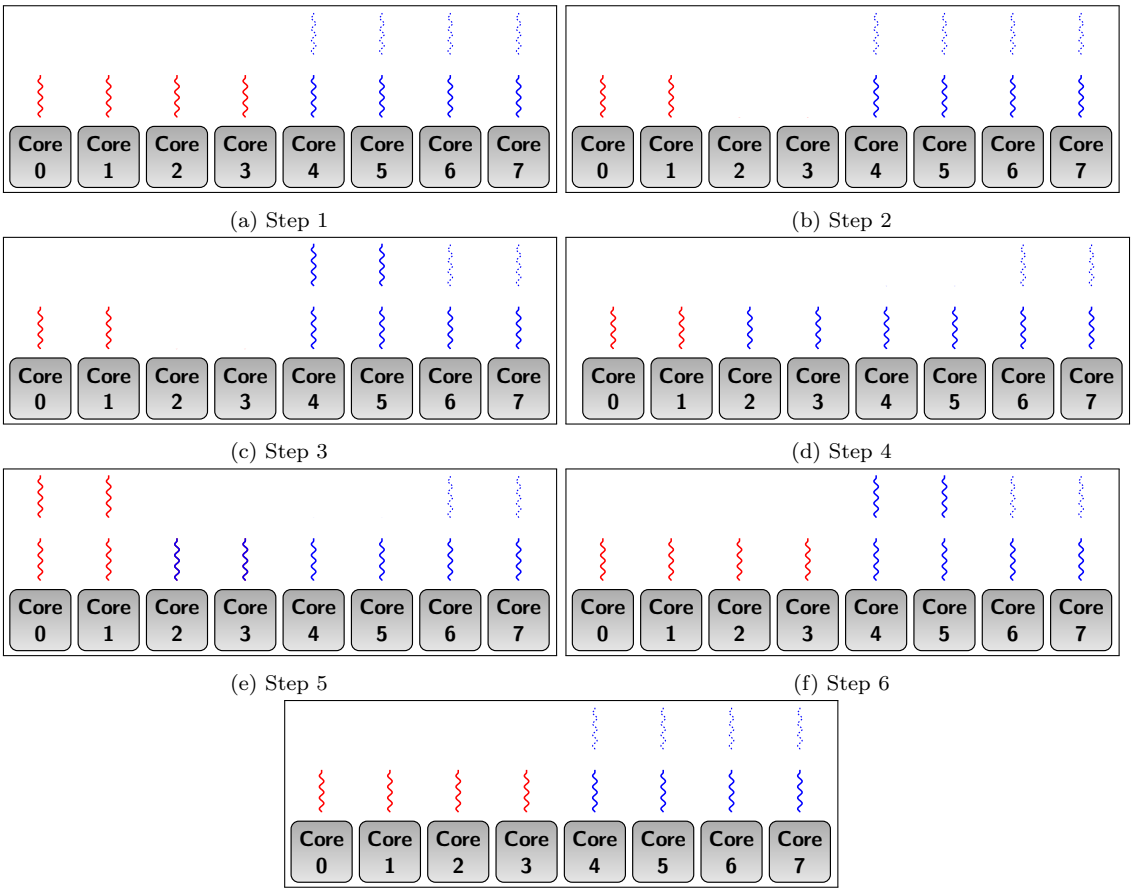
Step 6 To reduce the performance penalty caused by oversubscription scenarios like Step 5, the EM applies *thread packing* to quickly reassign "stolen" cores to non-malleable program. This is accomplished by modifying the affinity mask of all threads in program M. In response to that change, Linux's load balancer updates the thread-to-core mappings, effectively reassigning the necessary cores to program N. At the same time, application M runs for some time with an active thread count greater than its number of allotted cores.

Step 7 We arrive back at the initial scenario, when the EM tells the malleable program to reduce its active worker count. This is effected by altering the contents of the shared memory region and signaling the runtime system, which, upon signal reception, blocks the necessary worker threads.

3.3.3 Extensions to OpenMP runtime for malleability

Our malleability extensions for OpenMP were incorporated into the GNU OpenMP runtime system that comes with GCC 11.2, also referred to as *libgomp*. Notably, the OpenMP standard already allows developers to request the utilization of different number of worker threads in the various parallel regions of an application. While this constitutes a form of malleability, its coarse granularity makes it impossible to react in our desired target time frame (a matter of milliseconds) in the vast majority of loop-based OpenMP programs.

To achieve a finer-granularity and react as soon as possible when a change in the application's thread count is requested by the kernel, we chose to implement the malleability control within loop-scheduling methods. As as proof of concept, we modified the `dynamic` and `guided` loop-scheduling methods. Note that the `static` loop-scheduling technique is not amenable to malleability. Indeed, in *libgomp's* implementation, each worker thread invokes up to one runtime call at the beginning of the loop to get its full share of work.



(a) Step 1

(b) Step 2

(c) Step 3

(d) Step 4

(e) Step 5

(f) Step 6

(g) Step 7

Figure 3.9: Elasticity Manager in action with two applications running on an 8-core system.

To adopt malleability within the GNU OpenMP runtime system, we made a number of modifications, some of which affect environment variables. Specifically, we introduced new environment variables to allow the user to enable or disable malleability and to indicate the maximum number of threads the application can run with. In our experiments, we systematically set the maximum thread count to the total number of cores in the system, as it is standard practice when exploiting malleability [36, 162]. To engage malleability within parallel loops, we employ a global a bit array, including an entry for each thread to indicate whether it is *active* or *inactive*. Within a loop, active threads act as normal OpenMP worker threads, while inactive ones remain blocked inside the runtime calls associated with removing loop-iterations from the pool in the `dynamic` or `guided` methods. In the `dynamic` method, each thread receives a fixed chunk size with every allocation, meaning each thread processes the same predefined number of iterations in each assignment. In contrast, the `guided` method adjusts the chunk size as more threads pick up work; to determine the current (monotonically decreasing) chunk when a thread removes iterations from the pool, we use the current number of active threads in the calculation rather than the total thread count.

Upon receiving the `SIGUSR1` signal delivered by the OS-level elasticity manager, the runtime system reads the new target thread count value from shared memory, modifies the bit array of active threads accordingly, and wakes up the necessary threads. When a newly active thread wakes up, it proceeds to remove iterations from the pool and executes them. Should the number of active threads decrease upon signal reception, the threads that were recently forced to become inactive will block when invoking any runtime call to obtain loop iterations from the shared pool.

3.3.4 Experimental evaluation

Our evaluation was performed on the `CascadeLake-16` platform (refer to Section 2.1). To assess the effectiveness of our proposal, we experimented with both POSIX threads (aka *pthreads*) and OpenMP programs. Specifically, we considered multithreaded applications from different benchmark suites, including NAS Parallel Benchmarks [18], PARSEC [22], and Rodinia [31], among others. We ran 24 application mixes (named M1-M24), each including a non-malleable (N) program with limited Thread Level Parallelism (TLP) and a malleable (M) OpenMP application. The detailed composition of each M*i* workload can be found in Chapter 5. For each workload, we measured the performance of each application (completion time) and the system throughput (STP metric).

We ran each workload in three different scenarios:

1. In the first one, referred to as *Stock-Linux*, no malleability OS extensions are loaded and applications use the unmodified version of the OpenMP runtime system. Under such circumstances, which are depicted in Figure 3.10, both applications use fixed thread counts (8 threads each). Moreover, the threads of

program N are mapped to CPUs 0-7 via user-enforced affinities, while threads of the other application run on the remaining cores (CPUs 8-15).

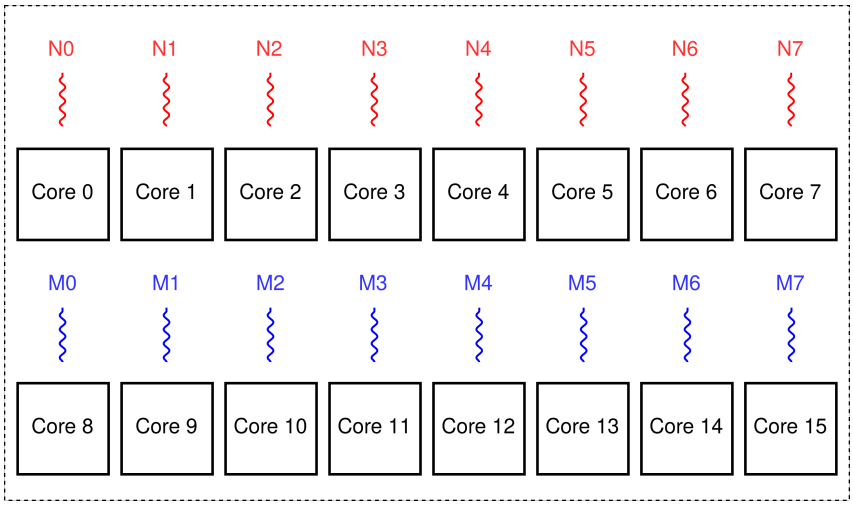


Figure 3.10: Stock-Linux scenario: 8 threads (N0-N7) belong to non-malleable program N, and 8 threads (M0-M7) to malleable program M.

2. In the second scenario, denoted as *Oversubscription*, we run the M-program with 16 threads and the N-program still uses 8 threads and fixed thread-to-core mapping (refer to Figure 3.11). This scenario does not employ malleability, but exploits the fact that program N temporarily leaves idle cores, which can be leveraged by additional threads (one thread per core) of program M. We considered this scenario to assess (1) how effectively the stock OS scheduler together with the unmodified runtime system administer idle cores, and (2) what is the impact that applying oversubscription has in the performance of program N.

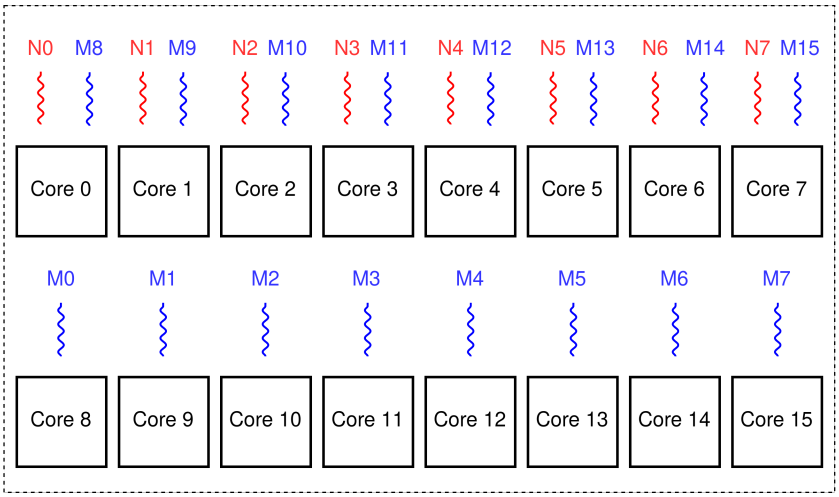


Figure 3.11: Oversubscription scenario: 8 threads (N0-N7) belong to program N, and 16 threads (M0-M15) to program M.

3. In the last scenario, shown in Figure 3.12, we enable our kernel-level elasticity

manager (EM) and activate the malleability support in the modified OpenMP runtime system. No user-enforced CPU affinities are used in this case and, as opposed to the remaining scenarios, the active thread count of the OpenMP (malleable) application is dynamically adjusted based on the number of idle cores left by the other multithreaded program.

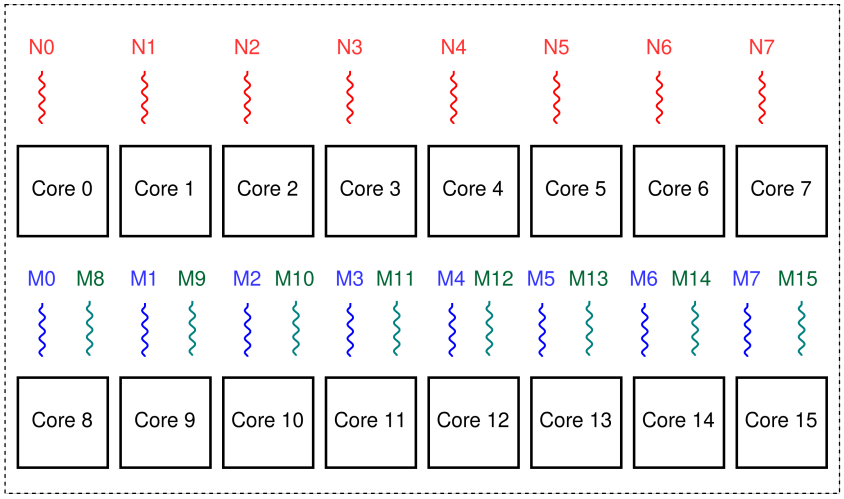


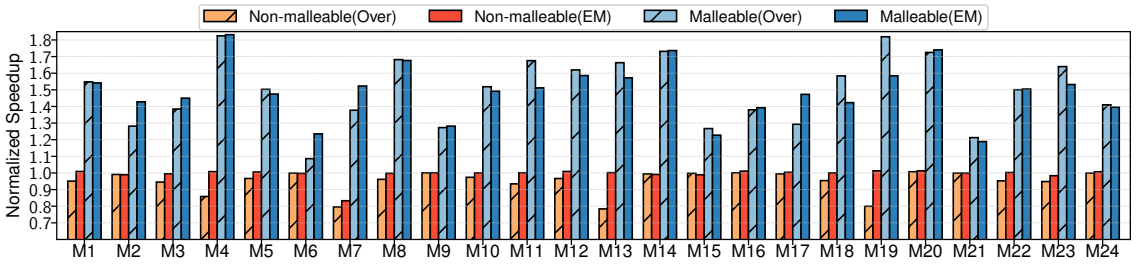
Figure 3.12: EM scenario: 8 threads (N0-N7) belong to the Non-Malleable (N) program, and 8 threads (M0-M7) to the Malleable (M) program. Eight additional worker threads are also created (M8-M15) but they are initially inactive (in green).

Figure 3.13 shows the normalized per-application speedup and throughput (using Stock-Linux as baseline) for each workload (M1 to M24). Chapter 5 provides a detailed discussion of the results presented in this figure. For the sake of simplicity in the discussion in this section, Figure 3.14 offers a compact summary of the results, displaying the distribution of the relative per-application speedup across workloads for non-malleable and malleable programs (Figure 3.14a), as well as the distribution of normalized throughput (Figure 3.14b).

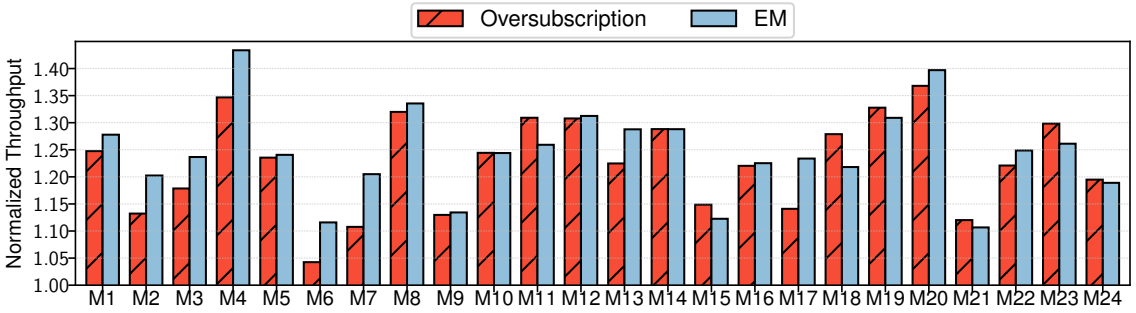
These were the main conclusions from our experiments:

- The results reveal that the EM delivers substantial performance gains for malleable programs, achieving speedup of up to 83%, as shown in Figure 3.14a. Note that Figure 3.13a also allows to see that this maximum benefit is obtained for the M4 workload. Overall, this acceleration is possible thanks to the effective utilization of idle cores left by non-malleable programs, all of which exhibit limited thread-level parallelism (TLP). Importantly, the performance of non-malleable applications remains mostly unaffected, as evidenced by their speedup values clustering closely around 1.0, with a maximum deviation of just 1.7% from the baseline provided by Stock-Linux.

- The substantial acceleration of malleable programs across the board also yields a considerable increase in system throughput. As Figure 3.14b reveals, the

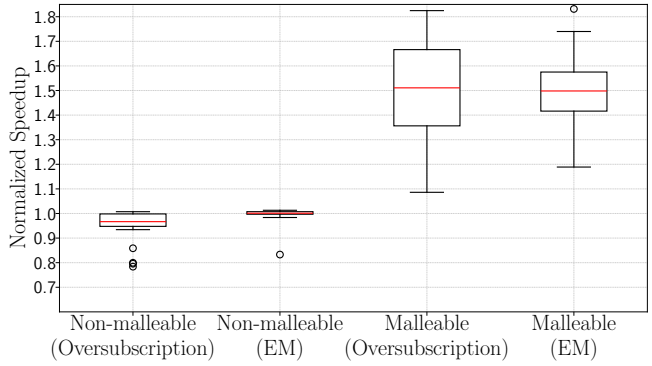


(a) Normalized speedup for each application under Oversubscription and EM

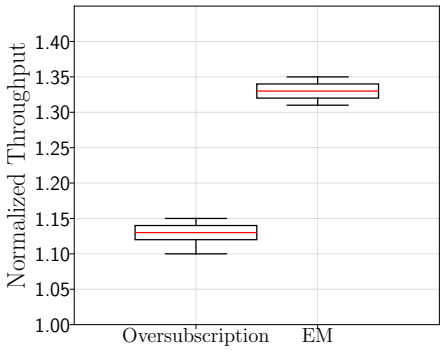


(b) Normalized system throughput

Figure 3.13: (a) Relative speedup for each application in the workload under Oversubscription and EM vs. Stock-Linux, and (b) normalized throughput for the various program mixes.



(a) Distribution of normalized speedup



(b) Distribution of normalized throughput

Figure 3.14: Distribution of the relative speedup (a) and normalized throughput (b) between oversubscription and the elasticity manager (normalized to Stock-Linux).

EM improves overall throughput by up to 43%, with an average gain of 34.5% compared to Stock-Linux.

- Leveraging oversubscription may bring additional throughput improvements in some cases, as shown in Figure 3.13b. Unfortunately, this also consistently negatively impacts the performance of the non-malleable program, as results in Figure 3.13a reveal; this is the case for M11, M18, and especially M19. Essentially, running the M application with 16 threads throughout the execution (what happens in Oversubscription), makes it possible to achieve higher performance for this program w.r.t. the EM scenario in many workloads. Unfor-

tunately, accelerating the M application under oversubscription usually comes at the expense of degrading the performance of the non-malleable program. Essentially, program N's threads often have to time-share a core with threads from the other application, thus causing uneven progress across threads and, in turn, deteriorating its performance. By contrast, the EM guarantees lower overheads for non-malleable programs, as it strives to avoid oversubscription when exploiting malleability dynamically.

3.4 Runtime and OS extensions for scheduling on asymmetric multicore systems

To advance the state of the art in throughput-optimized scheduling for asymmetric multicore processors (AMPs), we designed, implemented, and evaluated a set of custom scheduling extensions for the `AlderLake-16` platform described in Section 2.1. Recall that the processor on this system features two types of cores: performance cores and efficient cores – in Intel's nomenclature, referred to as P-cores and E-cores, respectively.

As indicated in the thesis introduction, a key innovation first introduced by Intel with the Alder Lake processor family is *Thread Director* (TD) [71, 131]. TD is a set of hardware extensions designed to assist the OS and other system software layers to decide on thread-to-core mappings in the workload. Note that this constitutes a challenging scheduling decision on asymmetric multicores, due to the diversity in performance that the various applications derive from the different core types present on the system. In this thesis, we make the following contributions to asymmetry-aware scheduling:

1. We conduct a survey on the current status of Linux's support for Intel's TD, detailing its inner workings. We also present key experimental data gathered with PMCSched that illustrate relevant limitations of Intel Alder Lake platforms. Furthermore, we perform an experimental evaluation of Intel's recently submitted TD scheduling patches (hereinafter referred to as Linux-ITD) for potential inclusion in the Linux kernel's *mainline* [108].

2. We propose OS scheduler extensions to accelerate loop-based OpenMP programs. Particularly, we present Flexible AID, a set of enhancements to asymmetric iteration distribution (AID) [138] of parallel loops in OpenMP programs. Flexible AID, implemented in GCC's runtime `libgomp` v11.2, includes loop-scheduling methods adapted to AMPs to deliver better performance and exploit coordinated scheduling decisions between the OS and the runtime system.

3. We leverage the synergistic cooperation between the OS and the runtime system. To this end, we created a PMCSched-based implementation of HSP [143], an OS scheduler designed to maximize system throughput on AMPs. We

created variants of HSP that utilize either TD hardware or PMCs to dynamically guide thread-to-core mappings. The effectiveness of these variants was assessed with various multi-application workloads, including unmodified OpenMP applications using Flexible AID.

The remainder of this section is organized as follows. Section 3.4.1 condenses our survey on the current status of Linux's support for TD. Section 3.4.2 presents Flexible AID. Section introduces the HSP scheduler, and outlines our HSP implementation with PMCSched, which leverages OS-runtime interaction 3.4.3. Finally, Section 3.4.4 enumerates the main conclusions drawn with our experimental analysis.

3.4.1 Study and evaluation of Intel Thread Director

Intel Thread Director (TD) is a set of hardware extensions designed to assist the OS in making thread scheduling decisions on Intel asymmetric multicore processors [70, 71], starting with Intel's Alder Lake microarchitecture. TD-enabled hardware allows interaction with the OS kernel via a set of privileged model-specific registers (MSRs) and a memory-resident table populated by the hardware, which exposes key information on the performance and energy-efficiency characteristics of the various cores. This table, referred to as the TD table, is only directly accessible from the OS kernel, which must map it to its virtual address space for proper utilization.

As a thread runs on a given logical processor (LP), the TD hardware continuously collects runtime statistics for the thread and assigns it a *TD class*, which may change over time. The hardware supports a fixed number of classes (depending on the processor model), with each class identified with an ID ranging from 0 to the total number of classes minus one. To read a thread's class ID on the current LP, the OS must periodically read a model-specific MSR. The entries in the TD table associated with that particular class ID provide the OS with performance and energy-efficiency scores for the various LPs. These scores, which range from 0 to 255, can be effectively leveraged from the system software to map the various threads to the most appropriate core type to optimize different target metrics (e.g., system throughput).

Figure 3.15 depicts the structure of the TD table present in the `AlderLake-16` platform, which we used for our experiments. This processor supports 4 TD classes (0-3). The TD table is composed of a global header – which features a timestamp and flags to indicate recent updates made by the hardware to the TD table –, and a set of feedback entries for different LPs that contain the aforementioned performance (PE) and energy-efficiency (EE) score values for each class ID. In our platform, the TD table has ten 8-byte feedback entries, with 1 byte for each score type and class. Each P-core has its own entry (the first eight entries), while each group of four small E-cores sharing an L2 cache also share a common feedback entry (making the last two entries). The TD table on our platform is populated when the OS first

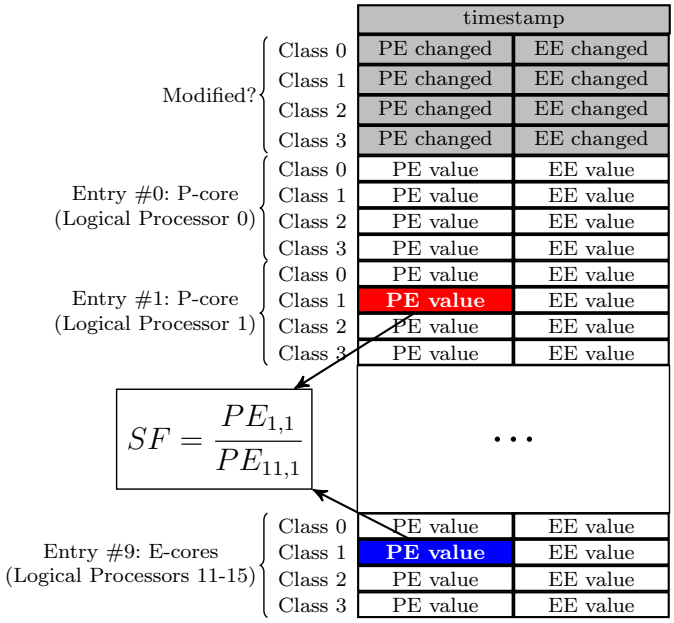


Figure 3.15: Structure of the Thread Director table in our Alder Lake experimental platform.

configures the TD hardware and remains unaltered thereafter. More information on the structure of the TD table can be found in the corresponding Intel manual [70].

Our work focuses on solutions for maximizing the system throughput automatically from the system software without requiring changes in the applications. In this context, scheduling must consider the performance benefit that each thread/task in the workload derives when it runs on a big core, compared to a small one. Henceforth, we will refer to this relative benefit as the thread's Speedup Factor (SF). Using TD, a thread's current SF can be approximated by dividing the P-core and E-core *PE* values for the thread's observed class ID (as reported by the hardware) from the TD table.

In this thesis, we conducted an analysis of the TD technology. To the best of our knowledge, this was the first experimental analysis of this technology on the Linux kernel, which still does not officially leverage TD-aided scheduling support in *mainline* to this date; the latest stable kernel at the time of this writing is v6.10. Critically, we found that a significant shortcoming of TD hardware on our platform is that, *while TD does report a valid class ID on P-cores most of the time, it always returns invalid class IDs* [4] *for any thread running on E-cores* [23, 146]. This poses a major limiting factor for the effectiveness of TD-based schedulers.

A recent paper by Intel [131] provides more details on the specific hardware implementation of TD in the Alder Lake microarchitecture, which complements the microarchitecture-agnostic description found in the Intel manual [70]. The article explains that class identification is conducted by a machine-learning algorithm that leverages telemetry data and runs on dedicated hardware. Moreover, it discloses

[4]The TD class is marked as invalid if the hardware cannot generate sufficient telemetry data. To indicate this, the lower byte of the `IA32_THREAD_FEEDBACK_CHAR` register, encoding the class ID, is valid only if bit 63 of the same register (the "valid bit") is set.

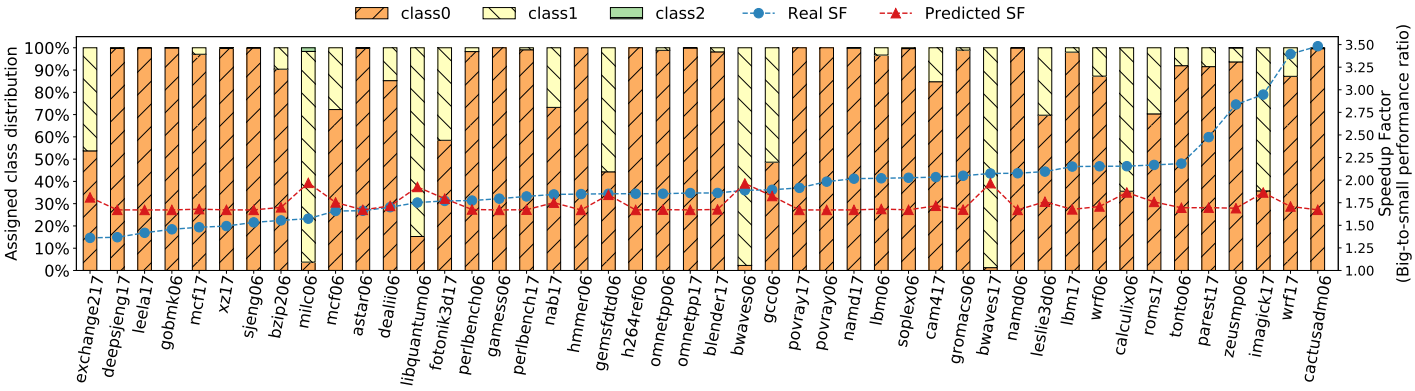


Figure 3.16: Proportion of assigned classes on big cores compared to actual and predicted speedup factor (SF) by Intel Thread Director.

the meaning of the four TD classes present on Alder Lake processors. TD Class 0 includes the majority of program phases, such as arithmetic, flow control, or data transfer intensive. TD Classes 1 and 2 are associated with code that experiences a substantial performance benefit (an average speedup of 2.75x) when running on P-cores, such as the ones that make extensive use of wide vectors, AI, and accelerated instructions. Lastly, TD Class 3 is reserved to identify threads running busy-waiting code (i.e., spin loops), which waste valuable CPU cycles, especially on P-cores. In summary, threads are assigned to the various TD classes based on the instruction mix that they run.

Figure 3.16 provides the distribution of classes assigned by TD to all SPEC CPU programs (from the CPU2006 and CPU2007 suites) throughout an isolated execution of each program on a big core of `AlderLake-16`. Recall that small cores do not provide valid class readings on our platform, a limitation also observed when using Intel's kernel patch. The TD class of every program was sampled at every tick, just as Linux-ITD does. We conducted this experiment using both our kernel-level extensions (implemented with PMCSched) and an instrumented version of the Linux-ITD patch [108]; in both cases, we obtained the same class distributions by gathering the per-class counts at the end of the execution. Each program listed in Figure 3.16 appears in ascending order of its *real* overall SF, computed as the ratio between the program's completion times on an E-core and a P-core. As a reference, in addition to the *real* SF, we also plot the overall SF estimated by TD, denoted as *predicted SF*. This prediction corresponds to a weighted average: the sum of the product of the frequency of each class in the program and the class's SF value. The results reveal that very little time of any application's execution timem is categorized as TD Class 2. Moreover, TD does not identify any program phases with TD Class 3, as these single-threaded benchmarks do not exhibit busy-waiting phases.

Figure 3.16 clearly shows that *there is not a consistent and reliable correlation between the predicted and real SF*. The discrepancies between actual and predicted SF values are significant. Take, for instance, the `cactusadm` program, which exhibited approximately twice as much real SF than predicted. These mispredictions could potentially lead to ineffective thread-to-core mappings performed by an asymmetry-aware scheduler that relies on TD. For example, the `milc` program, often catego-

rized as TD Class 1, would be preferentially assigned to a big core due to its high predicted SF (almost 2), whereas the `cactusadm` application – despite having the highest actual big-to-small speedup across the board – could be relegated to small cores due to its low TD-predicted value (1.67). Overall, we conclude that making performance predictions based on the instruction mix employed by TD's implementation in Alder Lake processors [131] does not render accurate speedup factor values.

We should highlight that the detailed discussion of our full experimental analysis of TD can be found in Chapter 6. There, we show that the performance-counter based SF estimation model proposed in [146] provides superior SF prediction accuracy compared to TD for SPEC CPU programs on our platform. Moreover, we also illustrate that Intel Thread Director successfully captures busy-waiting phases by reporting TD Class 3 when threads execute spin loops on a big core. We developed a custom micro-benchmark to showcase this behavior.

3.4.2 Improved OpenMP loop-scheduling for AMPs

AID (Asymmetric Iteration Distribution) [138] is a set of previously proposed loop-scheduling methods specifically tailored for efficient execution of loop-based OpenMP programs on asymmetric multicores. AID addresses the load imbalance that naturally arises on AMPs from running several application threads on cores that deliver different performance. To illustrate this issue, consider running a regular OpenMP application consisting of a single parallel loop on an asymmetric platform with 4 big cores and 4 small ones, as the one depicted in Figure 3.17. Suppose further that this is a legacy application designed for symmetric platforms. Since the loop's iterations involve roughly the same amount of work, OpenMP's `static` loop-scheduling is appropriate for symmetric CMPs. If we run this unmodified application on the AMP system with 8 threads, with the mapping shown in Figure 3.17 (left), it would poorly utilize the big cores. Essentially, threads running on the faster cores (T0-T3) will complete their share of the parallel loop earlier than threads on the small cores (T4-T7). As a result, big-core threads would wait in the implicit barrier for the other threads to complete their work, as shown in Figure 3.17 (right). In fact, under these circumstances, the performance of the application running on a platform with only 8 small cores would be very similar to that of the AMP system, which further underscores the poor utilization of the big cores.

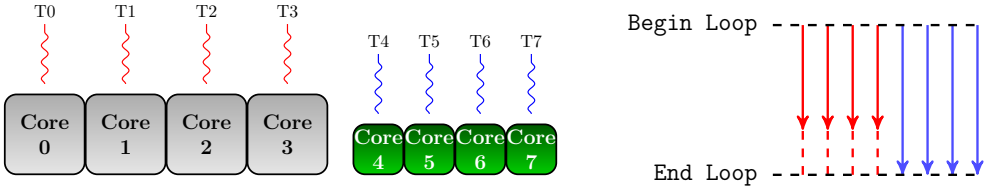


Figure 3.17: Typical imbalance scenario of a regular loop-based parallel program running on an AMP system.

To address this issue and deliver performance portability to unmodified loop-based OpenMP programs across symmetric and asymmetric multicores, AID's design considers two key observations about the execution of these programs on AMPs [138]. First, the speedup factor (SF) may differ substantially across parallel loops, even within the same application. Hence, using a single application-wide SF value (e.g., obtained beforehand through profiling or compiler support) fails to provide effective scheduling. Second, systematically assigning more loop iterations to big-core threads than to small-core ones, based on the loop's observed SF, does not always guarantee perfect load balance across worker threads. This stems from the fact that inherent imbalance may be present in some loops, due to varying amounts of work required to complete different loop iterations.

In what follows in this chapter, we first describe AID's original implementation and then present our Flexible AID proposal, which extends AID and addresses its major limitations.

3.4.2.1 Asymmetric Iteration Distribution (AID)

AID's original implementation consists of three new loop-scheduling methods designed as natural AMP-aware replacements for the standard `static` and `dynamic` loop-scheduling strategies. These existing OpenMP-standard methods are known to perform poorly on asymmetric multicores [138]. Unlike the standard methods, AID strategies leverage knowledge on the core type where worker threads run, allowing threads on big cores to potentially complete more loop iterations than those on small cores based on the observed big-to-small performance ratio (SF) of the loop.

AID consists of three loop-scheduling methods: `AID-static`, `AID-hybrid`, and `AID-dynamic`. The first two methods are asymmetry-agnostic alternatives for the `static` schedule on AMPs, while the third one is an AMP replacement for the conventional `dynamic` schedule. These approaches are designed with performance portability in mind, ensuring good performance on AMPs for regular loop-based OpenMP applications that were already optimized for a given symmetric multicore. For instance, if the most suitable schedule for a certain loop in a symmetric CMP is `dynamic`, `AID-dynamic` would be the appropriate choice for AMPs. Thus, the runtime system could automatically select the most suitable AID method without requiring changes to the application.

AID methods employ a common technique to determine the loop's SF at runtime [138]. This is done by running a *sampling phase*, during which all threads track the completion times of *chunk* iterations of the loop, where *chunk* is a configurable parameter. By factoring in the average time it takes for threads assigned to big and small cores to complete *chunk* iterations, AID approximates the loop's SF, and determines how subsequent iterations are scheduled. Specifically, Figure 3.18 illustrates how the SF is calculated during a sampling phase. In this example, an OpenMP program with 8 threads (from T0 to T7) runs on an AMP system with four big cores ($N_{big} = 4$) and four small cores ($N_{small} = 4$). T0-T3 run on big cores (henceforth referred to as big-core threads) and T4-T7 are assigned to small cores

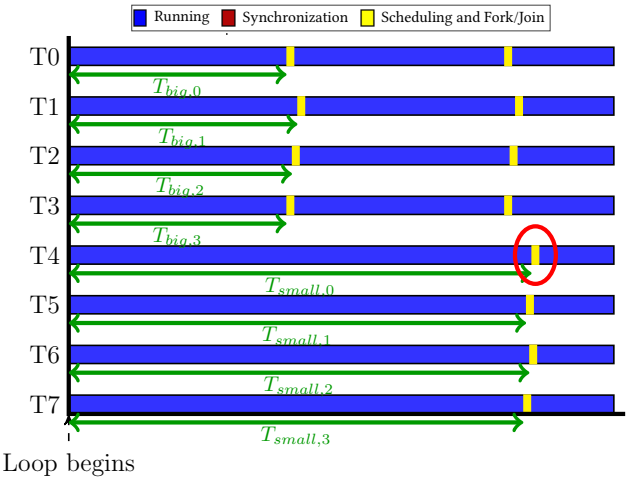


Figure 3.18: Example of `AID` sampling period, with four worker threads running on big cores and four threads assigned to small cores.

(henceforth referred to as small-core threads). As it is evident, the last thread to complete chunk iterations is T7, and so it will be responsible for calculating the SF. To this end, the following equation is used, where $T_{small,i}$ and $T_{big,j}$ denote the amount of time that a given thread i running on a small core (or thread j assigned to a big core, respectively) takes to complete *chunk* iterations:

$$SF = \frac{\frac{1}{N_{small}} \cdot \sum_{i=0}^{N_{small}-1} T_{small,i}}{\frac{1}{N_{big}} \cdot \sum_{j=0}^{N_{big}-1} T_{big,j}} \tag{3.1}$$

The various AID methods work as follows. `AID-static` employs just one sampling phase at the beginning of the loop and assigns loop iterations to threads in proportion to the performance of each thread's core type. For example, if the SF value observed during the sampling phase is 2, each big-core thread will be assigned twice as many loop iterations as each small-core thread. The `AID-hybrid` method is a variant of `AID-static` that caters to cases where the loop iterations involve similar, but not exactly the same, amount of work. In these scenarios, the sampling phase provides slightly inaccurate SF predictions, giving rise to load imbalance. `AID-hybrid` addresses this issue by distributing only a configurable fraction of the total loop iterations with the `AID-static` method; the remaining iterations are scheduled using OpenMP's standard `dynamic` method to help balance the load when scheduling this last set of iterations. Lastly, `AID-dynamic` constitutes an efficient replacement of the conventional OpenMP's `dynamic` schedule, designed for loops where the various iterations demand a different amount of computation. Essentially, `AID-dynamic` alternates between phases where all threads remove the same number of iterations (*chunk*) from the shared pool [5], and phases similar to `AID-static`, where iterations

[5]In the OpenMP runtime we used (GNU's libgomp), the implementation of loop-scheduling

are distributed unevenly. A detailed description of this elaborate loop-scheduling method, which involves several sampling phases within the same loop, can be found in prior work [138].

3.4.2.2 Flexible AID

AID's original implementation makes two simplifying assumptions [138]. First, the parallel program executes alone on the platform, thus taking advantage of all the large cores available; the number of big cores (N_{BC}) is communicated to the runtime system via a custom environment variable. Second, AID methods assume a fixed thread-to-core binding when applying the various AID loop-scheduling methods. It is up to the user to enforce the associated mapping by defining the corresponding environment variable of the runtime system; this requires user knowledge on the topology of the asymmetric system. For example, threads with IDs ranging from 0 to $N_{BC} - 1$ could be mapped to big cores, while the remaining threads run on small cores. Relying on such a fixed mapping greatly simplifies the implementation and, at the same time, automatically accelerates many inherently serial phases of the program that the master thread typically runs, such as initialization code or regions between parallel loops [37, 138, 140]. Unfortunately, all these assumptions greatly limit AID's applicability to more general workload scenarios.

In this thesis, we propose Flexible AID, which extends the original AID's capabilities in the following ways:

1. Unlike AID, Flexible AID does not require the user to impose fixed thread-to-core mappings or to explicitly inform the OpenMP runtime system on the CPU IDs associated with big and small cores. To transparently remove this burden from the user, Flexible AID exploits the runtime-OS interaction support provided by PMCSched (see Section 3.1.2). Specifically, each worker thread requests the creation of a memory region shared between the runtime system and the kernel for bidirectional communication through PMCSched. In our setup, the necessary OS kernel support was implemented inside a PMCSched plugin, which maintains a `core_type` field in the per-thread shared memory structure. This field indicates the core type on which the thread is currently running. The OS updates this field appropriately after cross-core-type migrations (e.g., when a thread is moved from a big to a small core). Flexible AID re-implements AID's loop-scheduling methods by leveraging the `core_type` field, enabling automatic decisions about how many loop iterations should be assigned to each thread, without requiring user intervention. This allows the OS and the runtime system to work efficiently and in synergy, even when multiple applications are running on the system, with thread-to-core mappings that may vary over time.

methods relies on a shared iteration pool. When running a parallel `for` construct, worker threads remove a number of iterations from the pool every time a loop-scheduling related runtime call is invoked.

2. Flexible AID also allows the runtime system to provide thread-mapping hints to an asymmetry-aware OS scheduler. Specifically, the aforementioned per-thread shared-memory region is also equipped with an `amp_prio` field, which is reserved for the runtime system to indicate to the OS kernel which threads could improve the most the application-wide performance when scheduled on the available big cores. For instance, in OpenMP, the master thread could be assigned a "high priority" to run on big cores (low `amp_prio` value) to ensure the scheduler favors its mapping on a big core relative to other application threads, thus accelerating the sequential code it may run. Our asymmetry-aware OS schedulers, implemented on top of PMCSched (described in the next section), take the per-thread `amp_prio` values into consideration when assigning application threads to big cores. In this thesis, we only explored the benefits of setting fixed per-thread `amp_prio` values throughout the execution. Nevertheless, designing runtime strategies that dynamically adjust per-thread `amp_prio` values in the runtime system is a promising avenue for future work.

3. Flexible AID incorporates a mechanism to reduce runtime system overhead in loops with very short iterations. We observed that, in loops of this kind, the overhead introduced by loop-scheduling runtime-system calls may significantly increase the loop's completion time, sometimes leading to performance degradation. Flexible AID's overhead mitigation technique is engaged when our runtime system detects that the loop iterations are shorter than the average duration of a runtime call. Under such circumstances, our AID-enabled runtime system automatically switches to `AID-static` – the AID method that requires fewer runtime calls. Notably, for lightweight detection of potentially conflicting loops (in terms of overhead), we leverage the completion times already recorded for the various worker threads during AID's sampling phase.

4. We implemented a new AID scheduling method referred to as `AID-guided`, which is an asymmetry-aware variant of the standard OpenMP `guided` schedule. Like `guided`, `AID-guided` also employs a monotonically decreasing chunk size. However, to factor in the platform's performance asymmetry, `AID-guided` adjusts based on the loop's SF and the core type where the thread runs, leveraging the relative performance associated with each worker thread. Specifically, let P_i be the relative performance of a worker thread i, where $P_i = 1$ for small-core threads and $P_i = SF$ for big-core threads. Let R denote the number of loop iterations remaining to be completed and N the total number of worker threads. When a worker thread T requests iteration removal from the pool under `AID-guided`, the runtime system assigns it $\frac{R \cdot P_T}{\sum_{i=1}^{N} P_i}$ iterations. This makes it possible for big-core threads to potentially process larger chunks than small-core threads in proportion to the loop's SF, improving load balance on AMPs.

3.4.3 The HSP scheduler and its implementation

Previous research [58, 79, 82, 102] has demonstrated that, to maximize throughput in the context of multi-program workloads, the scheduler needs to be able to (1) determine at runtime the benefit (i.e., *speedup*) that each thread in the workload derives from running on a big core relative to a small one, and then (2) preferentially map to big cores those application threads that may derive a larger relative performance benefit from such cores. Because applications may go through different program phases, continuous per-thread performance monitoring and dynamic adjustments of the thread-to-core mappings are necessary to fully leverage asymmetric cores [79, 102, 143].

For single-threaded applications, its speedup matches the SF of its only thread. For multi-threaded programs, determining the application-wide speedup and identifying which threads are more profitable candidates for running on big cores poses an important challenge. Note that, in this case, other factors beyond the per-thread SF need to be considered, such as the current degree of thread-level-parallelism (TLP) – some applications may exhibit serial or low-TLP phases [58] –, synchronization effects [140], or aspects handled exclusively by an underlying runtime system, like parallel-loop scheduling, load balancing, and interdependencies among tasks within the application [37, 38, 138].

In this thesis, we implemented the HSP (High SPeedup) scheduler [143] as a PMCSched plugin. This constitutes the first OS-level implementation of HSP in Linux that does not require source code changes at the kernel's core, where this kind of schedulers are usually implemented. Instead, we leverage PMCsched's extensible kernel module to augment the Linux scheduler's functionality.

The HSP scheduler aims to maximize the system throughput on AMPs by placing threads in the workload to the *most appropriate core type*, this is, on the core type that delivers the greatest relative benefit from the point of view of system throughput. To this end, HSP performs thread-to-core mappings based on applications' speedup predictions. These predictions are obtained online by combining runtime information gathered with PMCs – for the threads' speedup factors (SFs) –, with the application's runnable thread count, serving as a proxy of its amount of TLP [140]. For each thread T, HSP employs the following *utility metric*, which provides an overall idea of how much the application (with N threads) would speed up when using all the available big cores (N_{BC}) [124, 143]: [6]

$$Utility_T = \frac{MIN(N_{BC}, N)}{N} \cdot (SF_T - 1) + 1 \tag{3.2}$$

A new feature of our implementation, not present in the original HSP proposal [143], is its inclusion of the necessary OS support for Flexible AID's loop scheduling meth-

[6] This formula assumes that the underlying runtime system evenly balances the load among the application's runnable threads.

ods. Specifically, the scheduler maintains the `core_type` field in the per-thread memory region shared with the runtime system. This field is properly updated during cross-core-type migrations to reflect the core type on which threads are running at all times. Additionally, our HSP implementation leverages user or runtime-provided hints on the preference for threads within the same application to be assigned to big cores. To this end, the scheduler factors in the per-thread `amp_prio` values found in the thread-specific shared memory region. In particular, the implementation periodically checks these per-thread `amp_prio` values; if a pair of threads from the same application, T_B and T_S, running on big and small cores, respectively, are found such that $amp\_prio_{T_B} > amp\_prio_{T_S}$, the scheduler swaps the threads between cores.

One of the main deltas among the various asymmetry-aware throughput-optimized schedulers proposed in the literature [79,82,143] (like HSP) is the underlying method employed to determine the SF online. In this thesis, we explored the effectiveness of two SF prediction methods: PMC-based estimation models [79,125,143] and reliance on specific hardware support for SF estimation [70, 71]. For the first prediction method, we use two PMC-based estimation models specifically designed for SF prediction from the big and small cores of an Intel Alder Lake processor (from prior work [146]). For hardware-aided SF prediction, we leverage the Intel Thread Director (TD) technology outlined in Section 3.4.1.

We implemented three HSP variants to evaluate different SF-prediction methods on system throughput. The first variant, referred to as HSP/TD, uses TD for SF estimates. In the Alder Lake processor we used, such estimates are directly accessible only when a thread runs on a big core. To address this, our implementation maintains a per-thread history table of TD-based big-core SF estimates for different program phases. This approach allows SF predictions to be made indirectly from small cores by leveraging the stored history. In the second HSP variant, denoted as HSP/BS, the OS continuously gathers a number of per-thread PMC metrics. An up-to-date SF prediction is obtained for a thread by using the metric values as input to the core-specific models for big and small cores [146]. Lastly, in the HSP/B variant, SF predictions on big cores are obtained via the same big-core model used by HSP/BS; however, on small cores, predicted SFs are obtained indirectly by reading a history table populated with past SF estimates retrieved on the big core. Note that this variant was implemented to conduct a fairer comparison with HSP/TD, where direct SF predictions on small cores are unavailable.

3.4.4 Experimental evaluation

To assess the effectiveness of our various proposals on asymmetry-aware scheduling on the AlderLake-16 platform, we considered three types of scenarios for evaluation. The first scenario (discussed in Section 3.4.4.1) quantifies the degree of throughput delivered by different OS-level schedulers for AMPs when running multi-application workloads consisting of single-threaded programs. The second scenario (analyzed in Section 3.4.4.2) evaluates the potential of FlexibleAID when running a single OpenMP program on the system. The last scenario (covered in Section 3.4.4.3)

Scheduler	Short description
HSP/TD	HSP scheduler using TD for SF estimates
HSP/BS	HSP scheduler using PMCs for SF estimates
HSP/B	HSP/BS variant using history table for small cores SF prediction
AARR	Asymmetry-aware Round Robin scheduler
Linux-ITD	Linux CFS scheduler with TD patches by Intel

Table 3.2: Scheduler variants and their descriptions used for evaluation.

assesses the efficacy of the proposed OS-runtime cooperation method for multi-application scenarios (HSP+FlexibleAID) by running workloads that include both single-threaded and multithreaded programs.

To measure the degree of throughput in multi-application scenarios, we relied on the Aggregate Speedup metric [58, 143, 146]. Additionally, for each workload and scheduler, we computed the Average Normalized Turnaround Time (ANTT) [152]. Both metrics are defined in Section 2.3.1.

The detailed results of the full experimental analysis, along with the composition of all workloads used for the evaluation, are described in Chapter 6. The remaining sections focus on outlining the main insights.

3.4.4.1 Multi-application workloads consisting of single-threaded programs

The goal of this section is to evaluate the impact that different mechanisms for estimating SF has on scheduling. For the experiments, we randomly generated 20 diverse workloads, each comprising 16 single-threaded programs. The program mixes (referred to as M1 to M20) covered a total of 46 different SPEC CPU applications. Details on the composition of each Mi workload are provided in Chapter 6.

In addition to the three aforementioned variants of HSP (BS, B, and TD), we experimented with an Asymmetry-Aware Round-Robin (AARR) scheduler [88] that equally shares big and small cores among applications (also implemented on top of PMCSched) and with Linux-ITD, the Linux kernel patched with the scheduler enhancements proposed by Intel [108], which leverage TD. For the sake of clarity, Table 3.2 includes a brief description of all schedulers used for the evaluation. Our experiments revealed that Linux-ITD delivers the same degree of performance variability observed in Linux's default scheduler, CFS [58]. This variability is primarily because the scheduler prioritizes filling big cores but does not ensure an equitable distribution of CPU cycles among their threads. As explained in prior work [58], this behavior, combined with potential variations in thread-to-core mappings across different executions, leads to discrepancies in execution times for repeated runs of the same experiment – ultimately making AARR a more reliable baseline for comparison.

Figure 3.19 shows the normalized throughput (top) and the reduction in the ANTT metric (bottom) across the workloads considered. A detailed discussion of these

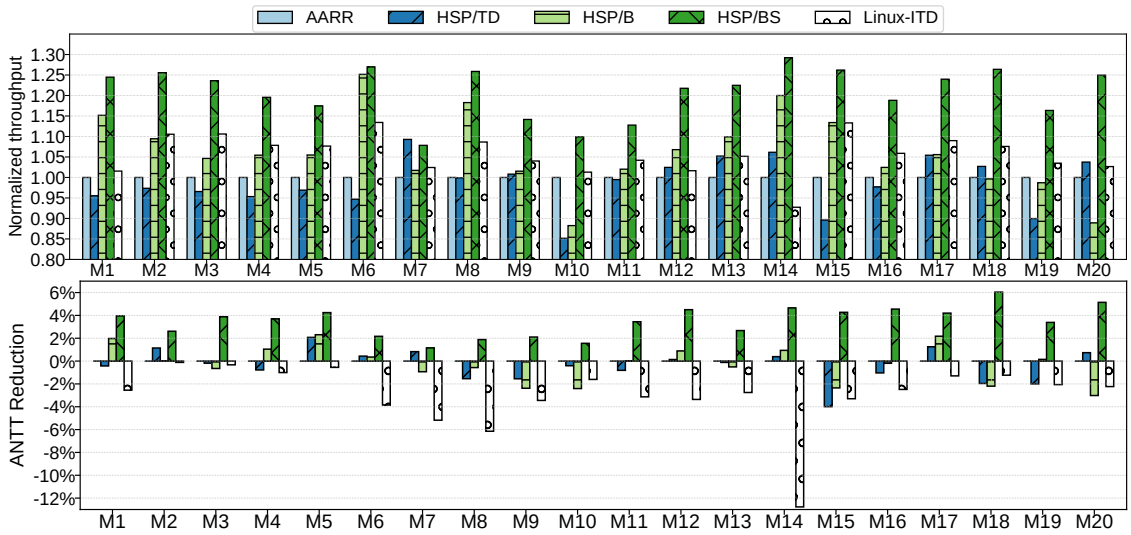
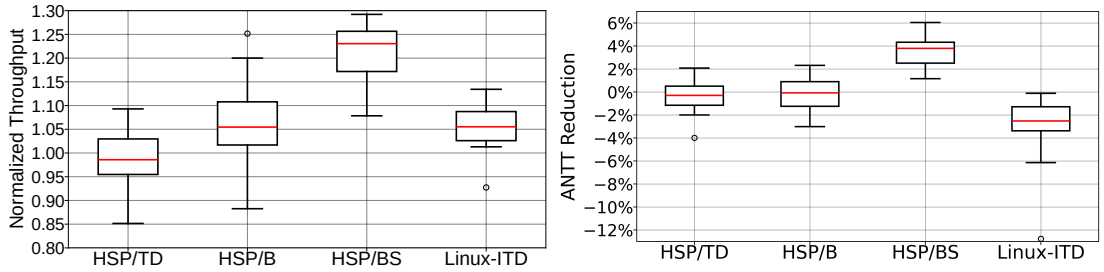


Figure 3.19: Normalized throughput and ANTT reduction associated with the various scheduling algorithms.



(a) (b)

Figure 3.20: Distribution of normalized throughput (a) and ANTT reduction (b) associated with the various scheduling algorithms.

results can be found in Chapter 6. To provide a compact discussion of the findings, Figure 3.20 presents the per-scheduler distribution of normalized system throughput (Figure 3.20a) and ANTT reduction (Figure 3.20b) across workloads. The main insights from these results are summarized below:

- As the results in Figure 3.20a reveal, HSP/BS outperforms the other schedulers for most workloads, achieving up to a 29% throughput gain w.r.t. AARR, and providing a 22.9% average improvement against the TD variant. The performance improvements are tightly related to the superior SF-estimation accuracy provided by the PMC-based models [146] for the big and small core, relative to that of TD, as discussed in Section 3.4.1. Overall, a higher SF-prediction accuracy allows HSP to better identify programs with a truly high SF, allowing the scheduler to grant more big-core cycles to those programs than to others. In doing so, HSP also reduces the average slowdown across applications in most workloads, achieving up to a 6% reduction in ANTT, as Figure 3.20b shows.

- Linux-ITD's system throughput figures differ substantially from those of the

baseline. Linux-ITD systematically assigns TD Class 0 to all small-core threads; in practice, that is the default class assigned by Linux-ITD to applications when the hardware repeatedly reports an unknown class. Recall that this is always the case for small-core threads, as TD fails to perform class readings on these cores [146]. Since the Linux scheduler is designed to avoid thread migrations when possible, and the TD performance score associated with TD Class 0 on big cores is no greater than that of classes 1 and 2, Linux-ITD never finds a thread running on a small core in our platform that exhibits a higher big-to-small speedup than any thread currently populating big cores. As a result, threads initially assigned to small cores "get stuck" on these cores, leading to highly variable throughput figures across runs. The impact of these nearly random assignments is shown in Figure 3.21, which displays Linux-ITD's distribution of normalized throughput and ANTT across 10 independent runs for each workload, highlighting substantial performance variations. Depending on the specific run, Linux-ITD may deliver either better or worse performance than the baseline, making the average numbers shown in Figure 3.20 potentially misleading if considered in isolation.

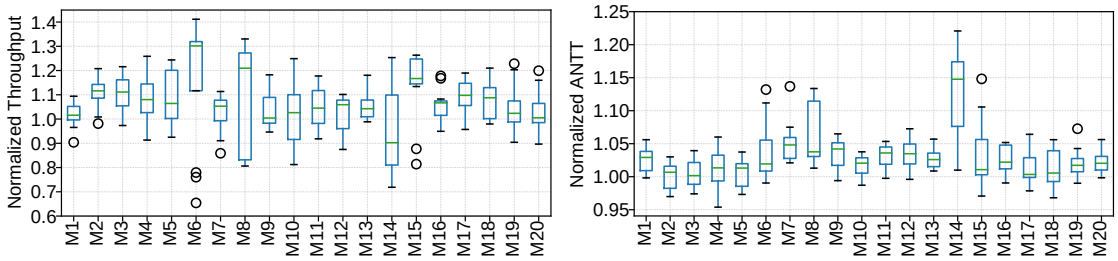


Figure 3.21: Distribution of normalized throughput (higher is better, left side) and ANTT values (lower is better, right side) under multiple runs of Linux-ITD.

- We further observe that using the big-core model alone in combination with the history table (HSP/B variant) provides substantially better throughput figures than HSP/TD (with an average improvement of 7.9%). However, in a few workloads, HSP/B fails to yield comparable performance to AARR. We found that this is caused by the extra thread migrations (and their associated overhead) triggered in response to frequent table phase misses, which aim to refresh the history table on big cores.

  Notably, the HSP variants that rely on the history table to retrieve SF predictions gathered on the big core (HSP/B and HSP/TD) trigger cross-core-type thread migrations more often than the other strategies. Despite this fact, we conclude that the PMC-based big-core model alone provides superior accuracy compared to TD, and that the per-thread history table constitutes a reasonably effective method to deal with scenarios where direct SF estimation may be unavailable on certain core types.

3.4.4.2 Evaluation of single-application workloads with Flexible AID

In our experiments, we compared the effectiveness of Flexible AID scheduling methods (`AID-static`, `AID-hybrid`, `AID-dynamic` and `AID-guided`) with OpenMP programs against the standard OpenMP `static`, `dynamic`, and `guided` strategies, which are asymmetry-unaware. To do so, we leveraged a diverse set of OpenMP programs from four different benchmarks suites – NAS parallel benchmarks, PARSEC, Rodinia, and SPEC CPU2017. A detailed discussion of per-application results can be found in Chapter 6.

From our evaluation, we were able to conclude that AID scheduling methods constitute good general replacements for the corresponding asymmetry-unaware approaches. Figure 3.22 presents the average normalized performance of OpenMP standard loop-scheduling strategies and Flexible AID's scheduling methods relative to the OpenMP `static` schedule, labeled as "static(SB)" in the figure. AID strategies provide comparable or better performance for most programs, yielding an average improvement of 6% and achieving substantial gains in specific applications such as `blackscholes` (32.8%), `bfs` (24.5%), `pathfinder` (17.8%), or `cactusbssn_s` (60.2%). Note that several programs obtained substantial performance gains (up to a 2.11x speedup) simply by placing the master thread on a big core. This improvement stems from the presence of extensive sequential execution phases in these applications, primarily during initialization. In fact, some OpenMP applications with large sequential regions benefited solely from the thread placement, with loop scheduling having little effect [138].

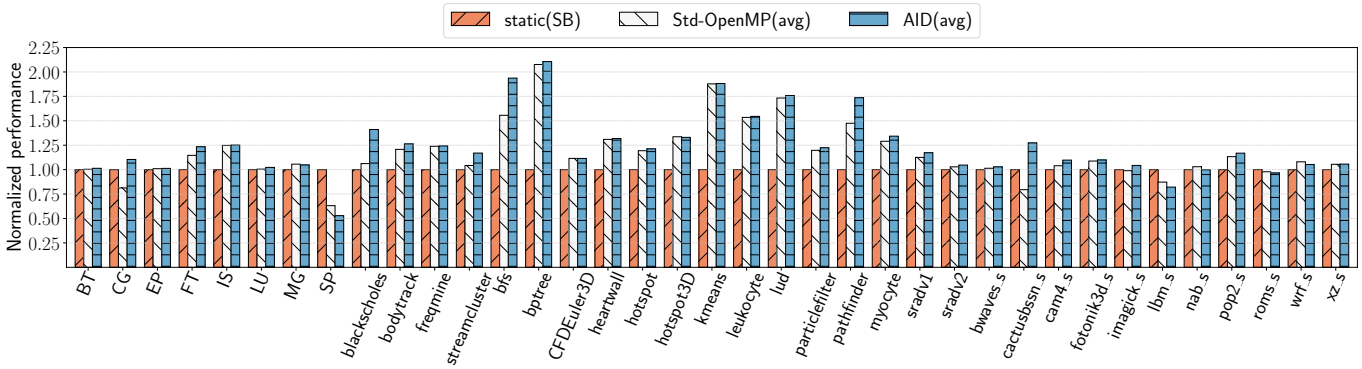


Figure 3.22: Average normalized performance across all standard OpenMP methods (std-OpenMP) and across all AID strategies for all the OpenMP programs explored.

3.4.4.3 Evaluation of multi-application workloads running under Flexible AID

The goal of this final experimental scenario was to analyze how HSP improves system throughput by (i) mapping threads to cores based on thread's utility factor (defined in Section 3.4.3), and (ii) leveraging OS-runtime interaction. Particularly, in OpenMP programs with FlexibleAID enabled, HSP informs the runtime system about which threads are mapped to big and small cores at all times. Therefore,

our modified runtime system can adjust the loop-iteration distribution dynamically based on the varying thread-to-core mappings.

To illustrate HSP's expected behavior, consider a hypothetical scenario with eight P-cores and eight E-cores (like the `AlderLake-16` platform), as shown in Figure 3.23. In this situation, four single-threaded applications and one multithreaded program (12 threads) run simultaneously. In the first step, HSP maps all single-threaded applications to the P-cores 0-3, maximizing the overall utility factor. At the same time, the rest of cores, including all eight E-cores, execute threads of the multithreaded application, likely including the master thread due to its higher priority (lowest `amp_prio` value). Subsequently, when two single-threaded applications leave cores idle (second step), Flexible AID migrates additional threads from the multithreaded application to the P-cores.

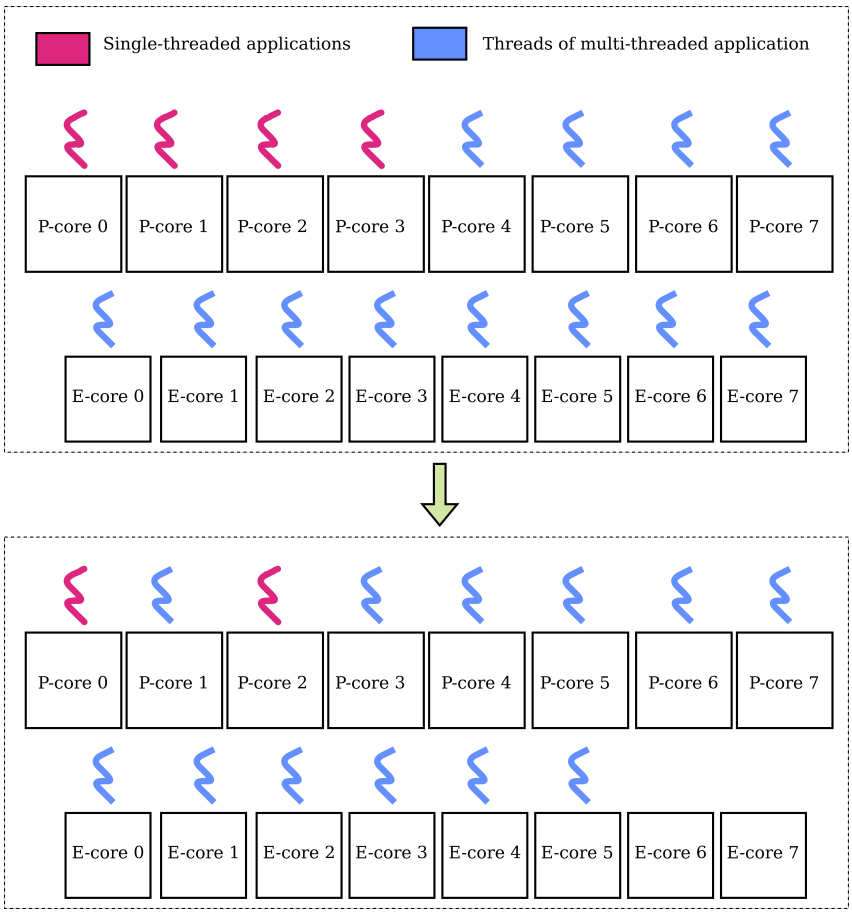


Figure 3.23: Scenario illustrating HSP and Flexible AID in action on a 16-core system (half P-cores and half E-cores).

HSP's ability to dynamically adapt to thread mappings stands in contrast with the original AID implementation [138], which only supports a single OpenMP application running on the system and requires imposing fixed thread-to-core-mappings (i.e., low-TID threads in the parallel program must be mapped to big cores). These limitations prevent the OS scheduler from fully controlling thread-to-core assignments, which is crucial in multi-application scenarios, such as those studied in this work. Flexible AID effectively addressed this issue by eliminating the need for fixed

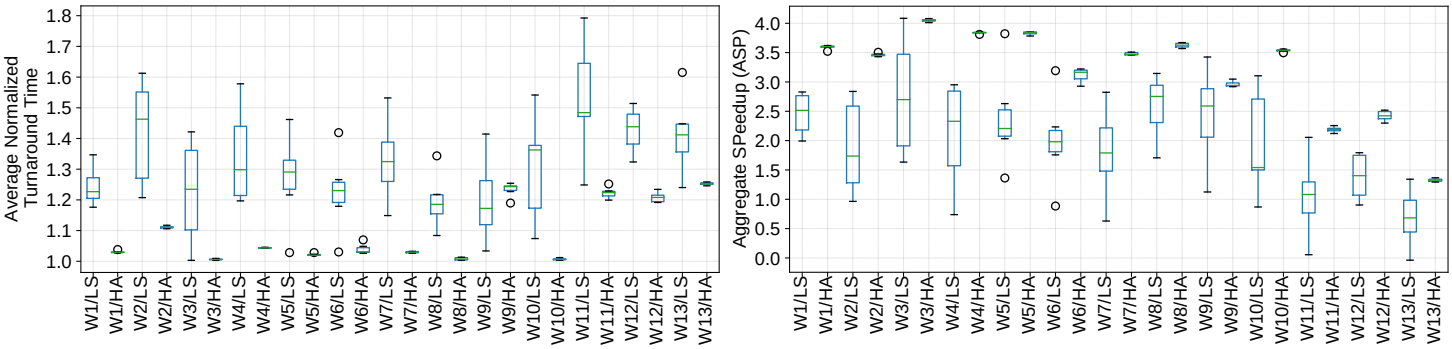


Figure 3.24: Distribution of ANTT (left side) and throughput (right side) for the various workloads under Linux-ITD with Standard OpenMP (LS) and HSP with AID loop-scheduling methods (HA).

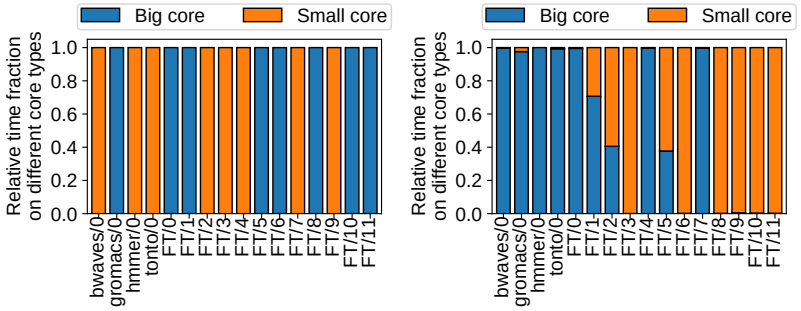
thread-to-core-mappings.

To validate HSP's desired behavior, we experimented with a set of multi-program workloads. Each randomly-generated mix included four sequential programs from SPEC CPU and one OpenMP application. In our experiments, we explored two different scenarios, referred to as *LS* and *HA*, where the OS scheduler entirely controls thread-to-core assignments dynamically (no CPU affinities are established for any program). In the *LS* (Linux-ITD + Standard OpenMP) scenario, the workload runs with the Linux's default scheduler with the Intel TD patch (Linux-ITD). In the *HA* (HSP + AID) scenario, the workload runs on top of the HSP scheduler (BS variant) – which we extended with support for Flexible AID –, and the parallel program is configured to use the best-performing AID method (as identified in our previous round of experiments). Note that under *LS*, the scheduler is expected to exhibit the behavior depicted in Figure 3.23, by populating any idle big cores with small-core threads, shortly after cores of this type become idle.

Figure 3.24 summarizes the results obtained for all workloads (named W1 to W13) running under the LS and HA scenarios; with the distribution of the lower-is-better ANTT metric and throughput (ASP metric) across all runs. As the results for ANTT reveal, our approach (the HA scenario) provides consistent results across runs and systematically lower ANTT values than Linux-ITD, averaging a 16.8% reduction in ANTT. At the same time, HSP + AID outperforms Linux-ITD across the board in terms of throughput (as the ASP results demonstrate), while delivering repeatable ASP figures across runs, as opposed to what Linux-ITD does.

To understand why our approach improves both ANTT and throughput with respect to Linux-ITD, and explain where the source of variability comes from on Linux, we recorded and analyzed the amount of time (in ticks) that all threads in each workload spend on big and small cores throughout the execution. As an illustrative example, Figure 3.25 shows the fraction spent by every thread on each core type registered on a sample run for one of the workloads under Linux-ITD and HSP. Clearly, Linux-ITD makes different thread-to-core assignments (see Figure 3.25a), and more importantly, *it fails to effectively map all sequential programs to big cores, thus missing opportunities for substantial throughput improvements.*

By contrast, as Figure 3.25b reveals, HSP systematically devotes big cores to run explicitly serial code. Essentially, threads belonging to a single-threaded program exhibit a high *utility* value (see Equation 3.2), so, when active, they are always mapped to a big core. Moreover, the HSP scheduler places the master thread in the OpenMP application on a big core (thread `FT/0`), by catering to its low `amp_prio` value, which indicates a high preference to run on a big core.



(a) Linux-ITD (b) HSP scheduler

Figure 3.25: Relative time each W2 threads spends on core types under different OS schedulers. The x-axis labels show the program name and TID, separated by a slash.

Our results also corroborated that, *when a single-threaded application terminates under HSP, the scheduler effectively migrates a thread from the OpenMP program to the idle big core* (as in the hypothetical scenario depicted in Figure 3.23). This behavior contrasts with Linux-ITD, which does not rapidly migrate threads to idle big cores but instead keeps threads on their initially assigned core type for as long as possible.

Chapter 4

Divide&Content: A fair OS-level resource manager for contention balancing on NUMA multicores

Full citation

Carlos Bilbao, Juan Carlos Saez, Manuel Prieto-Matias, *Divide&Content: A Fair OS-Level Resource Manager for Contention Balancing on NUMA Multicores*, IEEE Transactions on Parallel and Distributed Systems, 10.1109/TPDS.2023.3309999, 34, 11, (2928-2945), (2023).

Impact metrics:

JCR 2023, Impact Factor: 5.6, Q1 in Computer Science, Theory & Methods.

Abstract

Chip multicore processors (CMPs) constitute the cherry-picked architecture for high-performance servers employed in supercomputers and cloud datacenters. In the last few years, Non-Uniform Memory Access (NUMA) multicore systems have become the dominant choice in these domains. Regardless of the technology advances enabling to pack an increasing number of cores and bigger caches on the same chip, contention for shared resources still represents an important challenge for the system software. Cores in CMPs typically share multiple resources, such as the last-level cache (LLC) or a DRAM controller. The competition for the usage of these resources leads to uneven performance degradation across co-running applications.

Previous research has demonstrated that contention effects on CMPs can be mitigated via smart partitioning of the LLC or by distributing threads across groups

of cores so as to even out the degree of competition on multiple LLCs or memory nodes. However, most existing resource-management strategies fail to effectively combine both contention-mitigating techniques, thus providing suboptimal results on NUMA multicores. In this paper, we analyze how to best combine these techniques to improve system-wide fairness, and, based on the conclusions of our analysis, propose a fair OS-level NUMA-aware resource manager that leverages dynamic contention-aware thread-to-socket mappings and cache-partitioning. We implemented our resource manager in the Linux kernel and assessed its effectiveness on a real dual-socket system featuring Intel Skylake processors. Our results show that it reduces unfairness by more than 17% on average compared to Linux and a state-of-the-art NUMA-aware resource manager.

Divide&Content: A Fair OS-Level Resource Manager for Contention Balancing on NUMA Multicores

Carlos Bilbao, Juan Carlos Saez, and Manuel Prieto-Matias

***Abstract*—Chip multicore processors (CMPs) constitute the cherry-picked architecture for high-performance servers employed in supercomputers and cloud datacenters. In the last few years, Non-Uniform Memory Access (NUMA) multicore systems have become the dominant choice in these domains. Regardless of the technology advances enabling to pack an increasing number of cores and bigger caches on the same chip, contention for shared resources still represents an important challenge for the system software. Cores in CMPs typically share multiple resources, such as the last-level cache (LLC) or a DRAM controller. The competition for the usage of these resources leads to uneven performance degradation across co-running applications. Previous research has demonstrated that contention effects on CMPs can be mitigated via smart partitioning of the LLC or by distributing threads across groups of cores so as to even out the degree of competition on multiple LLCs or memory nodes. However, most existing resource-management strategies fail to effectively combine both contention-mitigating techniques, thus providing suboptimal results on NUMA multicores. In this paper, we analyze how to best combine these techniques to improve system-wide fairness, and, based on the conclusions of our analysis, propose a fair OS-level NUMA-aware resource manager that leverages dynamic contention-aware thread-to-socket mappings and cache-partitioning. We implemented our resource manager in the Linux kernel and assessed its effectiveness on a real dual-socket system featuring Intel Skylake processors. Our results show that it reduces unfairness by more than 17% on average compared to Linux and a state-of-the-art NUMA-aware resource manager.**

***Index Terms*—Multicore processors, NUMA, cache-partitioning, fairness, linux kernel, resource management, operating system.**

I. Introduction

EVER since their introduction, multicore processors have incessantly grown in popularity. Today, they constitute the architecture of choice for servers in cloud datacenters [1], [2], [3], as well as the dominant general-purpose solution for HPC platforms [4]. In both scenarios, NUMA multicores have become widespread due to their decentralized and scalable nature [5]. They comprise multiple memory nodes, each one featuring a set of cores that often share an on-chip DRAM controller. While local memory accesses happen via the local DRAM controller, remote ones occur via a cross-chip interconnect [6].

Despite the outstanding technological and microarchitectural advances, shared-resource contention in NUMA multicores still poses a relevant challenge to the system software. Specifically, cores in a memory node typically share critical resources with the neighboring cores, such as a last-level cache (LLC) and the local DRAM controller. Co-running applications and virtual machines (VMs) may intensively compete with each other for such shared resources, leading to uneven and hard-to-predict application/VM's performance degradation [1], [7], [8], [9]. The higher latency of remote memory accesses, and the interconnect contention in NUMA, are known to aggravate this problem further [10].

Shared-resource contention introduces undesirable effects on the system that make it difficult to enforce system-wide fairness [7], [11], [12] and QoS (Quality of Service) constraints [1], [2], [13]. For instance, an application's completion time and tail latency may largely depend on the application's co-runners [9], [13]. Moreover, the slowdown (i.e. relative performance degradation w.r.t. an isolated execution) of equal-priority applications that run simultaneously on the system may differ substantially under contention, leading to unfairness and other issues [2], [7], [8], [11].

Contention-induced performance disparities are exacerbated in NUMA multicores –and more generally in systems with multiple LLCs (each one shared by different groups of cores)–, where an application's slowdown may even differ substantially across multiple runs of the same multi-program workload, depending on the specific thread-to-core mappings. To illustrate this issue, we performed ten runs of a multi-program workload –consisting of compute-intensive applications from SPEC CPU and Rodinia– on an Intel-based dual-socket NUMA experimental system (more details on this platform can be found in Section VI). Fig. 1 shows the distribution of slowdowns of each program across the various runs provided by Linux's default scheduler, provided that no user-supplied thread-to-core bindings are imposed and memory-related shared resources are not partitioned. In this scenario, where the total number of threads matches the platform's core count, the degree of LLC contention and/or memory-bandwidth contention on each socket greatly depends on the (random) thread-to-core assignments performed by Linux. This causes a large disparity in applications' relative performance degradation; while the performance of some (contention-insensitive) programs remains similar across runs, others exhibit a highly variable slowdown (it rises up to 4.4x),

Manuscript received 31 January 2023; revised 21 July 2023; accepted 24 August 2023. Date of publication 30 August 2023; date of current version 15 September 2023. This work was supported in part by the Spanish MCIN under Grants PID2021-126576NB-I00 and MCIN/AEI/10.13039/501100011033, in part by "ERDF A way of making Europe", and also in part by Comunidad de Madrid under Grant S2018/TCS-4423. Recommended for acceptance by A. Sussman. *(Corresponding author: Juan Carlos Saez.)*

The authors are with the Facultad de Informática, Complutense University of Madrid, 28040 Madrid, Spain (e-mail: cbilbao@ucm.es; jcsaezal@ucm.es; mpmatias@ucm.es).

Digital Object Identifier 10.1109/TPDS.2023.3309999

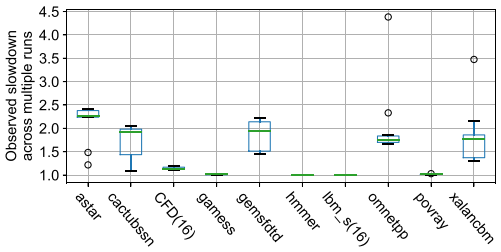


Fig. 1. Distribution of per-application slowdown across 10 runs of the same multi-program workload under the Linux default scheduler. The numbers in parentheses by the names of parallel programs indicate the number of threads they run with.

which greatly depends on the application's co-runners on the same socket. This performance variability also leads to unpredictable system throughput across runs. Clearly, performance divergences like these are unacceptable from the user satisfaction and fairness standpoints [14], making it also difficult to prioritize critical applications [13], to offer performance guarantees [2], [15] or to ensure correct billings in commercial cloud-like computing services [16]. Notably, uneven thread progress caused by contention may also greatly limit the scalability of parallel applications [17], [18].

Aware of these issues, previous work has aimed to mitigate shared-resource contention effects, mostly by exploiting two main types of control mechanisms. One of them is mapping threads to groups of cores –sharing LLC and/or DRAM controller- so as to ensure a balanced degree of contention across core groups [5], [6], [9], [11]. For these techniques to be effective, the system software has to be cognizant of the underlying hardware and the memory -and LLC- related behavior of the various applications, which may vary dynamically with program phases [9]. Another popular control mechanism is to partition the LLC, enabling to impose a certain degree of isolation among applications/VMs [1], [7]. After more than two decades of research on cache-partitioning [19], [20], the fairly recent adoption of hardware partitioning extensions in commodity processors [21], [22], [23] has given rise to a growing interest in the design of LLC-partitioning policies [7], [13], [21], [24]. Some recent partitioning strategies mainly target interactive latency-critical services [3], [13], [25]. In this work, we focus instead on compute-intensive workloads (CPU and/or memory bound) like other recent research [7], [14], [21], [26].

Crucially, most prior efforts on shared-resource contention fail to effectively combine the two aforementioned control mechanisms -a task that we will show to be far from trivial. In particular, some techniques only exploit thread-placement strategies [5], [11] but do not partition the LLC. Conversely, most recent partitioning strategies that primarily target compute-intensive workloads were specifically designed for UMA systems with a single LLC [7], [14], [24]. More importantly, as we demonstrate in this work, solely applying fairness-aware LLC-partitioning independently within each NUMA socket, does not allow to optimize system-wide fairness on NUMA multicores. Accomplishing this requires the simultaneous exploitation of dynamic contention-aware thread placement and resource partitioning.

To fill this gap, we propose Divide&Content (DC), a fair OS-level NUMA-aware resource management policy that dynamically combines LLC-partitioning with contention-aware thread-to-socket placement. DC was designed to optimize fairness with minimal impact on throughput. It strives to ensure that equal-priority applications that could potentially suffer from contention experience a similar performance degradation when sharing the system with others. Moreover, unlike Linux's default scheduler, DC provides consistent application performance and similar system throughput figures across multiple runs of the same compute-intensive workload. The main contributions of this paper are as follows:

1) By leveraging a simulation tool [27] –whose functionality had to be extended for this work– we conduct a comprehensive study to gain an understanding of how to best combine thread placement and LLC-partitioning for fairness optimization. Our research reveals that addressing thread placement and LLC-partitioning as separate and orthogonal optimization problems results in suboptimal solutions in terms of fairness. To maximize the effectiveness of fair resource partitioning, it is essential to leverage thread-to-socket assignments that even out the system-wide pressure for the various types of memory-related resources shared among cores in each NUMA node. We consider this insight crucial, and recommend factoring it in when designing NUMA-aware resource managers.
2) We designed DC based on the conclusions of our simulations, and implemented it in the Linux kernel. DC does not require any a priori application knowledge or profiling, and it performs coordinated dynamic LLC-space allocation and contention-conscious thread-to-core mappings at runtime, adapting to program phases.
3) For partitioning the LLC in DC, we built a variant of the LFOC+ partitioning strategy [7]. This variant was specifically designed to deal with multi-LLC systems.
4) For a comprehensive evaluation of DC, we compare it with our kernel-level implementation of DINO [5] (originally evaluated via a userspace prototype).
5) We performed an exhaustive experimental evaluation of DC on a real dual-socket NUMA multicore system, using a wide range of compute-intensive workloads that combine single- and multithreaded programs. We show that DC substantially improves fairness with respect to both DINO and the Linux default scheduler.

The remainder of the paper is organized as follows. Section II presents the motivation behind the creation of our a NUMA-aware resource manager. Section III discusses related work. Section IV covers our simulation analysis and enumerates the main insights leveraged by DC. Section V outlines DC's design and implementation. Section VI covers the experimental evaluation, and Section VII concludes the paper.

II. Background and Motivation

The main goal of this section is to briefly present the features of the LFOC+ partitioning policy [7], which is one of the building blocks of our proposal, as well as to discuss the limitations of partitioning policies designed for UMA systems

(such as LFOC+) when trivially adapted to NUMA systems. Before covering these aspects, we introduce the metrics used in our work to assess the degree of fairness and throughput of the various analyzed policies.

A. Fairness and Throughput Metrics

To quantify the performance degradation of an application a_i, part of N-application workload $W = \{a_1, a_2, \ldots, a_N\}$, we use the *Slowdown* metric:

$$Slowdown_{a_i} = \frac{CT_{res,a_i}}{CT_{alone,a_i}} = \frac{IPC_{alone,a_i}}{IPC_{res,a_i}} \quad (1)$$

where CT_{res,a_i} is the completion time of application a_i when running together with the other applications in W, under a specific resource management policy. Conversely, CT_{alone,a_i} is the time of a_i let run alone on the same system. In our experimental evaluation (Section VI), we calculate an application's slowdown based on its observed completion time. Nevertheless, as shown in Eq. 1, the slowdown can also be defined in terms of the Instructions Per Cycle (IPC) registered by the application when running in isolation (IPC_{alone,a_i}) and the achieved in the multi-program scenario (IPC_{res,a_i}). Considering the IPC, it becomes possible to measure a thread's slowdown in specific program phases by factoring in the phase's IPC in isolation and that observed in the multi-program scenario. LFOC+ [7], one of the building blocks of our proposal, relies on the IPC to approximate the slowdown of specific program phases at runtime.

Previous research on fairness for multicore systems [7], [11], [12] defines a policy as fair if equal-priority applications in a workload are subjected to the same slowdown product of sharing the system. To adopt this notion of fairness, we employ the *unfairness* metric, which has been extensively used in previous work [7], [11], [16], [17]. For a workload W, this lower-is-better metric is defined as follows:

$$Unfairness = \frac{MAX(Slowdown_{a_1} \cdots Slowdown_{a_N})}{MIN(Slowdown_{a_1} \cdots Slowdown_{a_N})} \quad (2)$$

The degree of unfairness is often reported together with system throughput figures. To this end, we employ the STP (System ThroughPut) metric [12], [28], defined as follows:

$$STP = \sum_{i=1}^{N} \left(\frac{CT_{alone,a_i}}{CT_{res,a_i}} \right) = \sum_{i=1}^{N} \left(\frac{1}{Slowdown_{a_i}} \right) \quad (3)$$

The STP is also known as the *Weighted Speedup*, and it has been widely used in computer architecture research [8], [12], [21], [29], [30]. The work by Eyerman and Eeckout [28] explains in detail why STP constitutes a throughput metric.

B. The LFOC+ Partitioning Policy

As applications run under LFOC+, the OS continuously gathers the value of several runtime metrics via performance monitoring counters (PMCs). Based on these metrics, applications are classified online into three categories: *light sharing*, *streaming* or *cache sensitive*. *Cache-sensitive* programs are those particularly susceptible to LLC contention; their performance degradation increases substantially as their LLC-way allotment is reduced. The *streaming* class encompasses bandwidth-intensive

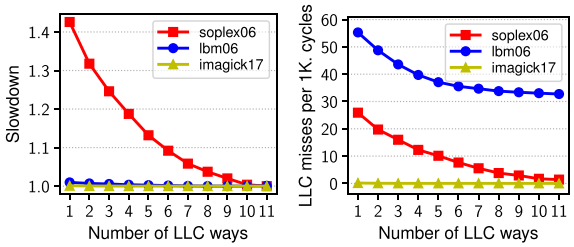


Fig. 2. Slowdown and LLCMPKC for different applications and ways.

programs that incur a high number of LLC misses per 1K cycles (LLCMPKC), and exhibit both a low LLC reuse rate and a low performance penalty for almost all way allocations when in isolation; these programs often degrade the performance of cache-sensitive applications co-located on the same LLC partition. *Light-sharing* programs are neither cache sensitive nor contentious to others, as most of their working set fits within the core's private cache levels. As an example, Fig. 2 illustrates how the application slowdown (left) and the LLCMPKC (right) vary with the number of assigned LLC ways for a cache-sensitive program (`soplex`), a streaming application (`lbm`), and a light-sharing one (`imagick`) on our Intel-based experimental platform (more details on this platform in Section VI).

LFOC+ features two operating modes: *fairness* and *sampling*. During the fairness mode, per-thread statistics are gathered using PMCs, and LFOC+'s partitioning algorithm is applied periodically at a configurable period. Algorithm 1 summarizes the partitioning strategy, which constitutes a cache-clustering (or partition-sharing) approach [21], [27]. Strategies of this kind may map a set of applications (referred to as *cluster* in Algorithm 1) to the same cache partition. When cache-sensitive programs are present in the workload, Step 1 of LFOC+'s algorithm takes care of confining streaming programs (if any) in up to 2 small LLC partitions. In Step 2, LFOC+ distributes the remaining LLC space (most of it) to cache-sensitive programs. To determine the number and size of LLC partitions for cache-sensitive applications, LFOC+ relies on a lightweight algorithm called *pair clustering*, extensively detailed in our earlier work [7]. This algorithm aims to minimize unfairness by assigning each cache-sensitive application to a private LLC partition or to a partition shared with another cache-sensitive program. Lastly, in Step 3, light-sharing applications are distributed among the various partitions by first populating partitions with streaming applications.

LFOC+'s *sampling mode* is used to determine an application's class, and is engaged in two specific cases: when an application enters the system (after a warm-up period), and when a class transition is detected. A class transition occurs when an application suddenly exhibits a performance profile that does not match its current class; for example, a supposedly streaming application begins to exhibit a low LLCMPKC. When the sampling mode is triggered, the application that initiated the transition into this mode is isolated from the rest in a LLC partition, referred to as the *sampling partition*, whose size is gradually increased (one

Algorithm 1: LFOC+'s Cache-Clustering Algorithm.

Input: ST, CS, and LS represent the sets of streaming (str), cache-sensitive and light-sharing applications, respectively; max_str_parts, $gaps\_per\_str$, and $ways\_str$ are configurable parameters (default values 5 and 3 and 2 respectively –see [7]), nr_ways is the number of ways of the LLC.
function LFOC+_part (ST, CS, LS, nr_ways)
  if $|CS| == 0$ **then**
    Create a single cluster S consisting of nr_ways;
    Map all applications in $ST \cup LS$ to S;
    return $\{S\}$
  $Clusters \leftarrow \varnothing$; $StreamingClusters \leftarrow \varnothing$;
  ▷Step 1: Create as many streaming clusters as needed
  if $|ST| > 0$ **then**
    $parts4str \leftarrow min(2, \lceil \frac{|ST|}{max\_str\_parts} \rceil)$;
    $<r, used> \leftarrow <\lceil \frac{|ST|}{parts4str} \rceil, parts4str * ways\_str>$;
  else
    $<parts4str, r, used> \leftarrow <0, 0, 0>$;
  for $i \leftarrow 1$ **to** $parts4str$ **do**
    Create new cluster C with $ways\_str$ ways;
    Map up to r apps from ST to C;
    Remove assigned apps from ST;
    Add C to $Clusters$ and to $StrClusters$;
  ▷Step 2: Distribute remaining space among apps in CS
  $SenClusters \leftarrow$ pair_clustering(CS, nr_ways-used);
  Add every cluster in $SenClusters$ to $Clusters$
  ▷Step 3: Assign apps in LS to existing clusters
  for each $TargetC \in StreamingClusters$ **do**
  $gaps\_avail \leftarrow r - |TargetC| * gaps\_per\_str$;
  if $|LS| > 0$ **and** $gaps\_avail > 0$ **then**
    Map up to $gaps\_avail$ apps from LS to $TargetC$;
    Remove assigned apps from LS;
  Distribute remaining applications in LS in a round-robin fashion among non-streaming clusters;
  return $Clusters$

LLC way each time). Meanwhile, the remaining applications are confined in a complementary LLC partition that shrinks over time, covering the remaining LLC space. During the sampling mode, LFOC+ observes the application performance (IPC) and the LLCMPKC for different way counts. By doing so, (1) the OS can quickly identify the application's class without exploring all possible LLC sizes (different heuristics are used to achieve this [7]), and (2) slowdown and miss-rate curves are built just for cache-sensitive programs. These curves represent normalized performance (reduction in IPC) and LLCMPKC, respectively, for different number of LLC ways, and are required for the *pair clustering* algorithm. As explained in prior work [7], LFOC+ estimates the slowdown of cache-sensitive programs for different ways by applying Eq. 1, which factors in the IPC observed for a specific number of ways and the actual IPC achieved by the application when the *sampling partition* reaches its maximum size (an estimate for IPC in isolation).

Unlike other partition-sharing strategies [12], [21], [24], [26], [27] that also rely on application classification but support single-threaded applications only, LFOC+ also has the ability to efficiently deal with regular data-parallel multithreaded applications, like the ones we experimented with in this work. For these applications, where all threads exhibit an almost identical PMC-related profile (as they do the same kind of work with different data), LFOC+ employs a lightweight classification method. Essentially, this method relies on tracking the PMC metrics of a selected *reference* thread in the application to guide its online classification. At the same time, LFOC+ always ensures that threads in the multi-threaded process are consistently mapped to the same LLC partition in either of LFOC+'s operating modes. When a change in the application's LLC partition is in order, LFOC+ effectively carries out the partition change by synchronously updating the partition-related per-core registers associated with all the application's threads. The low-level implementation aspects of this feature in LFOC+, including the selection of the *reference* thread, are described in our previous work [7], which also discusses ways to support additional types of multithreaded applications efficiently.

C. LLC-Partitioning and Mapping in NUMA Platforms

The vast majority of LLC-partitioning policies have been designed for UMA systems where cores share a single LLC [7], [12], [21], [26], [31]. Notably, no previous LLC-partitioning policy makes dynamic decisions on the thread-to-core assignments; in fact, most of them rely on fixed user-enforced thread-to-core mappings to function.

A straightforward way to extend these UMA policies to NUMA systems is to apply the partitioning strategy in question separately within each NUMA node (provided each one features a separate LLC), regardless of the specific thread-to-core mappings imposed by the user or the operating system. However, we observed that the effectiveness of a partitioning policy is largely affected by the underlying thread placement, as the placement substantially impacts the degree of pressure (contention) on shared resources within a NUMA node. Specifically, assigning multiple applications that aggressively compete for the same shared resource (LLC space and/or memory bandwidth) to the same node greatly limits the potential of the partitioning policy, particularly regarding fairness enforcement.

To illustrate this fact, we considered a multi-application workload consisting of 10 programs, which we ran on our 40-core Intel-based NUMA platform (two 20-core sockets with an 11-way LLC each) with six different fixed thread-to-socket assignments, and employing LFOC+'s partitioning algorithm. Previous work [7] demonstrates that this algorithm provides a near-optimal cache-clustering solution in terms of fairness for systems with a single LLC. For our experiment, we created a direct extension of the LFOC+ policy for NUMA, which applies the partitioning algorithm separately to the set of threads mapped to each socket. More information on the implementation of this

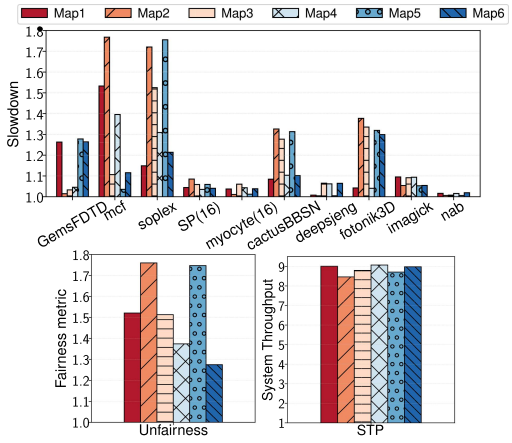


Fig. 3. Per-application slowdown (top), unfairness and throughput figures (bottom) registered for 6 different thread-to-LLC mappings of a 10-application workload under the LFOC+ LLC-partitioning policy. The numbers in parentheses by the names of the `SP` and `myocyte` (parallel) programs indicate the number of threads they run with.

LFOC+ variant can be found in Section V-A. Fig. 3 shows the per-application slowdown as well as the value of the Unfairness and STP metrics for the different mappings. As it is evident, the degree of fairness delivered by LFOC+ in this context greatly depends on the mapping. In particular, the worst fairness-wise mapping (Map2) increases unfairness by 28% compared to the best one (Map6), which also delivers a higher degree of throughput with a 6.1% improvement.

To understand why these divergences in unfairness occur, we focus on the slowdown associated with the `soplex` and `mcf` programs. Both applications are cache sensitive, and demand a substantial amount of dedicated space in the LLC to match the performance they deliver in isolation. Specifically, to reduce the slowdown of any of these applications below 1.05 (5%) under a scenario with no memory-bandwidth contention, each application requires a devoted LLC partition of over 55% of the total LLC size (i.e., at least 6 cache ways out of 11 – as depicted in Fig. 2 for `soplex`). Therefore, in mappings where both programs are assigned to the same NUMA node, such as Map2, it is simply unfeasible to simultaneously fulfill the LLC space requirements of both programs, regardless of the underlying partitioning algorithm. By contrast, under Map6, where said programs are assigned to different NUMA nodes, and memory-bandwidth consumption is balanced across nodes, LFOC+ can grant a bigger LLC share to both applications (assigned to different LLCs), thus reducing their slowdown and improving unfairness. This observation indicates that when the demand for a particular shared resource (such as the LLC) is uneven across NUMA nodes, it becomes impossible to fulfill the requirements of all the programs. This uneven demand inherently leads to unfairness, thus limiting the potential of the underlying partitioning policy.

The results also enable us to draw two insightful conclusions. First, optimizing system-wide fairness is not possible solely by minimizing unfairness at each socket separately, namely without explicit control of thread-to-socket placements. Hence, previous fairness-aware cache-clustering policies [7], [12], [32] designed for UMA systems cannot automatically optimize system-wide fairness on NUMA multicores, as they make no decisions on thread-to-socket placements. Second, an LLC-partitioning policy that operates independently in different NUMA nodes cannot guarantee by itself repeatable performance and fairness across runs. Therefore, to optimize system-wide fairness and deliver repeatable results, a fairness-conscious LLC-partitioning policy must be complemented with a consistent strategy to map threads to sockets, ensuring a balanced demand for the memory-related resources shared among cores in each NUMA node.

Notably, leaving the decision to the end user on where to place threads constitutes a significant burden. Making effective educated mapping decisions requires the gathering of substantial information about each application, including a detailed performance profile for different LLC-way counts [27], and some notions on its sensitivity to different levels of bandwidth contention [33]. Obtaining this information offline is unrealistic in many general-purpose settings due to the time required to conduct the associated experiments, and becomes unfeasible when the cloud provider has no direct access to the applications [1]. Moreover, even with detailed information, determining the optimal placement and LLC-partitioning in these contexts is largely impractical due to the exponential growth of the possible choices as the number of cores and NUMA nodes increase. We elaborate on this aspect in Section IV.

In this work, we propose an OS-level solution to consistently improve system-wide fairness on NUMA multicores by combining contention-aware thread-to-socket assignments and LLC-partitioning. The analysis conducted in this section raises a number of questions on the challenges associated with designing this approach:

1) Does the optimal fairness-wise mapping alone, without partitioning the LLC, constitute a good starting point to leverage fair LLC-partitioning later?
2) Should optimal thread-mapping and optimal LLC-partitioning (with partition sharing) be treated as separate optimization problems to be handled in sequence, or should they be addressed in a coordinated way to optimize system-wide fairness?
3) Is LFOC+ still effective in NUMA systems, so that it can serve as an appropriate building block of a fairness-aware resource manager?
4) How can we efficiently determine a good thread-to-socket mapping without extensively exploring all available mapping choices?

The simulation-based analysis in Section IV provides answers to these questions.

III. Related Work

In analyzing related work, we separately discuss cache-partitioning and contention-aware scheduling strategies, and then other optimizations and tools for contention mitigation on NUMA systems. Our work mainly differs from previous

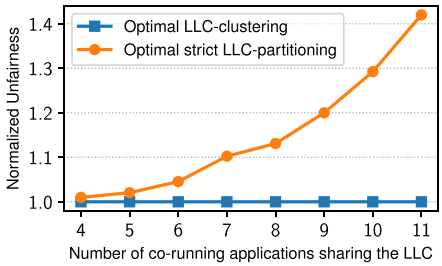


Fig. 4. Comparison between optimal strict LLC-partitioning and optimal cache-clustering for different application counts [32].

research in that we focus on how to make coordinated cache partitioning and NUMA-aware thread placement decisions within the OS kernel to improve fairness.

Cache Partitioning: A plethora of LLC-partitioning strategies was proposed to mitigate contention effects [2], [3], [7], [13], [19]. Recent works focused on the design of *cache-clustering* (also known as *partition-sharing*) algorithms [21], which constitute a generalization of strict cache-partitioning. While strict cache-partitioning policies [13], [14], [20], [25], [31] assign applications to separate partitions, cache-clustering strategies [7], [21], [24], [26], [27] may map several applications to the same partition. This type of strategy emerged to deliver better performance and fairness [7], [12], [21] on commodity processors with hardware LLC-partitioning support [22], [23], which allows creating a fairly limited number of coarse-grained partitions. Moreover, unlike strict cache-partitioning policies [13], [14], [20], [31], cache-clustering ones still work when the number of applications exceeds the number of LLC ways [27].

Garcia et al. [32] demonstrated that the utilization of LLC-clustering instead of strict LLC-partitioning leads to higher reductions in unfairness as the number of applications grows [32]. This trend is depicted in Fig. 4 (extracted from [32]), which shows the average increase in unfairness provided by optimal strict partitioning relative to optimal LLC-clustering for different workload sizes using the exact same processor model we used for our experiments (see Section VI). This trend stems from the fact that strict LLC-partitioning policies [13], [14], [20], [25], [31], [34] allocate a separate LLC partition of at least 1 way to every application, which impedes to systematically devote big LLC partitions to cache-sensitive programs, thereby degrading unfairness. By contrast, strategies based on cache-clustering [7], [12], [21], [24], [27] (such as our proposal) address this problem by allowing different programs to share the same LLC partition when beneficial.

Unlike our DC proposal, recent cache-clustering strategies targeting compute-intensive workloads [7], [14], [21], [24] were implemented and evaluated only on single-LLC UMA systems, and do not assign threads to specific groups of cores. To partition the LLC within DC, we adopted a variant of LFOC+ [7]; we created this variant (described in Section V-A) to support multi-LLC systems. We opted to use LFOC+ as one of DC's building blocks because it delivers greater fairness than its cache-clustering predecessors [12], [21], [24], it was the only one successfully evaluated using a lightweight OS-level implementation [7] – a better fit for our OS-level proposal–, and explicitly supports multithreaded programs, unlike [12], [21], [24], [26].

Fairness-aware strategies explicitly targeting container-based environments have also been proposed. Of special attention is CoPart [31], which leverages partitioning of the LLC coupled with memory bandwidth limitation to improve isolation among containerized applications. As discussed in Section V-D, our OS-level proposal could be leveraged in container-based environments as well. In addition, unlike CoPart, DC specifically targets NUMA systems. Note also that, in contrast to our cache-clustering based proposal, CoPart employs strict cache partitioning, so it is subject to its inherent limitations discussed earlier.

Other proposals exploit cache-partitioning to enforce QoS constraints in scenarios where high-priority applications, typically latency sensitive, run together with best-effort programs [2], [3], [13], [34]. CLITE [34] partitions not only the LLC but also other resources like disk and memory bandwidth. Drawing upon the insights of the PARTIES resource manager [13], CLITE introduces the capability for simultaneous Bayesian exploration of multiple resources, optimizing their allocation and enabling the co-location of latency-critical jobs to meet QoS requirements. However, CLITE has some limitations, including difficulties in adapting to rapid workload changes, a restricted ability to handle co-locations of more than 5 applications [25], and a considerable computational overhead. DRLPart [25], a deep-learning based model, exemplifies the combination of multi-resource management with the enforcement of throughput guarantees. However, DRLPart falls short in addressing fairness concerns, explicit support for NUMA platforms, or leveraging the advantages of cache-clustering strategies in contrast to strict cache partitioning.

Most of these QoS-oriented techniques [2], [25], [31], [34] were evaluated on UMA platforms. Moreover, the scarce works employing a NUMA system for evaluation [3], [13], assume externally fixed application-to-socket assignments; as the proposed techniques simply do not decide on these assignments, but handle resource partitioning only. Our proposed OS-level approach does simultaneously leverage dynamic thread placement with LLC-partitioning, to enforce fairness, and guarantee repeatable results across runs. Given that the effectiveness of a LLC-partitioning technique largely depends on the mapping (as shown in Section II-C), our proposal addresses a challenge that all previous work ignores. Note, however, that QoS-focused resource management policies are largely complementary to our research, as they generally adopt a best-effort approach towards throughput and fairness optimization. As shown by Park et al. [31] QoS-oriented techniques can be used in combination with those that focus on the optimization of system-wide metrics [7], [12], [21], [24], like our proposal, to improve system fairness further while enforcing QoS.

Contention-Aware Thread Scheduling: A large body of work has focused on designing contention-aware thread-placement policies, many of which were conceived for UMA systems consisting of multiple core groups, each one sharing a LLC [9], [11], [35]. Kundan et al. [36] propose a scheduler tailored explicitly to oversubscription scenarios on UMA. These are the same scenarios addressed by other recent works [16], [37], which –unlike our

proposal– leverage co-scheduling on UMA platforms instead of NUMA-aware thread placement. In the context of NUMA multicores, Majo et al. proposed N-MASS [6], a strategy that couples memory management and process scheduling. Unlike our proposal, N-MASS makes no distinction between bandwidth and LLC contention.

Blagodurov et al. [5] proposed DINO, a NUMA-aware strategy, which outperformed previous contention-aware schedulers. DINO combines contention-aware thread-to-core mappings with automatic page migration. Unlike DC, DINO does not partition the LLC. To decide on thread placement, DINO classifies threads at runtime based on its current LLC misses per 1 K instructions (LLCMPKI) into three classes: *turtles* (low LLCMPKI), *devils* (medium LLCMPKI) and *superdevils* (high LLCMPKI). It spreads threads of the various classes across core groups so as to even out the aggregated LLCMPKI among them. To reduce the number of migrations, DINO updates threads classes and readjusts thread-to-core mappings with a coarse granularity [5].

To perform an experimental comparison of our OS-level approach against a state-of-the-art NUMA-aware thread-placement policy like DINO, we created an OS-level implementation of the latter. This implementation, like DC's, uses Linux's automatic NUMA balancing feature [38], enabled by default in recent kernel versions. NUMA balancing exploits lightweight detection of page reuse and automatic page migration. When an application is migrated to another memory node, NUMA balancing automatically migrates referenced remote pages to the current node. Notably, this kernel feature was unavailable when DINO was proposed. To detect page reuse and drive page migration decisions, DINO's authors resorted to using a user-level architecture-specific performance-counter based monitor and had to be carefully configured to keep overheads under control [5].

Other Techniques: Several works explored complementary contention-aware solutions to scheduling [39], [40]. Recent studies [4], [41] –largely orthogonal to ours– exploit page interleaving techniques, page replication and migration to reduce contention and improve performance on NUMA platforms. Denoyelle et al. [40] acknowledged the complexity of thread placement and adopted a statistical approach, predicting whether a thread was sensitive to locality. This approach required sophisticated offline analysis, relying on a machine learning algorithm that had to be trained for the target hardware platform. NumaPerf [42] is a profiling tool to optimize source code for NUMA systems, but it requires static code instrumentation, instead of relying on the OS to abstract the difficulties of NUMA contention. CuttleSys [43] introduces a distinct approach to co-scheduling that explicitly targets reconfigurable multicores. Given its dedicated focus on QoS, it follows a best-effort approach to maximize throughput that lacks explicit fairness considerations.

IV. Fairness-Optimized Thread Placement and LLC-Partitioning in NUMA Systems

The design of our resource management proposal was driven by the conclusions of the simulation-based study described in this section. The main goal of this study was to determine how to best combine cache-clustering with thread placement, so as to optimize system-wide fairness on machines with multiple groups of cores (or sockets), each one integrating a separate LLC.

Before presenting the simulation environment and discussing the results, we summarize the main insights of our study upfront, which provide answers to the questions formulated at the end of Section II-C.

1) The thread-to-group mapping that optimizes fairness does not necessarily constitute the best mapping to apply fairness-aware LLC-partitioning later. This mapping often leads to uneven pressure on the different shared resources in the various core groups, thus limiting the effectiveness of LLC-partitioning substantially.
2) LLC-partitioning and thread-to-group mappings should be performed in a fully coordinated fashion to optimize system-wide fairness. In particular, to maximize the effectiveness of LLC-partitioning, the various threads must be assigned to core groups in a way that does not optimize fairness on its own. However, when optimal LLC-partitioning is applied on top of this mapping, the optimal degree of fairness can be achieved.
3) The LFOC+ partitioning algorithm, which constitutes a near-optimal fairness-wise cache-clustering strategy for single-LLC systems [7], serves as a good building block for fair NUMA-aware resource managers. Specifically, this heuristic partitioning algorithm, coupled with an effective thread-to-group mapping, can deliver fairness values close to those of the best solution among all possible cache-clustering choices.
4) The analysis of the global optimal fairness solution, which combines thread-to-socket mappings and LLC-partitioning to optimize fairness, reveals that a *good* thread-to-group mapping (i.e., amenable to LLC-partitioning) is one that guarantees a balance across groups in terms of the degree of competition for memory-bandwidth and for the demand of space in the LLC. This is the main idea that our OS-level proposal (DC) leverages to avoid the extensive exploration of the available mapping choices.

To arrive at the aforementioned insights, we had to determine a number of optimal solutions for various workloads, enforcing specific thread-to-group mappings and/or fairness-optimized LLC-partitioning. Determining these optimal solutions (described in detail later) requires extensive exploration of the search space. Unfortunately, the complexity of the optimal thread placement problem combined with the NP-hard nature of the optimal cache-partitioning/clustering problems [7], [9], [27], [36] requires the exploration of a vast search space, whose size grows exponentially with the number of applications and LLC ways. This constitutes an important challenge.

To make the problem tractable, we had to use a simulation tool, whose functionality had to be extended to make our analysis feasible. Specifically, we employed PBBCache [27], an open-source tool for rapid prototyping of LLC-partitioning policies. For a given multi-program workload and partitioning strategy, PBBCache allows obtaining the degree of throughput and

fairness under different LLC and system configurations. To achieve this, it relies on offline-collected applications' performance data – such as instructions per cycle, LLCMPKI, etc. – obtained beforehand for different LLC sizes on a target platform (a real system in our case). This information, the LLC-space distribution enforced by the partitioning strategy, and other system features (e.g., maximum memory bandwidth) are used by PBBCache to approximate the slowdown an application suffers as a result of both LLC sharing and competition for memory bandwidth [27]. Subsequently, it determines different system-wide metrics to assess the effectiveness of the partitioning strategy. Moreover, PBBCache implements a parallel algorithm to find the optimal partitioning/cache-clustering solution for diverse optimization objectives.

Notably, the original version of PBBCache [27] only supports the evaluation of partitioning algorithms on systems with a single LLC. Thus, to carry out our study we had to extend the simulator's capabilities in different ways. First, we augmented the underlying slowdown prediction model to support target systems consisting of multiple core groups with a separate LLC, that either make up a UMA platform (with a single DRAM controller), or a NUMA one (with as many memory nodes and DRAM controllers as the number of core groups). Second, the simulator's API was extended to allow the evaluation of strategies that combine thread-to-group placement with independent per-group LLC-partitioning. Third, a parallel optimizer was created to determine the optimal fairness-wise application-to-group placement and LLC-partitioning for a workload.

For the simulations we considered two small-sized NUMA platforms, referred to as Platform A and B. Platform A –with 8 cores in total– consists of two memory nodes, each one featuring four cores that share a 16-way 16 MB LLC; each group of four cores sharing an LLC (L3) cache has the specifications of the so-called Core-CompleX (CCX) present in the AMD EPYC Rome 7742 processor [44], where we gathered the offline application data for SPEC CPU programs required as input to the simulations. Platform B –12 cores in total– spans three memory nodes, each one including a 4-core group with the same features as in the first platform. For simplicity, our simulations in these NUMA configurations utilize exclusively single-threaded programs, and assume that all application pages are mapped to the local memory node where the program runs. Moreover, the simulations only consider techniques that enforce the same application-to-group and application-to-partition mapping throughout the execution. Nevertheless, our proposed OS-level approach (DC) does perform dynamic mapping and LLC-partitioning, as discussed in Section V.

For our study, we randomly generated 240 workloads by combining SPEC CPU2006 and CPU2017 programs. Specifically, 120 of these workloads consist of 8 applications each (for Platform A, with 8 cores), and the remaining 120 ones comprise 12 applications each (for Platform B, with 12 cores). For program characterization, PBBCache uses the average value of different performance metrics associated with the execution of the first 150 billion instructions of each program running alone on the aforementioned AMD processor with different LLC way counts (from 1 to 16).

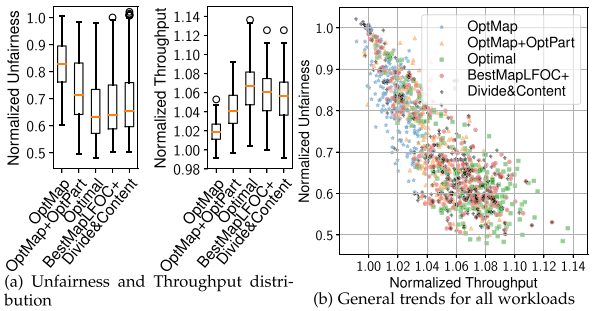
TABLE I
NUMBER OF POSSIBLE SOLUTIONS IN THE SEARCH SPACE FOR THE OPTMAP AND OPTIMAL SOLUTIONS

Platform (application count)	Number of different thread-to-group mappings	Possible solutions for mapping + cache-clustering
A (8 applications)	35	83370
B (12 applications)	5775	20634075

For each workload we obtained four theoretical solutions: OptMap, OptMap+OptPart, Optimal and BestMapLFOC+. OptMap explores all possible thread-to-group mappings (without partitioning the LLC) and selects the mapping that provides the optimal (minimal) unfairness. OptMap+OptPart starts off with OptMap's mapping solution, and, on top of that, it finds the best way to partition the LLC (via cache-clustering) so that unfairness is minimized. Optimal provides the all-encompassing optimal fairness solution; it combines optimal application-to-core mapping and optimal cache-clustering for the workloads, by exploring all possible mappings, and finding the optimal cache-clustering solution for each mapping. Lastly, BestMapLFOC+ is the best solution, among all possible mappings, that minimizes unfairness and where the LLC in each group is partitioned with LFOC+'s cache-clustering algorithm, summarized in Section II-B. Notably, if several solutions exist with the same unfairness value, these theoretical approaches pick the choice that yields the highest STP value.

Table I shows the size of the search space to be explored when determining the OptMap and Optimal solutions, respectively, for a single workload on both platforms. In both optimization problems, the search space size grows exponentially with the application count. However, it is clear that the number of options in the second optimization problem, which combines thread-to-group mapping and cache-clustering, is several orders of magnitude bigger than the first one, reaching more than 20 million possible choices for Platform B. In a hypothetical 16-core NUMA platform consisting of 4-core groups, the search space associated with determining Optimal for a 16-application workload rises up to 12.5 billion choices. This trend underscores the importance of utilizing a simulator for our analysis. Moreover, note that for each explored choice in finding Optimal, the distribution of LLC ways between clusters (i.e., groups of programs sharing the same LLC partition) that minimizes unfairness also needs to be determined. PBBCache enables us to efficiently obtain this distribution using a parallel branch-and-bound algorithm [27].

Fig. 5 depicts the overall throughput and unfairness numbers obtained for the workloads by the aforementioned theoretical approaches, plus the mapping&partitioning algorithm implemented by our DC proposal, described in detail in Section V. The distribution of unfairness and throughput values in Fig. 5(a) reveals that a huge gap exists (more than a 20% average reduction in unfairness) between Optimal and OptMap. Moreover, applying cache-clustering on top of OptMap (i.e, OptMap+OptPart) is still far from Optimal. This indicates that *Optimal cannot be reached by solving the optimal thread mapping and optimal cache-clustering problems separately*, and so *coordinated*



(a) Unfairness and Throughput distribution (b) General trends for all workloads

Fig. 5. Simulation results for the 240 workloads. Throughput (STP) and Unfairness values have been normalized to the results provided by a random application-to-group mapping and using no LLC-partitioning.

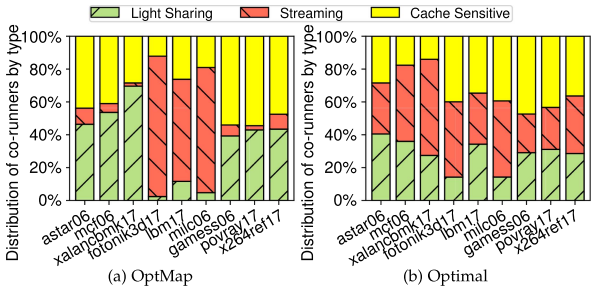
TABLE II
CACHE-SENSITIVITY CLASS OF THE DIFFERENT SPEC BENCHMARKS

Class	List of benchmarks
Cache Sensitive	`astar06`, `mcf06`, `omnetpp17`, `soplex06`, `xalancbmk17`
Streaming	`fotonik3D17`, `GemsFDTD06`, `lbm17`, `milc06`
Light Sharing	`gamess06`, `povray17`, `x264ref17`

mapping and LLC-partitioning decisions are paramount to optimize fairness. The results also highlight that relying on LFOC+'s cache-clustering algorithm enables BestMapLFOC+ to get very close to Optimal in both throughput and fairness. Our proposed heuristic policy (DC) operates in a close range of BestMapLFOC+ (1.5% on average on both metrics), and it does so without requiring the exhaustive exploration of all possible mappings.

A thorough analysis of the specific solutions provided by the theoretical approaches revealed other interesting observations. To ensure clarity in our discussion, we adopt the cache-sensitivity application classification used by the LFOC+ strategy, as introduced in Section II-B, which categorizes programs into three classes: *cache sensitive*, *light sharing* and *streaming*. Table II shows the cache-sensitivity class of a subset of the SPEC CPU benchmarks that were used, which are explicitly mentioned in specific examples. Notably, Section VI indicates the class of all the benchmarks used for all the experiments in this work.

To begin with, we find that a key to improving fairness is to separate streaming from cache-sensitive programs. To arrive at this conclusion we analyzed which kind of applications are assigned together in the same core group (referred to as co-runners) by OptMap and Optimal in the 240 workloads. Fig. 6 illustrates the distribution of co-runners organized by class for a few selected benchmarks from different classes, considering OptMap and Optimal across all workloads. For instance, the figure indicates that in workloads including the `astar` benchmark, OptMap predominantly places this application with co-runners that are either light sharing (47%) or cache sensitive (42%). Specifically, the results reveal that OptMap separates streaming from cache-sensitive programs by assigning them to different core groups. Clearly, more than 87% of the programs that OptMap places in the same group together with streaming programs –such as `fotonik3d` or `lbm` – are either light-sharing or streaming.



(a) OptMap (b) Optimal

Fig. 6. Distribution of co-runners (by type) assigned to the same core group for 9 selected programs –3 sensitive, 3 streaming and 3 light sharing (in the order used in the x axis)–, under OptMap and Optimal.

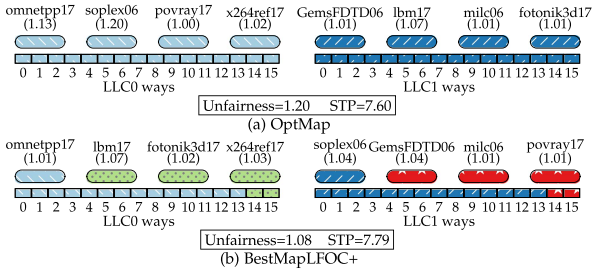


Fig. 7. Application-to-group mapping and LLC-space distribution within each group made by (a) OptMap and (b) BestMapLFOC+, for a sample workload.

Conversely, cache-sensitive programs –such as `astar` or `mcf` – seldom have streaming programs as co-runners in the same group; moreover, when this happens under OptMap, it is because other core groups are already packed with streaming programs. *By contrast, under Optimal, LLC isolation between streaming and cache-sensitive programs is effected by spreading both program types across groups, and ensuring that a streaming and a cache-sensitive program in the same group never share the same partition*. As shown in Fig. 6(b), Optimal may often assign streaming and cache-sensitive applications to the same core group. BestMapLFOC+'s and DC's solutions exhibit very similar trends as for using LLC-partitioning for isolating cache-sensitive from streaming programs.

Fig. 7 illustrates key recurrent mapping and partitioning patterns used by OptMap and BestMapLFOC+, by depicting the solutions provided for a specific workload, which includes 4 streaming applications, 2 cache-sensitive programs, 2 light-sharing ones (the class of each benchmark is indicated in Table II). The example in Fig. 7(b) highlights the main working principles of our proposed DC approach, which attempts to approximate BestMapLFOC+. Like this theoretical approach, DC improves fairness by balancing the number of cache-sensitive and streaming applications assigned to the various core groups; distributing cache-sensitive programs across groups allows the allocation of bigger LLC partitions for these programs, which is crucial to reduce their slowdown (as done with `omnetpp` and `soplex` in the example). Moreover, getting as close as possible to fulfilling the LLC-space requirements of cache-sensitive programs, and reducing LLC pollution by

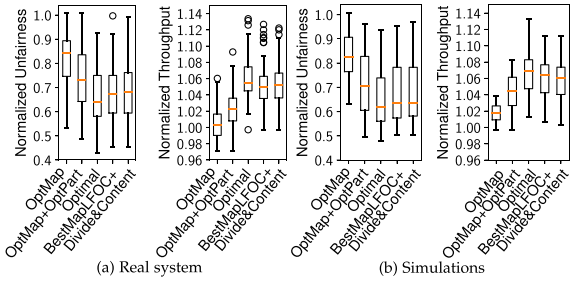


Fig. 8. Validation results for the simulations. Throughput (STP) and Unfairness values have been normalized to the results provided by a random application-to-group mapping and using no LLC-partitioning.

isolating them from streaming programs, also enables to minimize the memory-bandwidth consumption of cache-sensitive programs. Finally, the highest consumers of memory bandwidth –streaming programs– are spread across different groups, which evens out system-wide memory-bandwidth consumption.

To conclude, it is worth noting that the original version of the PBBCache simulator was validated using two Intel-based platforms [27], one of them featuring the processor model of the NUMA system employed in our experimental evaluation of DC (see Section VI). To validate the results provided by the extended version of the simulator used in this analysis, we conducted experiments on a real dual-socket system consisting of two AMD EPYC Rome 7742 processors. This system was employed to replicate the simulated Platform A (8 cores) by utilizing two four-core CCXs from different sockets to run 8-application workloads. To conduct the validation experiments, we ran the first 80 workloads of our simulations, all consisting of 8 applications. For these experiments, we followed an approach similar to that of earlier work [7]. Essentially, we employed fixed (static) cache-partitioning and enforced specific thread-to-group mappings throughout the execution, based on the output data provided by the simulator for each workload and strategy. This enabled us to determine the degree of throughput and fairness for each workload under the different theoretical or heuristic strategies.

Fig. 8(a) and (b) summarize the results gathered for the 80 workloads on the real system and the simulations, respectively. As we can see, Fig. 8(a) depicts similar trends to those observed in the simulator's counterparts. Clearly, Optimal achieves the best results in both throughput and fairness, and BestMapLFOC+'s and DC's fairness numbers closely align with those of Optimal. The biggest difference between the real system and simulation results is observed in the relative throughput gains, which are slightly smaller by up to 2.5% for certain workloads. These disparities stem from the inaccuracies in the simulator's bandwidth-contention model, particularly the underprediction of the performance degradation for some memory-intensive applications, as well as from the fact that the simulator does not consider individual program phases, but instead accepts input data on the overall program behavior. Despite these divergences, the insights drawn from our simulation results remain applicable to the real scenario. In Section VI, we conduct a comprehensive experimental evaluation of the OS-level implementation of DC on a real NUMA platform.

V. Design and Implementation

We developed Divide&Content (DC) in the Linux kernel v5.10.114. For its implementation, we leveraged PMCSched, [45] an open-source OS-level framework that makes it possible to implement scheduling and resource management strategies as *plugins* within a kernel module, which can be loaded in unmodified (vanilla) versions of the kernel.

To ensure a scalable design, our implementation leverages the *core group* abstraction provided by PMCSched. Essentially, cores in the system are organized into different sets (*core groups*) based on the platform's topology, such as cores sharing LLC or NUMA node. The framework's plugins assign threads to specific core groups by setting affinity masks. Enforcing load balance across cores within a group is up to the Linux load balancer, which respects affinity masks. The data structure describing a core group includes spinlock-protected linked lists to keep track of active threads and multithreaded processes mapped to it. This enables us to drive resource management decisions independently for threads assigned to different groups.

At a high level, DC works as follows. When a thread enters the system, it is assigned to one of the core groups, relying on the Linux scheduler's default functionality, which factors in the system's load across cores. As applications run, the OS continuously classifies them online into the aforementioned *light sharing*, *streaming* and *cache sensitive* classes. The online classification procedure is that of the LFOC+ cache-clustering strategy, summarized in Section II-B, and described in detail in our earlier work [7]. Periodically, and on a per-core-group basis, DC checks whether the degree of contention of the local group differs significantly from that of any of the remaining core groups. As we discuss in Section V-B, we consider the overall pressure on the different shared resources within each core group to determine its degree of contention. If a remote core group is found such as there is a clear contention-wise imbalance between groups, a balancing algorithm is executed to address this imbalance. Thread migrations are triggered as requested by the contention-balancing algorithm, and once completed in the local core group, DC applies LFOC+'s partitioning algorithm (see Section II-B) to improve fairness. This partitioning algorithm is also re-applied when threads mapped to the core group transition into a different application class.

In what follows, we first describe how we extended the LFOC+ cache-clustering strategy for multi-LLC systems, and enable its full integration within DC. Next, we present DC's balancing algorithm, and then describe interactions between DC's key building blocks and automatic NUMA balancing. Lastly, we discuss additional usages of DC beyond automatic thread placement and resource management for multi-program workloads at the OS level.

A. Variant of LFOC+ for DC

LFOC+ [7] is one of the key building blocks of DC. Because LFOC+ was originally designed for UMA systems with

a single LLC, substantial changes were required in its original implementation [7] to allow the independent utilization of the partitioning strategy within each core group (as done in the experiments of Section II-C). In particular, all global data structures to keep track of active applications/threads, to handle LFOC+'s *sampling* mode (see. Section II-B), and to perform application-to-partition assignments had to be replicated for each core group (LLC). The data structure that represents a process in our scheduling framework [45] had to be extended to track the set of LLC partitions associated with the various threads within the same multithreaded application, as these threads may be assigned to different core groups/LLC partitions. Lastly, it was also necessary to replicate a number of Linux kernel resources used by the implementation, such as spin locks, kernel timers and workqueues [46], to enable periodic SMP-safe OS activations and the execution of deferred work in blocking context (to update per-core LLC-partitioning registers) for each core group separately.

While these changes make it possible to apply LFOC+ independently within each core group, additional modifications were necessary for its full integration with DC, where thread migrations are triggered across core groups. Specifically, our LFOC+ variant reacts to thread migrations to transfer per-application statistics (e.g., slowdown curves) to another group when needed (i.e., the first thread of an application is migrated onto a different group). Moreover, it exposes LFOC+'s current operating mode in each group so that DC inhibits contention-related migrations in the group when LFOC+'s sampling mode is enabled (note that this mode leads to unstable application performance [7]).

Most recent fine-grained partitioning strategies [12], [14], [21], [24], [26], including LFOC+, were designed to work on situations without oversubscription. For this kind of scenario, DC does not partition the LLC, but just evens out the degree of contention across core groups by triggering thread migrations using the algorithm presented in Section V-B.

B. The Contention Balancing Algorithm

The goal of Divide&Content's balancing algorithm is to equalize the degree of contention between two core groups –the local and a remote one–, so as to later improve fairness via LLC-partitioning. This algorithm is motivated by the insights summarized in Sections II-C and IV, which indicate that balancing the pressure for shared resources across core groups increases the effectiveness of LLC-partitioning.

To approximate the degree of contention in a core group, we consider the pressure that each program assigned to the group applies to the local memory channels (measuring memory bandwidth demand) and its demand for the shared LLC (representing the required space to reduce the application's slowdown). To this end, we define two per-application indicators: *bandwidth load* (L_{BW}) and *LLC load* (L_{LLC}). In particular, an application's *bandwidth load* is defined as its total memory-bandwidth consumption reported by the hardware [22], [23] during the current monitoring interval. The *LLC load* of an application matches its *LLC critical point* as defined in [7]. Specifically, it represents the number of LLC ways at which the application's slowdown due to cache sharing falls below 5%. The higher the *LLC load* or critical point, the more LLC space the application requires to reduce its slowdown. For cache-sensitive programs, LFOC+ automatically updates the critical point when a new slowdown curve is obtained. For light-sharing and streaming applications their LLC load is set to be 1 and 2 LLC ways, respectively, for the specific Intel processor used in our study. By definition, a light sharing program is characterized by having a working set that fits entirely or almost entirely in the private cache levels, so allotting a single LLC way is sufficient to guarantee a low slowdown (< 1.05); hence $L_{LLC} = 1$. Regarding streaming applications, many of them experience a low slowdown as well using a 1-way LLC partition when running alone. However, our previous work [7] demonstrated that using 2-way LLC partitions to confine these type of programs can further increase system throughput, as that significantly reduces the bandwidth consumption of these programs, which can be substantial when using 1-way partitions. This approach allows LFOC+ to deliver a good throughput-fairness trade-off, especially when the workload includes bandwidth-intensive multithreaded programs [7]. Hence, as a conservative measure, we employ 2-way LLC partitions to confine streaming programs (parameter $ways\_str = 2$ in Algorithm 1), and thus assume that $L_{LLC} = 2$ for streaming programs.

In DC's contention balancing algorithm, applications currently running on the local and remote groups, are (one by one) re-assigned to one of the groups. Application placement decisions are based on the aggregated bandwidth load, and aggregated *LLC load* –denoted as AG_{BW} and AG_{LLC}– of each group; this is, the sum of the associated contention indicators (L_{BW} and L_{LLC}, respectively) for all the applications already assigned to a group in previous steps of the algorithm. The per-group AG_{BW} and AG_{LLC} are properly updated after assigning each application, and so is each group's *slot* counter, which records the number of free *slots* (i.e., yet unassigned cores) on it.

DC's contention balancing algorithm comprises 4 steps:

Step 1: The algorithm traverses all active applications in both core groups, and builds 3 linked lists for light-sharing, streaming and cache-sensitive programs, respectively. Henceforth, we will refer to these lists as *LS*, *ST* and *CS*. Applications in *ST* and *CS* are sorted in descending order by its L_{BW} and L_{LLC}, respectively. For efficiency reasons, an *MT* linked list is also maintained to keep track of active multithreaded applications irrespective of its class. Notably, in this first step of the algorithm –and in an attempt to reduce the number of migrations– applications that have been migrated recently (according to the `min_migration_period` parameter) or those with user-provided CPU affinities are automatically assigned to their current core group; these applications are not included in any of the aforementioned lists.

Step 2: The algorithm goes through all multithreaded applications in MT, and for each one it tries to improve data locality by ensuring that all of its threads are packed onto a single group (as in [5], [41]), while reducing the number of migrations. This thread compaction procedure is only possible if the number of slots in one of the groups is greater or equal than the application's

thread count. If so, all threads will be assigned to one of the groups. Otherwise, threads will remain assigned to their current group. Notably, in placing threads from applications in MT, DC handles applications in decreasing order by their thread count; applications whose thread count exceed the number of cores in a core group are skipped in this step. This makes it possible to improve the locality of many multithreaded programs.

Step 3: The algorithm now places single-threaded applications that have not been assigned yet. It first traverses applications in *CS*; each application is assigned to the core group with the lowest AG_{LLC}, provided that their slot counter ≥ 0. In doing so, it tries to even out the competition for LLC space between groups, thus maximizing the opportunities for the application to be mapped to a large LLC partition by LFOC+, which reduces its slowdown. Next, applications in *ST* are processed; each streaming program is assigned to the group with available slots that exhibits the lowest AG_{BW}, so as to balance memory bandwidth consumption across groups. Finally, the remaining sequential programs (LS list) are assigned so as to minimize the number of migrations. We should highlight that as soon as a group runs out of slots when assigning applications, the algorithm prioritizes the group with the greatest slot counter.

Step 4: Lastly, the algorithm tries to even out the memory-bandwidth consumption across groups further, but it does so only if the difference between the AG_{BW} of both groups exceeds a certain `bw_load_thr` threshold. This imbalance may be present due to the fact that streaming multithreaded applications typically have much higher bandwidth consumption than single-threaded programs [7]. Because Step 2 may pack all the threads from a streaming multithreaded application into a single group, this could introduce a large imbalance between the memory-bandwidth demands in both core groups, which could still persist even after Step 3. To address this issue, the algorithm iteratively swaps threads from a streaming multithreaded application assigned to the group with the highest AG_{BW} with light-sharing threads from the opposite group (preferably from another multithreaded application), until a balance is reached or no further swap partners are found.

C. Interactions Between DC's Building Blocks

Algorithm 2 shows the pseudo-code of the function that our resource-management strategy executes periodically on a per-core-group basis. Essentially, no actions are performed when the group is in a contention-wise unstable state, namely, when the LFOC+ instance for the current group is now going through sampling mode, or when thread migrations are yet pending on the group. Otherwise, the function (lines 5-8) traverses the set of remote groups until one group eligible for balancing is found and the balancing algorithm is able to address the detected contention imbalance via thread migrations. The function `balance_groups()` implements the balancing algorithm presented in Section V-B, and returns the number of threads migrations triggered.

A remote group is considered eligible for balancing (see Algorithm 3) when the following conditions are met (i) it is not in a contention-wise unstable state, (ii) it was not recently selected for contention balancing, and (iii) its pressure on the LLC or its memory-bandwidth consumption is substantial and differs significantly from that of the local group. To detect if such contention-wise imbalance exists, the `compare_bw_load()` and `compare_llc_load()` functions leverage the current AG_{BW} and AG_{LLC} of the local and remote groups, the number of ways in the LLC (LLC_{ways}), as well as 3 configurable thresholds (`bw_load_thr`, `low_bw_thr` and `llc_load_thr`). If the loop of Algorithm 2 (lines 5-8) does not carry out any

Algorithm 2: Divide&Content Algorithm.

```
1: function divide_and_content(cur_group)
   ▷Ensure it is safe to balance groups or partition the LLC
2:   if in_sampling_mode(cur_group) or
3:     migrations_pending(cur_group) then
4:       return;
   ▷Find a remote group so as to even out contention
5:   for each remote_group ∈ AllGroups − {cur_group}
   do
6:   if eligible_for_balancing(remote_group, cur_group)
7:     and balance_groups(remote_group, cur_group) > 0
8:       then return;
   ▷If no balancing was performed, apply LFOC+
  partitioning
9:   lfoc_plus_partitioning(cur_group);
```

Algorithm 3: Auxiliary Functions Used by Divide&Content.

```
1: function eligible_for_balancing(remote, local)
   ▷Check if balancing is feasible at the remote side
2:   if in_sampling_mode(remote) or
   migrations_pending(remote)
3:     or recently_balanced(remote)
4:   then return false;
   ▷Check if a clear imbalance in terms of AG_BW or
  AG_LLC exists
5:   return compare_llc_load(remote,local) or
6:     compare_bw_load(remote,local);
   ▷Returns true if the AG_LLC in both groups is high and
  uneven
7: function compare_llc_load(group1, group2)
8:   if AG_LLC(group1) ≤ LLC_ways and
   AG_LLC(group2) ≤ LLC_ways
9:   then return false;
10:  return |AG_LLC(group1) − AG_LLC(group2)| >
   llc_load_thr;
   ▷Returns true if the AG_BW in both groups is high and
  uneven
11: function compare_bw_load(group1, group2)
12:  if AG_BW(group1) ≤ low_bw_thr and
13:    AG_BW(group2) ≤ low_bw_thr
14:  then return false;
15:  return |AG_BW(group1) − AG_BW(group2)| >
   bw_load_thr;
```

balancing actions, then LFOC+'s partitioning algorithm is invoked (line 9). This algorithm is also called when all migrations triggered by a previous invocation of `balance_groups()` complete.

Notably, DC has been implemented so as to work in synergy with Linux's automatic NUMA balancing feature [38]. DC's `min_migration_period` parameter –used in Step 1 of the balancing algorithm– was introduced to ensure that applications that have been migrated recently between core groups remain pinned for some time to their current group. This allows to keep automatic page migrations under control, and permits NUMA balancing to migrate a substantial amount of the application's reused pages, which improves locality and reduces interconnect contention [5], [41].

To establish the value of `min_migration_period` we conducted a simple experimental study. Specifically, to track the activity of NUMA balancing in our experimental platform we monitored the remote memory bandwidth of different applications after migrating them between core groups. For memory-intensive programs from SPEC CPU, NUMA balancing takes 6.2 s on average to complete page migrations of the application's used pages. For other programs (mostly light sharing ones) automatic page migration is not even engaged due to their small working set, often stored in the cache hierarchy. Based on our study's conclusions, we set `min_migration_period` to 8 seconds, thus allowing memory-intensive programs to enjoy a 30% of extra time over the aforementioned average with improved locality.

D. Considerations on Affinities, Containers and VMs

In this work we assess the benefits of DC when delivering fully automatic thread-to-socket placements and resource-management to compute- and memory- intensive multiprogram workloads running natively on top of the OS. Nevertheless, DC could be leveraged in other workload scenarios. In particular, our implementation respects user-provided affinities, so users could forcefully assign specific applications/threads to particular core groups if required. Applications with a huge memory footprint or those exploiting custom page placement policies [4], [41] could greatly benefit from specific user-driven static mappings. In this scenario, DC's contention-balancing algorithm tries to even out the degree of contention by migrating (if needed) threads that do not have user-provided mapping constraints, and by still partitioning the LLC dynamically.

Moreover, due to its integration into the Linux kernel, DC could be leveraged for efficient resource management of multiple containers (e.g., Docker-based) or KVM-based virtual machines, with or without user-enforced CPU affinities. After all, VMs and containers are exposed to the Linux scheduler as multithreaded processes or groups of tasks, respectively. In the context of virtual environments consisting of multiple servers, our node-level proposal could be extended to harmoniously work with higher-level coarse-grained approaches that mostly focus on VM-to-node mappings [1] but do not address or react to program phases unlike DC.

VI. Experimental Evaluation

In this section we begin by describing our experimental setup, the workloads and the methodology employed in our experiments (Section VI-A). The detailed discussion of the results can be found in Section VI-B. Finally, Section VI-C analyzes the overhead of the evaluated resource-management strategies.

A. Experimental Setup, Methodology and Workloads

Our evaluation was performed on a dual-socket NUMA machine (2 core groups) with 96 GB DRAM that integrates two 20-core Intel Xeon Gold 6138 (Skylake) processors where cores run at 2Ghz. Each processor has an 11-way 27.5 MB LLC (L3) with way-partitioning support (via Intel RDT [22]). All cores have two private cache levels (64 KB L1 + 1 MB L2).

To assess the effectiveness of Divide&Content (DC), we experimentally compared it with three strategies: Stock-Linux, the unmodified Linux kernel, which does not partition the LLC; DINO [5], a contention-aware thread-placement NUMA-aware strategy that does not leverage LLC-partitioning either; and DINO/LFOC+, a variant of DINO that partitions the LLC with DC's LFOC+ version. We created DINO/LFOC+ to analyze the impact of combining two existing complementary approaches [5], [7] that make uncoordinated contention-aware decisions, namely where thread placement is done without knowing that the LLC is going to be partitioned afterward. Notably, DINO was originally evaluated via a user-space prototype [5], which relied on Linux's affinity-related Linux system calls. For our evaluation, we created a kernel-level version of DINO, by leveraging the PMCSched framework [45]. This implementation is not subject to system-call related overheads, as migrations are handled directly in the kernel with the same mechanism used for DC's implementation.

To pick appropriate values for DC's configurable parameters (see Section V), we conducted multiple sensitivity studies, whose conclusions we summarize here. The value of the memory-bandwidth related parameters was set as a percentage of the maximum per-socket bandwidth reported by the `stream` benchmark (76 GB/s in our platform). In particular, the `low_load_thr` and `bw_load_thr` parameters were set to 15% and 35% respectively. The `llc_load_thr` parameter was established to 5 cache ways (roughly 45% of the total way count). As for LFOC+'s settings, we used exactly the same parameter values as in LFOC+'s original work [7], where a UMA machine was used but with the same processor model. Lastly, to conduct a fairer comparison with DINO we set the activation period of the balancing algorithm in both DC and DINO to 1 s, as in DINO's first evaluation [5]. The low and high LLCMPKI thresholds for application classification in DINO were set to 3 and 24 respectively for our platform, which features a much bigger LLC than the platform used by DINO's creators [5].

For our evaluation, we followed the standard practice when assessing the effectiveness of LLC-partitioning strategies for compute-intensive workloads [2], [7], [12], [21], [24], [26], [31], namely, we consider randomly generated mixes comprising long-running programs from standard benchmark suites. Specifically, we used workloads that combine single-threaded

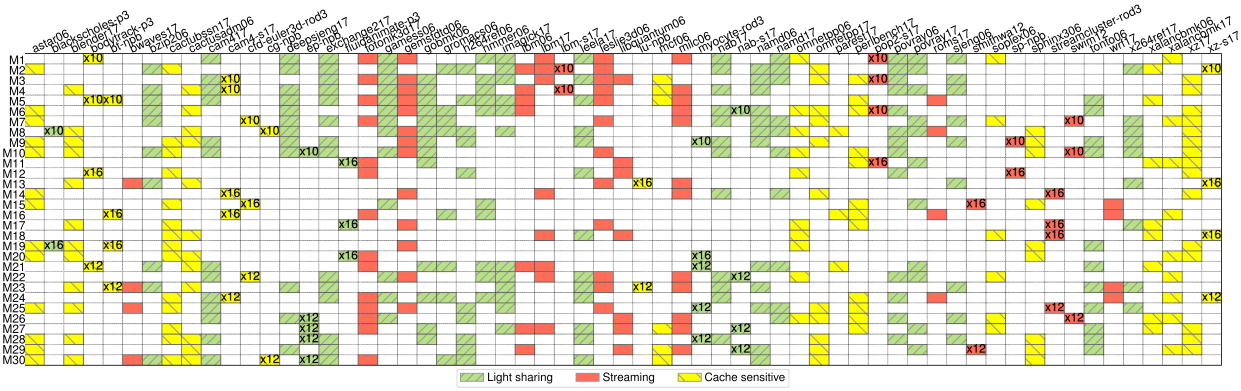


Fig. 9. Each table row M_i depicts the composition of the i-th workload used in our experiments. For each benchmark, the dominant application class observed during the execution is displayed in the associated column. When a program is not included in a workload a blank cell is used in the corresponding row. The "xN" mark indicates that the associated benchmark runs with N threads. A suffix has been added to each benchmark's name to indicate the corresponding suite (e.g., "17" denotes SPEC CPU2017, "-p3" is for PARSEC3 and "-npb" for NAS Parallel Benchmarks).

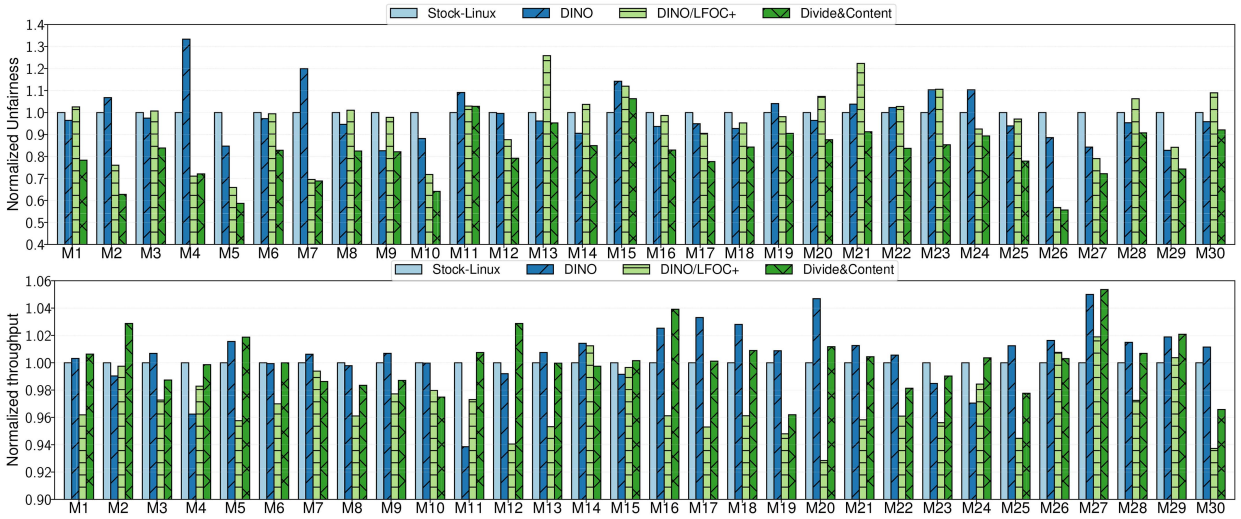


Fig. 10. Normalized unfairness and STP values obtained by the various strategies.

and multithreaded applications, and utilize 61 different programs from 6 benchmark suites: SPEC CPU2006, CPU2017, PARSEC3, SPEC OMP2012, NAS Parallel Benchmarks and Rodinia. Fig. 9 depicts the composition of the 30 randomly generated program mixes used for our experiments, which combine varying amounts of streaming, cache-sensitive and light-sharing applications. In all workloads, the total thread count was set to match the number of cores on our platform (40). Particularly, as indicated in Fig. 9 all workloads include 2 multithreaded programs, and a varying number of single-threaded applications (20 in M1-M10, 8 in M11-M20 and 16 in M21-M30). Multithreaded programs run with 10 (M1-M10), 16 (M11-M20) or 12 threads (M21-M30). These workloads allow us (1) to perform a fair comparison with DINO [5], where similar workloads were used, (2) to explore scenarios with different degree of competition for shared resources, including a varying number of programs of diverse contention classes (as shown in Fig. 9), and (3) to demonstrate that DC delivers fairness and does not introduce important overheads, even with large application counts. Notably, our experimental platform features a larger core count than any of those used in prior work on cache-clustering [2], [7], [12], [21], [24], [26].

In running the workloads we follow a similar methodology to that of [7], [17], [47], which deals with the divergences in the applications' completion times by keeping the system load constant throughout an experiment. Specifically, we ensure that all applications in the mix are started simultaneously, and when one of them completes, it is restarted repeatedly until the longest application in the set completes three times. We then measure unfairness and STP, by considering the completion times for each program. All the workloads were launched 10 times under each strategy so as to assess the strategy's degree of variability across multiple runs of the same workload.

TABLE III
AVERAGE, AND MAXIMUM GAINS/REDUCTIONS FOR VARIOUS METRICS WITH RESPECT TO STOCK-LINUX

Strategy	mean	max
DINO	1.34%	17.39%
DINO/LFOC+	5.41%	43.23%
Divide&Content	18.67%	44.35%

(a) Unfairness reductions

Strategy	mean	max
DINO	0.56%	5.00%
DINO/LFOC+	-2.91%	1.89%
Divide&Content	0.12%	5.36%

(b) Throughput gains

Strategy	mean	max
DINO	0.63%	9.36%
DINO/LFOC+	-2.31%	16.65%
Divide&Content	1.94%	17.95%

(c) Reductions in ANTT

Strategy	mean	max
DINO	-1.07%	28.28%
DINO/LFOC+	2.28%	53.14%
Divide&Content	33.24%	59.65%

(d) Reductions in UnfairnessCoV

B. Discussion of Experimental Results

Fig. 10 shows the degree of unfairness and throughput (normalized w.r.t. Stock-Linux) delivered by the different contention-aware strategies. For each workload and strategy, the figure shows the arithmetic mean of the various metrics recorded in the multiple runs of each workload. Note that we use separate charts (discussed later) to illustrate the enormous performance variability provided by Stock-Linux from run to run. Results in Fig. 10 clearly reveal that our DC approach outperforms the remaining strategies in terms of fairness for the vast majority of the program mixes. As reported in Table III A, DC reduces unfairness by up to 44.35% (M26 workload) and 18.67% on average relative to Stock-Linux. Compared to DINO and DINO/LFOC+, DC improves fairness by 17% and 13.3% on average, respectively. As for throughput, the differences among approaches are smaller; in fact, for most of the workloads, all strategies operate in a tight 4% range. Nevertheless, DC is able to provide substantial fairness gains vs. Stock-Linux without a significant impact on average throughput (see Table III b).

For the sake of completeness, Table III c and III d provide the average and maximum reductions in other popular system-wide metrics [12]: the Average Normalized Turnaround Time (*ANTT*), and an alternative metric to assess the degree of unfairness, referred to as *UnfairnessCoV*, based on the Coefficient of Variation (CoV) across per-application slowdowns. Regarding ANTT, DC is able to reduce it by almost 2% on average. We also observed that the *UnfairnessCoV* metric greatly magnifies fairness improvements with respect to the *Unfairness* metric, so providing both values as a reference is in order for proper comparison across works. In particular, DC averages a 33.24% reduction in *UnfairnessCoV*, whereas the other approaches achieve a similar average value than Stock-Linux.

As discussed in Section I, Stock-Linux fails to provide repeatable results across multiple runs of the same workload. Fig. 11(a) and (b) depict the variability in unfairness and throughput for the first 10 workloads. As it is evident, DINO and DC are capable to deliver more consistent results (so is DINO/LFOC+, despite the omission of its results here for the sake of clarity in the charts). However only DC reaps substantial fairness benefits across the board. Stock-Linux's variability stems from the fact that the Linux scheduler initially performs random thread-to-core-group mappings, and then tries to keep threads on the same core they run for as long as possible. Thread migrations are only triggered to enforce load balance, while avoiding migrations across NUMA nodes if possible. These actions do not ensure a balanced and consistent degree of LLC and memory-bandwidth contention. As an illustrative example, Fig. 11(c) shows the impact of Stock-Linux's random mappings for workload M4. Clearly, cache-sensitive programs (like `mcf`) and bandwidth-intensive ones (e.g., `milc`) are highly exposed to this variable degree of contention which causes substantial slowdown divergences from run to run. By contrast, DC effectively addresses this variability while reducing the per-application slowdown significantly; take for instance the case of the M4 workload, depicted in Fig. 11(d).

We turn back our attention to Fig. 10, which reveals DINO's mixed results. Particularly, we observe that DINO provides modest throughput improvements over Stock-Linux in many cases, while ensuring consistent performance across runs. These improvements are the result of spreading among core groups those applications with a substantial LLCMPKI (*superdevils* and *devils*). In doing so, DINO evens out the aggregate number of LLC misses across NUMA nodes, which contributes to reducing the degree of bandwidth contention and, in turn, improves the performance of streaming bandwidth-intensive programs. DC also reduces bandwidth contention, but it may slightly degrade the performance of streaming programs (hence the lower throughput in some cases). This stems from its reliance on LFOC+ [7], which confines streaming programs into a few LLC partitions for effective isolation from cache-sensitive applications.

While in some scenarios DINO reduces unfairness by more than 15% w.r.t. Stock-Linux (e.g. M5, M9 or M27), it has a substantially unfairer behavior than DC in most cases. One of the main reasons behind this trend is DINO's reliance on the LLCMPKI metric to classify applications at runtime, which poses a number of issues. Specifically, the LLCMPKI is known to be a misleading indicator of the application's degree of sensitivity to shared-resource contention [7]. This metric does not enable contention-aware strategies to distinguish between cache-sensitive and streaming programs, as both types of programs could exhibit an equally high LLCMPKI at runtime [7]. In spreading *superdevils* and *devils* across multiple core groups, DINO seldom separates streaming from cache-sensitive programs, which severely degrades the performance of the latter. This stands in contrast with what the theoretically optimal fairness-wise placement does when cache-partitioning is not used (see Section IV), where this separation is paramount. Of special attention is the case of M4, where DINO systematically maps the cache-sensitive `mcf` program and streaming aggressor

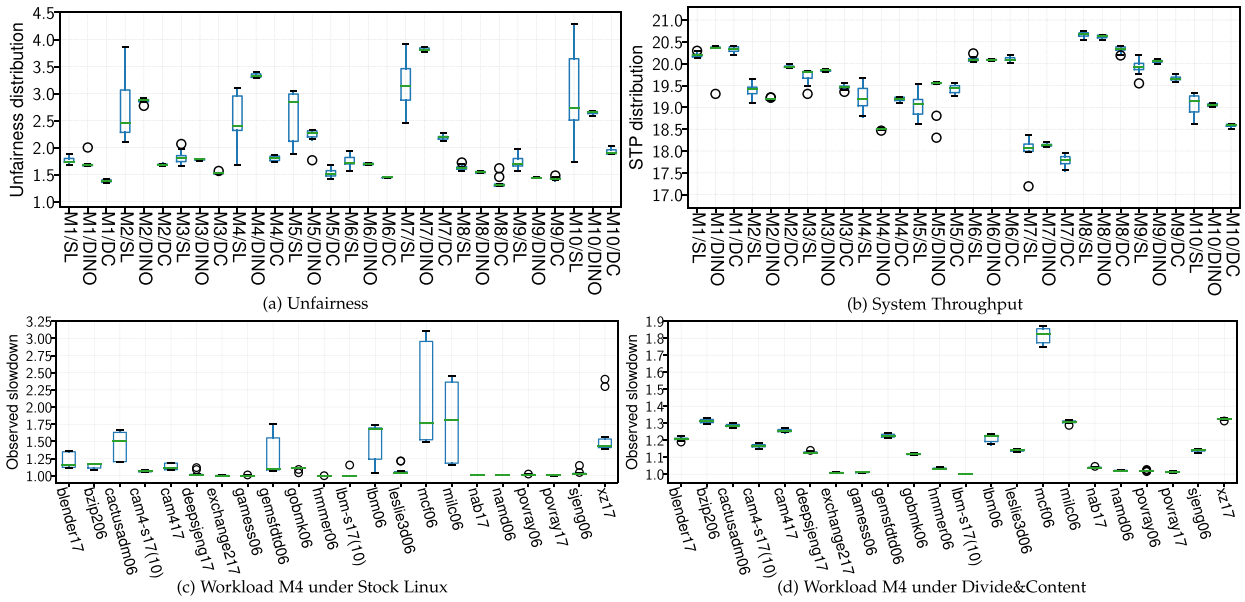


Fig. 11. Variability of the unfairness -Fig.(a)- and the STP metric -Fig.(b)- across multiple runs of workloads M1-M10 with Stock-Linux (SC), DINO and Divide&Content (DC). Figs (c) and (d) show the distribution of the per-application slowdown values observed in different runs of the M4 workload with Stock-Linux and DC, respectively.

applications on the same core group, slowing down `mcf` by a factor of 3.35x, which introduces substantial unfairness. Note also that the DINO class assigned to an application at runtime depends –as the LLCMPKI– on its actual LLC occupancy, which, in turn, largely depends on their group co-runners at that time (recall that DINO does not partition the LLC). Take for instance, the `xalancbmk` or `mcf` programs which are classified most of the time as *superdevils* when the cache occupancy is lower than 3 cache ways, and as *devils* for high LLC-space allocations. This, in turn, causes class oscillations triggered by changes in the behavior of the remaining programs, which ultimately leads to extra thread migrations (hence to overheads). By contrast, under DC, these applications are classified as cache sensitive most of the time, and the LFOC+ policy effectively assigns them to a partition not shared with streaming programs, irregardless of their current LLCMPKI or their cache occupancy.

Lastly, we zoom in on DINO/LFOC+'s results in Fig. 10. Applying the LFOC+ partitioning policy on top of DINO does bring substantial fairness benefits in some cases (e.g., an extra 32% improvement in M5). However, and contrary to our initial expectations, DINO/LFOC+ underperforms DINO in many program mixes. Two main factors are behind this behavior. First, the thread-to-group mappings performed by DINO based on the application's classes often lead to an uneven distribution of cache-sensitive programs among core groups, thus concentrating many of these programs in one group. As pointed out in Section II-C, reducing unfairness is challenging under such a high degree of LLC-space competition. So, the LFOC+ strategy does not yield substantial benefits in this context. Second, in many workloads– such as M13, M21 or M28– DINO/LFOC+ substantially increases the number of cross-group migrations w.r.t. DINO, and the associated overhead negates the benefits of using LFOC+. Essentially, the higher migration rate comes from frequent oscillations in the application classes caused by the effect of LLC-partitioning. Under DINO/LFOC+, the LLC is not partitioned while cross-group thread migrations are in progress, and once completed the cache-partitioning algorithm is executed. Since the DINO class is heavily dependent on an application's LLC occupancy, changes in occupancy lead to oscillations in the application classes, and, in turn, to extra migrations. These two issues are not present in Divide&Content, which makes the most out of the LFOC+ cache-clustering policy by carefully balancing the degree of competition for LLC space across core groups.

TABLE IV
AVERAGE STATISTICS OBTAINED FOR WORKLOADS M1-M10

Strategy	Cross-group thread migrations per sec.	Average time contention alg.	Average time LFOC+
DINO	5.97	21.16μs	N/A
DINO/LFOC+	5.77	21.73μs	14.72μs
Divide&Content	2.32	11.15μs	12.99μs

C. Overhead Analysis

Table IV shows the average cross-group thread migration rate, as well as other key overhead indicators for the various strategies, gathered during the execution of the first 10 workloads, which include the largest number of applications (i.e. 22) across all program mixes. These statistics were collected with the SystemTap kernel tracing tool in separate launches of the workloads. As it is evident, the cross-group thread migration rate under DINO and DINO/LFOC+ is roughly 2.5 times higher than that of DC. DC's lower migration rate is the result of

its various mechanisms to avoid migrations (see Section V), coupled with LFOC+'s classification method, which does not lead to as many class changes as in DINO. Table IV also reflects that DINO's contention-aware placement algorithm takes almost twice as much to run on average (21.16 μs) than DC's contention balancing algorithm (11.15 μs). Nevertheless, devoting tens of microseconds every second on one core of the system does not constitute a substantial overhead.

A detailed overhead analysis of LFOC+'s partitioning algorithm can be found in [7], where the same processor model was used, but in a single-socket setting. By following the same procedure for the overhead analysis, we corroborated that our LFOC+ implementation within DC introduces a similarly low overhead in our NUMA setting. PMCs –required by LFOC+'s classification method– are sampled continuously using coarse-grained instruction windows as in [7], which introduce negligible overheads. LFOC+'s sampling mode was engaged on average on a per-core-group basis only for 0.42% of the workloads' total execution time. Its partitioning algorithm (invoked twice per second and per core-group under DC's) has an average execution time of 12.99 μs (as shown in Table IV), which leads to low overhead.

VII. Conclusions and Future Work

In this paper, we introduced Divide&Content (DC), a novel OS-level resource manager that effectively combines contention-aware thread placement and cache-partitioning so as to improve fairness in NUMA multicores. DC's design was driven by the insights drawn from our comprehensive simulation study on how to best combine LLC-partitioning and thread placement. We conclude that coordinated thread-mapping and cache-partitioning decisions are paramount to optimize fairness. Our experimental evaluation demonstrates that DC substantially improves fairness with respect to Linux and a NUMA-aware contention-conscious strategy [5], while providing consistent performance and throughput across runs. Notably our DC implementation in a kernel module, which leverages the PMCSched framework [45], can be loaded in unmodified (vanilla) versions of the Linux kernel with standard tracing support enabled.

As for future work, we plan on exploiting hardware extensions for bandwidth limitation [22] together with LLC-partitioning to improve isolation among applications/VMs in CMPs. We will also conduct an experimental demonstration of the scalability and fairness benefits of DC on NUMA systems including more than two LLC core groups, like processors with the EPYC Milan microarchitecture, which feature multiple core groups/LLCs within a single socket.

References

[1] M. Shahrad, S. Elnikety, and R. Bianchini, "Provisioning differentiated last-level cache allocations to VMs in public clouds," in *Proc. ACM Symp. Cloud Comput.*, 2021, pp. 319–334.

[2] K. Nikas et al., "DICER: Diligent cache partitioning for efficient workload consolidation," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.

[3] D. Lo et al., "Heracles: Improving resource efficiency at scale," in *Proc. 42nd Annu. Int. Symp. Comput. Archit.*, 2015, pp. 450–462.

[4] D. Gureya et al., "Bandwidth-aware page placement in NUMA," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 546–556.

[5] S. Blagodurov et al., "A case for NUMA-aware contention management on multicore systems," in *Proc. USENIX Annu. Tech. Conf.*, 2011, pp. 557–558.

[6] Z. Majo and T. R. Gross, "Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead," *SIGPLAN Not.*, vol. 46, no. 11, pp. 11–20, Jun. 2011.

[7] J. C. Saez, F. Castro, G. Fanizzi, and M. Prieto-Matias, "LFOC+: A fair OS-level cache-clustering policy for commodity multicore systems," *IEEE Trans. Comp.*, vol. 71, no. 8, pp. 1952–1967, Aug. 2022.

[8] L. Subramanian, D. Lee, V. Seshadri, H. Rastogi, and O. Mutlu, "BLISS: Balancing performance, fairness and complexity in memory access scheduling," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 10, pp. 3071–3087, Oct. 2016.

[9] S. Zhuravlev et al., "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surveys*, vol. 45, no. 1, pp. 4:1–4:28, Dec. 2012.

[10] H. Xu, S. Wen, A. Gimenez, T. Gamblin, and X. Liu, "DR-BW: Identifying bandwidth contention in NUMA architectures with supervised learning," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 367–376.

[11] D. Xu et al., "Providing fairness on shared-memory multiprocessors via process scheduling," in *Proc. 12th ACM SIGMETRICS Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 295–306.

[12] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez, "Application clustering policies to address system fairness with intel's cache allocation technology," in *Proc. IEEE 26th Int. Conf. Parallel Archit. Compilation Techn.*, 2017, pp. 194–205.

[13] S. Chen et al., "PARTIES: QoS-aware resource partitioning for multiple interactive services," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2019, pp. 107–120.

[14] R. B. Roy, T. Patel, and D. Tiwari, "SATORI: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains," in *Proc. IEEE/ACM 48th Annu. Int. Symp. Comput. Archit.*, 2021, pp. 292–305.

[15] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory bandwidth management for efficient performance isolation in multicore platforms," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 562–576, Feb. 2016.

[16] J. Feliu, J. Sahuquillo, S. Petit, and J. Duato, "Perf&Fair: A progress-aware scheduler to enhance performance and fairness in SMT multicores," *IEEE Trans. Comput.*, vol. 66, no. 5, pp. 905–911, May 2017.

[17] A. Garcia-Garcia, J. C. Saez, and M. Prieto-Matias, "Contention-aware fair scheduling for asymmetric single-ISA multicore systems," *IEEE Trans. Comp.*, vol. 67, no. 12, pp. 1703–1719, Dec. 2018.

[18] F. V. Zacarias et al., "Intelligent colocation of HPC workloads," *J. Parallel Distrib. Comput.*, vol. 151, pp. 125–137, 2021.

[19] S. Mittal, "A survey of techniques for cache partitioning in multicore processors," *ACM Comput. Surveys*, vol. 50, no. 2, pp. 27:1–39, 2017.

[20] M. Qureshi and Y. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. IEEE/ACM 39th Annu. Int. Symp. Microarchit.*, 2006, pp. 423–432.

[21] N. El-Sayed, A. Mukkara, P. -A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A hybrid cache partitioning-sharing technique for commodity multicores," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 104–117.

[22] Intel 64 and IA-32 architectures developer's manual: vol. 3b, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[23] AMD, "AMD64 technology platform QoS extensions," 2022. [Online]. Available: https://developer.amd.com/wp-content/resources/56375.pdf

[24] L. Pons, J. Sahuquillo, V. Selfa, S. Petit, and J. Pons, "Phase-aware cache partitioning to target both turnaround time and system performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 11, pp. 2556–2568, Nov. 2020.

[25] R. Chen et al., "DRLPart: A deep reinforcement learning framework for optimally efficient and robust resource partitioning on commodity servers," in *Proc. 30th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2021, pp. 175–188.

[26] L. Pons et al., "Cache-poll: Containing pollution in non-inclusive caches through cache partitioning," in *Proc. 51st Int. Conf. Parallel Process.*, 2022, pp. 1–11.

[27] A. Garcia-Garcia et al., "PBBCache: An open-source parallel simulator for rapid prototyping and evaluation of cache-partitioning and cache-clustering policies," *J. Computat. Sci.*, vol. 42, 2020, Art. no. 101102.

[28] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, no. 3, pp. 42–53, May/Jun. 2008.

[29] S. Eyerman and L. Eeckhout, "Restating the case for weighted-IPC metrics to evaluate multiprogram workload performance," *IEEE Comput. Archit. Lett.*, vol. 13, no. 2, pp. 93–96, Jul./Dec. 2014.
[30] H. Vandierendonck and A. Seznec, "Fairness metrics for multi-threaded processors," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 4–7, Jan./Jun. 2011.
[31] J. Park et al., "CoPart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers," in *Proc. 14th EuroSys Conf.*, 2019, pp. 1–16.
[32] A. Garcia-Garcia et al., "LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 14:1–14:10.
[33] T. Morad et al., "EFS: Energy-friendly scheduler for memory bandwidth constrained systems," *J. Parallel Distrib. Comput.*, vol. 95, pp. 3–14, 2016.
[34] T. Patel and D. Tiwari, "CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2020, pp. 193–206.
[35] J. Feliu, S. Petit, J. Sahuquillo, and J. Duato, "Cache-hierarchy contention-aware scheduling in CMPs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 03, pp. 581–590, Mar. 2014.
[36] S. Kundan et al., "A pressure-aware policy for contention minimization on multicores," *ACM Trans. Archit. Code Optim.*, vol. 19, no. 3, pp. 1–26, 2022.
[37] T. Marinakis and I. Anagnostopoulos, "Performance and fairness improvement on CMPs considering bandwidth and cache utilization," *IEEE Comput. Architect. Lett.*, vol. 18, no. 2, pp. 1–4, Jul./Dec. 2019.
[38] D. Parker and S. Radvan, "Red Hat Enterprise Linux 7. Virtualization tuning and optimization guide," 2014. [Online]. Available: https://linux.web.cern.ch/centos7/docs/rhel/Red_Hat_Enterprise_Linux-7-Virtualization_Tuning_and_Optimization_Guide-en-US.pdf
[39] O. Papadakis et al., "You can't hide you can't run: A performance assessment of managed applications on a NUMA machine," in *Proc. 17th Int. Conf. Managed Program. Lang. Runtimes*, 2020, pp. 80–88.
[40] N. Denoyelle et al., "Data and thread placement in NUMA architectures: A statistical learning approach," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.
[41] M. Dashti et al., "Traffic management: A holistic approach to memory placement on NUMA systems," in *Proc. 18th Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2013, pp. 381–394.
[42] X. Zhao et al., "NumaPerf: Predictive NUMA Profiling," in *Proc. ACM Int. Conf. Supercomput.*, 2021, pp. 52–62.
[43] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi, "CuttleSys: Data-driven resource management for interactive services on reconfigurable multicores," in *Proc. IEEE/ACM 53rd Annu. Int. Symp. Microarchit.*, 2020, pp. 650–664.
[44] AMD, "High performance computing: Tuning guide for AMD EPYC 7002 series processors," 2020. Accessed: May 21, 2021. [Online]. Available: https://developer.amd.com/wp-content/resources/56827--1-0.pdf
[45] C. Bilbao, J. C. Saez, and M. Prieto-Matias, "Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake," *Concurrency Comput. Pract. Experience*, In press, 2023, Art. no. e7814, doi: 10.1002/cpe.7814.
[46] R. Love, *Linux Kernel Development*, 3rd ed. Reading, MA, USA: Addison-Wesley, 2010.
[47] D. Shelepov et al., "HASS: A scheduler for heterogeneous multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 2, pp. 66–75, 2009.

Carlos Bilbao received the MSc degree in computer engineering from Virginia Tech in 2021. He is now an operating systems developer and working toward the PhD degree in computer engineering with Complutense University of Madrid (UCM). His research interests include system software, scheduling, virtualization and multicore systems. His recent research activities focus on shared resource contention and resource-aware scheduling on multicore systems.

Juan Carlos Saez received the PhD degree in computer science from the Complutense University of Madrid (UCM) in 2011. Currently, he is an associate professor with the Department of Computer Architecture, and the Campus Representative, UCM of the USENIX international association. His research interests include system software, scheduling, runtime systems, performance monitoring, and resource management. His recent research activities focus on improving the system software support for emerging hardware platforms.

Manuel Prieto-Matias received the PhD degree from the Complutense University of Madrid (UCM) in 2000. Currently, he is a full professor with the Department of Computer Architecture at UCM. His research interests include parallel computing and computer architecture. His current research addresses emerging issues related to heterogeneous systems and energy-aware computing, with a special emphasis on the interaction between the OS and the underlying architecture. He has co-written numerous articles in journals and international conferences on parallel computing and computer architecture.

Chapter 5

Exploiting Elasticity via OS-runtime Cooperation to Improve CPU Utilization in Multicore Systems

Full citation

Javier Rubio, Carlos Bilbao, Juan Carlos Saez, and Manuel Prieto-Matias, *Exploiting Elasticity via OS-runtime Cooperation to Improve CPU Utilization in Multicore Systems* 2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Dublin, Ireland, 2024

Abstract

The chip multicore processor (CMP) architecture has become the predominant design choice for contemporary general-purpose systems across multiple sectors of commercial technology. Thanks to technological progress, CMP systems can now feature hundreds of cores. While multithreaded applications may potentially benefit from the increasing core counts, leveraging all available cores is not always feasible due to limited Thread-Level Parallelism (TLP), load imbalance among threads, and other scalability bottlenecks.

Colocating multiple applications on the same node is becoming a popular practice to maximize processor utilization. In HPC, malleability –the ability to dynamically alter the number of active threads within the same application–, is also being exploited at the runtime-system level to better deal with scenarios exhibiting time-varying scalability. In the cloud, application colocation is leveraged along with different forms of coarse-grained elasticity to cater to the varying resource demands. This work introduces an operating system (OS) level elastic mechanism designed to

efficiently leverage idle CPU periods in workloads consisting of unmodified applications, many of which do not rely on a runtime system to function. This mechanism constitutes a form of fine-grained vertical elasticity that leverages cooperation between the runtime system and the operating system to maximize CPU utilization. To this end, it opportunistically increases the active thread count of malleable applications during idle periods. We implemented our proposed operating system extensions in the Linux kernel, and augmented the GNU's OpenMP runtime to show a proof of concept of the required OS-runtime interaction. By using diverse multithreaded programs, we demonstrate the ability of the proposed operating system support to substantially improve the system throughput.

Exploiting Elasticity via OS-runtime Cooperation to Improve CPU Utilization in Multicore Systems

Javier Rubio, Carlos Bilbao, Juan Carlos Saez, and Manuel Prieto-Matias
Facultad de Informática
Complutense University of Madrid
Madrid, Spain
Email: {jrubio05,cbilbao,jcsaezal,mpmatias}@ucm.es

***Abstract*—The chip multicore processor (CMP) architecture has become the predominant design choice for contemporary general-purpose systems across multiple sectors of commercial technology. Thanks to technological progress, CMP systems can now feature hundreds of cores. While multithreaded applications may potentially benefit from the increasing core counts, leveraging all available cores is not always feasible due to limited Thread-Level Parallelism (TLP), load imbalance among threads, and other scalability bottlenecks.**

Colocating multiple applications on the same node is becoming a popular practice to maximize processor utilization. In HPC, malleability –the ability to dynamically alter the number of active threads within the same application–, is also being exploited at the runtime-system level to better deal with scenarios exhibiting time-varying scalability. In the cloud, application colocation is leveraged along with different forms of coarse-grained elasticity to cater to the varying resource demands. This work introduces an operating system (OS) level elastic mechanism designed to efficiently leverage idle CPU periods in workloads consisting of unmodified applications, many of which do not rely on a runtime system to function. This mechanism constitutes a form of fine-grained vertical elasticity that leverages cooperation between the runtime system and the OS to maximize CPU utilization. To this end, it opportunistically increases the active thread count of malleable applications during idle periods. We implemented our proposed OS extensions in the Linux kernel, and augmented the GNU's OpenMP runtime to show a proof of concept of the required OS-runtime interaction. By using diverse multithreaded programs, we demonstrate the ability of the proposed OS support to substantially improve the system throughput.

***Index Terms*—Multicore processors, operating system, elasticity, Linux kernel, runtime system, OpenMP, malleability.**

1. Introduction

The chip multicore processor (CMP) architecture stands today as the *de facto* design choice for modern general-purpose systems. CMPs find a place in embedded devices, personal computers, high-performance servers, and virtually all sectors of commercial technology. Thanks to the striking advances in technology and processor design, CMP systems already reach hundreds of cores in server platforms. Likewise, a substantial increase in the core count is also taking place in the desktop market [1]. Take for instance the 14th generation of Intel Core i9 processors, with up to 24 cores.

Multithreaded applications constitute a straightforward way to benefit from increasing core counts. However, not every multithreaded program can effectively leverage all the system's cores, due to program phases with a limited degree of Thread-Level Parallelism (TLP) –e.g., explicit sequential sections– [2], the presence of load imbalance between worker threads [1], or other scalability bottlenecks associated with contention on shared memory-related resources between cores [3], [4].

In HPC environments, colocating multiple applications on the same node is becoming a popular practice towards improving processor utilization [4], [5]. Several colocation approaches have been explored in this context. One option is to statically partition the system's CPU resources, so that each co-running applications is assigned an exclusive set of cores throughout the execution [4]. However, the dynamic degree of parallelism that many HPC application exhibit over time makes this approach not only challenging but suboptimal [5]. Another alternative is to dynamically adjust the number of active threads of the co-running applications, so that they use an appropriate number of cores that caters to the needs of its current execution phase [6]. To this end, the application must support dynamic *malleability*, this is, the ability to utilize a varying number of threads/cores at runtime. In HPC, delivering malleability support to applications is simpler than in other scenarios, as HPC applications often make use of runtime systems; implementing malleability support at the runtime system level often requires little or no changes in the application code. In turn, cooperation between the multiple runtime systems of applications colocated in the same node makes it easier to implement policies that optimize resource utilization [6], [7].

In the cloud, maximizing resource utilization is paramount to increase revenue and reduce utility costs [8]. Increasing resource usage is effected via oversubscription [9] and elasticity [10]. Specifically, elasticity can be exploited horizontally (i.e., by increasing the number of instances of the application using multiple containers/VMs),

or vertically (i.e. by dynamically increasing the associated CPU and memory resources). Cloud elasticity mechanisms are based on the continuous monitoring of resource usage and/or the external requests that an application receives. Previous research has highlighted that, while this is suitable for client-server applications, it does not meet the needs of workloads that do not depend on external requests, as scientific applications [10]. Another limitation of current cloud dynamic elasticity mechanisms is that they are applied at coarse granularities; this often makes it difficult to effectively utilize cores that go idle just for a few seconds.

In this work, we propose an operating-system (OS) level elastic mechanism to efficiently harness CPU resources during idle periods on a multicore system. Specifically, we showcase the potential of automatically maximizing CPU utilization and improving system throughput when running a mix of unmodified applications, some of which feature malleability support in the runtime system. To do so, our proposal leverages explicit interaction between the OS and the runtime system, and reacts to OS-visible thread activations and deactivations (i.e., a thread blocks when waiting). Our proposal is meant to be used as a complementary method for vertical elasticity in the cloud, providing a means to increase CPU utilization and throughput that operates at a much shorter timescales than existing methods of vertical elasticity [8]. Moreover, it can be also used in general-purpose OSs to opportunistically improve CPU utilization in desktop environments.

To the best of our knowledge, this proposal is the first to leverage elastic OS-runtime scheduling for workloads spanning both unmodified malleable and non-malleable application types, all without the need for instrumentation or alterations to the applications' code. While previously-proposed techniques to improve CPU utilization also rely on application colocation and/or perform elastic scheduling decisions [4], [6], [7], they require that every application in the workload use a customized runtime system. Unlike our approach, these techniques do not leverage OS-runtime interaction, and are incompatible with legacy software, which may not rely on a runtime system to function.

Our work makes the following contributions:

1) We designed an OS-level mechanism that leverages idle cores left by non-malleable applications to opportunistically increase the active thread count of malleable applications. The proposed OS extension also reclaims the extra cores populated by malleable programs automatically, when non-malleable ones increase their amount of TLP.
2) We implemented the proposed OS-level support as a loadable kernel module, which can be loaded in unmodified Linux kernels (no kernel patches required).
3) As a proof of concept of malleability delivered by the runtime system, we extended the GNU's OpenMP runtime with malleability support for loop-based parallel programs. To benefit from this support, no changes in the application source code are required.
4) To assess the effectiveness of our proposal, we employ program mixes that include malleable and non-malleable applications. Our experimental evaluation shows that our approach brings substantial throughput gains (24.5% average improvement w.r.t. the default Linux scheduler). This is accomplished by accelerating malleable applications without causing a significant decline in the performance of non-malleable programs.

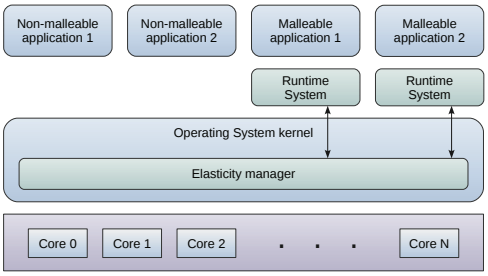


Figure 1: System overview

The remainder of this paper is organized as follows. Sec. 2 describes the design and implementation of our proposal. Sec. 3 covers the experimental evaluation. Sec. 4 discusses related work. Finally, Sec. 5 concludes the paper.

2. Design and implementation

Our proposal leverages cooperation between the runtime system and the OS kernel to maximize CPU utilization and improve system throughput via dynamic malleability.

Fig. 1 depicts the interaction between the various components of our framework and its intended use. The workload combines a mix of applications; each application may run directly on top of the operating system or inside a separate container. Some applications run unmodified programs, potentially legacy software, where threads not doing useful work block. These applications can be single- or multi-threaded, and they may or may not use a runtime system. In case a runtime is used, it does not interact with the OS or exploits malleability. For simplicity, we refer to these programs as *non-malleable* applications. Other programs leverage malleability via a cooperative runtime system that runs at user-space and interacts with our proposed OS-level elasticity manager. As a proof of concept to show the potential of this OS-runtime interaction, we augmented the GNU OpenMP runtime system with malleability support in *parallel for* constructs. This brings malleability to a wide range of well-known OpenMP benchmarks that allow us to quantify the benefits when using unmodified applications.

At a high level, our proposed system works as follows. The OS-level elasticity manager continuously monitors the CPU utilization of non-malleable applications. When at least one non-malleable application consistently leaves idle cores, the kernel reassigns them to malleable programs. To do so, it communicates the target active thread count to the runtime system through a shared memory region, where changes are notified by OS via a signal delivered to the associated process. The runtime system is ultimately responsible for enforcing the desired active thread count. In addition, the

elasticity manager exploits thread packing to confine malleable and non-malleable programs in different core groups, so as to minimize interference between applications.

We implemented our elasticity manager as an OS scheduler extension in the Linux kernel v5.16.20. Specifically, this extension is bundled as a plugin of the PMCSched framework [1], which can be dynamically loaded in unmodified kernels (no patch required) via a customizable kernel module. The malleability extensions for OpenMP were incorporated into the GNU OpenMP runtime system that comes with GCC 11.2, also referred to as *libgomp*.

In the remainder of this section, we first describe the way in which the kernel-level elasticity manager interacts with the user-level runtime system. We then present the inner workings of the elasticity manager. Lastly, we explain how the runtime system enforces the necessary active thread count in parallel for OpenMP constructs.

2.1. OS-Runtime interface

Communication between the malleability-enabled runtime system and the OS kernel takes place through a per-application shared memory region. In our implementation, this region holds a simple two-field data structure. The first field, called `is_malleable`, is a flag that the runtime system sets to 1 to inform the OS that the application supports malleability. In our OpenMP runtime implementation, this flag is set to 1 when the program enters the first parallel section, although the runtime has the potential to selectively disable it when the application enters phases that are not amenable to malleability, such as explicit sequential regions. The second field (`target_threads`) is exclusively modified by the elasticity manager and is used to inform the runtime system of the maximum allowed number of active threads for the application. Every time this integer field is altered, the elasticity manager sends the SIGUSR1 signal to the application process. We opted to deliver that signal as it is one reserved for user-defined purposes in the POSIX standard. The runtime system must install the associated signal handler and carry out the necessary implementation-specific actions upon signal reception to enforce that maximum target thread count.

We should highlight that allowing an application to communicate with the OS kernel via shared memory to exchange scheduling-relevant information is inspired by the `schedctl()` system call present in the Solaris operating system [11]. This call returns a pointer to a scheduling-related data structure, allowing efficient bidirectional communication between user and kernel spaces without the need for additional system calls. To create the per-application memory region shared with the Linux kernel, we leverage the built-in support provided by PMCSched [1]. This allows us to seamlessly create the region by relying on the existing set of Linux system calls. To this end, the main thread must open the */proc/pmc/schedctl* special file –exported by PMCSched– and then pass the returned file descriptor to the `mmap()` system call. Internally, PMCSched associates the custom data structure with the corresponding process descriptor (Linux `task struct`), thus making it available to any PMCSched plugin, including our elasticity manager, described in the next section.

2.2. Elasticity manager

The elasticity manager (EM) is an OS scheduler extension that distributes the system's cores among malleable and non-malleable applications in the workload. It works cooperatively with the Linux process scheduler by dynamically adjusting CPU affinities and by communicating with the runtime system of each malleable application. EM reacts to key scheduling events, such as when a new process/thread is created, when a thread terminates or blocks, and when it becomes runnable again. Additionally, EM's code is activated periodically in a system-wide fashion to assign cores to the various applications on the system. EM was implemented as a plugin of the PMCSched tool [1]; the interface of a PMCSched plugin consists of a set of callbacks allowing to capture the aforementioned scheduling events [12]. The periodic activation of EM is also triggered by PMCSched.

EM maintains three global linked lists: *active* applications (processes with at least one runnable thread), *idle* applications (non-malleable programs with no runnable threads at this point), and *malleable* applications whose target thread count and affinity masks needs to be updated. The first two lists may require updating when a thread enters the system, becomes runnable, blocks, or exits. The third list is populated by the periodic system-wide core allocation algorithm described next, which is activated every `em_period` ms. Notably, `em_period` is a configurable parameter of EM.

Between executions of the core allocation algorithm, EM continuously monitors changes in threads' states (i.e., blocked and runnable) to keep track of the amount of time that each application spends with different runnable thread counts. For this purpose, it maintains a per-application *activity vector* – an array whose i-th entry stores the amount of time (in nanoseconds) the application spends with i runnable threads. Activity vectors of non-malleable applications are taken into consideration by EM to decide how to distribute cores among processes; these vectors are reset by the core allocation algorithm so as to start a new monitoring interval, which lasts a full `em_period`. For efficiency reasons, the activity vector is accompanied by a bitmask, where each bit indicates the validity of a particular entry. Maintaining this bitmask allows us to efficiently determine the application's usual thread count, and makes it possible to reset the activity vector without zero-filling all its entries (the entire bitmask is cleared instead). We should also highlight that EM makes changes in a process's activity vector only when one of its threads becomes inactive (blocks or terminates) or runnable again; these events are captured by leveraging specific callbacks provided by PMCsched. When the system is idle, these callbacks are not called, and the periodic core allocation algorithm is not engaged; therefore, EM's code is not invoked during idle system periods.

Alg. 1 depicts the pseudo-code of EM's periodic core allocation algorithm, which comprises two steps. In Step 1,

Algorithm 1 Core allocation algorithm

```
1:  function periodic_core_allocation(void)
2:      nr_remaining_cores = num_online_cpus()
    ▷ lists for malleable and non-malleable applications
3:      m_list=[] ; n_list=[]
    ▷ STEP 1: Determine application types and number of cores used by
    non-malleable applications
4:      for each app in active_list do
5:          if app.shared_region and app.shared_region.is_malleable then
6:              m_list.append(app)
7:          else
8:              nr_remaining_cores -= get_typical_thread_count(app)
9:              n_list.append(app)
10:         reset_activity_vector(app)
    ▷ STEP 2: Assign remaining cores to malleable applications
11:     nr_remaining_apps = len(m_list)
12:     if len(idle_apps) > 0 then
13:         nr_remaining_cores−−
    ▷ Ensure there is at least one core for each malleable application
14:     if nr_remaining_cores < len(m_list) then
15:         nr_remaining_cores = len(m_list)
16:     non_mall_cores = num_online_cpus() - nr_remaining_cores
17:     for each app in m_list do
18:         nr_fair_cores = max(1, ⌊nr_remaining_cores / nr_remaining_apps⌋)
19:         nr_remaining_apps−−
20:         nr_remaining_cores -= nr_fair_cores
21:         if app.shared_region.target_threads != nr_fair_cores then
22:             app.shared_region.target_threads = nr_fair_cores
23:             update_affinity_mask(app, nr_fair_cores, non_mall_cores)
24:             apps_to_update.append(app)
25:     if len(apps_to_update) > 0 then
26:         wake_up_kernel_thread()
```

EM traverses the list of active applications and divides them into two classes: malleable and non-malleable. To determine the application's class, EM attempts to obtain a reference to the structure stored in the memory region shared with the runtime system. If present, this structure is retrieved from the task descriptor of the process's group leader (the task structure from the main thread). The application is classified as malleable only if the shared memory region exists, and the current value of the `is_malleable` attribute is one (recall that the runtime system may dynamically alter this flag). Otherwise, the application is considered to be non-malleable. The loop in Step 1 also calculates the number of cores utilized by each non-malleable application, by invoking the `typical_thread_count()` auxiliary function. This function traverses the application's activity vector from higher to smaller thread counts, disregarding invalid entries, and returns the highest thread count whose entry stores a number no smaller than the `min_threshold` parameter. This configurable parameter establishes the minimum amount of CPU time a process must run within the `em_period` to be considered sufficiently CPU intensive. Note also that EM reserves one core for all idle applications. This substantially reduces the number of context switches in the event a thread from these applications suddenly becomes runnable. This is a common scenario in compute-intensive programs like those we used for evaluation (see Sec. 3).

Step 2 of the algorithm takes care of evenly distributing the remaining cores[1] (variable `nr_remaining_cores`) among malleable applications. To do so, it traverses the local list of malleable applications, and for each one it assigns a target thread count that matches its fair share of `nr_remaining_cores`. If the desired target thread count is different from the current one (i.e., the value stored in the shared memory region), EM updates the `target_threads` attribute in the shared structure. This step of the algorithm also takes care of determining the new CPU affinity mask of each malleable application, so that their threads are confined in specific sets of cores, different from those used by non-malleable applications. Note that in doing so, EM leverages thread-packing [13], as the total number of threads of malleable applications is typically greater than the number of cores assigned to it (see Sec. 2.3). Affinity masks are assigned in such a way that the `nr_remaining_cores` CPUs (cores) with the highest IDs are assigned to malleable programs. Therefore, all CPUs with IDs smaller than `nr_remaining_cores` are devoted to running non-malleable programs. Clearly, this implementation does not consider scenarios where the user explicitly reserves specific cores to run critical threads (e.g. a resource manager), or the fact that threads of non-malleable applications may be pinned to specific cores. As future work we plan to augment EM with a more sophisticated thread-to-core assignment policy that factors in these constraints.

Lastly, the algorithm activates a kernel thread to signal processes whose maximum thread count need adjusting, and to impose the designated per-application affinity masks to all of their threads. We should highlight that a kernel thread is required in this scenario, because the functions of the Linux kernel API that deliver signals and enforce affinity masks –which may trigger thread migrations– are blocking calls. Therefore, they must be invoked from *process context*, which is not the context type where the periodic core allocation algorithm runs (*interrupt context*).

2.3. Malleability in the OpenMP runtime system

The OpenMP runtime system already allows to request the utilization of different number of worker threads in the various parallel regions of a program. While this constitutes a form of malleability, its coarse granularity makes it impossible to react in our target time frame (a matter of milliseconds) in the vast majority of loop-based OpenMP programs. To achieve a finer-granularity and react as soon as possible when a thread change is requested from the kernel, we chose to implement the malleability control within loop-scheduling methods. As as proof of concept, we modified the `dynamic` and `guided` loop-scheduling methods. Note that the `static` loop-scheduling technique is not amenable to malleability. Indeed, in the *libgomp* implementation each worker thread invokes a single runtime call at the beginning of the loop, to get its full share of it.

To adopt malleability within the GNU OpenMP runtime system, we made a number of modifications, some of which affect environment variables. Specifically, we introduced two new environment variables: `GOMP_MALLEABLE` and `GOMP_MAX_THREADS`. These variables allow the user to enable or disable malleability and to indicate the max-

1. Exploring uneven core distribution among applications is out of the scope of this work. However, previous research [6] suggests that this would bring further throughput improvements.

imum number of threads the application can run with, respectively. In our experiments, we systematically set `GOMP_MAX_THREADS` to the total number of cores in the system, which is standard practice when exploiting malleability [6]. Note also that when malleability is enabled, the purpose of the standard `OMP_NUM_THREADS` environment variable was altered to allow specifying the initial number of *active* threads the application uses. Likewise, the `omp_set_num_threads()` OpenMP API function was also modified to allow establishing the same property. Regarding global settings, our modified runtime system automatically sets the default wait policy in synchronization primitives to the `passive` mode, just like in previous work [6]. By allowing worker threads to sleep instead of busy-wait on synchronization primitives (e.g., barriers), we effectively avoid wasting CPU cycles. This is paramount in oversubscription scenarios, which become apparent in multi-program settings, where EM grants a number of cores to the application that is often smaller that its total thread count.

To engage malleability within parallel loops, we employ a global a bit array, including an entry for each thread, to indicate whether it is *active* or *inactive*. Within a loop, active threads act as normal OpenMP worker threads, while inactive ones remain blocked inside the runtime calls associated with removing loop-iterations from the pool in the `dynamic` or `guided` methods. To determine the current chunk when a thread removes iterations from the pool in the `guided` method, we use the current number of active threads in the calculation rather than the total thread count. Note also that to enforce synchronization, we employ standard POSIX mutexes and condition variables. Within the runtime system implementation, the application's master thread takes care of initializing all these global resources. It also requests the creation of the per-application memory region shared with the EM (kernel space), and installs the signal handler for the `SIGUSR1` signal, which EM delivers when a change of the active thread count is in order.

Upon receiving the `SIGUSR1` signal, the runtime system reads the new target thread count value from shared memory, modifies the bit array of active threads accordingly, and wakes up the necessary threads (those in the active state). When a newly active thread wakes up, it proceeds to remove iterations from the pool and executes them. Should the number of active threads decrease upon signal reception, the threads that were recently forced to become inactive will block upon invocation of any runtime call for removing loop-iterations from the shared pool. Lastly, we should highlight that in our implementation all threads (active and inactive) must synchronize with each other in the implicit barrier present at the end of most parallel loops. To this end, when an active thread detects that all iterations for this parallel loop have been completed, it will awake all blocked (inactive) threads, ensuring they can arrive at the barrier.

3. Experimental evaluation

In this section, we begin by describing our experimental setup and introducing the applications and methodology employed in our experiments (Sec. 3.1). The detailed discussion of the results can be found in Sec. 3.2.

3.1. Experimental setup and methodology

Our evaluation was performed on a 16-core Intel Xeon Gold 5218 "Cascade Lake" processor, where cores run at 2.3Ghz. All cores feature two private cache levels (64KB L1 + 1MB L2) and share a 22MB last-level (L3) cache.

To assess the effectiveness of our proposal, we experimented with both POSIX threads (aka *pthreads*) and OpenMP programs. Specifically, we considered multi-threaded applications from different benchmark suites, encompassing NAS Parallel Benchmarks [14], PARSEC [15] and Rodinia [16]. In addition, we included `RNASeq` [17], an OpenMP-based RNA sequencing application, and two pthreads-based programs: `BLAST`, a bioinformatics application, and `FFTW3D` [2], a scientific benchmark performing the fast Fourier transform. For the NAS and PARSEC benchmark suites, we employed the `C` and `native` input sets, respectively. Given the exceedingly short execution time of Rodinia benchmarks when using their default command-line settings, we opted to use the alternative input sets employed by previous research [1], [18]. The exact command-line parameters we used can be found in prior work [18].

All OpenMP programs (unmodified) were compiled with a customized GCC 11.2 compiler [1], which contains a simple patch that just alters the default loop-scheduling method from `static` to `runtime`. This allows the user to enforce the desired loop-scheduling method and other parameters (via environment variables) for all parallel loops not including the `schedule` clause; this is the case of the vast majority of the loops in the considered applications. We should also highlight that application loops where the programmer explicitly established the `dynamic`, `guided`, or `runtime` scheduling methods also make use of our malleabillity extensions, which are globally enabled when the `GOMP_MALLEABLE` environment variable is set to 1.

In building the workloads, we first proceeded to identify parallel applications that exhibit execution phases with limited thread-level parallelism (TLP). These applications often leave idle cores, allowing our elasticity manager to opportunistically increase the thread count of other (malleable) programs within the workload, thus maximizing the processor's utilization. To identify applications of this kind, we built a PMCSched plugin that efficiently monitors the amount of time an application spends with different number of runnable (i.e., non-blocked) threads. Fig. 2 shows the fraction over the total execution time that different programs spend with various runnable thread counts. In this experiment, we launched the programs with 8 threads, as this is the thread count used to run non-malleable programs in our workloads. The results reveal that some programs spend more than 40% of its execution time running with just a single runnable thread. This is the case of one OpenMP application (`pathfinder`) and three pthreads-based programs: `blackscholes`, `BLAST` and `FFTW3D`. The limited

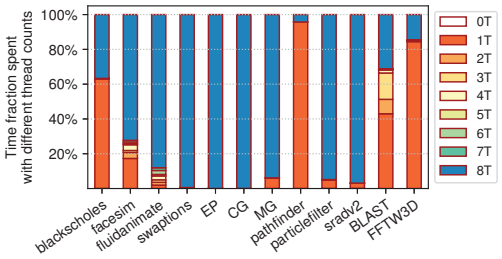


Figure 2: Time fraction spent with different runnable thread counts in various applications.

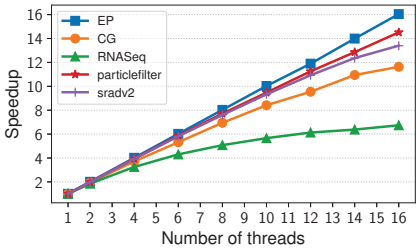


Figure 3: Scalability of several OpenMP programs in our experimental platform.

TLP in this context stems from the presence of long sequential phases (predominantly in initialization code), coarse-grained critical sections, or severe load imbalance among worker threads. By contrast, other applications, like `EP` or `particlefilter`, consistently run with the maximum thread count during most of their execution time.

To find OpenMP programs that potentially benefit from malleability in the presence of idle cores, we ran the various analyzed applications with different thread counts (from 1 to 16). Fig. 3 shows the speedup of a number of OpenMP programs that experience diverse performance improvements as the thread/core count increases.

In constructing the workloads, we selected 12 programs from the full set of benchmarks considered. Among these programs, four exhibit limited TLP (`blackscholes`, `BLAST`, `FFTW3D` and `pathfinder`), while the rest are OpenMP programs that experience significant benefit when increasing their thread counts. Table 1 presents the 24 random program mixes employed in our experiments. Each workload includes a non-malleable (N) program with limited TLP, and a malleable (M) OpenMP one. Both programs are launched with 8 threads each on our 16-core experimental system. Note also that OpenMP programs run with the best-performing loop-scheduling method among those supporting malleability (`guided` or `dynamic`).

We ran each workload in three different scenarios. In the first one, referred to as *Stock-Linux*, no malleability OS extensions are loaded, and applications use the unmodified version of the OpenMP runtime system. Under these circumstances both applications use a fixed thread count – 8 threads each. Moreover, the threads of the N-program are mapped to CPUs 0-7 via user-enforced affinities, while threads of the other application run on the remaining cores (CPUs 8-15). In the second scenario –denoted as *Oversubscription*– we run the M-program with 16 threads, whereas the N-program still uses 8 threads; affinities are only imposed to threads of the N-program, which are pinned to CPUs 0-7. This scenario does not leverage malleability, but exploits the fact that the N-program temporarily leaves idle cores, which can be used by the additional threads (one thread per core) used by the M-program. We considered this scenario to assess (1) how effectively the stock OS scheduler together with the unmodified runtime system deal with idle cores, and (2) what is the impact of leveraging oversubscription in the performance of the non-malleable program. In the last scenario, we enable our kernel-level elasticity manager (EM), and activate the malleabillity support in our modified OpenMP runtime system. No user-enforced CPU affinities are used in this case, and, as opposed to the remaining scenarios, the active thread count of the OpenMP (malleable) application is dynamically adjusted based on the number of idle cores left by the other multithreaded program. Notably, we conducted a sensitivity study (omitted due to space constraints) to determine a suitable choice of the `em_period` and `min_threshold` configurable parameters of EM in our platform. Based on the results, we opted to set these parameters to 100ms and 2ms, respectively.

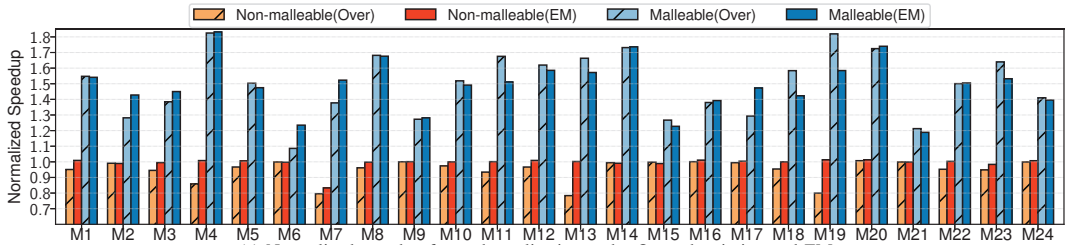
For each workload and scenario we measure the application performance (completion time) and system throughput (STP metric [19]). It is worth highlighting that running each program in the workload just once until completion, provides misleading performance results, especially when the N-program is shorter than the M-program. In that case, the M-program automatically experiences a performance boost under the elasticity manager, simply because threads of this program effectively populate all the platform's cores whe the N-program terminates. For accurate assessment of the performance gains provided by the elasticity manager when both applications run together the whole time, we employ a similar method than in prior work [20], so as to keep the system's load uniform throughout an experiment. To elaborate, we ensure that all applications in the mix commence simultaneously. When one of them finishes its execution, it is consistently restarted until the program with the longest runtime completes three iterations. Then, we measure system throughput based on the geometric mean of the completion times observed for each application.

3.2. Discussion of experimental results

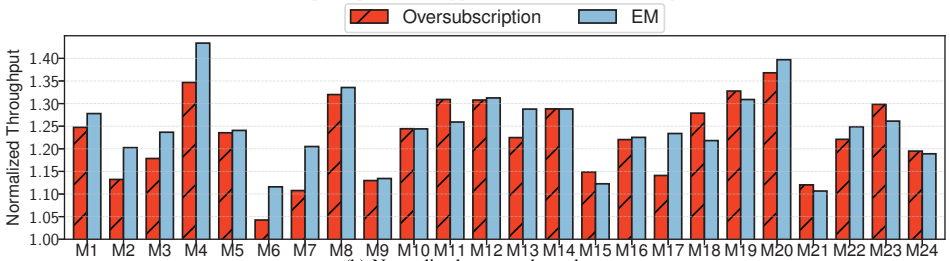
Fig. 4a shows the relative performance that each application in the workload obtains under EM and Oversubscription vs. Stock Linux. Clearly, EM delivers substantial performance gains for M-programs (up to 83%, for workload M4). This is due to the effective utilization of idle cores left by N-programs, all with limited TLP. In most cases this is accomplished without significantly degrading the performance of the N-program. For the vast majority of the workloads, the performance of the non-malleable applications remains within a 1.7% range of that obtained with Stock-Linux. The substantial acceleration of M-programs across the board also leads to a considerable increase in the system throughput,

TABLE 1: Each table column M_i depicts the composition of the i-th workload used in our experiments. The (N) suffix in the application name (see row labels) indicate that the application is non-malleable. Conversely, malleable programs are listed with the (M) suffix. When a program is included in a workload, a black dot is displayed; otherwise, the associated cell is blank.

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13	M14	M15	M16	M17	M18	M19	M20	M21	M22	M23	M24
blackscholes(N)	•				•				•				•				•				•			
BLAST(N)		•				•				•				•				•				•		
FFTW3D(N)			•				•				•				•				•				•	
pathfinder(N)				•				•				•				•				•				•
CG(M)						•										•								
EP(M)				•									•						•					
heartwall(M)					•					•													•	
leukocyte(M)			•									•						•						
particlefilter(M)	•										•									•		•		
RNASeq(M)								•							•						•			
sradv2(M)		•					•										•							
streamcluster(M)									•					•										•



(a) Normalized speedup for each application under Oversubscription and EM



(b) Normalized system throughput

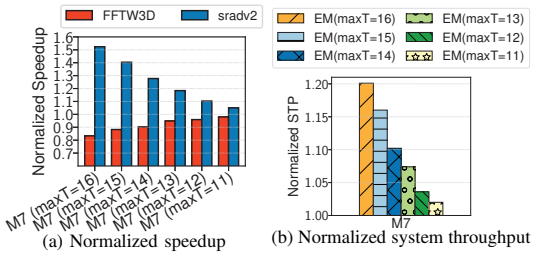
Figure 4: (a)Relative speedup for each application in the workload under Oversubscription and EM vs. Stock-Linux, and (b) normalized throughput for the various program mixes.

as Fig. 4b reveals. Compared to Stock-Linux, EM improves throughput by up to 43%, and by 24.5% on average.

Now we turn our attention to the results under Oversubscription (*Over*), and compare them with the EM counterparts. Running the malleable OpenMP application with 16 threads throughout the execution (what happens under *Over*), makes it possible in many workloads to achieve higher performance for this program w.r.t. EM. Despite the issues and overheads stemming from oversubscription [9], [21], the loop-scheduling methods used in our experiments (dynamic and guided) are key in improving the performance of the M-program. In this context, these methods automatically assign more work to threads running on a dedicated core than to threads time-sharing a core with other threads. This partially mitigates the imbalance arising from oversubscription. Unfortunately, the acceleration of the malleable OpenMP application under *Over* usually comes at the expense of degrading the performance of the N-program. Take for instance the results of M19 workload in Fig. 4a, where a substantial performance gain is accomplished for the M-program in exchange for a 21% performance degradation of the N-program. This stems from the fact that, under *Over*, threads of the N-program often have to time-share a core with threads from the other application; this causes uneven progress among threads[2], thus deteriorating the N-program's performance. All in all, we conclude that leveraging oversubscription may bring additional throughput improvements in some cases (e.g. see results for M8, M11, or M19 in Fig. 4b), but it also negatively (and consistently) impacts the performance of the non-malleable program. Conversely, EM guarantees lower overheads of non-malleable programs. After all, EM strives to avoid oversubscription when exploiting malleability dynamically.

The results provided by EM for the M7 and M23 workloads –both including the `FFTW3D` (non-malleable) application– are also of special attention. Specifically, as shown in Fig. 4a, the acceleration of the malleable partner

2. The N-programs we used do not exploit dynamic load balancing among threads; instead, threads are assigned a fixed amount of work upon creation. Hence, oversubscription gives rise to uneven thread progress.



(a) Normalized speedup (b) Normalized system throughput

Figure 5: Limiting the active thread count of the malleable application of workload M7 under EM.

comes at the expense of noticeable performance degradation of `FFTW3D`, being particularly noteworthy the 16.6% performance penalty in M7. After a thorough analysis of these scenarios, we determined that this degradation is caused by severe bandwidth contention on the platform. Essentially, `FFTW3D` is a highly bandwidth-intensive program, and when it shares the system with another bandwidth-intensive application, the system's bandwidth saturation point is reached. This situation is further aggravated by the effect of the elasticity manager. Granted, increasing the number of threads of a bandwidth-intensive program causes its bandwidth consumption to grow [22], thereby exacerbating the bandwidth contention problem. This especially deteriorates the performance of the non-malleable program, which does not enjoy a dynamic increase in its thread count.

We found that this issue can be addressed by limiting the number of threads assigned to the malleable application by our elasticity manager. As an illustrative example, Fig 5 shows the impact on application performance and system throughput that comes from imposing an upper limit to the number of threads assigned to the malleable application in workload M7, ranging from 16 threads (maxT=16) to 11 threads (maxT=11). Clearly, setting a lower limit on the number of threads utilized by the malleable program (`sradv2`) reduces the performance degradation of the non-malleable one (`FFTW3D`). However, this also comes at the expense of a consistent reduction of both `sradv2`'s performance and the system throughput. Moreover, throughput gains reach a peak value when no capping is applied (16 threads maximum, matching the total core count). Therefore, this experiment underscores that in specific cases a trade-off must be made to reap throughput benefits without deteriorating the performance of non-malleable programs. More importantly, memory bandwidth contention –in addition to the idle core count– should be taken into consideration when leveraging dynamic malleability to avoid performance degradation. Incorporating contention awareness into our elasticity manager constitutes a promising research avenue.

4. Related work

Elasticity is a key feature in the cloud, as it allows providers to offer scalable resources on demand while maximizing profits [8]. Notably, existing CPU elasticity mechanisms operate at higher time frames and coarser granularity than our node-level OS-oriented approach. For example, virtual machines or containers may be migrated to another node in response to a request to increase CPU resources [8], [9], which can take several seconds.

Huang et al. [9] propose an OS-level elasticity strategy that exploits various optimizations in the synchronization primitives. While this approach is also based on rapid scheduling of runnable threads during short idle periods, it focuses on the efficient handling of CPU oversubscription, rather than on the dynamic adaptation of thread counts in malleable programs. Moreover, unlike our proposal, Huang's work [9] does not propose any kind of OS-runtime cooperation method, or explicitly supports malleable applications. Other works not leveraging malleability either [1], [21], [23] acknowledge the importance of the coordination and information exchange between the OS and the application. For example, [21], [23] underscore the potential of making the OS scheduler aware of threads that are busy waiting in synchronization primitives by issuing notifications from the application level. Saez et al. [23] exploit this OS-runtime interaction in an actual OS scheduler implementation. Our proposal could be augmented so as to leverage information on busy waiting threads to improve effective CPU usage.

Malleability has been widely exploited in HPC in a more general form, this is, by performing dynamic resource reallocation across nodes, including adjustments in the number of worker processes/threads, and the necessary data redistribution [24]. By contrast, we looked at a more specific type of malleability that leverages active thread adjustments in a single node and is orchestrated by the OS. While our proposal focuses on describing the OS-runtime interaction and uses malleable OpenMP parallel for work-sharing constructs as a proof of concept, the OpenMP tasking model also lends itself (even more) to dynamic thread count adjustments. A wide range of techniques that leverage malleability, such as free-agent threads [25]–[27], have been proposed for OpenMP, most of which operate entirely at the runtime system level or in the cluster/job scheduler (e.g., Slurm) [5], [28]. These techniques could also benefit from OS-runtime cooperation and are exciting avenues for future research.

In elastic OpenMP [29], authors propose a mechanism to provide elasticity support for OpenMP applications that make the dynamic provisioning of cloud resources possible. This proposal supports the dynamic adjustment of resources (vCPUs) and a set of additional routines to enable the configuration of the elastic execution. Our OpenMP prototype has also adapted the inner workings of the parallel construct, but the problems we have addressed are orthogonal.

Other works also applied dynamic thread adjustment under colocation. Cho et al. introduced NuPoCo [6], a runtime-system level solution that strives to efficiently utilize both the CPU and memory controller when multiple parallel applications share the system. Our proposal conversely places its emphasis on the exploitation of idle cycles left by unmodified applications through elasticity support within the OS. More importantly, unlike our approach, NuPoCo requires that every application in the workload use a customized runtime system, thus being incompatible with legacy single-

threaded and multithreaded software.

5. Conclusions and Future Work

In this work, we have proposed an OS-level elasticity mechanism that strives to maximize CPU utilization and system throughput, when running workloads consisting of a mix of non-malleable programs and malleable applications (i.e., with the ability to vary their thread count). To accomplish its goals, our proposal leverages idle cores left by non-malleable applications (even for short periods of time), and exploits synergistic cooperation between the OS kernel and the runtime system for the dynamic adjustment of active threads in malleable applications.

To demonstrate the effectiveness of the proposed OS extension, we have implemented it in the Linux kernel. The associated OS-level support is bundled as a loadable kernel module that can be loaded in unmodified (unpatched) Linux kernels. We also augmented the GNU OpenMP runtime system to provide a proof-of-concept malleability support at the runtime system level that exploits the OS-runtime interaction required by our proposal. This malleability support can be utilized by a wide range of unmodified loop-based parallel applications. Our experiments reveal that our proposal delivers improved system throughput, while having minimal impact in the performance of non-malleable programs. This is accomplished thanks to the opportunistic utilization of idle cores to accelerate malleable applications.

As for future work, we plan on augmenting our proposal to explicitly deal with contention on shared resources among cores (e.g., shared last-level-cache). Providing the necessary support to deliver benefits to applications beyond compute-intensive applications and loop-based OpenMP programs is also an interesting avenue for future work.

Acknowledgments

This work has been supported by the Spanish MCIN under Grant PID2021-126576NB-I00, funded also by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe".

References

[1] C. Bilbao, J. C. Saez, and M. Prieto-Matias, "Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake," *Concurrency and Computation: Practice and Experience*, p. e7814, 2023.

[2] M. Annavaram *et al.*, "Mitigating Amdahl's Law through EPI Throttling," in *Proc. of ISCA '05*, 2005, pp. 298–309.

[3] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps," in *Proc. of ASPLOS '08*, 2008, p. 277–286.

[4] F. V. Zacarias *et al.*, "Intelligent colocation of HPC workloads," *Journal of Parallel and Distributed Computing*, vol. 151, pp. 125–137, 2021.

[5] D. Álvarez, K. Sala, and V. Beltran, "nos-v: Co-executing hpc applications using system-wide task scheduling," https://arxiv.org/abs/2204.10768, 2022.

[6] Y. Cho, C. A. C. Guzman, and B. Egger, "Maximizing system utilization via parallelism management for co-located parallel applications," in *PACT*, ser. In Proc. of PACT '18, 2018.

[7] T. Harris, M. Maas, and V. J. Marathe, "Callisto: Co-scheduling parallel runtime systems," in *In Proc. of EuroSys '14*, 2014.

[8] Y. Al-Dhuraibi *et al.*, "Elasticity in cloud computing: State of the art and research challenges," *IEEE Transactions on Services Computing*, vol. 11, no. 2, pp. 430–447, 2018.

[9] H. Huang *et al.*, "Towards exploiting cpu elasticity via efficient thread oversubscription," ser. In Proc. of HPDC'21, 2021, p. 215–226.

[10] G. Galante and L. C. Erpen De Bona, "A programming-level approach for elasticizing parallel scientific applications," *Journal of Systems and Software*, vol. 110, pp. 239–252, 2015.

[11] J. Mauro and R. McDougall, *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Prentice Hall, 2006.

[12] C. Bilbao and J. C. Saez, "PMCSched website," https://github.com/Zildj1an/pmcsched-website, GitHub, 2022, accessed: 2023-10-23.

[13] J. Park *et al.*, "Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers," in *Proc. of PACT '18*, 2018, pp. 5:1–5:14.

[14] D. H. Bailey, E. Barszcz, and J. T. Barton et al., "The NAS parallel benchmarks—summary and preliminary results," in *Supercomputing '91*, 1991, pp. 158–165.

[15] C. Bienia *et al.*, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proc. of PACT'08*, 2008.

[16] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc of IISWC '09*, 2009, pp. 44–54.

[17] H. Chitsaz *et al.*, "A partition function algorithm for interacting nucleic acid strands," *Bioinformatics*, vol. 25, no. 12, pp. i365–i373, 2009.

[18] T. Suzuki, A. Nukada, and S. Matsuoka, "Efficient execution of multiple cuda applications using transparent suspend, resume and migration," in *Proc. of Euro-Par '15*, 2015.

[19] S. Eyerman and L. Eeckhout, "System-level performance metrics for multiprogram workloads," *IEEE Micro*, vol. 28, pp. 42–53, 2008.

[20] D. Shelepov *et al.*, "HASS: a Scheduler for Heterogeneous Multicore Systems," *Oper. Syst. Review*, vol. 43, no. 2, pp. 66–75, 2009.

[21] J. H. M. Korndörfer *et al.*, "How do os and application schedulers interact? an investigation with multithreaded applications," in *Proc. of Euro-Par '23*, 2023, pp. 214–228.

[22] M. A. Suleman, M. K. Qureshi, and Y. N. Patt, "Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs," in *Proc. of ASPLOS '08*, 2008, p. 277–286.

[23] J. C. Saez *et al.*, "Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors," in *Proc. of CF' 10*, 2010, pp. 31–40.

[24] J. I. Aliaga *et al.*, "A survey on malleability solutions for high-performance distributed computing," *Applied Sciences*, no. 10, 2022.

[25] J. Criado *et al.*, "Exploiting openmp malleability with free agent threads and dlb," in *ISC High Performance 2022 International Workshops*, 2022.

[26] ——, "Role-shifting threads: Increasing openmp malleability to address load imbalance at mpi and openmp," *The International Journal of High Performance Computing Applications (In press)*, 2023.

[27] OpenMP Architecture Review Board, "Openmp technical report 12: Version 6.0 preview 2," https://www.openmp.org/wp-content/uploads/openmp-TR12.pdf, 2023, Accessed: 2023-11-12.

[28] M. D'Amico *et al.*, "Drom: Enabling efficient and effortless malleability for resource managers," in *Proc. of ICPP Workshops '18*, 2018.

[29] G. Galante and R. da Rosa Righi, "Adaptive parallel applications: From shared memory architectures to fog computing (2002-2022)," *Cluster Computing*, vol. 25, no. 6, p. 4439–4461, 2022.

Chapter 6

Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake

Full citation

Bilbao C, Saez JC, Prieto-Matias M. *Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake.* Concurrency Computat Pract Exper. 2023; 35(25):e7814. doi: 10.1002/cpe.7814

Impact metrics:

JCR 2023, Impact Factor: 1.5, Q2 in Computer Science, Theory & Methods.

Abstract

Asymmetric multicore processors (AMPs) couple high-performance big cores and power-efficient small ones, all exposing a shared instruction set architecture to software, but with different microarchitectural features. The energy efficiency benefits of AMPs, together with the general-purpose nature of the various cores, have led hardware manufacturers to build commercial AMP-based products, first for the mobile and embedded domains, and more recently, for the desktop market segment, as with the Intel Alder Lake processor family. This trend indicates that AMPs may become a solid and more energy efficient replacement for symmetric multicores in a wide range of application domains. Previous research has demonstrated that the system software can substantially improve scheduling – critical to get the most out

of heterogeneous cores – by leveraging hardware facilities that are directly managed by the OS, such as performance monitoring counters, or the recently introduced Intel Thread Director technology. Unfortunately, the OS-level support enabling access to these hardware facilities may often take a long time to be adopted in operating systems, or may come in forms that make its utilization challenging from specific levels of the system software stack, especially in production systems. To fill this gap, we propose PMCSched, an open-source framework enabling rapid development and evaluation of custom scheduling-related support in the Linux kernel. PMCSched greatly simplifies the design and implementation of a wide range of scheduling policies for multicore systems that operate at different system software layers without requiring to patch the kernel. To demonstrate the potential of our framework, we conduct a set of experimental case studies on asymmetry-aware scheduling for Intel Alder Lake processors.

Received: 29 December 2022 | Revised: 30 March 2023 | Accepted: 12 May 2023

DOI: 10.1002/cpe.7814

SPECIAL ISSUE PAPER

WILEY

Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake

Carlos Bilbao | Juan Carlos Saez | Manuel Prieto-Matias

Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain

Correspondence
Juan Carlos Saez, Facultad de Informática, Universidad Complutense de Madrid, Madrid, Spain.
Email: jcsaezal@ucm.es

Funding information
Comunidad de Madrid, Grant/Award Number: S2018/TCS-4423; European Regional Development Fund; Ministerio de Ciencia, Innovación y Universidades, Grant/Award Number: PID2021-126576NB-I00

Summary

Asymmetric multicore processors (AMPs) couple high-performance big cores and power-efficient small ones, all exposing a shared instruction set architecture to software, but with different microarchitectural features. The energy efficiency benefits of AMPs, together with the general-purpose nature of the various cores, have led hardware manufacturers to build commercial AMP-based products, first for the mobile and embedded domains, and more recently, for the desktop market segment, as with the Intel Alder Lake processor family. This trend indicates that AMPs may become a solid and more energy efficient replacement for symmetric multicores in a wide range of application domains. Previous research has demonstrated that the system software can substantially improve scheduling—critical to get the most out of heterogeneous cores—by leveraging hardware facilities that are directly managed by the OS, such as performance monitoring counters, or the recently introduced Intel Thread Director technology. Unfortunately, the OS-level support enabling access to these hardware facilities may often take a long time to be adopted in operating systems, or may come in forms that make its utilization challenging from specific levels of the system software stack, especially in production systems. To fill this gap, we propose PMCSched, an open-source framework enabling rapid development and evaluation of custom scheduling-related support in the Linux kernel. PMCSched greatly simplifies the design and implementation of a wide range of scheduling policies for multicore systems that operate at different system software layers without requiring to patch the kernel. To demonstrate the potential of our framework, we conduct a set of experimental case studies on asymmetry-aware scheduling for Intel Alder Lake processors.

KEYWORDS

asymmetric multicore processors, Intel Alder Lake, Intel Thread Director, Linux kernel, loop scheduling, OpenMP, operating systems, runtime systems, scheduling

1 | INTRODUCTION

Energy efficiency has become one of the most critical constraints of processor design.[1] The quest for improved efficiency has substantially contributed to the proliferation of heterogeneous architectures that combine different types of cores and processing units within the same platform.[2] Asymmetric multicore processors (AMPs) constitute an attractive type of heterogeneous architecture where high-performance big cores and

power-efficient small ones—all exposing a shared ISA (instruction set architecture)—are combined on the same system. The common ISA, in conjunction with the general-purpose nature of the AMP cores, allows the execution of legacy (unmodified) software. These facts, along with AMPs' energy efficiency benefits, have drawn the attention of major hardware players, leading to the massive release of commercial AMP products for mobile platforms, such as those based on the ARM big.LITTLE processor.[3] Today, the Intel Alder Lake processor family and the Apple M1 SoC, are clear examples of the expansion of AMPs toward the desktop market segment.[4] The increase in the number of cores in later generations of these Intel-based AMPs processors[5] makes them even more attractive for the execution of parallel code and multi-application workloads.

Despite the remarkable benefits of AMPs,[6] effectively scheduling diverse programs/tasks on heterogeneous cores poses a significant challenge to the various system software layers.[2,7-10] When a single multithreaded application runs alone on an AMP system, smart user-level scheduling within the runtime system is the key to making the most out of its heterogeneous cores.[7,11] However, in multi-application scenarios, and especially under the presence of legacy programs, the OS scheduler plays an essential role in transparently delivering the benefits of AMPs to the end user.[2,9,12,13] In our work, we focus on designing and implementing effective OS and runtime-level schedulers to deal with single- and multi-application scenarios, by leveraging smart interaction between system software layers when possible. Regarding parallel applications, we focus on the acceleration of unmodified loop-based OpenMP programs. Many of these applications are broadly used in scientific and engineering domains, and constitute the dominant application type in many parallel benchmark suites extensively used for performance evaluation on symmetric and asymmetric multicore systems.[7,11,14-19] Some of our insights on effective OS-runtime interaction techniques could be leveraged for other types of programs, such as the increasingly popular task-based parallel applications, where runtime scheduling has also drawn special attention.[10,20-22]

Previous research has demonstrated that the runtime system and the OS scheduler can perform optimizations on AMPs by leveraging hardware features that are directly controlled by the OS kernel and exposed to user space, such as performance monitoring counters (PMCs)[4,12,23] or dynamic voltage and frequency scaling (DVFS).[24,25] Often, the support to conveniently access new scheduling-relevant hardware features from the system software may take time to be adopted in operating systems,[4] or it may come in the form of architecture-specific interfaces that limit application portability or make its utilization impossible from particular levels of the software stack.[26,27] Take for instance the Linux kernel, which does not currently feature official support for the Intel Thread Director (TD) technology,[4] unlike the proprietary Windows 11 kernel. TD is a set of scheduling-related hardware facilities—introduced with Alder Lake processors—that provide the system software with performance and energy efficiency hints to aid in carrying out effective thread-to-core mappings on Intel AMPs. We provide an in-depth description of TD in Section 4. Implementing custom mechanisms in the OS kernel to leverage these new—yet unsupported—features directly from the OS scheduler, or exposing them to user space involves a substantial development effort, due to the inherent challenges associated with kernel-level programming.[26,28] At the same time, custom OS-level extensions could be difficult to be adopted in production systems, where patching the OS kernel may be impractical.

To address these issues, we propose PMCSched, an open-source framework for the Linux kernel that enables rapid development of the OS-level support required to create custom scheduling and resource-management schemes on both symmetric and asymmetric multicore systems. Unlike other existing frameworks that require patching the Linux kernel to function,[28-31] PMCSched makes it possible to incorporate new scheduling-related OS-level support in Linux via a kernel module that can be loaded in unmodified kernels, making its adoption easier in production systems. Notably, the main focus of this framework is to simplify the creation of novel scheduling and resource-management strategies that are either implemented entirely in the OS kernel, or require changes in different layers of the system software, so as to benefit from coordinated decisions between the runtime system and the OS scheduler.[2,32,33]

To demonstrate the effectiveness and flexibility of our Linux-based framework we conduct several experimental case studies on an AMP system featuring an Intel Core i9-12900K "Alder Lake" processor. Apart from our proposed framework, we make the following contributions:

- We carry out a brief survey on the current status of Linux's support for Intel TD. This survey also describes TD's inner workings based on the scarce related literature, and provides key experimental data gathered with PMCSched that showcases important limitations of this scheduling-related technology on Intel Alder Lake platforms.
- We propose flexible asymmetric-iteration distribution (AID), a set of enhancements for improved AID[11] of parallel loops in OpenMP programs. Flexible AID consists of a set of loop-scheduling methods that leverage awareness on the platform's asymmetry to improve performance, and exploit explicit OS-runtime interaction for coordinated scheduling decisions between both system software layers. We implement flexible AID in GCC v11.2, and evaluate its effectiveness using a broad collection of unmodified loop-based OpenMP applications.
- By leveraging PMCSched, we implement HSP,[34] an OS scheduler that strives to maximize system throughput on AMPs. This constitutes the first OS-level implementation of HSP in Linux that does not require source code changes in the kernel itself, but instead can be dynamically loaded on top of an unmodified Linux kernel. We created several variants of this scheduler to guide thread-to-core mappings dynamically by using either TD hardware or PMCs from kernel space. To assess the effectiveness of our HSP implementation we considered a wide range of multi-application workloads, some of them including unmodified OpenMP applications that leverage synergistic OS-runtime interaction via flexible AID.
- For the sake of completeness, we also conduct an experimental evaluation of the TD scheduling-related patches recently submitted by Intel for potential inclusion in Linux kernel's *mainline*.[35] The analysis of the associated kernel code (described in Section 4.2) was integral to our efforts, as it enabled us to thoroughly examine the impact of TD on our Alder Lake based system along with Intel's proposed OS support.

The remainder of the article is organized as follows. Section 2 discusses related work. Section 3 provides an overview of PMCSched's design, and introduces its main implementation challenges. We offer an in-depth analysis of the Intel Thread Director technology and the observed limitations in Section 4. Section 5 revisits AID, and showcases the key features and novelties of our flexible AID proposal. Section 6 covers the experimental case studies on asymmetry-aware scheduling for Alder Lake processors, including OS scheduling alone, parallel-loop scheduling, and OS-runtime interaction. Lastly, Section 7 concludes the article.

2 | RELATED WORK

A large body of work has proposed asymmetry-aware scheduling strategies for adoption on either runtime systems[9,10,20] or OS kernels.[2,4,12] Frequently, such endeavors culminate in tools and frameworks that aim to ease the development and analysis of new scheduling algorithms; these are likewise some of the main goals of this article.

Recent studies have shown that scheduling algorithms that come in stock general-purpose OSs exhibit suboptimal behavior for different workloads on a wide range of processor architectures.[2,7,36] At the same time, making the required changes in an OS kernel to build effective scheduling policies specifically tailored to custom workloads or microarchitectures may be a significant burden to the average developer.[28,31] On monolithic kernels such as Linux, developing a new scheduling policy constitutes a labor-intensive task, as the kernel itself needs to be modified. More specifically, testing any scheduling-related kernel modification requires compiling and reinstalling the kernel, and finally rebooting the machine for the changes to take effect. Testing an individual change in this way may as well take a full coffee break, depending on the features and resources of the target platform and the development host.

To overcome these problems, some researchers have resorted to evaluating their proposed OS-level schedulers via simplistic user space prototypes.[24,37,38] Even though this approach may allow to draw interesting insights and also benefit from leveraging application-level metrics, strategies implemented in this way suffer from the limitations imposed to userland, such as the additional overhead of context switches and extra system calls required for dynamic thread affinity and performance monitoring,[28] or the inability to react to low-level scheduling-related events quickly (e.g., a thread blocks due to I/O or a page fault), thus wasting CPU cycles.[26] In addition, user-level scheduling prototypes cannot access hardware extensions not currently exposed by the OS kernel.

Other scheduling frameworks,[28,29,31] such as PMCSched itself, aim to overcome some of the aforementioned limitations. LUSH[31] permits the creation of user-level schedulers for AMPs without special execution privileges, and introduces kernel-level changes to allow fine-grained access to PMCs from user space. Mvondo et al.[28] propose extending existing operating system APIs to allow the development of kernel-level schedulers programmable from user space in a safe and controlled environment. The LITMUS project[29] constitutes a substantial fork of the Linux kernel, with extensions to facilitate the prototyping of real-time kernel-level scheduling algorithms. Contrary to such solutions—some of them restricted to specific domains[29,31]—PMCSched allows to create custom scheduling-related OS-level modifications without actually patching the kernel.

We should underscore that other OS-oriented projects such as LITMUS[29,39] or Popcorn Linux[40,41] require profound modifications of the Linux kernel. Unfortunately, getting such kernel modifications accepted upstream can be a difficult and lengthy process, particularly for research-oriented tools. As a result, many researchers treat their software as research prototypes with no real hope of production integration in sight,[28] even after conducting necessary security audits. Rather than attempting to get their modifications accepted upstream, these projects maintain multiple forks for specific releases of the Linux kernel. However, this approach can shorten the lifespan of the project,[42] as it becomes challenging to keep up with the rapid development pace of the kernel. For instance, the latest kernels used by the LITMUS[39] and Popcorn Linux[41] projects are forks based on stable Linux v4.9.30 (May 2017) and v5.2.21 (October 2019), respectively, whereas the latest stable Linux kernel at the time of writing is v6.2 (PMCSched does support this version). In contrast, PMCSched takes a different approach to quickly adapt to new kernel versions, by relying on two main aspects that avoid patching the kernel: (1) bundling its full functionality in a loadable kernel module, and (2) leveraging standard tracing technologies available in recent kernels. We elaborate on these aspects of PMCSched in Section 3.

Other studies explore the challenges of OS scheduling on highly heterogeneous architectures.[9] Of special attention is the case of Popcorn Linux,[40] which targets heterogeneous systems consisting of nodes with different ISAs, opening the door to parallel ISA-heterogeneous OpenMP runtime scheduling.[43] These efforts are orthogonal to ours, formulating a problem with several interconnected computing nodes with different processor architectures (e.g., x86 and ARM).

3 | PMCSCHED: IMPLEMENTATION CHALLENGES AND DESIGN

3.1 | Motivation and challenges

PMCSched is implemented on top of PMCTrack, a performance monitoring tool[44] for Linux that was open sourced back in 2015.[45] Unlike Perf Events[42]—the default Linux subsystem to access hardware facilities, such as PMCs—PMCTrack was not primarily designed to only expose hardware

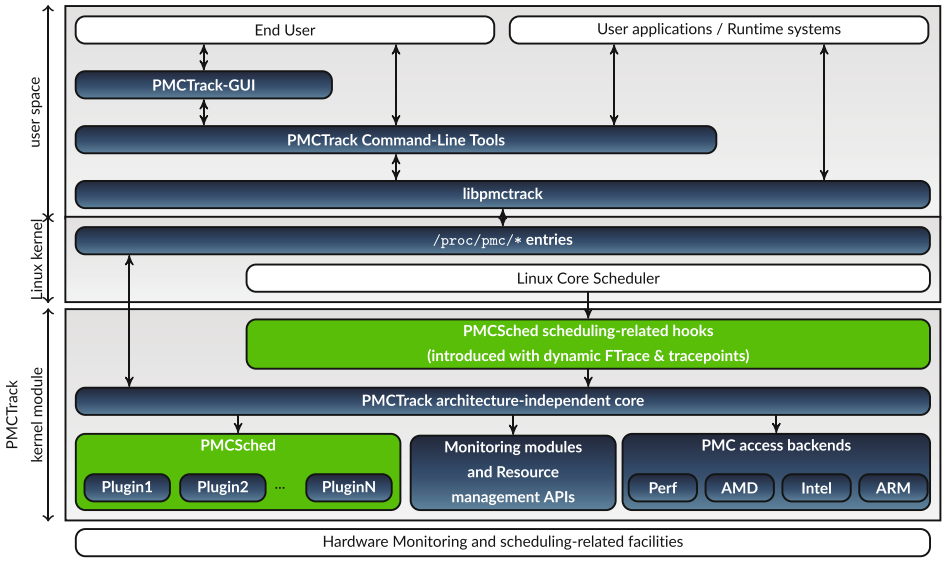


FIGURE 1 PMCSched components (in green) inside PMCTrack's architecture.

monitoring facilities to user space, but to assist the system software when performing runtime optimizations based on these hardware facilities. The operations for which the system software can benefit from PMCTrack include scheduling[2,34] and resource management.[26] The main advantages of relying on PMCTrack for such tasks are its ability to foster new OS-level features as part of an extensible loadable kernel module, and its lightweight architecture-independent API to access PMCs within the kernel on a wide range of processor architectures (x86, ARMv7, ARMv8, etc.). Figure 1 depicts the various components of PMCTrack and their relationship, described in detail in our earlier work.[44]

With the PMCSched framework we take PMCTrack's potential one step further by enabling rapid development of OS support for scheduling and resource management for Linux *within a loadable kernel module*. We need to highlight this because hitherto new scheduling policies could not be implemented as a kernel module,[28,30] since no specific API exists for that purpose within the Linux scheduler. When creating novel OS-level schedulers for Linux without modifying the kernel, three main challenges have repeatedly appeared: (1) the inability to execute code in a kernel module in immediate response to the occurrence of a set of key scheduling-relevant events—context switches, process creation or destruction and so forth—(2) the lack of a standardized method to seamlessly extend the Linux task structure with new per-thread scheduling related fields that custom schedulers typically require to function, and (3) how to efficiently customize the behavior of the Linux load balancer. Notably, the first two barriers also arise when attempting to manage performance counters at the low level, and for that reason, most PMC tools require changes in the kernel; PMCTrack adds the associated functionality via a small portable kernel patch.[44]

3.2 | Design and implementation of PMCSched

PMCSched addresses the three aforementioned issues without patching the kernel as follows. First, to be aware of key scheduling events from a kernel module, PMCSched installs scheduling-related hooks (callbacks) by leveraging two modern tracing facilities of the Linux kernel: *dynamic ftrace*[46] and *tracepoints*.[47] These two tracing technologies rely on dynamic and static kernel instrumentation, respectively. Noticeably, both are supported on a wide range of processor architectures, and can be found enabled by default on the most popular Linux distributions. Unlike other kernel instrumentation facilities (like Kprobes), these technologies make it possible for a module to be notified when a kernel function is invoked or when a static tracepoint is reached with virtually no overhead.[46,47] Not only do PMCSched hooks—depicted in Figure 1—enable the implementation of custom schedulers in a kernel module, but they also allow us to eliminate the need for the PMCTrack kernel patch entirely.[48] We should highlight that although *dynamic ftrace*[46] was introduced in Linux several years ago, its implementation has not been robust enough to adopt our scheduling-related hooks till Linux stable version v5.9.[49] Essentially, to accomplish its goals, PMCSched must be fully aware of context switches, being notified right before any process is scheduled out from its CPU (*switch out*), and also when a process is scheduled in a CPU (*switch in*). Unfortunately, a number of ftrace-related bugs[50,51] made it impossible to fully capture all switch-in events; the corresponding bugfixes were incorporated into the aforementioned stable kernel. For legacy platforms not officially supporting kernel versions newer than v5.8.y (such as the ARM-based Odroid-XU4 board[11,52]), PMCSched can still be used, but requires a patched Linux kernel that either includes the necessary ftrace (backported) bugfixes, or incorporates the changes of the old PMCTrack patch.

Second, PMCSched provides a seamless mechanism to extend the task structure with new thread-specific data without modifying the kernel. To this end, whenever a thread enters the system, PMCSched associates a *dummy* software event from the Perf Events subsystem with the thread, by inserting the event into the event list present in Linux's task structure (`perf_event_list` field). The structure of this dummy event (`struct perf_event`) contains a void pointer field (`pmu_private`) that can be utilized to point to any other structure. To simplify the integration of PMCSched in PMCTrack, we use the event's void pointer to point to PMCTrack's per-thread structure (`pmon_prof_t`). PMCSched scheduling fields can be seamlessly added without modifying the kernel, by extending the structures definition inside PMCTrack's kernel module sources.

To make it possible to implement custom load balancing policies, PMCSched introduces the *core group* abstraction. Essentially, cores in the system are organized into different sets (or *core groups*) based on their type (for AMP systems) and their hierarchical relationship in the platform's topology (e.g., cores sharing a last-level cache, or part of the same NUMA node). PMCSched automatically divides cores into different core groups based on system topology, but considering a configurable granularity (LLC, socket or NUMA domain). To implement custom and scalable OS-level load balancing policies or perform specific thread-to-core mappings, a scheduler implemented in PMCSched must assign threads to specific core-groups by using affinity masks. In using this approach, enforcing load balancing across cores within the same group is up to the Linux load balancer, which respects affinity masks. We should also highlight that PMCSched associates a set of linked lists to each core group, making it possible to keep track of active threads or multithreaded processes associated with each core group. This design approach allows to make scheduling decisions independently for threads assigned to different core groups, and favors scalable designs that reduce contention in accesses to core-group specific data structures.

A new scheduling or resource management algorithm can be implemented by creating a *scheduling plugin*, which—as illustrated in Figure 1—becomes a part of the PMCSched subsystem within PMCTrack's kernel module. Building a plugin boils down to instantiating an interface of scheduling operations and implementing the corresponding interface functions in a separate $\varepsilon.c\varepsilon$ file within the module sources. The various algorithm-specific operations are invoked from the core part of the scheduling framework when a key scheduling-related event occurs, such as when a thread enters the system, terminates, becomes runnable/non-runnable, or when tick processing is due to update statistics. The framework also provides a set of callbacks to carry out periodic scheduling activations from interrupt context (timer) and process context (kernel thread) on each core group separately, thus making it possible to invoke a wide range of blocking and non-blocking scheduling-related kernel API calls, such as those to map a thread to a specific CPU or core group. This modular approach to creating scheduling algorithms resembles the one used by *scheduling classes* (algorithms) inside the Linux kernel, but with a striking advantage: PMCSched scheduling plugins can be bundled in a kernel module that can be loaded on unmodified kernels. Moreover, plugin developers have access to a rich set of APIs available in PMCTrack, empowering them to configure performance counters seamlessly and retrieve PMC values in a per-thread fashion, to gather data from other hardware monitoring features,[4,44] or to govern hardware facilities for shared-resource contention mitigation (e.g., LLC partitioning) available on Intel and AMD processors.[26,53]

PMCSched opens the door to a wide range of opportunities that involve rapid prototyping of OS scheduling support. We are committed to making PMCSched accessible to all developers and, to this end, we provide online resources with detailed instructions on how to use PMCSched on its official website.[54] Due to the large amount of technical details already discussed in this article, we opted to leave on its website the specifics regarding PMCSched API for plugin creation and utilization, as well as source code examples on how to leverage its ability to gather runtime PMC metrics. In its current version, PMCSched is primarily a research tool. To be used in production systems, the entire PMCTrack tool requires to undergo an exhaustive security audit. However, we plan to make this the case in the near future. Among other design improvements, we are already working on the support for creating scheduling plugins using the Rust programming language. This will provide developers with a more secure alternative to C thanks to Rust's ground-breaking memory safety and synchronization guarantees, and will benefit from the recent inclusion of Rust into the Linux kernel.[55]

4 | A BRIEF SURVEY ON THE INTEL THREAD DIRECTOR (TD) TECHNOLOGY

This section begins by presenting our experimental platform. Then it describes the Intel Thread Director technology, and its specifics on Intel Alder Lake processors. The section concludes with a brief analysis on TD conducted in our experimental platform.

4.1 | Experimental platform

In this work, we experimented with an Intel Core i9-12900K "Alder Lake" processor, which combines 8 "Golden Cove" big cores, and 8 "Gracemont" small cores. Intel refers to big cores as P-cores (for performance) and to small cores as E-cores (for efficiency). Figure 2 shows the topology of our Alder Lake based platform. As it is evident, E-cores (cores 8-15) are grouped into two 4-core clusters, where each group shares a 2 MB L2 cache. By contrast, each P-core (cores 0-7) has a private 1.25 MB L2 cache. All platform cores integrate a private L1 cache, but share a 30 MB L3 (LLC) with the remaining E and P cores. For our experimental analysis we employed Debian GNU/Linux 11 along with the Linux kernel v5.16.9. Specifically, for the experiments leveraging PMCSched, we used the *vanilla* (unmodified) kernel.

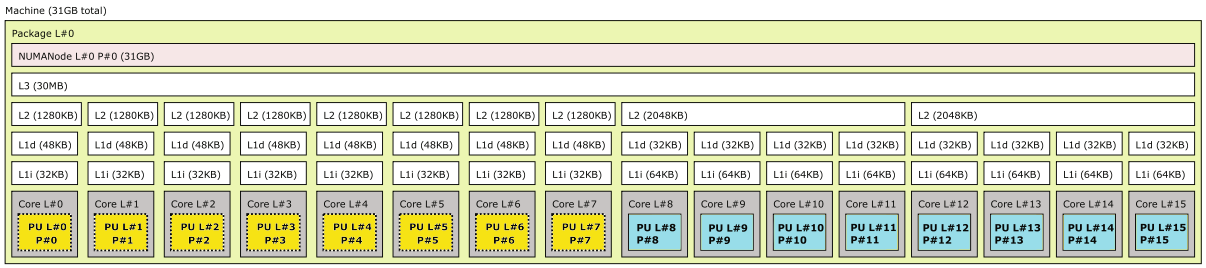


FIGURE 2 The system employed in our experiments. The eight E-cores (right side in blue color) are grouped into two 4-core clusters, sharing a 2 MB L2 cache. P-cores (left side in yellow color) have a private 1.25 MB L2 cache. All E-cores and P-cores share a 30 MB L3 (LLC).

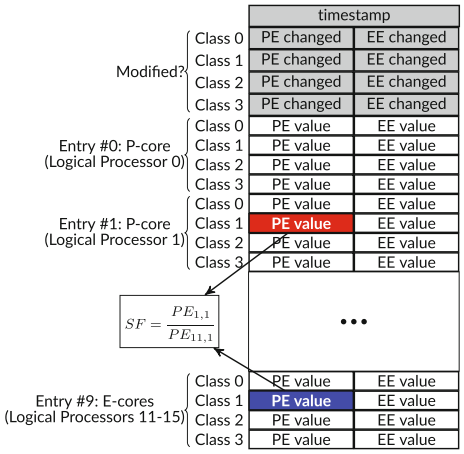


FIGURE 3 Structure of the Thread Director table on our Alder Lake experimental platform.

4.2 | TD on Intel Alder Lake processors

Intel TD is a set of hardware extensions designed to assist the OS in making thread scheduling decisions on Intel hybrid multicore processors.[13,56] TD-enabled hardware allows the interaction with the OS kernel via a set of privileged model-specific registers (MSRs), and a memory-resident table populated by the hardware that exposes key information on the performance and energy-efficiency characteristics of the various cores. This table, referred to as the TD table, is only directly accessible from the OS kernel, which must map it to its virtual address space for proper utilization.

As a thread runs on a given logical processor (LP), the TD hardware continuously collects runtime statistics for the thread, and assigns it a *TD class*, which may change over time. The hardware supports a fixed number of classes (depending on the processor model), where each class is identified with an ID ranging between 0 and the total number of classes minus one. To read a thread's class ID on the current LP, the OS must periodically read the `IA32_THREAD_FEEDBACK_CHAR` MSR. The entries of the TD table associated with that particular class ID provide the OS with a performance and an energy-efficiency score on the various LPs. These scores range from 0 to 255, and can be effectively leveraged from the system software to map the various threads to the most appropriate core type to optimize different target metrics (e.g., system throughput).

Figure 3 depicts the specific structure of the TD table found in our 16-core "Alder Lake" processor. This processor supports 4 TD classes (0-3). The TD table is composed of a global header—which features a timestamp and flags to indicate recent updates made by the hardware to the TD table–, and a set of feedback entries for different LPs that contain the aforementioned performance (*PE*) and energy-efficiency (*EE*) values (scores) for each class ID. In our platform, the TD table has ten 8-byte feedback entries (1 byte for each score type and class). Each big P-core has its own entry (the first eight entries), while each group of four small E-cores sharing an L2 cache also share a common feedback entry (making the last two entries). Our platform's TD table is populated when the OS first configures TD hardware, and remains unaltered after that. More information on the structure of the TD table can be found in the corresponding Intel manual.[56]

In this work, we focus on solutions for maximizing the system throughput automatically from the system software without requiring changes in the applications. In this context, scheduling must be done by taking into account the performance benefit that each thread/task in the workload

TABLE 1 Performance of big and small cores of our system, and classes' assigned thresholds for speedup factor.

	Performance score		
Class	Big core	Small core	Speedup factor
0	65	39	1.67
1	77	39	1.97
2	102	39	2.62
3	51	39	1.31

derives when it runs on a big core, relative to a small one. Henceforth we will refer to this relative benefit as the thread's speedup factor (SF). Using TD, a thread's current SF can be approximated by dividing the P-core and E-core *PE* values for the thread's observed class ID (as reported by the hardware) from the TD table. For instance, as indicated in Figure 3 if a thread runs on logical processor 1 (P-core) on our system, and its current class ID is 1, the SF can be computed with the ratio of the performance scores of the current LP and that of any of the E-cores for class 1. More specifically, Table 1 shows the performance scores for big and small cores found at the TD table of the processor we used, and the associated SF value for each class.

It is important to point out that reading the `IA32_THREAD_FEEDBACK_CHAR` register may not always yield a valid class ID, as the hardware may be unable to generate sufficient telemetry data in order to accurately determine it.[56] Specifically, the lower byte of the register, which encodes the class ID, is only considered valid when bit 63 of the same register (the "valid bit") is set to one. If the valid bit is zero, then the retrieved class ID should be disregarded, and the most recent valid class ID obtained should be used for scheduling decisions instead.[56]

In our earlier work[4] we conducted a preliminary analysis of the Thread Director technology on the same platform. To the best of our knowledge, it was the first experimental analysis of this technology on the Linux kernel, which still does not officially feature Thread Director scheduling support in *mainline* to this date; the latest stable kernel at the time of this writing is v6.2. Using the scarce technical information available on the TD technology at the time of our earlier work[4,56,57], we implemented basic support in Linux to carry out our evaluation. We found that a significant shortcoming of TD hardware on our platform is that *while the* `IA32_THREAD_FEEDBACK_CHAR` *register does report a valid class ID on P-cores most of the time, it always returns invalid class IDs for any thread running on E-cores*. This poses a major limiting factor for the effectiveness of TD-based schedulers, as we demonstrate in Section 6.1.

Only recently, Intel has disclosed key information on the inner workings of TD on Intel Alder Lake processors,[35,58] which have made it possible to corroborate our early findings on the effectiveness of this technology.[4,59] First, we have found that the recent set of TD-related Linux kernel patches submitted by Intel for review and potential inclusion in *mainline*,[35] exhibit the same limitation of TD hardware we had previously observed:[59] the inability to obtain valid class-ID readings on E-cores on the processor we used. Henceforth, we will refer to the TD-related scheduler extensions provided by these Intel patches as *Linux-ITD*. Linux-ITD provides the low-level kernel support to manage and leverage TD features from the process scheduler. More specifically, every scheduler tick, Linux-ITD retrieves the class ID of the thread running on the current CPU. The scheduler only updates the thread's class (stored in a new field added to the Linux task structure) when the last four consecutive class readings match the same class ID. Linux-ITD also extends the functionality of the load balancer with awareness of relative performance scores of the tasks running on the various cores. This makes it possible to map preferentially on big cores threads that exhibit a higher relative performance on those cores. In Section 6.1, we experimentally evaluate Linux-ITD, whose performance is greatly affected by the inability to get valid class readings on E-cores.

A recent paper by Intel[58] showcases more details on the hardware implementation of TD in the Alder Lake microarchitecture, which complements the microarchitecture-agnostic description found in the Intel manual.[56] Of special attention is the fact that class identification is conducted by a machine-learning based algorithm that leverages telemetry data and runs on dedicated hardware. More importantly, the article discloses the actual meaning behind each of the four TD classes, an information that we did not have in our earlier works.[4,59] Class 0 encompasses the majority of program phases, such as arithmetic, flow control, and data transfer intensive. Classes 1 and 2 are associated with code that experiences a substantial performance benefit (a 2.75× average speedup[58]) when running on P-cores, such as the ones that make extensive use of wide vectors, AI, and accelerated instructions. Lastly, Class 3 is reserved to identify threads running busy-waiting code (i.e., spin loops), which waste valuable CPU cycles, especially on P-cores. The overarching conclusion is that threads are assigned to the various TD classes based on the instruction mix that they run.

4.3 | Analysis of the big-to-small speedup prediction accuracy of TD

In our experiments, we observed that SF predictions provided by TD hardware are not accurate enough to effectively guide a throughput-optimized scheduler for AMPs. To illustrate these inaccuracies, Figure 4 provides the distribution of classes assigned to all SPEC CPU programs (from the CPU2006 and CPU2007 suites) throughout an isolated execution of each program on a big core. The class of every program was sampled every tick,

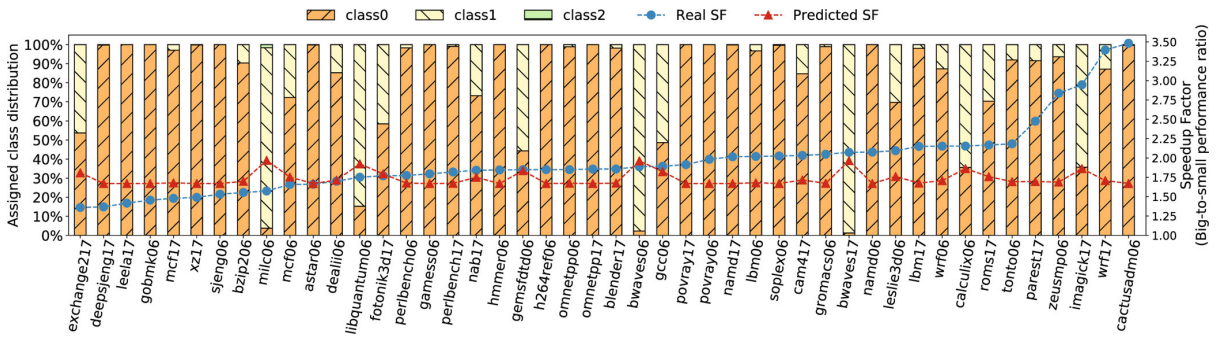


FIGURE 4 Proportion of assigned classes on big cores versus actual and TD-predicted speedup factor.

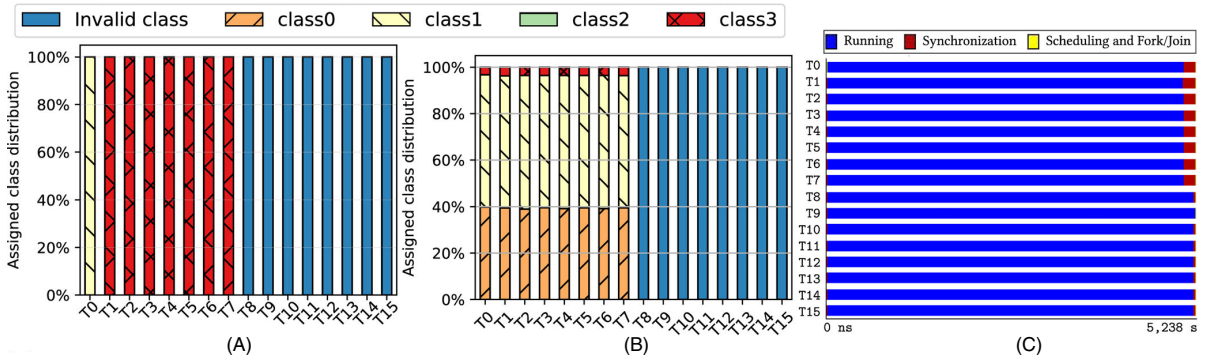


FIGURE 5 Demonstration of the effectiveness of the spin-loop detection feature in Thread Director with (A) the `barrier` microbenchmark and (B) the EP program. (C) Depicts the load imbalance situation under EP that leads T0–T7 threads to busy wait for a while. (A) Class distribution for threads in `barrier`. (B) Class distribution for threads in EP. (C) Execution trace of EP.

just like Linux-ITD does. We conducted this experiment using both our kernel-level extensions (implemented with PMCSched), and an instrumented version of the Linux-ITD patch;[35] in both cases we obtained the same class distributions by gathering the per-class counts at the end of the execution. Recall that small cores do not provide valid class readings on our platform, even using the Intel patch. Each program listed in Figure 4 appears in ascending order by its *real* overall SF, computed as the ratio between the program's completion times on an E-core and a P-core. As a reference, in addition to the *real* SF, we plot the overall SF estimated by TD, denoted as *predicted SF*. This prediction corresponds to a weighted average: the summation of the product of the frequency of each class on the program, and the class's SF value (as indicated in Table 1).

The results reveal that very little time of the execution phase of any application is categorized as class 2 by TD. According to the recent Intel paper,[58] it is also unsurprising that no benchmark was identified as belonging to class 3 at any time (reserved for busy-waiting code), and that most benchmarks are categorized as class 0 for the majority of their execution time, given that this is the default class for what TD considers normal workloads.

From our experiments, it is clear that there is not a consistent and reliable correlation between the predicted and real SF. The discrepancies between actual and predicted SF values are significant. Take for instance the `cactusadm` program, that showed approximately twice as much real SF than predicted. These mispredictions could potentially lead to ineffective thread-to-core mappings performed by an asymmetry-aware scheduler that relies on TD. For example, the `milc` program often categorized as class 1 would be preferentially assigned to a big core due to its high predicted SF (almost 2), whereas the `cactusadm` application (with the highest big-to-small actual speedup across the board) could be relegated to small cores due to the low TD-predicted value (1.67). So overall, we conclude that making performance predictions based on the instruction mix employed by TD implementation in Alder Lake processors[58] does not render accurate speedup factor values.

To validate the ability of TD hardware to detect busy-waiting code, we built a custom microbenchmark, referred to as `barrier`. This microbenchmark launches one POSIX thread per core, albeit only Thread 0 (T0) performs useful and computationally-intensive work; the remaining threads just wait for T0 to complete at a synchronization barrier, which employs busy waiting. Threads with a lower TID value (0–7) were mapped to big cores, and the remaining ones to small cores. Figure 5A shows that threads T1–T7 are identified with TD class 3 for most of the execution

time; this class is reserved for busy-waiting code. As proposed in prior efforts,[32] this information could be, in theory, leveraged by the OS scheduler to preempt long-running busy-waiting threads in favor of others doing useful work, albeit at a slower rate, on small cores. Unfortunately, given that E-cores provide no valid class ID readings (see T8–T15 in Figure 5A), the OS may select a small-core thread that is also busy waiting; this would be exactly the case in this particular program.

Note that non-negligible busy-waiting phases can also be present in real multithreaded applications. This is particularly the case of regular OpenMP programs running with the `static` OpenMP loop-scheduling method on AMP platforms; `static` assigns an even number of loop iterations to all threads. To illustrate this aspect, we ran the EP program from NAS parallel benchmarks using all available cores on our Alder Lake platform, employing the same thread-pinning scheme as in `barrier`. EP consists of a single parallel loop that encompasses most of its execution. While the `static` loop-scheduling method usually provides acceptable performance for EP on symmetric multicore platforms, it typically introduces load imbalance under performance asymmetric cores, because big-core threads usually complete their share of the loop earlier than small-core threads.[7,11] When using adaptive synchronization (the default setting in our GCC-based OpenMP environment), threads busy-wait for a while and they may later block after a certain number of frustrated attempts in synchronization primitives, such as barriers.[32] Figure 5C shows an execution trace built with Paraver[60] for the EP program. Red regions at the end of the execution of big-core threads (T0–T7) indicate the existence of load imbalance. These threads complete their allotted iterations earlier than small-core threads, so they busy-wait in the synchronization barrier at the end of the loop. As Figure 5B reveals, threads T0–T7 are identified with class 3 (busy-waiting code) during a non-negligible fraction of their execution time.

Motivated by the TD's inaccuracies in SF prediction, in our previous work[4] we explored the utilization of machine learning (ML) to estimate the SF online by leveraging runtime statistics gathered with PMCs. Our analysis resulted in two PMC-based estimation models making it possible to predict the SF from a big core or a small one. These models were designed to better capture the SF across different program phases, which is known to be the key for effective thread-to-core assignments in multi-program workloads.[2,9,34] The inference procedure for the PMC-based models, based on additive-regression, can be efficiently conducted via a simple iterative algorithm in the OS kernel-free of any floating-point code—that runs in just a few microseconds.

Figure 6 shows a compact summary of the results presented in our earlier work[4] on the SF prediction accuracy of the PMC-based models, which we later leverage in several scheduler implementations (Section 6.1). Specifically, the figure shows the comparison of the actual SF and the predicted one (with TD and our PMC-based models) on a big core for a few benchmarks from SPEC CPU. The chart plots the predictions observed over time, enabling to gain a better understanding of the per-phase accuracy of the various prediction methods. As it is evident, the big-core PMC-based model provides superior estimation accuracy than TD for these programs. The results also reveal that relying on a reduced number of classes to drive predictions leads TD to producing very coarse SF estimations; the predicted SF values for these programs are either 1.67 or 1.97 with TD (values associated with class 0 and 1, respectively, as shown in Table 1). We should highlight that, unlike TD, the PMC-based models for the big-core and small-core do not leverage classes or exclusively rely in the instruction mix. Instead they produce an estimation using a lightweight iterative algorithm (additive regression) based on the values of different performance metrics that capture diverse sources of processor stall cycles.[61] Stall cycles are known to be a good indicator to capture the performance differences observed across core types.[3,12,62] The specific performance events

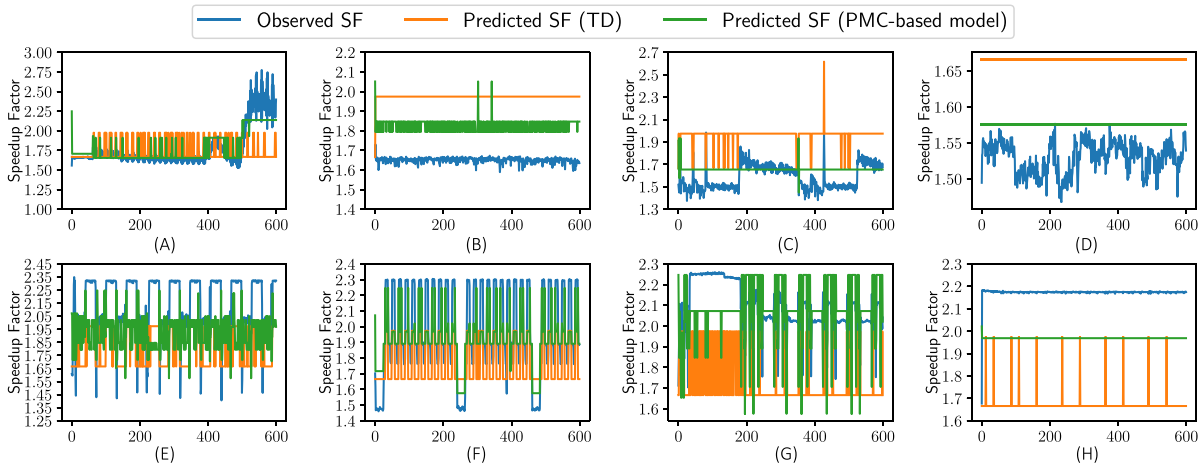


FIGURE 6 Comparison between real speedup factor (SF) and that predicted by TD and a PMC-based model[4] on big cores. The figure plots the predictions observed over time for a total of 30B instructions (SF values sampled every 50M retired instructions). Actual SF values are obtained with the IPC ratio of each 50M-instruction window gathered on both core types, as explained in detail in our earlier work.[4] (A) fotonik3d17. (B) imagick17. (C) milc06. (D) sjeng06. (E) calculix06. (F) nab17. (G) cam417. (H) lbm17.

the PMC-models depend upon, the methodology used to build these models, and an extensive experimental analysis on their accuracy can be found in prior work.[4]

5 | ASYMMETRY-AWARE LOOP-SCHEDULING FOR OPENMP PROGRAMS

In this section, we begin by describing AID and its main limitations. Next we present flexible AID, which we designed and implemented in GCC v11.2.

5.1 | Asymmetric iteration distribution

The rationale behind creating AID (asymmetric iteration distribution)[11] was to offer asymmetry-aware replacements to conventional loop-scheduling OpenMP methods. Unlike these techniques, AID strategies use knowledge of the underlying core type where the different worker threads run to improve load balance and consequently enhance application performance on AMPs. This is accomplished by letting threads running on big cores (henceforth, big-core threads) potentially complete more loop iterations than threads running on small cores (henceforth, small-core threads) based on the observed big-to-small performance ratio (aka. SF).

The experimental analysis of prior work[11] revealed two key insights on parallel-loop execution for asymmetric multicores. First, the SF may differ substantially across parallel loops even within the same application, so using a single application-wide SF value may fail to provide effective scheduling. Second, systematically assigning more loop iterations to big-core threads than to small-core ones in proportion to the observed SF does not always guarantee perfect load balance across worker threads. This stems from the fact that inherent imbalance may be present in some loops, namely imbalance not caused by the platform's performance asymmetry but instead due to the different amount of work associated with completing the various loop iterations.

Based on these insights, a variety of AID loop-scheduling methods were proposed,[11] all of them featuring a shared stage at the beginning of each parallel loop, referred to as the *sampling phase*. Essentially, during this phase, all threads run `chunk` iterations (being 1 the default value of this configurable parameter) and track these iterations' completion time*. When the last thread completes its `chunk` iterations, this thread approximates the loop's SF by dividing the average time that small-core threads take to complete these iterations by the average completion time of the iterations for big-core threads. The way that the remaining loop iterations are scheduled differs across AID methods.

AID[11] comprises three asymmetry-aware scheduling methods: `AID-static`, `AID-hybrid` and `AID-dynamic`. The first two strategies constitute an alternative to the conventional OpenMP `static` schedule. Under AID-static, each thread receives a number of iterations in proportion to the performance they get on the core they are currently running on, relative to that of the small core. Specifically, let P_i be the relative performance of thread i, where $P_i = 1$ for small-core threads, and $P_i = SF$ for big-core threads. Let NI denote the number of iterations of a parallel loop, and N the number of worker threads. The number of iterations that `AID-static` assigns to a thread T is $\frac{NI \cdot P_T}{\sum_{i=1}^{N} P_i}$. So, for example, in a loop with 120 iterations, $SF = 2$, and considering that 2 threads are running on each core type ($N = 4$), big-core threads would be assigned 40 iterations each and small-core threads 20 iterations each. The `AID-hybrid` method, by contrast, distributes only a configurable portion of the total iterations (being 80% the default value) as `AID-static`; the remaining iterations are scheduled with the conventional `dynamic` method. This hybrid approach compensates for slight divergences in the amount of work required by the various iterations, as well as for potential inaccuracies in SF prediction. Finally, `AID-dynamic` was conceived as a lower-overhead replacement for the conventional `dynamic` loop-scheduling method, where small-core threads employ a fixed chunk (with the same meaning as in `dynamic`), whereas big-core threads normally use a bigger chunk, which is dynamically adjusted based on the SF observed during the loop's execution.

AID's original implementation[11] makes two simplifying assumptions. First, the parallel program executes alone on the platform, thus taking advantage of all the large cores available; the number of big cores (N_{BC}) is notified to the runtime system via a custom environment variable. Second, AID methods assume specific thread-to-core bindings when applying the various AID loop-scheduling methods; specifically, threads with IDs ranging between 0 and $N_{BC} - 1$ are mapped to big cores, and the remaining threads run on small cores. It is up to the user to enforce the associated mapping by defining the corresponding environment variable of the runtime system (e.g., `GOMP_CPU_AFFINITY` in GNU's implementation). The default mapping considered by AID's original implementation, greatly simplifies the implementation, and at the same time allows to automatically accelerate many inherently serial phases of the program that the master thread (thread with TID = 0) usually runs, such as initialization code or regions in between parallel loops.[7,11,32] Unfortunately, the aforementioned assumptions greatly limit AID's applicability to more general workload scenarios.

5.2 | Flexible AID

In this work, we created flexible AID, which extends AID's functionality in the following ways. First, unlike the original AID, flexible AID allows the exploitation of asymmetry-aware loop-scheduling without requiring fixed thread-to-core mappings; this enables the OS to make thread placement

decisions even with multiple applications running on the system (as we show in Section 6.3). This enhancement is realized via PMCSched, and opens the door to future malleable implementations. Second, our proposal is equipped with a new loop-scheduling method, referred to as `AID-guided`, an asymmetry-aware variant of the standard OpenMP `guided` schedule. Third, flexible AID implements a mechanism to reduce the runtime overhead in loops with very short iterations, where runtime system's calls may constitute a significant fraction of the loop's completion time.

The first new feature leverages explicit interaction between the OpenMP runtime system and the OS scheduler, which we implemented on top of a vanilla Linux kernel thanks to PMCSched. Essentially, when the runtime system initializes the per-thread structures in a distributed fashion, each worker thread requests the creation of a memory region shared between the runtime system and the kernel. This memory region stores a structure with several thread-specific attributes (or fields) that allow different system software layers to exchange scheduling-relevant information[†]. To enable the creation of per-thread shared memory regions, PMCSched exports a special file (`/proc/pmc/schedctl`); the thread in question invokes the `open()` and `mmap()` system calls on that file to retrieve the pointer (user virtual address) for the memory region. At the same time, the OS stores a per-thread field in the task structure that holds the runtime thread-specific structure. This enables any PMCSched plugin to access the associated fields, making it possible to conduct a wide range of runtime optimizations. Relying on a special file in a pseudo-file system like *procfs* (not backed up on disk) allows us to implement the request for the creation of the memory region without adding new system calls. Note that creating new system calls in Linux requires patching kernel, which we wanted to avoid doing.

Our flexible AID implementation utilizes two thread-specific attributes in the shared memory region—`core_type` and `amp_prio`. The `core_type` field indicates the core type where the thread is currently running on; this attribute is maintained by the OS scheduler, which appropriately updates its value after a cross-core-type migration (e.g., from a big to a small core). Flexible AID's loop-scheduling methods use the `core_type` field to decide on how many iterations assign to the various threads. By contrast, the `amp_prio` integer attribute indicates the thread's priority when it comes to using the available big cores, relative to the other threads in the same application; larger values of `amp_prio` indicate a lower priority. Unlike the `coretype` field, the `amp_prio` attribute is set by the runtime system, so as to provide hints to the OS scheduler on which threads contribute the most to improve the application-wide performance when scheduled on the available big cores. For example, the master thread could be given a high priority (low `amp_prio` value), to ensure the scheduler favors its mapping on a big core relative to other application threads, thus accelerating the sequential code it may run. While other methods exist to establish the proposed OS-runtime interaction and to determine the required information from the runtime system, using a shared memory region brings important benefits, such as the avoidance of entering the kernel and removing the need of architecture-specific mechanisms for CPU identification from the runtime system (e.g., leveraging the x86 `cpuid` instruction to determine the core type in hybrid CMPs). This reduces the overhead of potentially frequent operations, and improves the portability of the runtime-system implementation.

Our asymmetry-aware OS schedulers implemented on top of PMCSched do take the per-thread `amp_prio` values into consideration when assigning application threads to big cores. Specifically, the implementation periodically checks the `amp_prio` values of threads assigned to big and small cores; if a pair of threads of the same application T_B and T_S running on big and small cores, respectively, are found such that $amp\_prio_{T_B} > amp\_prio_{T_S}$, the scheduler swaps both threads between cores. We should highlight that in our experiments the thread's `amp_prio` values are established via a custom environment variable `GOMP_AMP_PRIO`, so they remain the same throughout the execution. This, together with the OS maintenance of the per-thread `coretype` fields, allows the runtime system to apply AID methods with or without pinning threads to cores, while leveraging big cores to accelerate serial code in the master thread, even with multiple applications running on the system. An interesting avenue of future work would be the design of runtime strategies that address other types of scalability bottlenecks in OpenMP programs (e.g., long `CRITICAL` sections) by dynamically adjusting per-threads `amp_prio` values in the runtime system.

During our experiments we found that some OpenMP programs actually benefit from the standard `guided` OpenMP schedule on AMP systems. This loop-scheduling method uses a monotonically decreasing chunk; every time that a thread removes iterations from the common pool (also used in the `dynamic`'s implementation) the number of iterations the thread receives is $\frac{R}{N}$, where R is the number of loop iterations remaining to be completed, and N the total number of worker threads. Because R decreases every time a thread removes iterations, the next thread attempting to remove iterations is assigned fewer iterations. On AMPs, this schedule provides lower runtime overhead than `dynamic` (as it normally requires fewer runtime-system calls due to the bigger variable-size chunks), and addresses in part the load imbalance that comes from running worker threads on performance-asymmetric cores. However, as opposed to `dynamic`, the `guided` method may accidentally assign substantially more work (iterations) to small-core threads than to big-core threads, especially at the beginning of the loop. Nothing prevents a small-core thread from removing iterations from the shared pool before a big-core thread, leading to more iterations, and hence to potential load imbalance.

To better cope with performance asymmetry under these circumstances, we created a new asymmetry-aware variant of `guided`, referred to `AID-guided`. This loop-scheduling method also employs a decreasing chunk, but caters to the loop's speedup factor and the core type where the thread runs, by leveraging the thread's relative performance—the P_i indicator defined in the previous subsection. Specifically, when a worker thread T requests iteration removal from the pool, the runtime system assigns it $\frac{R \cdot P_T}{\sum_{i=1}^{N} P_i}$ iterations. This makes it possible for big-core threads to potentially use larger chunks than small-core threads in proportion to the loop's SF, which contributes to improving load balance on AMPs.

Finally, flexible AID incorporates an optimization that seeks to limit the runtime system's overhead in loops with very short iterations. We found that for this kind of loops, the benefits that come from AID methods other than `AID-static`, which require more than two runtime calls per thread each loop, could be overshadowed by the cost associated with the runtime system calls (averaging 1.5 μs on our platform). In some cases frequent

runtime calls could even lead to performance degradation. To overcome this shortcoming, our AID-enabled runtime system automatically switches to `AID-static` (the AID method that requires fewer runtime calls) as soon as it detects that loop iterations no longer surpass the average time required by runtime calls. Significantly, for lightweight detection, we leverage the completion times already recorded for the various worker threads during AID's sampling phase. This conservative optimization naturally detects situations where fine-grained loop-scheduling (e.g., `dynamic` with a small chunk) is not beneficial, and effectively avoids the overhead.

6 | EXPERIMENTAL CASE STUDIES

To demonstrate the applicability of PMCSched, we considered three experimental case studies that leverage our framework for the adoption of various asymmetry-aware optimizations in the system software. In the first case study (Section 6.1), we evaluate how effectively an OS-level scheduler can improve the overall system throughput on AMPs using workloads consisting of multiple single-threaded programs. In the second one (Section 6.2), we assess the effectiveness of flexible AID's asymmetry-aware loop-scheduling strategies for OpenMP programs running alone on our Alder Lake platform. In the last case study (Section 6.3), we analyze the impact of an asymmetry-aware runtime system and an OS-level scheduler working in synergy toward optimizing the overall system throughput; to this end, we experiment with multi-application workloads combining sequential and parallel (OpenMP) programs.

6.1 | Maximizing throughput under mixes of compute-intensive sequential programs

Improving system throughput on AMPs with the HSP scheduler. Previous research[2,6,9,12] has demonstrated that to maximize throughput in the context of multi-program workloads, the scheduler needs to be able to (1) determine at runtime the benefit (i.e., *speedup*) that each thread in the workload derives from running on a big core relative to a small one, and then (2) map preferentially to big cores those application threads that may derive a larger relative performance benefit from such cores. Because applications may go through different program phases, continuous per-thread performance monitoring and dynamic adjustments of the thread-to-core mappings are in order to make the most out of asymmetric cores.[9,12,34]

For single-threaded applications, its speedup matches the SF of its only thread. For multi-threaded programs, determining the application-wide speedup and identifying which threads are more profitable candidates for running on big cores poses a big challenge. Note that other factors beyond the per-thread SF need to be considered in this case, such as the current degree of thread-level-parallelism (TLP)—some applications may exhibit serial or low-TLP phases[2]—, synchronization effects,[32] or aspects handled exclusively by an underlying runtime system, like parallel-loop scheduling, load balancing or interdependencies among tasks within the application.[7,10,11]

In this work we implemented the HSP (High SPeedup) scheduler[34] on top of PMCSched. Our implementation leverages PMCSched's extensible kernel module to augment the functionality of the default Linux scheduler (CFS) with asymmetry-aware thread-to-core mappings, by imposing dynamic core-group level thread affinities. These mappings are driven based on applications' speedup predictions, which are obtained online by combining runtime information gathered with PMCs—for the threads' speedup factors (SFs)—, with the application's runnable thread count, serving as a proxy of its amount of TLP.[32] For each thread T, HSP employs the following *utility metric*, which provides an overall idea on how much the application would speed up when using all the available big cores (N_{BC}), and assuming (for simplicity) that the underlying runtime system equally balances the load among the application's runnable threads (N threads):[34,63]

$$\text{Utility}_T = \frac{MIN(N_{BC}, N)}{N} \cdot (SF_T - 1) + 1. \quad (1)$$

Threads in the workload with a higher value of the utility metric are preferentially assigned to big cores by the HSP scheduler. Despite the simplicity of this metric, previous work has already widely demonstrated its effectiveness when it comes to throughput maximization under compute-intensive multi-program workloads.[2,34] In particular, the metric properly reflects the fact that sequential code derives larger performance benefits from using the available big cores than highly-parallel code (with $N >> N_{BC}$). For example, sequential programs from SPEC CPU may run up to to 3.5 times faster (on average) on a big core rather than a small one (See SF values in Figure 4). In this case, the utility metric's value of its single runnable thread matches the speedup factor (SF_T). By contrast, as we show in Section 6.2, parallel applications without significant serial bottlenecks derive much smaller performance benefits (up to 1.6× performance increase on our platform). More importantly, achieving noticeable performance improvements requires the smart utilization of all the platform's big cores from the runtime system.

To ensure that high-utility threads are preferentially assigned to big cores, HSP leverages thread swaps, just like the vast majority of asymmetry-aware schedulers.[2,9,64-67] Swaps are preferred to one-way migrations as they do not disrupt load balance. Every so often, HSP activates a system-wide process that repeatedly checks if the thread with minimum $Utility_T$ running on big cores (T_B), has a utility value that is no smaller than that of the small-core thread with the highest utility (T_S). If this is not the case, T_B and T_S are swapped. Our implementation additionally features an optimization for OpenMP programs, which affects HSP's thread swapping policy. By default, HSP avoids triggering swaps between threads of

the same application. The rationale behind this is the assumption that the runtime system internally balances the load across workers, by possibly assigning more work (e.g., more loop iterations) to big-core threads. Nevertheless, HSP still factors in per-thread `amp_prio` values established by a runtime system leveraging flexible AID (see Section 5.2). To this end, it ensures that active threads of the same application with the lowest `amp_prio` values (high priority), are considered first when populating big cores. So, two threads of the same OpenMP application running on different core types can be swapped only in the event the big-core thread has a lower priority than that of the small-core thread. This optimization favors locality (as it reduces the number of thread migrations), and at the same time, caters to the affinity hints provided by the runtime system.

SF prediction under HSP. One of the main deltas among the various asymmetry-aware throughput-optimized schedulers[6,12,34] is the underlying method employed to determine the SF online. In this work, we explore the effectiveness of two SF prediction methods: PMC-based estimation models,[3,12,34] and reliance on specific hardware support for SF estimation.[13,56] Regarding the first prediction method, we use the two PMC-based estimation models proposed in our earlier work,[4] which were specifically built for SF prediction from the big and small cores of an Intel Alder Lake processor. For the hardware-aided SF prediction we leverage the Intel TD technology, outlined in Section 4.

To evaluate the impact of the various SF-prediction methods on system throughput we implemented three variants of HSP. The first variant—referred to as HSP/TD—employs TD to obtain SF estimates. Because in the Alder Lake processor we used such estimates are only accessible directly when the thread runs on a big core (as discussed in Section 4), our implementation continuously stores TD-based big-core SF estimates on a per-thread history table for different program phases, making it possible to obtain SF predictions indirectly from small cores by taking previous history into account. A large body of work has leveraged history tables to observe patterns from previous samples and predict current and future performance.[2,4,31] In particular, our per-thread history table was implemented following a similar strategy to that described in prior work,[2,4] where two control PMC metrics are used to allow the efficient matching between the SF values stored on the table and the corresponding program phases. To deal with frequent *phase misses* when accessing the history table from small cores (i.e., the table does not hold valid SF predictions for certain phases), our implementation triggers migrations to big cores to gather new big-core estimates, and also implements a throttling mechanism to limit the number of profiling-related migrations.[2]

In the second HSP variant, denoted as HSP/BS, the OS continuously gathers a number of per-thread PMC metrics; an up-to-date SF prediction is obtained for a thread by using the metric values as input to the core-specific models proposed in our earlier work[4] for the big and the small core. Lastly, under the HSP/B variant, SF predictions on big cores are obtained via the same big-core model used by HSP/BS; however, on the small core, predicted SFs are obtained indirectly by reading a history table, populated with past SF estimates retrieved on the big core. Note that this variant was implemented to conduct a fairer comparison with HSP/TD, where direct SF predictions on the small core are unavailable.

Experiments and discussion. In addition to the three aforementioned variants of HSP (BS, B and TD), we experimented with an Asymmetry-Aware Round-Robin (AARR) scheduler[8] that equally shares big and small cores among applications (also implemented on top of PMCsched), and with Linux-ITD (see Section 4.2), the Linux kernel patched with the recent scheduler enhancements proposed by Intel,[35] which leverage TD.

Previous work has demonstrated that the Linux default scheduler (CFS) in the unmodified kernel provides highly variable completion times for the same application across multiple runs of the same experiment on Intel Alder Lake processors[4] and other AMP systems.[2] Essentially CFS may map an application to a big core for a certain run, and then to a small core in another run, irrespective of its co-runners. This scheduler is designed to minimize the number of thread migrations; this coupled with the highly random mappings across runs, introduces large throughput differences, making CFS a misleading baseline.[2] Our experiments reveal that Linux-ITD also exhibits the same degree of performance variability, as we discuss next, so the AARR scheduler makes a better baseline for comparison.

For our experiments we randomly generate 20 diverse workloads, comprising of 16 single-threaded programs each. The composition of the various program mixes (*Mi*) is depicted in Figure 7, and covers 46 different SPEC CPU applications in total. All programs were compiled with GCC v11.2 with the `-O3` and `-mtune=alderlake` compiler switches. In launching each program mix, we follow a similar procedure to that of previous works,[2,34,37] so as to ensure the machine's load is constant throughout the experiment despite the divergences among the applications' completion time. All programs in the workload are started simultaneously, and when one of them completes, the program is restarted repeatedly until the slowest application completes three times. We use the geometric mean of the completion times for each program to calculate the degree of throughput, by using the aggregate speedup metric.[2,4,34] For each workload, and scheduler we also measured the average normalized turnaround time (ANTT) metric.[68] The lower-is-better ANTT metric captures the average performance degradation suffered by the workload's applications relative to running alone on the system (with all the big cores available). We should also highlight that Linux-ITD is the only scheduler among those considered whose degree of throughput varies substantially from run to run. To better capture Linux-ITD's overall behavior, we repeated the experiments with this scheduler 10 times, and report the geometric mean of the various performance metrics considered. The remaining schedulers distribute big-core and small-core cycles consistently among threads in different executions; in multiple runs of the same workload their throughput oscillates in a tight 1.3% range.

Figure 8 shows the normalized throughput (top) and the reduction of the ANTT metric (bottom), for the various scheduling algorithms relative to AARR. These results undoubtedly reveal that HSP/BS outperforms the other schedulers for most workloads, achieving up to a 30% throughput gain w.r.t. AARR, and providing a 22.9% average improvement against the TD variant. The performance improvements are tightly related to the superior SF-estimation accuracy provided by the PMC-based models[4] for the big and small core, relative to that of Thread Director, as discussed in

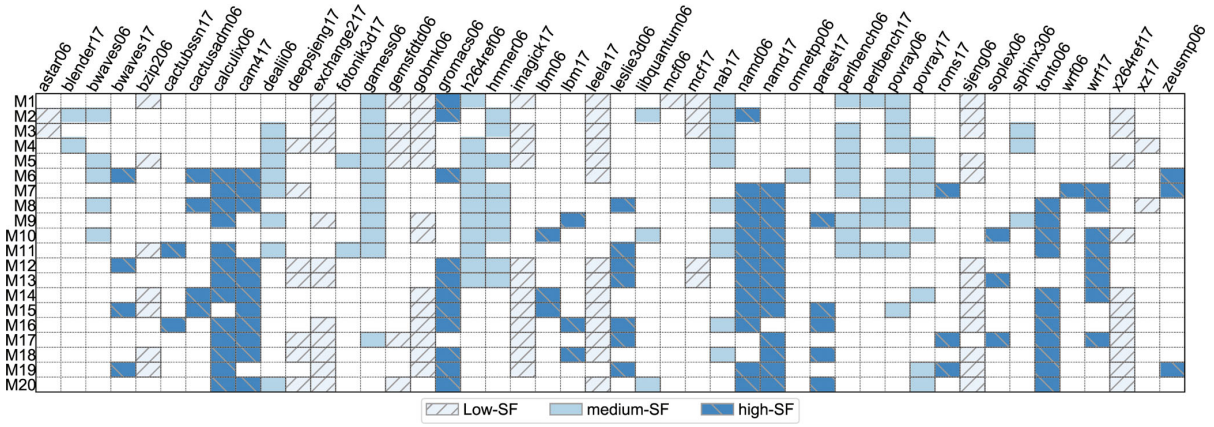


FIGURE 7 Workloads used for our experiments. Each row M_i depicts the composition of the ith workload. A blank cell indicates that the associated program is not included in the workload. Applications whose average SF is lower than 1.7 are considered low-SF programs in our platform, and those with an SF value greater than 2.05 are classified as high-SF. The remaining ones are labeled as medium-SF.

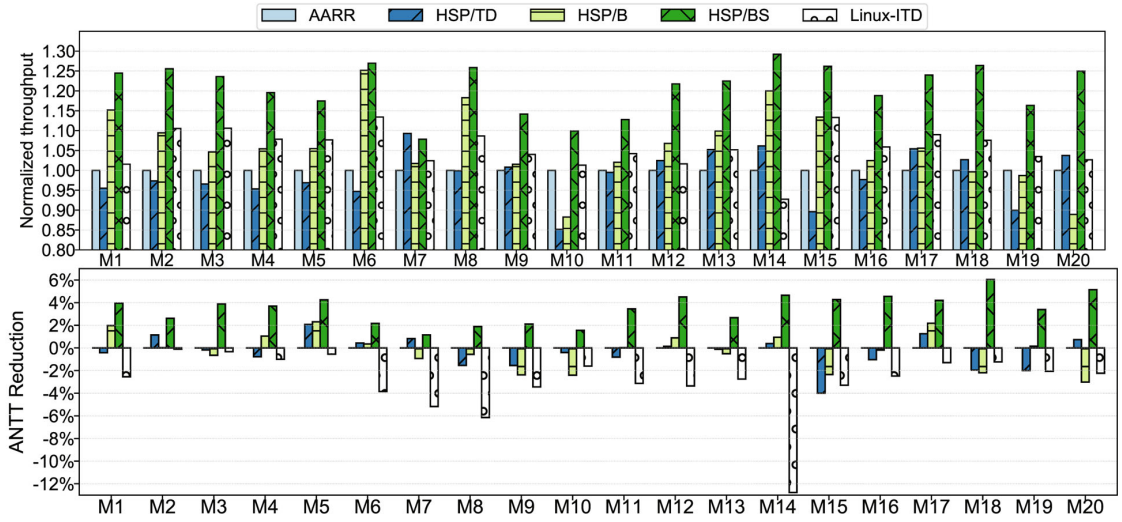


FIGURE 8 Normalized throughput and ANTT reduction associated with the various scheduling algorithms.

Section 4.3. Overall, a higher SF-prediction accuracy allows HSP to identify programs with a truly high SF better, and, as a result, the scheduler can grant more big-core cycles to them than to other threads. In doing so, the average slowdown across applications is also reduced across the board; leading to up to a 6% reduction in the ANTT metric (see M18 in Figure 8 bottom).

We further observe that using the big-core model alone in combination with the history table (HSP/B variant), provides substantially better throughput figures than HSP/TD (averaging 7.9% improvement). However, in a few workloads, such as M10 and M20, HSP/B fails to yield comparable performance to that of AARR. We found that this is caused by the extra thread migrations (and hence the overheads), triggered in response to frequent table phase misses, and aimed at refreshing the history table on big cores. As a reference, Table 2 shows the average number of thread swaps per second of the various asymmetry-aware schedulers across all workloads. As it is evident, the HSP variants that rely on the history table to retrieve SF predictions gathered on the big core (HSP/B and HSP/TD), trigger cross-core-type thread migrations more often than the other strategies. Despite this fact, we conclude that the PMC-based big-core model alone provides superior accuracy than TD, and that the per-thread history table constitutes a reasonably effective method to deal with scenarios where direct SF estimation may be unavailable on certain core types.

Finally, we turn our attention to the results of Linux-ITD. To illustrate the enormous performance divergences observed in multiple executions of the same workload under Linux-ITD, Figure 9 shows the distribution of normalized throughput and ANTT across 10 independent runs of each

TABLE 2 Average number of thread swaps per second observed for workloads M1-M20 under the various scheduling algorithms.

Algorithm	Thread swaps per second
AARR	2.01
HSP/TD	2.33
HSP/B	2.45
HSP/BS	1.11

Note: These statistics were gathered with the SystemTap kernel tracing tool in separate launches of the workloads (different to those of Figure 8 to avoid interference with the measurements).

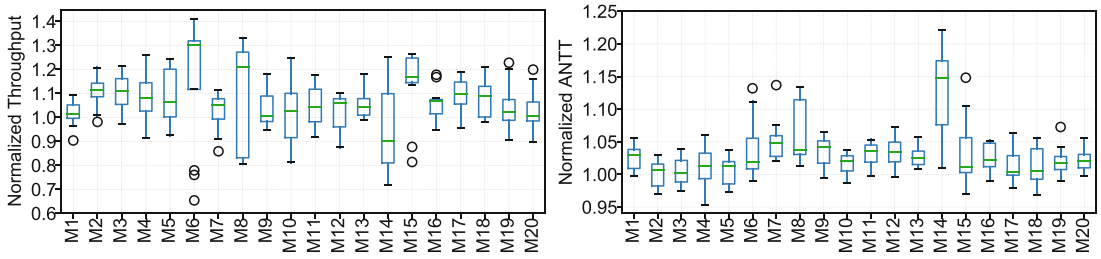


FIGURE 9 Distribution of normalized throughput (higher is better) and ANTT values (lower is better) under multiple runs of Linux-ITD.

TABLE 3 Multi-application workloads consisting of single-threaded and multithreaded programs.

Workload	Applications
W1	`calculix06, gamess06, nab17, namd17, CG`
W2	`gromacs06, hmmer06, tonto06, bwaves17, FT`
W3	`calculix06, gromacs06, namd06, perlbench17, IS`
W4	`gromacs06, perlbench06, tonto06, bwaves17, blackscholes`
W5	`gromacs06, namd06, sphinx306, namd17, bodytrack`
W6	`namd17, perlbench17, povray17, x264ref17, freqmine`
W7	`gamess06, gromacs06, namd17, perlbench17, CFDeuler3D`
W8	`dealii06, h264ref06, tonto06, namd17, myocyte`
W9	`calculix06, h264ref06, cam417, exchange217, particlefilter`
W10	`gamess06, namd06, cam417, x264ref17, pathfinder`
W11	`calculix06, sphinx306, cam417, leela17, cactubssn_s`
W12	`gromacs06, namd06, cam417, deepsjeng17, cam4_s`
W13	`hmmer06, perlbench06, sphinx306, deepsjeng17, pop2_s`

workload. Depending on the run selected, Linux-ITD may deliver either better or worse performance than the baseline, making the average numbers displayed in Figure 8 potentially misleading if considered in isolation. Of special attention is the case of workload M6 under Linux-ITD where one of the runs achieves a 41% throughput increase w.r.t the baseline, and another run degrades throughput by more than 35%. This variability stems from the fact that Linux-ITD randomly assigns thread to cores at the beginning of the execution by populating big cores first (as the other schedulers do), but instead leave the threads on the same core type they run throughout the execution in most cases. The reason of this unexpected behavior is the inability of the TD hardware implementation on our platform to feed the OS with valid class IDs for threads running on small cores. While SPEC CPU programs running on the big core are classified over time in classes 0-2 (as shown in Section 4), Linux-ITD systematically assigns class 0 to all small-core threads; in practice, that's the default class assigned by Linux-ITD when the hardware repeatedly reports an unknown class. Because the Linux scheduler is designed to avoid thread migrations when possible, and the TD performance score associated with class 0 on big cores is no greater than that of classes 1 and 2 (see Table 3), Linux-ITD never finds a thread running on a small core on our platform that exhibits a higher

big-to-small speedup than any of the threads currently populating big cores. Therefore, threads initially assigned to small cores get stuck on these cores, leading to highly variable throughput figures across runs (see Figure 9), and providing higher ANTT values than the other schedulers across the board (see Figure 8). Notably, Section 6.3 provides evidence of the consistent throughput results of the HSP from run to run, even when parallel applications are included in the workload.

6.2 | Experiments with OpenMP programs

In this section, we proceed to analyze the performance of various loop-scheduling methods using a wide range of OpenMP programs from four different benchmarks suites—NAS parallel benchmarks, PARSEC, Rodinia and SPEC CPU2017—running on the Intel Core i9-12900K processor. To the best of our knowledge this is the first comprehensive evaluation of the performance of OpenMP programs on an Intel Alder Lake platform.

All programs were compiled with GCC v11.2 using the same compiler switches mentioned in Section 6.1. To activate the runtime system in all parallel loops, we also applied the small compiler patch proposed in prior work,[11] which changes the default loop-scheduling method from `static` to `runtime`; this also allows us to specify the selected loop-scheduling method and its parameters via environment variables for OpenMP parallel loops not including the `schedule` clause. For NAS, PARSEC and SPEC benchmarks we used the *C*, *native* and *reference* input sets, respectively. Since the default commands provided for Rodinia applications lead to very short execution times on our platform (less than a second in most cases), we slightly increased the benchmarks' data input size as in earlier work.[69] The processor's Turbo Boost feature was disabled during the experiments.

In our experiments, we compare the effectiveness of flexible AID scheduling methods—`AID-static`, `AID-hybrid`, `AID-dynamic` and `AID-guided`—with all OpenMP programs, to that of the OpenMP standard `static`, `dynamic`, and `guided` strategies, which are asymmetry unaware. Because we did not modify the source code of any OpenMP program, each loop-scheduling technique evaluated was applied to all the parallel loops missing the `schedule` clause; this is the case of the vast majority of the loops (over 93%) in the benchmarks we used. Note that assessing the impact of applying a mix of loop-scheduling methods to different loops in the same OpenMP program running on an AMP requires making changes in the application, so we leave this promising analysis for future work.

As shown by previous research,[11] running the master thread (with TID = 0) on a big core may deliver a substantial performance boost in many OpenMP programs, as this thread is in charge of the execution of explicitly sequential regions. To isolate the benefits and potential overheads stemming from specific loop-scheduling methods, we experimented with two thread-to-core mapping strategies: *SB* (small–big) and *BS* (big–small). Under *SB*, threads with a low TID (from 0 to 7) are assigned to small cores, and the remaining threads (with TIDs $\in \{8..15\}$) are pinned to big cores. *BS* makes the opposite mapping, this is, big cores are reserved for threads with TIDs ranging between 0 and 7. Therefore, under *SB* the master thread is assigned to a small core, and under *BS* it runs on a big core. For simplicity, and to reduce the bar count in the corresponding charts, the reported Flexible-AID results correspond to the *BS* mapping.

Figure 10A–D show the results obtained for all the OpenMP programs with the different loop-scheduling techniques, and running with as many threads as the number of cores in our Alder Lake platform (16). We ran each program five times; for our analysis, we considered the geometric

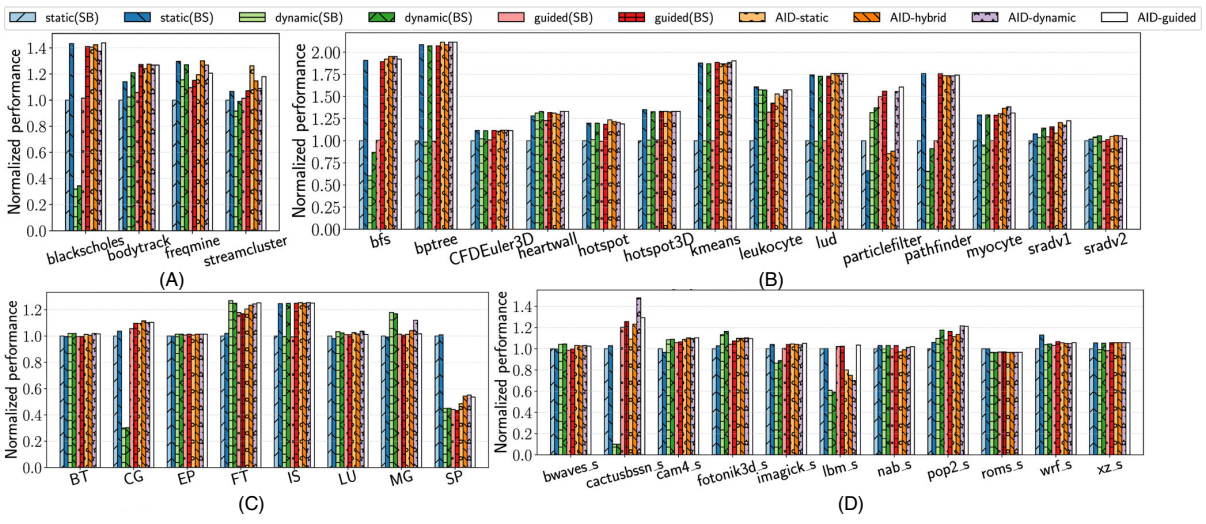


FIGURE 10 Normalized performance of the various OpenMP applications obtained for different loop-scheduling strategies. (A) PARSEC. (B) Rodinia. (C) NAS parallel benchmarks. (D) SPEC CPU2017.

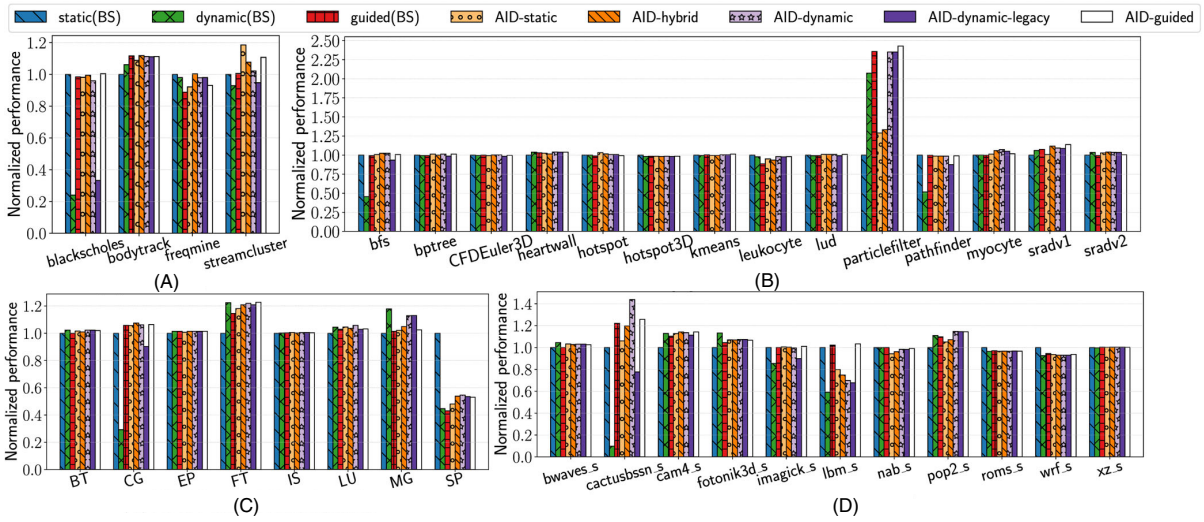


FIGURE 11 Performance of the various OpenMP applications obtained for different loop-scheduling strategies with the BS mapping normalized to `static(BS)`. (A) PARSEC. (B) Rodinia. (C) NAS parallel benchmarks. (D) SPEC CPU2017.

mean of the completion times of the application across the various runs. As in previous work,[7,11,70] we used the default chunk values defined by the runtime system for all the loop-scheduling methods. Figure 10A–D show the relative performance of each benchmark and loop-scheduling technique, normalized to the performance of `static(SB)`. For the sake of completeness, we also report the results of all the strategies using the BS mapping normalized to the performance of `static(BS)` in a separate figure (Figure 11). As a reference, this figure also includes an additional bar with the results of the old implementation of AID-dynamic[11]—denoted as `AID-dynamic-legacy`–, which does not feature the overhead-control mechanism described in Section 5.2.

The results in Figure 10A–D reveal that several programs obtain substantial performance gains (up to a 2.11× speedup) just by running the master thread on the big core. This is the case of *blackscholes*, *bfs*, *bptree*, *kmeans* or *IS*, where `static(BS)` greatly outperforms `static(SB)`, and other loop-scheduling methods provide modest or no additional performance improvements. Notably, for some programs like *freqmine*, *hotspot3D*, *leukocyte* or *pathfinder*, we observe that loop-scheduling approaches that could, in theory, cope with the platform's performance asymmetry (such as `dynamic` or the different AID variants) are unable to deliver performance improvements due to their higher overhead, which stems from the higher number of runtime-system calls required. Of special attention are the results of the `SP` program (Figure 10C), where it is extremely short loop iterations can be only handled safely by minimizing the interaction with the runtime system (as `static` does).

The results also underscore that no single OpenMP loop-scheduling method from the standard set constitutes the best choice across the board. In particular, *dynamic* becomes the best-performing strategy only in a few cases (*FT*, *MG* and *fotonik3D*); these programs feature coarse-grained loop-iterations that require an uneven amount of work, so a fine-grained loop-scheduling method like `dynamic` is the most appropriate choice here, even under performance-asymmetric cores. Conversely, by focusing on Figure 10A–D, we also observe that under different circumstances, the overhead introduced by `dynamic` backfires by degrading performance substantially, even to the extent of delivering lower performance than the baseline for some programs. Take for instance the *blackscholes* or *CG* programs, which slow down by a factor of 2.9× and 3.3×, respectively with `dynamic(BS)` w.r.t. the baseline. Other clear examples are *bfs* and *cactusbssn_s*.

Notably, the AID alternative for `dynamic` (`AID-dynamic`) is able to substantially reduce the overhead and reap benefits in these scenarios, thanks to the utilization of bigger and dynamic chunks, coupled with the overhead-control mechanism, which triggers a transition into AID-static under the presence of short loop iterations. By zooming on in the comparison between `AID-dynamic` and `AID-dynamic-legacy` (the latter does not feature the overhead-control mechanism) in Figure 11, we can see that both implementations perform similarly for most applications. However, the effect of the overhead-mitigation technique is quite noticeable in programs where dynamic iteration scheduling is counterproductive. This is the case of *blackscholes*, *bfs*, *pathfinder*, *CG*, *cactusbssn_s* and *imagick_s*, for which `dynamic(BS)` introduces an enormous overhead. While `AID-dynamic-legacy` partially mitigates this overhead by using bigger chunks, `AID-dynamic` substantially outperforms `AID-dynamic-legacy` across the board, even yielding performance improvements over the baseline in some cases (e.g., 6% for *CG* and 43% for *cactusbssn_s*).

Overall, and by zooming in on the results using the *BS* mapping, we conclude that AID scheduling methods constitute good general replacements for the corresponding asymmetry-unaware approaches. In other words, AID-static and/or AID-hybrid typically outperform the `static` schedule,

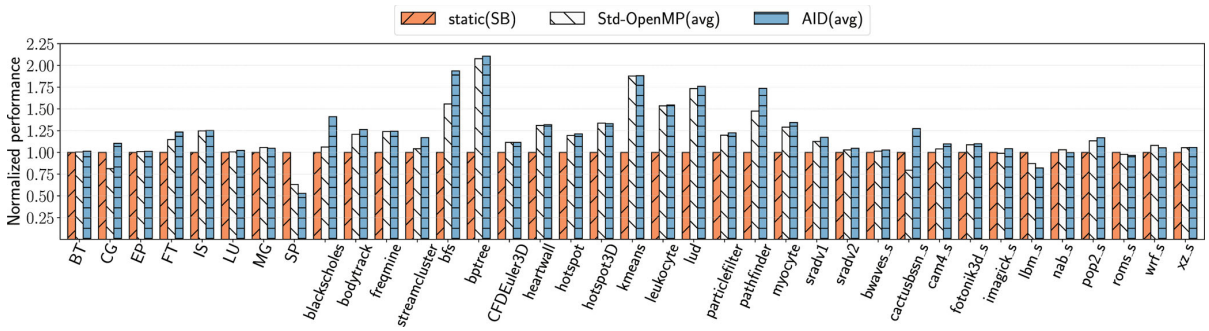


FIGURE 12 Average normalized performance across all standard OpenMP methods with the *BS* mapping (Std-OpenMP) and across all AID strategies for all the OpenMP programs explored.

and AID-dynamic provides better average performance than `dynamic` for most programs. Moreover, the new AID-guided loop-scheduling method, which we implemented as part of flexible AID, also outperforms `guided` across the board. To capture the overall benefits that come from AID scheduling methods, Figure 12 shows the average normalized performance delivered by standard OpenMP loop-scheduling methods (`static`, `dynamic`, and `guided`) with the *BS* mapping, and by AID methods across all OpenMP programs. As it is evident, AID strategies provide comparable or better performance for most programs (6% average improvement), achieving substantial gains for *blackscholes* (32.8%), *bfs* (24.5%), *pathfinder* (17.8%), or *cactusbssn_s* (60.2%).

AID performance numbers reported in Figure 10A–D correspond to program executions where threads are mapped to specific cores with the default Linux scheduler. We also experimented with the same programs running alone on the system on top of the HSP scheduler and using the per-thread `amp_prio` field to provide asymmetry-aware placement hints instead of mapping threads to specific core IDs. As discussed in Section 5.2, our asymmetry-aware scheduler takes the per-thread `amp_prio` values into consideration to decide which runnable threads of the application are mapped to big cores. Notably, `amp_prio` values constitute a soft version of strict CPU affinities, as threads with a higher "AMP priority" are not pinned to a particular CPU ID, but could be assigned to any big core. By assigning low `amp_prio` values (high priority) to threads with TID 0-7, and high `amp_prio` values (low priority) to threads with TID 8-15, the HSP scheduler indirectly achieves a mapping similar to *BS* when the application runs alone on the platform. Under these circumstances we observed very similar performance to that reported in Figure 10A–D for all programs across the various AID methods (average normalized performance in a 1.5% range). After all, once HSP places threads on the various core types in accordance with the associated `amp_prio` values, no further thread swaps are triggered; as pointed out in Section 6.1, HSP avoids triggering thread swaps across threads of the same OpenMP application for improved locality.

6.3 | Multi-application workloads with flexible AID

As stated in Section 5.2, original AID's implementation[11] only supports a single application running on the system, and requires imposing fixed thread-to-core-mappings (i.e., low-TID threads in the parallel program must be mapped to big cores). With these restrictions, the OS scheduler is not allowed to fully control thread-to-core assignments, which is crucial in multi-application scenarios. Flexible AID effectively addresses this issue, as no fixed thread-to-core-mappings are required. To illustrate flexible AID's full potential, we experimented with a set of multi-program workloads, whose composition is depicted in Table 3. Each randomly-generated mix includes four sequential programs from SPEC CPU, and one OpenMP application (the last program listed in the corresponding row). To utilize all the platform's cores when running the workload, the parallel application runs with 12 threads. In experimenting with these workloads, our main goals are (1) to evaluate the degree of throughput provided by HSP and Linux-ITD when OpenMP programs are present in the workload, and (2) to mimic scenarios where the number of big cores devoted to running the parallel application may vary over time, due to potential thread migrations triggered by the OS scheduler.

In our experiments we explored two different scenarios, referred to as *LS* and *HA*, where the OS scheduler entirely controls thread-to-core assignments dynamically (no CPU affinities are established for any program). In the *LS* (Linux-ITD + Standard OpenMP) scenario, the workload runs with the Linux's default scheduler with the Intel TD patch (Linux-ITD), and the parallel application uses the best-performing standard OpenMP loop-scheduling method (i.e., `static`, `dynamic` or `guided`) as identified in our previous round of experiments (Figure 10A–D). In the *HA* (HSP + AID) scenario, the workload runs on top of the HSP scheduler (BS variant)—which we extended with support for flexible AID—, and the parallel program is configured to use the best-performing AID method, as reported in Figure 10A–D. Moreover, to ensure consistent performance of the OpenMP program across runs, we assigned fixed `amp_prio` values to the various threads by using an environment variable. Particularly, for the

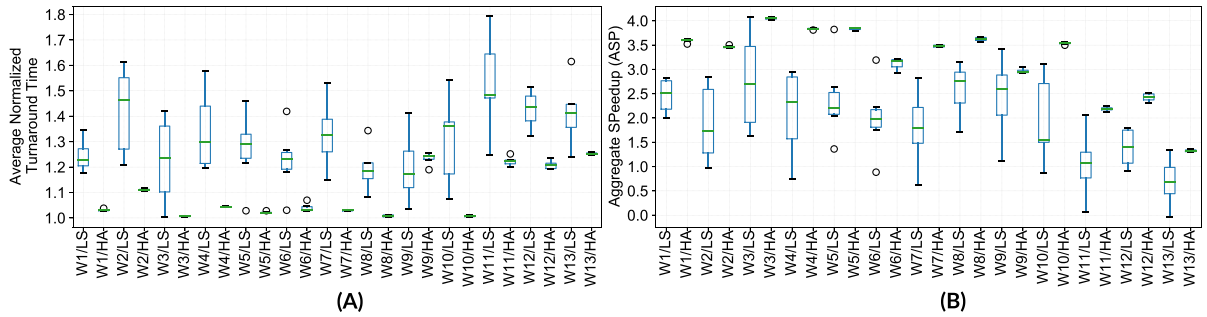


FIGURE 13 Distribution of (A) average normalized turnaround time (ANTT) and (B) throughput (ASP metric) for the various workloads under Linux-ITD with standard OpenMP (LS) and HSP with AID loop-scheduling methods (HA).

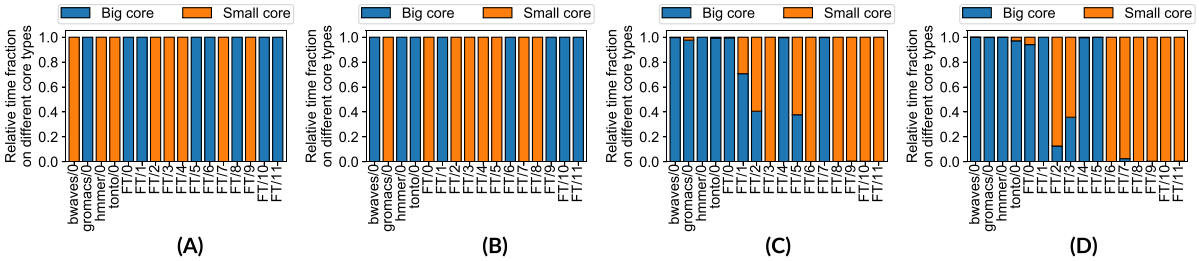


FIGURE 14 Relative time fraction that each thread on W2 spends on the different core types throughout its execution under the different OS schedulers. The labels found in the x-axis indicate the program's name and the TID, both separated by a slash. (A) Linux-ITD (Run1). (B) Linux-ITD (Run2). (C) HSP scheduler (Run1). (D) HSP scheduler (Run2).

master thread (TID = 0) `amp_prio` was set to 0 (maximum priority to be assigned to a big core); `amp_prio`'s values 1 and 2 were assigned to TIDs 1–7 and TIDs 8–11, respectively.

In running each workload we launch all the programs simultaneously. Because on our platform SPEC CPU programs take much longer to run than many of the OpenMP programs, we even out the completion times by running sequential programs for 10 billion instructions only, while conducting a full execution of the parallel program. In doing so we observed that some sequential applications may even terminate a bit earlier than the parallel program in some cases, potentially leaving idle big cores for running other threads in the workload. These situations make it possible to assess the ability of the system software to benefit from the available big cores throughout the workload's execution.

Figure 13 summarizes the results obtained for all workloads running under the LS and HA scenarios. To capture the huge variability provided by the Linux-ITD scheduler we ran each workload 10 times under both scenarios, and display the distribution of the lower-is-better ANTT metric and throughput (ASP metric) across all runs. As Figure 13A reveals, our approach (the HA scenario) provides consistent results across runs and systematically lower ANTT values than Linux-ITD, averaging a 16.8% reduction in ANTT. At the same time, HSP+AID outperforms Linux-ITD across the board in terms of throughput (as Figure 13B reveals), while delivering repeatable ASP figures across runs, as opposed to what Linux-ITD does. Note that the higher-is-better ASP metric[2] is also workload dependent; workloads whose threads exhibit a low average speedup factor normally lead to low ASP values. Despite this, HSP increases ASP by 68.5% on average, with respect to Linux-ITD.

To understand why our approach improves both ANTT and throughput w.r.t. Linux-ITD, and explain where the source of variability comes from on Linux, we recorded and analyzed the amount of time (in ticks) that all threads in each workload spend on big and small cores throughout the execution. As an illustrative example, Figure 14 shows the fraction spent by every thread on each core type registered on the first two runs for workload W2 under Linux-ITD and HSP. Clearly, Linux-ITD makes different thread-to-core assignments on both runs (see Figure 14A,B), and more importantly, it fails to effectively map all sequential programs to big cores, thus missing opportunities for substantial throughput improvements. By contrast, as Figure 14C,D reveal, HSP systematically devotes big cores to run explicitly serial code. Essentially, threads belonging to a single-threaded program exhibit a high *utility* value (see Equation 1), so, when active, they are always mapped to a big core. Moreover, the HSP scheduler places the master thread in the OpenMP application on a big core (thread `FT/0`), by catering to its low `amp_prio` value, which indicates a high preference to run on a big core.

Our experiments also corroborated that when a single-threaded application terminates under HSP, the scheduler effectively migrates a thread from the OpenMP program to the idle big core. This stands in contrast with Linux-ITD's behavior, which does not rapidly migrate threads to idle

big cores, but instead keeps threads on the core type where they were initially assigned for as long as possible (Figure 14A,B show no evidence of migrations across core types). Notably, under HSP the thread to be migrated upon a newly idle big core, is selected by catering its utility (speedup) and to the `amp_prio` value established by the runtime system; for swap selection among threads of the same application only `amp_prio` values are taken into consideration. Therefore, according to the aforementioned per-thread `amp_prio` settings, threads with TID 1-7 are more likely to be mapped to a big core by the scheduler, than those with TID 8-11. Under W2, we observed that `hmmer` and `tonto` terminate earlier than the parallel program, so HSP migrates threads with TID lower than 8 to big cores. This is reflected in Figure 14D, where threads with TID 2 and 3 (`FT/2` and `FT/3`) have a chance to run for a while on big cores (during the last part of the workload's execution); by contrast HSP assigns threads `FT/0`, `FT/1`, `FT/4`, and `FT/5` to big cores from the beginning of the execution, hence the larger big-core time fraction.

7 | CONCLUSIONS AND FUTURE WORK

In this article, we have presented PMCSched, an open-source framework for Linux that enables to implement the custom OS kernel support to assist novel scheduling strategies for multicore systems. A key distinctive feature of our framework is that it empowers developers and researchers to add new kernel-level scheduling-related support via a loadable module that can be inserted in vanilla (unmodified) versions of the Linux kernel. This favors the adoption of custom, and potentially sophisticated, scheduling strategies implemented at one or multiple levels of the system software stack.

To demonstrate the flexibility of the framework, we leveraged PMCSched's modular plugin-based design to implement different kinds of custom scheduling-related extensions, which were evaluated on an Intel Alder Lake (hybrid) multicore processor. First, we incorporated into Linux the support for the Intel Thread Director technology and carried out an experimental analysis that reveals its potential and key limitations on Alder Lake processors. Second, we implemented and evaluated several asymmetry-aware OS-level schedulers (i.e., AARR and HSP), which exploit PMCSched's flexible API to conduct thread migrations and seamlessly access hardware performance monitoring facilities. Third, we proposed a number of flexible AID methods that leverage the synergistic cooperation between the runtime system and the OS scheduler to improve performance in both single- and multi- application scenarios.

PMCSched is already publicly available at PMCTrack's source code repository;[48] technical documentation on our framework can be found on its official website.[54] As for future work, we plan on exploring the potential of PMCSched for the design and implementation of resource-management policies by leveraging its API for system-software guided cache-partitioning[71] on both symmetric and asymmetric multicore platforms. We also envision leveraging a similar approach to the OS-runtime interaction for other types of multithreaded programs, as well as to augment flexible AID's functionality with support for malleability. Another interesting research avenue would be the experimentation with future Intel hybrid processors with improved TD support that allows the direct and independent gathering of accurate per-thread big-to-small performance predictions from any core type.

ACKNOWLEDGMENTS

This work has been supported by the Spanish MCIN under Grant PID2021-126576NB-I00 (funded by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe") and also in part by Comunidad de Madrid under Grant S2018/TCS-4423.

DATA AVAILABILITY STATEMENT

Research data are not shared.

ENDNOTES

*On Linux, measuring the elapsed time is done very efficiently, as obtaining timestamps does not require regular syscalls but virtual ones (*vsyscalls*) or virtual dynamic shared object (vDSO) calls, which do not require entering the OS kernel.

†Using shared memory to accelerate the communication between different system software layers for improved scheduling is a widely used practice.[2,18,33]

ORCID

Carlos Bilbao https://orcid.org/0000-0002-4750-5124

Juan Carlos Saez https://orcid.org/0000-0003-1343-7108

Manuel Prieto-Matias https://orcid.org/0000-0003-0687-3737

REFERENCES

1. Hennessy JL, Patterson DA. A new golden age for computer architecture. *Commun ACM*. 2019;62(2):48-60.
2. Garcia-Garcia A, Saez JC, Prieto-Matias M. Contention-aware fair scheduling for asymmetric single-ISA multicore systems. *IEEE Trans Comput*. 2018;67(12):1703-1719.
3. Pricopi M, Muthukaruppan TS, Venkataramani V, Mitra T, Vishin S. Power-performance modeling on asymmetric multi-cores. *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. IEEE; 2013:1-10.

4. Saez JC, Prieto-Matias M. Evaluation of the Intel Thread Director technology on an Alder Lake processor. *13th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'22)*. Association for Computing Machinery; 2022:61-67.
5. Intel. 13th Generation Intel Core i9 Processors. https://ark.intel.com/content/www/us/en/ark/products/series/230485/13th-generation-intel-core-i9-processors.html
6. Kumar R, Tullsen D, Ranganathan P, Jouppi N, Farkas K. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. *Proceedings 31st Annual International Symposium on Computer Architecture (ISCA 04)*. IEEE; 2004:64-75.
7. Chronaki K, Moretó M, Casas M, et al. On the maturity of parallel applications for asymmetric multi-core processors. *J Parallel Distrib Comput*. 2019;127:105-115. doi:10.1016/j.jpdc.2019.01.007
8. Li T, Brett P, Knauerhase R, Koufaty D, Reddy D, Hahn S. Operating system support for overlapping-ISA heterogeneous multi-core architectures. *2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE; 2010:1-12.
9. Mittal S. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Comput Surv*. 2016;48(3):45:1-45:38.
10. Chronaki K, Rico A, Badia RM, Ayguadé E, Labarta J, Valero M. Criticality-aware dynamic task scheduling for heterogeneous architectures. *Proceedings of the 29th ACM on International Conference on Supercomputing*. Association for Computing Machinery; 2015:329-338.
11. Saez JC, Castro F, Prieto-Matias M. Enabling performance portability of data-parallel OpenMP applications on asymmetric multicore processors. *Proceedings of the 49th International Conference on Parallel Processing*. Association for Computing Machinery; 2020:1-11.
12. Koufaty D, Reddy D, Hahn S. Bias scheduling in heterogeneous multi-core architectures. *Proceedings of the 5th European Conference on Computer Systems*. Vol 10. Association for Computing Machinery; 2010:125-138.
13. Intel. Optimizing software for x86 hybrid architecture. Intel White Paper; 2021.
14. Che S, Boyer M, Meng J, et al. Rodinia: a benchmark suite for heterogeneous computing. *Proceedings of the 2009 IEEE International Symposium on Workload Characterization*. IEEE; 2009:44-54.
15. SPEC CPU 2017 Overview. What's New? 2019. https://www.spec.org/cpu2017/Docs/overview.html
16. Müller MSea. SPEC OMP2012—an application benchmark suite for parallel systems using OpenMP. *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*. Springer; 2012:223-236.
17. Jin HQ, Frumkin M, Yan J. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report NAS System Division: NAS-99-011; 1999.
18. Cho Y, Guzman CAC, Egger B. Maximizing system utilization via parallelism management for co-located parallel applications. *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. Association for Computing Machinery; 2018:1-14.
19. Joao JA, Suleman MA, Mutlu O, Patt YN. Utility-based acceleration of multithreaded applications on asymmetric CMPs. *40th Annual International Symposium on Computer Architecture*. Association for Computing Machinery; 2013:154-165.
20. Torng C, Wang M, Batten C. Asymmetry-Aware Work-Stealing Runtimes. *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*. IEEE; 2016:40-52.
21. Butko A, Bruguier F, Gamatié A, Sassatelli G. Efficient programming for multicore processor heterogeneity: OpenMP versus OmpSs. OpenSuCo; 2017:HAL Id: lirmm-01723762.
22. Nishikawa H, Shimada K, Taniguchi I, Tomiyama H. Energy-aware scheduling of malleable fork-join tasks under a deadline constraint on heterogeneous multicores. *SIGBED Rev*. 2019;16(3):57-62.
23. Servat H, Teruel X, Llort G, et al. On the instrumentation of OpenMP and OmpSs tasking constructs. In: Caragiannis I, Alexander M, Badia RM, et al., eds. *Euro-Par 2012: Parallel Processing Workshops*. Springer; 2013:414-428.
24. Salami B, Noori H, Naghibzadeh M. Online energy-efficient fair scheduling for heterogeneous multi-cores considering shared resource contention. *J Supercomput*. 2022;78(6):7729-7748.
25. Costero L, Igual FD, Olcoz K, Tirado F. Energy efficiency optimization of task-parallel codes on asymmetric architectures. *2017 International Conference on High Performance Computing & Simulation*. IEEE; 2017:402-409.
26. Garcia-Garcia A, Saez JC, Castro F, Prieto-Matias M. LFOC: a lightweight fairness-oriented cache clustering policy for commodity multicores. *Proceedings of the 48th International Conference on Parallel Processing*. Association for Computing Machinery; 2019:14:1-14:10.
27. Intel. User space software for Intel(R) resource director technology; 2022. Accessed October 5, 2022. https://github.com/intel/intel-cmt-cat
28. Mvondo D, Barbalace A, Lozi JP, Muller G. Towards user-programmable schedulers in the operating system kernel. Proceedings of the 11th Workshop on Systems for Post-Moore Architectures; 2022:4.
29. Calandrino JM, Leontyev H, Block A, Devi UC, Anderson JH. LITMUS-RT: a testbed for empirically comparing real-time multiprocessor schedulers. *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. IEEE; 2006:111-126.
30. Lepers B, Gouicem R, Carver D, et al. Provable multicore schedulers with Ipanema: application to work conservation. *Proceedings of the Fifteenth European Conference on Computer Systems*. Association for Computing Machinery; 2020:1-16.
31. Xu VML, McShane LW, Mossé D. LUSH: lightweight framework for user-level scheduling in heterogeneous multicores. *2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE; 2021:396-404.
32. Saez JC, Fedorova A, Prieto M, Vegas H. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors. *7th International Conference Computing Frontiers*. Association for Computing Machinery; 2010:31-40.
33. Harris T, Maas M, Marathe VJ. Callisto: co-scheduling parallel runtime systems. *Proceedings of the 9th European Conference on Computer Systems*. Association for Computing Machinery; 2014:1-14.
34. Saez JC, Pousa A, Castro F, Chaver D, Prieto-Matias M. Towards completely fair scheduling on asymmetric single-ISA multicore processors. *J Parallel Distrib Comput*. 2017;102:115-131.
35. Neri R. Intel kernel patches posted to kernel mailing list. Accessed December 19, 2022. https://lore.kernel.org/lkml/20220909231205.14009-1-ricardo.neri-calderon@linux.intel.com/
36. Lozi JP, Lepers B, Funston JR, Gaud F, Quéma V, Fedorova A. The Linux scheduler: a decade of wasted cores. *Proceedings of the 11th ACM European Conference on Computer Systems (Eurosys'16)*. Association for Computing Machinery; 2016:1-16.
37. Blagodurov S, Fedorova A, Zhuravlev S, Kamali A. A case for NUMA-aware contention management on multicore systems. *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE; 2011:557-558.

38. Feliu J, Sahuquillo J, Petit S, Duato J. Perf&Fair: a progress-aware scheduler to enhance performance and fairness in SMT multicores. *IEEE Trans Comput*. 2017;66(5):905-911.
39. The LITMUS-RT kernel; 2019. Accessed March 14, 2023. https://github.com/LITMUS-RT/litmus-rt
40. Barbalace A, Lyerly R, Jelesnianski C, et al. Breaking the boundaries in heterogeneous-ISA datacenters. *ACM SIGPLAN Notices*. Vol 52. Association for Computing Machinery; 2017:645-659.
41. Popcorn Linux kernel; 2019. Accessed March 14, 2023. https://github.com/ssrg-vt/popcorn-kernel
42. Weaver VM. Linux perf event features and overhead. FastPath Workshop; 2013.
43. Lyerly R, Bilbao C, Min C, Rossbach CJ, Ravindran B. An OpenMP runtime for transparent work sharing across cache-incoherent heterogeneous nodes. *ACM Trans Comput Syst*. 2022;39:1.
44. Saez JC, Pousa A, Rodriíguez-Rodriíguez R, Castro F, Prieto-Matias M. PMCTrack: delivering performance monitoring counter support to the OS scheduler. *Comput J*. 2017;60(1):60-85.
45. Saez JC, Casas J, Serrano A, et al. An OS-oriented performance monitoring tool for multicore systems. In: Hunold S, Costan A, Giménez D, et al., eds. *Euro-Par 2015: Parallel Processing Workshops*. Springer; 2015:697-709.
46. Rostedt S. ftrace: where modifying a running kernel all started. https://kernel-recipes.org/en/2019/talks/ftrace-where-modifying-a-running-kernel-all-started/
47. Linux. Using the Linux Kernel Tracepoints; Accessed March 14, 2023. https://www.kernel.org/doc/html/latest/trace/tracepoints.html
48. PMCTrack. Github repository; 2015. https://github.com/jcsaezal/pmctrack
49. Kroah-Hartman G. Linux 5.9.7; 2020. Accessed March 14, 2023. https://lwn.net/Articles/836772/
50. Rostedt S. ftrace: Handle tracing when switching between context; 2020. Accessed March 14, 2023. https://www.spinics.net/lists/stable/msg426317.html
51. Rostedt S. ftrace: handle tracing when switching between context; 2020. Accessed March 14, 2023. https://www.mail-archive.com/linux-kernel@vger.kernel.org/msg2362422.html
52. Latest Linux kernel version for Odroid XU4 board; 2019. Accessed March 14, 2023. https://github.com/hardkernel/linux/tree/odroid-5.4.y
53. AMD. AMD64 technology platform QoS extensions. https://developer.amd.com/wp-content/resources/56375.pdf
54. Bilbao C, Saez JC. PMCSched website: documentation and examples; 2022. Accessed December 23, 2022. https://github.com/Zildj1an/pmcsched-website
55. The open-source community. The rust for Linux project; 2022. Accessed December 23, 2022. https://github.com/Rust-for-Linux
56. Intel. Intel® 64 and IA-32 architectures software developer's manual Vol. 3: system programming guide; 2021. Accessed March 15, 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html
57. Neri R. Intel kernel patches posted to kernel mailing list. Accessed December 19, 2022. https://lore.kernel.org/lkml/20220108013038.GB19633@ranerica-svr.sc.intel.com/T/
58. Rotem E, Yoaz A, Rappoport L, et al. Intel Alder Lake CPU architectures. *IEEE Micro*. 2022;42:13-19.
59. Bilbao C, Saez JC, Prieto-Matias M. Rapid development of OS support with PMCSched for scheduling on asymmetric multicore systems. In: Singer J, Elkhatib Y, Blanco Heras D, Diehl P, Brown N, Ilic A, eds. *Euro-Par 22: Parallel Processing Workshops*. Springer; 2022:184-196.
60. BSC. Paraver: A Flexible Performance Analysis Tool; 2018. Accessed March 19, 2019. https://tools.bsc.es/paraver
61. Yasin A, Haj-Yahya J, Ben-Asher Y, Mendelson A. A metric-guided method for discovering impactful features and architectural insights for Skylake-based processors. *ACM Trans Archit Code Optim*. 2019;16(4):46.
62. Van Craeynest K, Jaleel A, Eeckhout L, Narvaez P, Emer J. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). *39th Annual International Symposium on Computer Architecture (ISCA 12)*. IEEE; 2012:213-224.
63. Pousa A, Saez JC, Castro F, Chaver D, Prieto M. Theoretical study on the performance of an asymmetry-aware round-robin scheduler. TR – 5028A. Department of Computer Architecture, UCM; 2012. https://pmctrack-linux.github.io/assets/papers/dacya-tr5028A.pdf
64. Becchi M, Crowley P. Dynamic thread assignment on heterogeneous multiprocessor architectures. *3rd International Conference Computing Frontiers (CF 06)*. Association for Computing Machinery; 2006:29-40.
65. Shelepov D, Alcaide JCS, Jeffery S, et al. HASS: a scheduler for heterogeneous multicore systems. *Oper Syst Rev*. 2009;43(2):66-75.
66. Van Craeynest K, Akram S, Heirman W, Jaleel A, Eeckhout L. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. *22nd International Conference Parallel Arch. Compilation Techniques (PACT 13)*. IEEE; 2013:177-187.
67. Zhang Y, Duan L, Li B, Peng L, Sadagopan S. Cross-architecture prediction based scheduling for energy efficient execution on single-ISA heterogeneous chip-multiprocessors. *Microprocess Microsyst*. 2015;39(4):271-285.
68. Selfa V, Sahuquillo J, Eeckhout L, Petit S, Gómez ME. Application clustering policies to address system fairness with Intel's cache allocation technology. *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE; 2017:194-205.
69. Suzuki T, Nukada A, Matsuoka S. Efficient execution of multiple CUDA applications using transparent suspend, resume and migration. *Euro-Par 2015: Parallel Processing*. Springer; 2015:687-699.
70. Butko A, Bessad L, Novo D, et al. Position paper: OpenMP scheduling on ARM big.LITTLE architecture. MULTIPROG: Programmability and Architectures for Heterogeneous Multicores; 2016: HAL Id: lirmm-01377630.
71. Saez JC, Castro F, Fanizzi G, Prieto-Matias M. LFOC+: a fair OS-level cache-clustering policy for commodity multicore systems. *IEEE Trans Comput*. 2022;71(8):1952-1967. doi:10.1109/TC.2021.3112970

How to cite this article: Bilbao C, Saez JC, Prieto-Matias M. Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake. *Concurrency Computat Pract Exper*. 2023;35(25):e7814. doi: 10.1002/cpe.7814

Chapter 7

Conclusions

The continuous reduction in transistor size, a trend since the earliest days of computing [43], is now approaching fundamental physical limits [68]. In response to these and other challenges (collectively referred to as the Power and Memory Walls), the semiconductor industry increasingly adopted Chip Multicore Processors (CMPs), which integrate multiple cores on the same platform. Over time, CMPs gained popularity and have now become the dominant general-purpose architecture in a wide spectrum of commercial products, ranging from mobile platforms to server systems for both cloud computing and HPC. Notwithstanding the benefits of multicore systems, they pose significant challenges to the system software. In this thesis, we focus on tackling three major challenges:

1. **Addressing shared-resource contention**. Applications running simultaneously on a CMP system typically compete for the use of key shared resources among cores such as the last-level cache (LLC), DRAM controllers, and I/O and memory bandwidth. Unfortunately, this competition may lead to contention, which usually has a negative and uneven impact on the performance of co-running applications [24, 45, 57, 152, 167, 171]. Contention complicates enforcing quality of service (QoS) [34], delivering fairness [136], and providing strict performance guarantees to specific applications [174]. Notably, performance degradation caused by contention is aggravated even further in NUMA multicores, particularly due to the increased latency of inter-node memory accesses [172] and interconnect contention [42].

   As part of this thesis, we aimed to tackle shared-resource contention on NUMA systems by combining, for the first time in a dynamic resource manager, the following two mitigation techniques: *contention-aware thread placement* [26, 34, 95, 171, 178] and *LLC-partitioning* [34, 46, 122, 136]. The latter technique has gained increasing interest both in the research community [34, 101, 136] and in the industry [92], due to the fairly recent adoption of specific hardware support in commercial platforms, such as the Intel's Resource Director Technology (RDT) [2] and the AMD's Technology Platform QoS extensions [11]. We wished to shed light on how to best combine these

mechanisms – something that existing management strategies have failed to do, leading to suboptimal fairness – and to develop the necessary system software support to automate this process in a NUMA-aware fashion.

2. **Maximizing CPU usage**. Maximizing resource utilization is essential to cloud providers for reducing utility costs and increasing revenue [7]. While colocating several applications on the same physical server can lead to increased CPU usage, the irregular diurnal patterns of current cloud workloads still make it challenging to eschew resource idleness, even with colocation. To address this problem, cloud providers leverage the execution of in-house workloads during periods of resource underutilization [33, 34, 92, 116, 175].

   This thesis explores a complementary approach to maximizing CPU usage that exploits the elastic execution of HPC/scientific workloads. An appealing property that most of these workloads exhibit is their inherent parallelism, which allows them to benefit from multiple cores throughout a vast portion of its execution. We leverage this observation to fully utilize idle core cycles present in the system – even during short time periods – by opportunistically running extra threads from HPC/scientific workloads. In this thesis, we have explored ways to achieve this without making changes in the applications, and by leveraging both OS support and explicit interaction between the OS and the runtime system.

3. **Scheduling on asymmetric multicore processors**. Today, multicore processors present in commercial server platforms consist of identical cores. This stands in contrast with the desktop market segment, where asymmetric multicore processors (AMPs) are rapidly gaining traction. Take for instance, popular commercial products such as Intel's Alder Lake and Raptor Lake processor families, [72] or Apple's M1 to M4 system on chips (SoCs) [14,15]. AMPs combine multiple core types with diverse performance and power characteristics while sharing a common Instruction Set Architecture (ISA). The shared ISA allows users to run unmodified software that was not explicitly built to leverage the performance asymmetry of these platforms.

   A large body of related work over the last two decades has widely showcased both the benefits and barriers of exploiting AMPs [13, 21, 58, 62, 66, 69, 81, 82, 102, 144, 147, 166]. Particularly, attempting to automatically bring the benefits of AMPs to unmodified applications (what we do in this thesis), gives rise to a number of scheduling-related challenges for the system software. Many of these challenges stem from the wide diversity in the benefit that various applications derive from utilizing big cores, compared to using only small cores.

   To advance the state of the art in asymmetry-aware scheduling, we explore ways to maximize system throughout via asymmetry-aware loop scheduling and by leveraging explicit interaction between the runtime system and the OS scheduler. Moreover, this thesis evaluates the effectiveness of the Intel's Thread Director technology, a hardware feature that provides the OS scheduler with hints to facilitate thread placement decisions at runtime on AMPs [16, 131, 146]. First introduced in the Alder Lake processor family, this

technology estimates the relative performance and energy consumption of running individual threads on the various core types.

To take on the first challenge, this thesis proposes *Divide&Content (DC), a fair OS-level resource manager for contention balancing on NUMA multicores*. DC leverages performance monitoring counters (PMCs) and other hardware monitoring facilities available on off-the-shelf multicore servers to measure per-application bandwidth consumption and classify programs based on their performance sensitivity to the LLC utilization. Based on this information, DC dynamically places and migrates threads across the various sockets of a NUMA platform, while applying cache-partitioning so as to optimize system-wide fairness. Thread placement and LLC-partitioning are performed such that the pressure on the LLC and the memory channels of each socket is balanced across NUMA nodes. Our comprehensive experimental evaluation of DC shows that it reduces unfairness by more than 17% on average, compared to both the Linux kernel and a state-of-the-art NUMA-aware resource manager [26].

The main takeaways drawn from our research on addressing contention balancing on NUMA multicores are as follows:

- In optimizing system-wide fairness or throughput, thread-to-core placement and LLC-partitioning cannot be treated as orthogonal problems. Rather, they must be applied in a rationally coordinated manner that guarantees a balanced degree of contention across sockets. Specifically, when multiple memory-intensive applications are included in the workload, this is accomplished by enforcing a similar pressure on the per-socket LLC and memory channels throughout the system. We reached this conclusion after conducting an extensive simulation-based analysis that considers various theoretical placement and partitioning strategies.

- Unlike what was often tacitly assumed in prior work [34, 92, 133], simply extending any existing LLC-partitioning policy to NUMA systems by using independent per-socket partitioning leads to suboptimal solutions. To be effective, these techniques must be complemented with a smart contention-aware thread placement policy, like the one that DC leverages, and must undergo non-trivial extensions so as to operate in synergy with the placement policy.

- Current hardware facilities present in commercial systems for monitoring cache-related behavior (e.g., Intel RDT [2]) and the performance of the various applications (i.e., PMCs), can be efficiently exploited to guide LLC-partitioning and thread-placement decisions at the OS level. Our experimental evaluation supports this observation, leveraging workloads consisting of long-running HPC and scientific applications.

To complement existing proposals that maximize resource usage on multicore architectures – thereby addressing the second challenge of this thesis – we propose an *elasticity framework to increase CPU usage on CMPs*. This framework consists of

an OS-level elasticity manager (EM) and specific runtime system support to exploit malleability (i.e., the ability to adjust the number of active worker/threads processes in an application at runtime). Our proposal continuously monitors the system's utilization to detect idle core cycles while a mix of unmodified applications runs on the system. In particular, the EM temporarily increases the number of cores assigned to HPC/scientific applications with malleability support when other applications leave cores idle, even for short time periods. Similarly, cores "stolen" from other applications are automatically reclaimed by the OS when these applications begin to exhibit a higher CPU utilization. This approach makes it possible to increase system throughput and accelerate HPC/scientific applications in an opportunistic fashion. To realize this potential without negatively impacting application performance, our proposal combines OS-level extensions with runtime system support, both operating in full coordination. In this thesis, we have explored ways to deliver malleability transparently to unmodified applications. To demonstrate this potential, we have designed a number of malleability extensions for the OpenMP runtime system.

The main conclusions that can be extracted from the design and experimental evaluation of our elasticity manager are as follows:

- An application's degree of CPU utilization, as exposed by GNU/Linux both at the kernel level and user-space, does not constitute a reliable metric to quantify the number of cores that the application uses over time. Particularly, it tends to under-predict the core count utilized during a given monitoring period. This makes Linux's CPU utilization an inaccurate metric to estimate the number of idle cores on the system. To obtain a realistic prediction of the number of cores that the application uses over time, our OS-level EM relies on another metric that factors in the amount of time that it spends with a different number of runnable threads. Crucially, this information can only be monitored efficiently from the OS kernel, where the EM operates.

- Our experimental analysis reveals that, in opportunistically leveraging idle core cycles, our proposal delivers substantial throughput improvements with respect to the unmodified Linux kernel. Furthermore, the EM boosts the performance of malleable – yet unmodified – loop-based OpenMP applications by up to 84%. These benefits are reaped without negatively impacting the performance of the remaining (non-malleable) applications, whose performance remains within a 1.5% range of that delivered by the Linux kernel.

- Leveraging oversubscription constitutes a complementary, yet naive, approach to increment CPU utilization and accelerate unmodified HPC/scientific workloads. However, as opposed to our proposal, this approach severely degrades the performance of the applications that leave idle cores, as the Linux kernel itself fails to efficiently reclaim "stolen" cores.

- The malleability extensions we implemented in the GNU's OpenMP runtime system deliver transparent malleability to a wide range of well-known regular OpenMP benchmarks. These extensions engage fine-grained malleability

actions when scheduling iterations in parallel loops. This serves as a proof of concept for automatic malleability support that can be adopted in other runtime systems. Moreover, our approach to malleability can also be exploited to turn existing applications into malleable ones by altering their source code.

To address the third challenge of this thesis, we proposed *a set of scheduling extensions to maximize throughput on asymmetric multicore systems*. These extensions were evaluated on a 16-core Intel Alder Lake processor. Our main contributions to asymmetry-aware scheduling include (i) a comprehensive experimental evaluation of the Intel Thread Director (TD) technology in the Linux kernel, (ii) a set of loop-scheduling methods for AMPs that builds upon previous work [138], and (iii) an OS-runtime cooperation strategy for multi-application scenarios that leverages asymmetry-aware thread placement together with loop scheduling. The main takeaways that come from the design and evaluation of the proposed scheduling extensions can be summarized as follows:

- The TD implementation in Alder Lake processors [131] produces rather inaccurate predictions of a thread's big-to-small speedup at runtime. This, coupled with the fact that TD-aided speedup predictions are not available for small cores [146], makes current TD implementations very limited in assisting the OS scheduler with effective thread placement decisions. In this thesis, we have demonstrated that an OS-scheduler that relies on existing PMC-based prediction models to estimate the speedup factor [146] yields substantially higher throughput and more consistent performance from run to run than a scheduler relying on TD-based speedup estimates.

- Existing loop-scheduling methods of the OpenMP standard fail to efficiently execute unmodified OpenMP programs on AMPs. In this thesis, we proposed a number of flexible asymmetric-iteration distribution methods – referred to as Flexible-AID – that leverage core-type awareness to decide how many loop iterations each worker thread runs. Our experimental evaluation shows that Flexible-AID methods outperform the standard OpenMP loop-scheduling strategies and offer a variety of scheduling options to cater to diverse OpenMP applications, such as those included in popular benchmark suites like Rodinia, PARSEC, and NAS Parallel Benchmarks.

- Another key feature of Flexible-AID's methods is that they exploit the synergistic cooperation between the runtime system and the OS scheduler, facilitating decision-making via information exchange through shared memory. On the one hand, the OS informs the runtime system on the core type where each worker thread runs; this is critical in scenarios where the OS scheduler may trigger thread migrations (i.e., CPU affinities are not imposed by the user). On the other hand, under Flexible-AID, the user and/or runtime system can also provide hints to the OS scheduler regarding which worker threads could derive a higher benefit from running on a big core. In this thesis, we showcased the effectiveness of this bidirectional cooperation via a custom OS scheduler implementation that interacts with an OpenMP runtime system enhanced

with Flexible-AID's extensions. Our experiments show that this cooperation is paramount to maximize system throughput in multi-application scenarios.

Clearly, all the proposals in this thesis have involved the design and implementation of scheduling or resource-management policies at the OS level, more specifically, within the Linux kernel. As physical barriers hinder hardware improvements [68, 90], the OS has become an increasingly appealing layer for optimization. Indeed, throughout this thesis, the various OS-level strategies proposed were essential in facing the major scheduling-related challenges considered in this thesis: mitigating shared-resource contention, increasing resource utilization, and addressing performance asymmetry. Devising these strategies – and, more importantly, implementing them in the OS in a timely fashion – introduced a key transversal objective in the early stages of this thesis: *overcoming the challenges of the time-consuming and steep learning curve inherent in the development within the Linux kernel and its interaction with upper system software layers.*

This goal led to the creation of PMCSched, an open-source framework integrated into the PMCTrack tool [145] that enables rapid development of scheduling and resource management strategies on Linux via an extensive loadable kernel module. Throughout each use case in this thesis, PMCSched underwent significant improvements to cater to diverse research scenarios, making the current version considerably more capable than the original. The release of PMCTrack version 3.0 in January 2023 [3, 145] marked a significant milestone, by incorporating the PMCSched framework as one of PMCTrack's new subsystems. PMCSched's official website [23] provides detailed documentation along with other online resources with technical information. All in all, the use of PMCSched across the research avenues in this thesis clearly demonstrated its value – namely, it illustrated the benefits of simplifying development both at the OS level and through coordination between the runtime system and the OS scheduler. Consequently, PMCSched stands as one of the main contributions of this thesis.

7.1 Future Work

A number of future research avenues arise as a natural continuation of the proposals of this thesis. In particular, we find especially promising the following research lines:

- **Augmenting PMCSched for handling container-based workloads and virtual environments**. The latest public release of PMCSched allows the creation of plugins that manage the various threads running on the system in much the same way as Linux's scheduler does; this is, by handling a set of task structures from one or multiple single-threaded or multithreaded processes. This system view helps plugins' developers create schedulers and resource managers for workloads that run natively on top of the OS, but it does not ease the development at higher-level abstractions like containers or virtual machines (VMs). Adding data types and facilities to PMCSched for simplifying

the management of containerized/virtualized workloads would strategically extend it. This would facilitate the rapid exploration of novel scheduling algorithms and resource management strategies specifically targeted to server platforms in cloud datacenters, where containers and VMs are massively exploited [113, 154].

When dealing with virtualized environments, another significant research challenge that PMCSched could address is bridging the *semantic gap* under multiple scheduling agents. This problem (also commonly referred to as *double scheduling*) arises in virtualized environments where at least two schedulers exist: one for the host OS and another within the guest VM. The term semantic gap refers to the lack of awareness between these schedulers – for example, the guest scheduler may be unaware of the NUMA topology of the underlying host, while the host scheduler lacks critical information, such as whether a virtual CPU (vCPU) in the guest has acquired a lock [32, 64].

- **Memory-bandwidth allocation for resource management**. Following a similar philosophy as DC, we could further reduce memory-bandwidth contention among applications by leveraging recent hardware extensions for bandwidth limitation/capping, which are implemented in commodity server processors [114] – such as Intel's Memory Bandwidth Allocation technology [1] or AMD's Platform Quality of Service Enforcement [2]. Given this, devising resource-management strategies that leverage cache-partitioning and dynamic memory-bandwidth allocation is an interesting avenue for future work.

- **Resource management on Multi-Chip Module (MCM) processors with several NUMA nodes per socket**. All Intel multicore processors used to experimentally assess the effectiveness of the various thesis proposals follow a monolithic design, including a single computing die per chip with multiple cores sharing a single LLC. This stands in contrast with Multi-Chip Module (MCM) processors, like recent AMD server processors, which exhibit multiple NUMA nodes per socket. A clear example of this particular MCM design is the AMD EPYC 7742 processor (presented in Section 2.1), which integrates multiple dies into a single package, and consists of multiple sets of 4-core groups, each sharing a separate LLC (L3) cache. Not only does this processor allow to partition the L3 cache of each 4-core group independently, but it also enables users to statically partition the different sets of cores and memory-channels into separate resource pools – exposed to the OS as separate NUMA nodes within the socket (see Section 2.1). This architecture's unique features give rise to numerous resource-management challenges and opportunities, opening the door to the exploration of novel proposals.

- **Addressing contention under memory expansion and disaggregation**. The CXL (Compute Express Link) protocol paves the way to deliver a

[1] `https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-memory-bandwidth-allocation.html`

[2] `https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24593.pdf`

seamless interaction among several heterogeneous computing resources (e.g., CPUs, TPUs, and GPUs) via a high-speed interconnect. It also allows access to extra memory resources in servers in a cache-coherent way. Particularly, CXL memory devices (Type 3 devices) expand sever memory capacity and bandwidth using CXL memory expansion cards or *memory disaggregation*, namely, accessing reserved memory banks placed in an off-node memory pool shared by multiple servers [96, 119, 151]. This approach helps mitigate the impact of the *memory wall* and reduces the total cost of ownership for servers in cloud datacenters [86]. The additional bandwidth that can be dynamically used with CXL memory devices contributes by itself to mitigating bandwidth contention, as separate memory channels are employed to access local and remote (CXL) memory banks. However, effectively exploiting these extra memory resources via dynamic memory allocation and page migration constitutes a substantial challenge for the system software [96], which fosters the design of future innovative resource-management strategies.

- **Design of contention-conscious resource managers for the cloud**. The evaluation of the various proposals in this thesis mainly involved workloads that consist of long-running HPC workloads and other applications from popular benchmark suites (like SPEC CPU or PARSEC), typically used in computer architecture research. In these applications, performance is measured exclusively by tracking the program's completion time under different settings. This stands in contrast with many cloud workloads, where avoiding quality of service (QoS) violations is a more critical concern that raw performance [92, 113]. Avoiding these violations usually strives to ensure that a service's tail latency does not exceed a certain threshold, or that an application is capable of processing a given (minimal) number of requests per second [34, 175].

  We strongly believe that PMCSched could also be used for rapid prototyping of schedulers and resource managers for cloud workloads. Such strategies could help enforce Service Level Agreements (SLAs) and reduce QoS violations. In the context of datacenters composed of multiple servers that enable virtual environments, PMCSched's functionality could also be expanded to make room for novel plugins that integrate with higher-level coarse-grained approaches, which primarily focus on VM-to-node mappings [154] but that do not directly address the effects of program phases [73, 74]. A PMCSched plugin could potentially trigger fine-grained resource-management actions to complement the decisions of a coarse-grained strategy.

- **Holistic shared-resource contention management**. In this thesis, we have primarily focused on addressing LLC contention; however, competition for other critical shared resources also plays a significant role in the overall system throughput and fairness. Zooming out from the cache hierarchy, contention may arise in I/O subsystems, leading to competition for disk access [92, 116] and network bandwidth [8, 175]. Conversely, zooming in on finer-grained resource sharing in the processor package, Simultaneous Multithreading (SMT) – which is not studied in this thesis – also constitutes a remarkable

source of contention, as stated in Section 2.1. In SMT-enabled CPUs, instruction streams from various threads execute on a single core, increasing CPU resources utilization and system throughput, especially when pairs of "compatible" threads (e.g., a memory-intensive thread alongside a CPU-intensive one) are co-scheduled [52,156]. Unfortunately, threads mapped to the same physical SMT-enabled core may compete heavily for pipeline resources (e.g., functional units) as well as for space and bandwidth in private cache levels (L1 and L2 in most commercial SMT-enabled server CPUs). Mitigating this source of contention requires specialized scheduling policies [52, 93, 106, 141, 153, 156]. All in all, the design of all-encompassing contention-aware strategies that holistically manage multiple shared resources – ranging from intra-core resources all the way to network and I/O bandwidth – constitutes a very promising research avenue.

- **Sophisticated interaction mechanisms between different software layers**. Some of the proposals of this thesis relied on the explicit interaction between the OS scheduler and the OpenMP runtime system. This interaction made it possible to deliver substantial performance gains to OpenMP applications, underscoring the enormous potential associated with OS-runtime cooperation, which surprisingly has not received much attention in the last decade. As for future work, we will leverage PMCSched's support to explore the design of more sophisticated OS-runtime interaction mechanisms. Beyond natural extensions of our research (e.g., extending Flexible AID loop-scheduling methods with malleability support), we aim to explore OS-aided elasticity for other application types, such as MPI programs or parallel ML inference engines.

In addition to the proposals that naturally follow from this thesis, there are other intriguing open problems in system software security that we consider particularly promising for future research. Among these are the following:

- **Resource management for confidential computing**. A notable area of current focus in systems research is Confidential Computing [39], which aims to address the security concerns of virtual guests in cloud environments by preventing unauthorized access to memory or registers by the virtual host. Confidential computing safeguards data-in-use (what customers execute), complementing traditional encryption methods for data-at-rest (e.g., disk encryption) and data-in-transit (e.g., trusted communication channels). Hardware extensions such as AMD's Secure Processor [76] can provide assurances of confidentiality and integrity for the virtualized guest. In contrast, availability aimed at preventing Denial-of-Service (DoS) attacks is not a primary objective in this context, as the host machines maintain control over the life cycle of the virtual guests and the resources available to them – including memory and virtual CPUs. At this early stage, all major hardware vendors are actively developing system software support for confidential computing [76, 80, 149], overcoming technical challenges such as the inclusion of patches for AMD's Secure Nested Paging [5, 132] in the Linux kernel.

Once such technical obstacles are addressed, the focus will likely shift to mitigating the inefficiencies introduced by the additional security steps and encryption/decryption processes. For instance, it is reasonable to expect that live migration, the process of transferring a VM guest to a different host without rebooting, will be significantly impacted by the encryption requirements and verification processes, such as attesting that the destination firmware is appropriate given the security requirements. In such scenarios, smart resource management could play a critical role in improving the user experience. Following the example of live migration, if the guest continues to execute on the original host machine until all memory is transferred (this is known as the *pre-copy* phase), any writes to an already migrated page will mark it as 'dirty'. This is done to label pages modified by a local processor that have not been synchronized and written back to the remote machine [94], needing additional data transfers and negatively affecting the duration of the migration process. Therefore, the insights gained in this thesis regarding profiling application memory behavior could guide decisions on prioritizing the migration order of encrypted memory pages to the destination machine.

- **Large Language Models (LLMs) in OS security**. While the application of AI, especially machine learning mechanisms, is not new in the context of OSs [44, 75, 146, 168], it's important to consider the immense variability of computing systems [73, 74]. In essence, integrating AI into system software may introduce the risk of unexpected outcomes. At this level of development, there is a need to address unforeseen consequences of actions taken by trained models, which may only become apparent when they are irreversible or cause system crashes. This risk is particularly pronounced in monolithic systems like the Linux kernel, where components are deeply interconnected.

  Combining AI with controlled system software environments, to avoid such variability, is an interesting avenue for future research in operating systems. System sandboxing [63,173] (a behavioral-based virus detection technique that logs actions and effects) is an exciting example, since it could be combined with LLMs to verify the detection of vulnerable or malicious source code. In the context of antivirus software, a false positive occurs when the antivirus mistakenly identifies a harmless application as malicious or infected with malware [20, 55]. There are antiviruses of very diverse nature (some leverage machine learning algorithms, others apply source code hashing, etc.) so false detection can occur due to many reasons, including but not limited to imperfect heuristics, software with a signature that closely resembles one from a malware database, unfamiliar applications, packaged software, and erroneous software updates. Such false positives can be a significant burden for both users and developers. The users may need to quarantine their system, causing disruption and inconvenience [40, 126]. For developers, false positives may generate distrust in their products and business, as well as increase their workload with the reviews of reports of false positives. Although current antivirus software may mistakenly flag non-existent threats, they typically provide mechanisms for users to report false positives directly. However, this process requires the

user to report to the vendor that will later have to investigate.

To mitigate false positives, LLMs could potentially verify the validity of system software antivirus decisions in real-time with automatically generated source code. In particular, a trained LLM model could create custom controlled system software environments, such as VMs or containerized solutions (e.g., Docker), tailored specifically to test the flagged software. Within this controlled environment, the LLM could assess whether the flagged application is genuinely malicious or infected. Whereas antivirus engines primarily apply signature-based scanning [169], behavioral analysis, and similar tools, LLMs could complement these efforts by exploring intricate relationships between system software components (e.g., attempts to access sensitive system files) and generating targeted ad-hoc tests, potentially effective even against zero-day OS vulnerabilities.

Appendix A

Rapid development of OS support with PMCSched for scheduling on asymmetric multicore systems

Full citation

Carlos Bilbao, Juan Carlos Saez, Manuel Prieto-Matias, *Rapid Development of OS Support with PMCSched for Scheduling on Asymmetric Multicore Systems*. Euro-Par 2022: Parallel Processing Workshops. Also part of the book series Lecture Notes in Computer Science (LNCS, volume 13835).

Abstract

Asymmetric multicore processors (AMPs) couple high-performance big cores and power-efficient small ones, all exposing a shared instruction set architecture to software, but with different microarchitectural features. The energy efficiency benefits of AMPs together with the general-purpose nature of the various cores, have led hardware manufactures to build commercial AMP-based products, first for the mobile and embedded domains, and more recently for the desktop market segment, as with the Intel Alder Lake processor family. This indicates that AMPs may become a solid and more energy efficient replacement to symmetric multicores in a wide range of application domains.

Previous research has demonstrated that the system software can substantially improve scheduling – critical to get the most out of heterogeneous cores – by leveraging hardware facilities that are directly managed by the the OS, such as performance monitoring counters, or the recently introduced Intel Thread Director technology. Unfortunately, the OS-level support enabling to access scheduling-relevant hardware support may take a long time to be adopted in operating systems, or may

come in forms that make its utilization challenging from specific levels of the system software stack, especially in production systems. To fill this gap, we propose the PMCSched framework, which enables the creation of custom OS support on Linux to aid in the design of novel scheduling and resource-management policies for multicores implemented at different layers of the system software, but without requiring to patch the kernel. To demonstrate the potential of our framework, we implement a set of OS-level schedulers for AMPs, that make use of custom OS extensions to access scheduling-relevant hardware facilities in an x86 AMP processor.

Differences with the associated journal article

Note that the journal article found in Chapter 6 is an extended version (special issue of journal) of this earlier workshop paper. While the workshop paper included in this chapter identified key issues regarding system software scheduling for AMPs, several aspects prompted further analysis and experimentation. The primary differences between this earlier submission and the special journal issue are as follows:

1. **Limited analysis of Intel Thread Director**: In the workshop paper, we provided a brief description of Intel's Thread Director (TD) technology, constrained by space limitations. This technology was utilized by one of the asymmetry-aware OS-level schedulers we evaluated. Later, Intel disclosed additional information about TD [131], along with the release of Linux kernel patches [108]. Consequently, the journal submission included a detailed survey on TD, offering an in-depth examination of its inner workings. It also evaluated its current support for Linux and presented experimental evidence demonstrating its limitations on Intel Alder Lake platforms, based on data gathered with the PMCSched framework.

2. **OS-level scheduling for AMPs**: This workshop paper exclusively focused on asymmetry-aware OS-level schedulers; albeit it also mentioned the intention to explore scheduling strategies leveraging the synergistic cooperation between the OS and a runtime system on AMPs. In the journal submission, we realized this concept by delving into OpenMP loop-scheduling and OS-runtime interaction. Specifically, we envisioned ways to improve the performance of loop-schedulers for OpenMP programs on asymmetric systems, which gave rise to our Flexible AID proposal. Flexible AID also leverages OS-runtime interaction to bring performance portability to unmodified OpenMP programs.

3. **Asymmetry-aware scheduling for multithreaded applications**: This workshop paper evaluated the asymmetry-aware HSF scheduler, designed for single-threaded workloads, on an Intel Alder Lake processor. In the journal submission, we replaced HSF with the HSP scheduler [143], which also supports multithreaded programs. Our experiments with both single-threaded and multithreaded workloads showcased the improved performance of HSP.

Moreover, we augmented the original HSP implementation [143] with OS-runtime interaction, which brings further benefits in multi-application scenarios, as demonstrated in the journal article.

4. **Evaluation of Intel's TD scheduling patches**: While this workshop paper did not assess Intel's TD-related Linux patches, our journal article included an experimental evaluation of these patches, which Intel had recently submitted for inclusion in the Linux kernel [108]. This analysis allowed us to examine the effects of TD on the Alder Lake system and assess the impact of Intel's proposed OS-level support.

Rapid development of OS support with PMCSched for scheduling on asymmetric multicore systems

Carlos Bilbao[1 [0000−0002−4750−5124]], Juan Carlos Saez[1 [0000−0003−1343−7108]], and Manuel Prieto-Matias[1 [0000−0003−0687−3737]]

Facultad de Informática, Universidad Complutense de Madrid, Spain
{cbilbao,jcsaezal,mpmatias}@ucm.es

Abstract. Asymmetric multicore processors (AMPs) couple high-performance big cores and power-efficient small ones, all exposing a shared instruction set architecture to software, but with different microarchitectural features. The energy efficiency benefits of AMPs together with the general-purpose nature of the various cores, have led hardware manufactures to build commercial AMP-based products, first for the mobile and embedded domains, and more recently for the desktop market segment, as with the Intel Alder Lake processor family. This indicates that AMPs may become a solid and more energy efficient replacement to symmetric multicores in a wide range of application domains.
Previous research has demonstrated that the system software can substantially improve scheduling –critical to get the most out of heterogeneous cores– by leveraging hardware facilities that are directly managed by the the OS, such as performance monitoring counters, or the recently introduced Intel Thread Director technology. Unfortunately, the OS-level support enabling to access scheduling-relevant hardware support may take a long time to be adopted in operating systems, or may come in forms that make its utilization challenging from specific levels of the system software stack, especially in production systems. To fill this gap, we propose the PMCSched framework, which enables the creation of custom OS support on Linux to aid in the design of novel scheduling and resource-management policies for multicores implemented at different layers of the system software, but without requiring to patch the kernel. To demonstrate the potential of our framework, we implement a set of OS-level schedulers for AMPs, that make use of custom OS extensions to access scheduling-relevant hardware facilities in an x86 AMP processor.

Keywords: Asymmetric multicore processors · Scheduling · Operating Systems · Runtime Systems · Linux kernel · Intel Alder Lake

1 Introduction

Energy efficiency has become one of the most critical constraints of processor design [14]. The quest for improved energy efficiency substantially contributed

to the proliferation of heterogeneous architectures that combine within the same platform different types of cores and processing units for diverse and specialized uses [10]. Asymmetric multicore processors (AMPs) constitute an attractive type of heterogeneous architecture where high-performance big cores and power-efficient small ones –all exposing a shared ISA (instruction set architecture)– are combined on the same system. The common ISA in conjunction with the general-purpose nature of the AMP cores, allows the execution of legacy (unmodified) software. These facts, along with AMPs' energy efficiency benefits, have drawn the attention of major hardware players, leading to the massive release of commercial AMP products for mobile platforms, such as those based on the ARM big.LITTLE processor [28]. Today, the Intel Alder Lake processor family and the Apple M1 SoC, are clear examples of the expansion of AMPs toward the desktop market segment [31]. Moreover, in the high performance computing (HPC) domain, the combination of different core types with a shared ISA has also been explored; the Sunway TaihuLight supercomputer is a representative case [6,8].

Despite the remarkable benefits of AMPs [19], effectively scheduling diverse programs/tasks on heterogeneous cores poses a significant challenge to the various system software layers [21,25,10,5,6]. When a single multithreaded application runs alone on an AMP system, smart user-level scheduling within the runtime system is the key to making the most out of its heterogeneous cores [6,30]. However, in multi-application scenarios, and especially under the presence of legacy programs, the OS scheduler plays an essential role in transparently delivering the benefits of AMPs to the end user [18,25,10,16].

Previous research has demonstrated that the runtime system and the OS scheduler can perform optimizations on AMPs by leveraging hardware features that are directly controlled by the OS kernel and exposed to user space, such as Performance Monitoring Counters (PMCs) [10,36,18] or Dynamic Voltage and Frequency Scaling (DVFS) [35,7]. Often, the support to conveniently access new scheduling-relevant hardware features from the system software may take time to be adopted in operating systems [31], or it may come in the form of architecture-specific interfaces that limit application portability or make its utilization impossible from particular levels of the software stack [11,17]. Take for instance the Linux kernel, that does not currently feature support for the Intel Thread Director (TD) technology [31], unlike the proprietary Windows 11 kernel. TD is a set of scheduling-related hardware facilities –introduced with Alder Lake processors– that provide the system software with performance and energy efficiency hints to aid in carrying out effective thread-to-core mappings on Intel AMPs. Implementing custom mechanisms in the OS kernel to leverage these new –yet unsupported– features directly from the OS scheduler, or exposing them to user space involves a substantial development effort, due to the inherent challenges associated with kernel-level programming [26,11]. At the same time, custom OS-level extensions could be difficult to be adopted in production systems, where patching the OS kernel may be impractical.

To address these issues, we propose PMCSched, a framework for the Linux kernel that enables rapid development of the OS-level support required to create

custom scheduling and resource-management schemes on both symmetric and asymmetric multicore systems. Unlike other existing frameworks that require patching the Linux kernel to function [4,20,26,39], PMCSched makes it possible to incorporate new scheduling-related OS-level support in Linux via a kernel module that can be loaded in unmodified kernels, making its adoption easier in production systems. Notably, the main focus of this framework is to simplify the creation of novel scheduling and resource-management strategies that are either implemented entirely in the OS kernel, or require changes in different layers of the system software, so as to benefit from coordinated decisions between the runtime system and the OS scheduler [13,10,30].

As a proof of concept of our framework, in this work we implement different asymmetry-aware OS-level schedulers on top of an unmodified Linux kernel v5.16, and evaluate their effectiveness by running different multi-application workloads on an Intel Alder Lake processor. These schedulers make use of PMCs and leverage the Intel Thread Director technology [15,16], by accessing such hardware facilities directly from kernel space.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 provides an overview of PMCSched design and introduces its main implementation challenges. Section 4 covers the experimental case study on scheduling for Alder Lake processors, and Section 5 concludes the paper.

2 Related Work

A large body of work has proposed asymmetry-aware scheduling strategies for adoption on either runtime systems [5,37,25] or OS kernels [18,10,31]. Frequently, such endeavors culminate in tools and frameworks that aim to ease the development and analysis of new scheduling algorithms; these are likewise some of the main goals of this paper.

Recent studies have shown that scheduling algorithms that come in stock general-purpose OSs exhibit suboptimal behavior for different workloads on a wide range of processor architectures [10,23,6]. At the same time, making the required changes in an OS kernel to build effective scheduling policies specifically tailored to custom workloads or microarchitectures may be a significant burden to the average developer [26,39]. On many monolithic kernels such as Linux, the development of new OS scheduling policies constitutes a labor intensive task, as the kernel itself needs to be modified. More specifically, testing any scheduling-related kernel modification requires compiling and reinstalling the kernel, and finally rebooting the machine for the changes to take effect. Testing an individual change in this way may as well take a full a coffee break, depending on the features and resources of the target platform and the development host.

To overcome these problems, some researchers have resorted to evaluating their proposed OS-level schedulers via simplistic user space prototypes [3,35,9]. Even though this approach may allow to draw interesting insights and also benefit from leveraging application-level metrics, strategies implemented in this way suffer from the limitations imposed to userland, such as the additional overhead

of context switches and extra system calls required for dynamic thread affinity and performance monitoring [26], or the inability to quickly react to low-level scheduling-related events (e.g., a thread blocks due to I/O or a page fault), thus wasting CPU cycles [11]. In addition, user-level scheduling prototypes cannot access hardware extensions not currently exposed by the OS kernel.

Scheduling frameworks, such as those proposed in [39,26,4] or PMCSched itself, aim to overcome some of the aforementioned limitations. LUSH permits the creation of user-level schedulers for AMPs without special execution privileges, and introduces kernel-level changes to allow fine-grained access to PMCs from user space. Mvondo et al. [26] propose the extension of existing OS APIs, so as to allow the development of kernel-level schedulers programmable from user-space using a safe and controlled environment. LITMUS [4], by contrast, constitutes a substantial fork of the Linux kernel with extensions to facilitate programming of real-time kernel-level scheduling algorithms. Contrary to such solutions –some of them restricted to specific domains [39,4]– PMCSched allows to create custom scheduling-related OS-level modifications without actually patching the kernel. This constitutes a major advantage, as getting profound modifications of the kernel accepted upstream is an arduous task; so much so that researchers tend to forget about that possibility altogether and treat their software as research prototypes with no hope of production integration in sight [26], even after conducting the required security audits. Conversely, the big effort required to maintain multiple project forks for various releases of the Linux kernel often shortens the lifespan of the associated projects [38].

Other studies explore the challenges of OS scheduling on highly heterogeneous architectures [25]. Of special attention is the case of Popcorn Linux [2], which targets heterogeneous systems consisting of nodes with different ISAs, opening the door to parallel ISA-heterogeneous runtime scheduling [24]. These efforts are orthogonal to ours, formulating a problem with several interconnected computing nodes with different processor architectures (e.g., x86 and ARM).

3 PMCSched: Implementation challenges and design

Motivation and challenges. PMCSched is implemented on top of PMCTrack, a performance monitoring tool [33] for Linux that was open sourced back in 2015 [32]. Unlike Perf Events [38] –the default Linux subsystem to access hardware facilities, such as performance monitoring counters (PMCs)– PMCTrack was not primarily designed to only expose hardware monitoring facilities to user space, but to assist the system software when performing runtime optimizations based on these hardware facilities. The operations for which the system software can benefit from PMCTrack include scheduling [34,10] and resource management [11]. The main advantages of relying on PMCTrack for such tasks are its ability to foster new OS-level features as part of an extensible loadable kernel module, and its efficient architecture-independent API to access PMCs within the kernel on a wide range of architectures (x86, ARMv7, ARMv8, etc.). Fig. 1

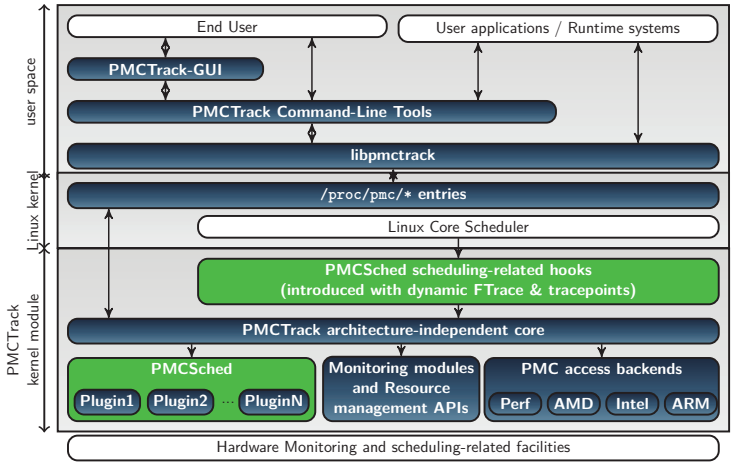


Fig. 1. PMCSched components (in green) inside PMCTrack's architecture.

depicts the various components of PMCTrack and their relationship, described in detail in [33].

With the PMCSched framework we take PMCTrack's potential one step further by enabling rapid development of OS support for scheduling and resource management for Linux *within a loadable kernel module*. We need to highlight this because hitherto new scheduling policies could not be implemented as a kernel module [20,26], since no specific API exists for that purpose within the Linux scheduler. When creating novel OS-level schedulers for Linux without modifying the kernel, three main challenges have repeatedly appeared: (1) the inability to execute code in a kernel module in immediate response to the occurrence of key scheduling-relevant events –context switches, thread creation/destruction, etc.– (2) the lack of a standardized method to seamlessly extend the Linux task structure with new per-thread scheduling related fields that custom schedulers typically require to function, and (3) how to efficiently customize the behavior of the Linux load balancer. Notably, the first two barriers also arise when attempting to manage performance counters at the low level, and for that reason, most PMC tools require changes in the kernel; PMCTrack adds the associated functionality via a small portable kernel patch [33].

Our solution. PMCSched addresses the three aforementioned issues without patching the kernel as follows. First, to be aware of key scheduling events from a kernel module, PMCSched installs scheduling-related hooks (callbacks) leveraging two modern tracing facilities of the Linux kernel: *dynamic ftrace* [29] and *tracepoints* [22]. These two tracing technologies rely on dynamic and static kernel instrumentation, respectively. Noticeably, both are supported on a wide range of processor architectures, and can be found enabled by default on the most popular Linux distributions. Unlike other kernel instrumentation facilities (like Kprobes), these technologies make it possible for a module to be notified when

a kernel function is invoked or when a static tracepoint is reached with virtually no overhead [29,22]. Not only do PMCSched hooks –depicted in Fig. 1– enable the implementation of custom schedulers in a kernel module, but also allowed us to eliminate the need for the PMCTrack kernel patch entirely [27]. Secondly, PMCSched provides a seamless mechanism to extend the task structure with new thread-specific data without modifying the kernel. To this end, whenever a thread enters the system, PMCSched associates a *dummy* software event from the Perf Events subsystem to the thread, by inserting the event into the event list present in Linux's task structure (`perf_event_list` field). The structure of this dummy event (`struct perf_event`) contains a void pointer field (`pmu_private`) that can be utilized to point to any other structure. To simplify the integration of PMCSched in PMCTrack, we use the event's void pointer to point to PMCTrack's per-thread structure (`pmon_prof_t`). PMCSched scheduling fields can be seamlessly added without modifying the kernel, by extending the structures definition inside PMCTrack's kernel module sources.

To make it possible to implement custom load balancing policies, PMCSched introduces the *core group* abstraction. Essentially, cores in the system are organized into different sets (or *core groups*) based on their type (for AMP systems) or their hierarchical relationship in the platform's topology (e.g., cores sharing a last-level cache, or part of the same NUMA node). PMCSched automatically divides cores into different core groups based on system topology, but considering a configurable granularity (LLC, socket or NUMA domain). To implement custom and scalable OS-level load balancing policies or perform specific thread-to-core mappings, a scheduler implemented in PMCSched must assign threads to specific core-groups by using affinity masks. In using this approach, enforcing load balancing across cores within the same group is up to the Linux load balancer, which respects affinity masks. We should also highlight that PMCSched associates a set of linked lists to each core group (spin-lock protected), making it possible to keep track of active threads or multithreaded processes associated with each core group. This design approach allows to make scheduling decisions independently for threads assigned to different core groups, and favors scalable designs that reduce contention in accesses to core-group specific data structures.

A new scheduling or resource management algorithm can be implemented by creating a *scheduling plugin*, which –as illustrated in Fig. 1– becomes a part of the PMCSched subsystem within PMCTrack's kernel module. Building a plugin boils down to instantiating an interface of scheduling operations and implementing the corresponding interface functions in a separate ".c" file within the module sources. The various algorithm-specific operations are invoked from the core part of the scheduling framework when a key scheduling-related event occurs, such as when a threads enters the system, terminates, becomes runnable/non-runnable, or when tick processing is due to update statistics. The framework also provides a set of callbacks to carry out periodic scheduling activations from interrupt (timer) and process (kernel thread) context on each core group separately, thus making it possible to invoke a wide range of blocking and non-blocking scheduling-related kernel API calls, such as those to map a thread to a specific

CPU or core group. This modular approach to creating scheduling algorithms resembles the one used by *scheduling classes* (algorithms) inside the Linux kernel, but with a striking advantage: PMCSched scheduling plugins can be bundled in a kernel module that can be loaded on unmodified kernels. Moreover, plugin developers have access to a rich set of APIs available in PMCTrack, empowering them to configure performance counters seamlessly and retrieve PMC values in a per-thread fashion, to gather data from other hardware monitoring features [33,31], or to govern hardware facilities for shared-resource contention mitigation (e.g., LLC partitioning) available on Intel and AMD processors [11,1].

OS-runtime interaction and Future Work. As previously stated, PMCSched could also be used as a tool to perform system-software optimizations that exploit synergistic interactions between a user-level runtime system and the OS [13,30]. To allow different types of interaction between user space and the kernel, the current version of PMCSched exports a set of special files under the /proc filesystem. For example, the value of configurable parameters of the currently active scheduling plugin can be retrieved/altered by reading/writing from/to those special files. PMCSched also supports the creation of a per-thread page-sized memory region that can be shared between kernel and user space, so as to allow the runtime system to share critical application-level metrics with the OS (e.g., QoS metrics for throughput or latency constraints) and, at the same time, enable the OS to expose information not directly accessible from the runtime system, such as Thread Director performance and energy-efficiency estimates for the current core type where the thread runs [31]. As for future work, and by leveraging this or other communication features –such as netlink sockets–, we plan to implement an OS/runtime interaction scheme to enable efficient execution of multiple data-parallel OpenMP programs on an AMP system, where both layers of the system software play an essential role [30,6].

4 Experimental case study

To demonstrate the applicability of the PMCSched framework, we experimented with a system equipped with an Intel Core i9-12900K "Alder Lake" processor and 32GB DDR4 SDRAM. This AMP processor combines 8 "Golden Cove" big (P) cores, and 8 "Gracemont" small (E) cores. E-cores are grouped into two 4-core clusters, each group sharing a 2MiB L2 cache. P-cores, by contrast, have a private 1.25MiB L2 cache. Every core in the platform integrates a private L1 cache, but shares a 30MiB L3 (LLC) with the remaining E and P cores. With our experiments we evaluate how effectively an OS-level scheduler implemented with our framework can improve the overall system throughput on an Intel Alder Lake processor.

Maximizing throughput on AMPs. Previous research has demonstrated that, to maximize throughput in the context of multi-program workloads, the scheduler needs to be able to (1) determine at runtime the performance benefit that each thread in the workload derives from running on a big core relative to a small one, and then (2) use big cores for running threads that exhibit

a larger relative performance benefit from such cores, while possibly readjusting the mappings dynamically based on program-phase changes. Henceforth, we will refer to the big-to-small performance benefit as the thread's Speedup Factor (SF). Similarly, we will now use the acronym HSF (i.e., High SF) to refer to a dynamic scheduling strategy that aims to maximize throughput by mapping high-SF threads to big cores. While this experimental analysis focuses on workloads consisting of compute-intensive single-threaded applications, it is worth noting that other factors beyond the SF need to be considered for multi-threaded programs, such as latency constraints [12], load balancing and synchronization [6,30], along with other interdependencies among tasks/threads in the application [5].

Implementation of scheduling algorithms. One of the main deltas among the various HSF implementations [19,18,34] is the underlying method employed to determine the SF online. In this work we explore the effectiveness of two SF prediction methods: PMC-based estimation models [18,28,34], and reliance on specific hardware support for SF estimation [16,15]. Regarding the first prediction method, we use the two SF-estimation models proposed in our earlier work [31], which were specifically built for SF prediction from the big and small cores of an Intel Alder Lake processor. The methodology used to build these estimation models [34], the specific performance events they depend upon, and a detailed discussion on their accuracy can be found in [31]. For the hardware-aided SF prediction we leverage the Intel Thread Director (TD) technology, a set of hardware facilities –first introduced in Alder Lake processors– enabling to guide the OS in making thread scheduling decisions on Intel hybrid multi-cores [15,16]. To predict a thread's current SF with TD, the OS must retrieve its TD class (i.e., an integer in {0..3} in the Alder Lake processor we used) by reading a model-specific register, and then calculate the ratio of two performance estimates (for big and small cores) associated with the current TD class; these performance estimates are stored in a memory-resident table that the hardware maintains, which is directly readable from the OS kernel alone.

We experimented with several asymmetry-aware schedulers implemented in PMCSched: an Asymmetry-Aware Round-Robin (AARR) scheduler [21] that equally shares big and small cores among applications; and three variants of the HSF scheduler, which optimize throughput. The first variant of HSF –referred to as HSF/TD– employs Thread Director (TD) to obtain SF estimates. Because in the Alder Lake processor we used such estimates are only accessible directly when the thread runs on a big core (i.e., a valid TD class is not reported from E-cores [31]), our implementation continuously stores TD-based big-core SF estimates on a per-thread history table for different program phases, making it possible to obtain SF predictions indirectly from small cores by accessing the history table. The utilization of history tables to observe patterns from previous samples and predict current and future performance has been widely explored by previous work [39,10]. To deal with frequent *phase misses* when accessing the history table from small cores, our implementation triggers migrations to big cores to gather new big-core estimates, and also implements a throttling

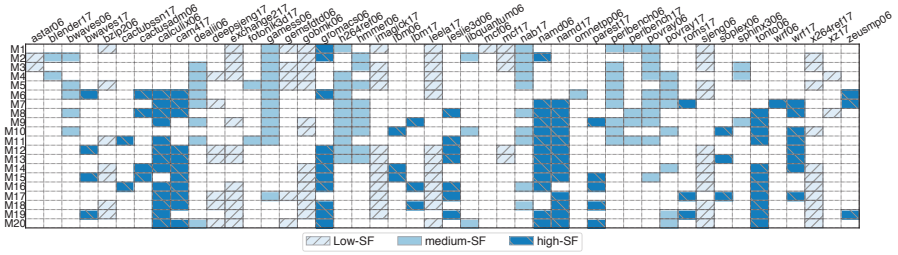


Fig. 2. Workloads used for our experiments. Each row M_i depicts the composition of the i-th workload. A blank cell indicates that the associated program is not included in the workload. Applications whose average SF is lower than 1.7 are considered low-SF programs in our platform, and those with an SF value greater than 2.05 are classified as high-SF. The remaining ones are labeled as medium-SF.

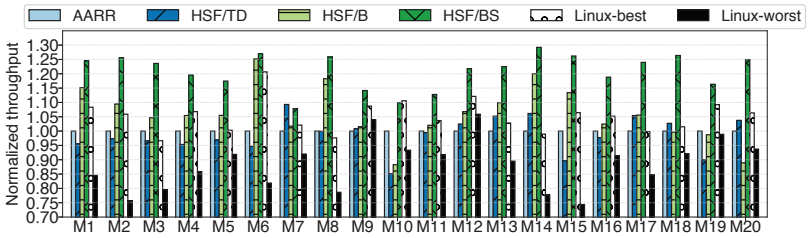


Fig. 3. Normalized throughput delivered by the various scheduling algorithms

mechanism to limit the number of profiling-related migrations, as in [10]. In the second HSF variant, denoted as HSF/BS, the OS continuously gathers a number of per-thread PMC metrics; an up-to-date SF prediction is obtained for a thread by using the metric values as input to the core-specific models proposed in [31] for the big and the small core (the model to use depends on the thread's current core type). Lastly, under the HSF/B variant, SF predictions on big cores are obtained via the same big-core model used by HSP/BS; however, on the small core, predictions are obtained indirectly by reading a history table, populated with past SF estimates retrieved on the big core. Note that this variant was implemented to conduct a fairer comparison with HSF/TD, where direct SF predictions on the small core are unavailable.

Experiments and discussion. For our experiments we randomly generate 20 diverse workloads, comprising of 16 single-threaded programs each. The composition of the various program mixes (Mi) is depicted in Fig. 2, and covers 46 different SPEC CPU applications in total. In launching each program mix, we follow a similar procedure to that of previous works [3,34], so as to ensure the machine's load is constant throughout the experiment. All applications in the workload are started simultaneously, and when one of them completes, the program is restarted repeatedly until the slowest application completes three times. We use the geometric mean of the completion times for each program to calculate the degree of throughput, by using the Aggregate Speedup metric [34,10,31]. All

programs were compiled with GCC 11.2 with the `-O3` and `-mtune=alderlake` compiler switches.

Fig. 3 shows the normalized throughput for the various scheduling algorithms relative to AARR. As a reference, we also provide the best and worst results obtained by Linux default scheduler (CFS) across 10 runs of each experiment, referred to as Linux-best and Linux-worst, respectively. This scheduler is designed to minimize the number of thread migrations, but it is still largely asymmetry unaware [10], and provides highly variable completion times for the same application across multiple runs of the same experiment on Intel Alder Lake processors. Essentially CFS may map an application to a big core for a certain run, and then to a small core in another run, irrespective of its co-runners. This causes large throughput differences across runs, making CFS a misleading baseline [10].

These experimental results undoubtedly reveal that HSP/BS outperforms the other schedulers for most workloads, achieving up to a 30% throughput gain w.r.t. AARR, and providing a 22.9% average improvement against the TD variant. These numbers are tightly related to the superior SF-estimation accuracy provided by the PMC-based models for the big and small core, relative to that of Thread Director, as shown in [31]. Overall, a higher SF-prediction accuracy allows HSF to identify programs with a truly high SF better, and, as a result, the scheduler can grant more big-core cycles to them than to other threads. We further observe that using the big-core model in combination with the history table (HSF/B variant), provides substantially better throughout figures than HSP/TD (averaging 7.9% improvement). However, in a few workloads, such as M10 and M20, HSF/B fails to yield comparable performance to that of AARR. We found that this is caused by the extra thread migrations (and hence the overheads), triggered in response to frequent table phase misses, and aimed at refreshing the history table on big cores. Despite this fact, we conclude that the PMC-based big-core model alone provides superior accuracy than TD, and that the per-thread history table constitutes a reasonably effective method to deal with scenarios where direct SF estimation is not available on certain core types.

5 Conclusions and Future Work

In this paper we have presented PMCSched, a framework for Linux that enables to implement the custom OS kernel support required by new scheduling and resource-management policies for multicore systems. A key distinctive feature of our framework is that it empowers developers and researchers to add new kernel-level scheduling-related support via a loadable module that can be inserted in vanilla (unmodified) versions of the Linux kernel. This favors the adoption in production systems of custom, and potentially sophisticated, scheduling strategies implemented at one or multiple levels of the system software stack. To demonstrate the flexibility of the framework, we leveraged PMCSched's modular plugin-based design to implement several asymmetry-aware OS-level schedulers, and evaluated their ability to improve system throughput under multi-application workloads on an Intel Alder Lake (hybrid) multicore processor.

As for future work, we plan to design novel scheduling and resource management strategies to improve performance when both single-threaded and multithreaded programs are present on the system, making emphasis on potential optimizations that come from the synergistic cooperation between the runtime system and the OS. Lastly, we should highlight that part of the core functionality of PMCSched is already publicly available in PMCTrack's source code repository [27], but that the full framework will be open sourced with the next public release of PMCTrack, scheduled for late 2022.

Acknowledgements Work supported by the EU (FEDER), the Spanish MINECO and CM, under grants RTI2018-093684-B-I00 and S2018/TCS-4423.

References

1. AMD: AMD64 Technology Platform QoS Extensions. `https://developer.amd.com/wp-content/resources/56375.pdf`
2. Barbalace, A., Lyerly, R., Jelesnianski, C., Carno, A., Chuang, H.R., Legout, V., Ravindran, B.: Breaking the boundaries in heterogeneous-ISA datacenters. In: ACM SIGPLAN Notices. vol. 52, pp. 645–659. ACM (2017)
3. Blagodurov, S., et al.: A case for NUMA-aware contention management on multicore systems. In: Proceedings of USENIX ATC '11. USA (2011)
4. Calandrino, J.M., et al.: LITMUS-RT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In: 2006 27th IEEE Int'l Real-Time Systems Symposium (RTSS'06). pp. 111–126 (2006)
5. Chronaki, K., et al.: Criticality-aware dynamic task scheduling for heterogeneous architectures. In: Proceedings of the 29th ACM on Int'l Conference on Supercomputing. pp. 329–338. ICS 2015 (2015)
6. Chronaki, K., et al.: On the maturity of parallel applications for asymmetric multicore processors. J. Par. Distrib. Comput. **127**, 105–115 (2019)
7. Costero, L., et al.: Energy efficiency optimization of task-parallel codes on asymmetric architectures. In: Proc. of HPCS '17. pp. 402–409 (jul 2017)
8. Dongarra, J.: Report on the sunway taihulight system. Tech Report University of Tennessee: UT-EECS-16-742 (2016)
9. Feliu, J., et al.: Perf&fair: A progress-aware scheduler to enhance performance and fairness in smt multicores. IEEE Trans. Comput. **66**(5), 905–911 (May 2017)
10. Garcia-Garcia, A., et al.: Contention-aware fair scheduling for asymmetric single-ISA multicore systems. IEEE Transactions on Computers **67**(12) (Dec 2018)
11. Garcia-Garcia, A., et al.: LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores. In: Proc. of ICPP'19. pp. 14:1–14:10 (2019)
12. Haque, M.E., et al.: Exploiting heterogeneity for tail latency and energy efficiency. In: 50th Ann. IEEE/ACM Int'l Symp. on Microarchitecture. pp. 625–638 (2017)
13. Harris, T., Maas, M., Marathe, V.J.: Callisto: Co-scheduling parallel runtime systems. In: Proc. of 9th European Conf. on Comput. Systems. EuroSys '14 (2014)
14. Hennessy, J.L., Patterson, D.A.: A new golden age for computer architecture. Commun. ACM **62**(2), 48–60 (Jan 2019)
15. Intel: Intel® 64 and IA-32 Architectures Software Developer's Manual Vol. 3: System Programming Guide (2021)
16. Intel: Optimizing software for x86 hybrid archiecture. Intel White Paper (Oct 2021)

17. Intel: User space software for Intel(R) Resource Director Technology. https://github.com/intel/intel-cmt-cat (2022)
18. Koufaty, D., Reddy, D., Hahn, S.: Bias Scheduling in Heterogeneous Multi-core Architectures. In: Eurosys 10. pp. 125–138 (2010)
19. Kumar, R., et al.: Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In: 31st Ann. Int'l Symp. Computer Architecture (ISCA 04). pp. 64–75 (2004)
20. Lepers, B., et al.: Provable multicore schedulers with Ipanema: Application to work conservation. In: Proc. of Eurosys '20 (2020)
21. Li, T., et al.: Operating system support for overlapping-ISA heterogeneous multicore architectures. In: Proc. of HPCA '10. pp. 1–12 (2010)
22. Linux: Using the linux kernel tracepoints. https://www.kernel.org/doc/html/latest/trace/tracepoints.html
23. Lozi, J.P., et al.: The linux scheduler: A decade of wasted cores. In: Proceedings of the 11th ACM European Conference on Computer Systems (Eurosys '16) (2016)
24. Lyerly, R., et al.: An OpenMP Runtime for Transparent Work Sharing Across Cache-Incoherent Heterogeneous Nodes. ACM Trans. Comput. Syst. (dec 2021)
25. Mittal, S.: A survey of techniques for architecting and managing asymmetric multicore processors. ACM Comput. Surv. **48**(3), 45:1–45:38 (Feb 2016)
26. Mvondo, D., et al.: Towards user-programmable schedulers in the operating system kernel. In: Proceedings of the 11th Workshop on Systems for Post-Moore Architectures, SPMA 2022 (Apr 2022)
27. PMCTrack: Github repository. https://github.com/jcsaezal/pmctrack (2015)
28. Pricopi, M., et al.: Power-performance modeling on asymmetric multi-cores. In: Proc. of CASES '13. pp. 15:1–15:10 (2013)
29. Rostedt, S.: "ftrace: Where modifying a running kernel all started" https://kernel-recipes.org/en/2019/talks/ftrace-where-modifying-a-running-kernel-all-started/
30. Saez, J.C., Castro, F., Prieto-Matias, M.: Enabling performance portability of data-parallel openmp applications on asymmetric multicore processors. In: 49th Int'l Conference on Parallel Processing. ICPP '20 (2020)
31. Saez, J.C., Prieto-Matias, M.: Evaluation of the Intel Thread Director technology on an Alder Lake processor. In: 13th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '22) (2022)
32. Saez, J.C., et al.: An OS-oriented performance monitoring tool for multicore systems. In: Proc. of Euro-Par 2015: Parallel Processing Workshops. pp. 697–709 (2015)
33. Saez, J.C., et al.: PMCTrack: Delivering performance monitoring counter support to the OS scheduler. The Computer Journal **60**(1), 60–85 (2017)
34. Saez, J.C., et al.: Towards completely fair scheduling on asymmetric single-ISA multicore processors. Journal of Parallel and Distributed Computing **102** (2017)
35. Salami, B., et al.: Online energy-efficient fair scheduling for heterogeneous multicores considering shared resource contention. J. Supercomput. **78**(6) (apr 2022)
36. Servat, H., et al.: On the instrumentation of OpenMP and OmpSs tasking constructs. In: Euro-Par 2012: Parallel Processing Workshops. pp. 414–428 (2013)
37. Torng, C., Wang, M., Batten, C.: Asymmetry-aware work-stealing runtimes. In: Proc. of ISCA '16. pp. 40–52 (2016)
38. Weaver, V.M.: Linux perf event features and overhead. FastPath Workshop (2013)
39. Xu, V.M., et al.: Lush: Lightweight framework for user-level scheduling in heterogeneous multicores. In: 2021 IEEE 14th Int'l Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC). pp. 396–404 (2021)

Appendix B

Revisiting Fairness and Throughput Metrics for Cache-Partitioning Policy Assessment: Insights and Recommendations

The workshop paper included in this chapter was presented at the 1st Workshop on Computer Architecture Modeling and Simulation (CAMS 2023). Note that contributions to the CAMS '23 workshops were not published, as the workshops did not have associated proceedings.

Abstract

The widespread adoption of hardware cache-partitioning support in commercial processors, especially in the server market, has rekindled interest in research on cache-partitioning policies. Partitioning the Last-Level Cache (LLC), often shared among a number of cores in current multicore processor designs, can help alleviate the negative effects stemming from shared resource contention. Particularly, LLC-partitioning policies significantly influence the degree of throughput and fairness extracted from the system. Unfortunately, recent research on LLC-partitioning systematically employ different sets of metrics, making it difficult to distinguish between the most effective proposals in terms of both throughput and fairness.

This work conducts a simulation-based analysis to shed light on the effect that the choice of different popular metrics for throughput and fairness assessment has on the relative gains associated with LLC-partitioning policies. In light of the results, we also propose a set of recommendations on the utilization and selection criteria for throughput and fairness metrics. Our study reveals that the chosen metric

indeed has a profound impact on the reported relative improvement for a partitioning policy, as each metric exhibits a very different improvement range, sometimes even duplicating the percentage increases with respect to other metrics. We find that, putting aside the differences in range, improvements reported by the analyzed throughput metrics show a strong correlation. In contrast, there exists a weaker correlation among many popular fairness metrics regarding their improvement trends.

Revisiting Fairness and Throughput Metrics for Cache-Partitioning Policy Assessment: Insights and Recommendations

Carlos Bilbao
Complutense University of Madrid
Madrid, Spain
cbilbao@ucm.es

Juan Carlos Saez
Complutense University of Madrid
Madrid, Spain
jcsaezal@ucm.es

Manuel Prieto-Matias
Complutense University of Madrid
Madrid, Spain
mpmatias@ucm.es

ABSTRACT

The widespread adoption of hardware cache-partitioning support in commercial processors, especially in the server market, has rekindled interest in research on cache-partitioning policies. Partitioning the Last-Level Cache (LLC), often shared among a number of cores in current multicore processor designs, can help alleviate the negative effects stemming from shared resource contention. Particularly, LLC-partitioning policies significantly influence the degree of throughput and fairness extracted from the system. Unfortunately, recent research on LLC-partitioning systematically employ different sets of metrics, making it difficult to distinguish between the most effective proposals in terms of both throughput and fairness.

This work conducts a simulation-based analysis to shed light on the effect that the choice of different popular metrics for throughput and fairness assessment has on the relative gains associated with LLC-partitioning policies. In light of the results, we also propose a set of recommendations on the utilization and selection criteria for throughput and fairness metrics. Our study reveals that the chosen metric indeed has a profound impact on the reported relative improvement for a partitioning policy, as each metric exhibits a very different improvement range, sometimes even duplicating the percentage increases with respect to other metrics. We find that, putting aside the differences in range, improvements reported by the analyzed throughput metrics show a strong correlation. In contrast, there exists a weaker correlation among many popular fairness metrics regarding their improvement trends.

KEYWORDS

Multicore, simulation, metrics, fairness, throughput, cache-partitioning

1 INTRODUCTION

Today, chip multicore processors (CMPs) constitute the dominant architecture choice for general-purpose computing, being present in a wide spectrum of commercial products from low-power embedded and mobile devices to high-performance servers. Despite the outstanding technological and microarchitectural advances, the contention in shared memory resources in a CMP –e.g., Last-Level Cache (LLC) [5, 17] and DRAM controller [9, 38]– still leads to a number of undesirable effects on the system, making it challenging to ensure system-wide fairness [30, 31, 37] and uphold QoS (Quality of Service) constraints [8, 32]. Contention also poses a substantial barrier towards delivering performance guarantees without degrading throughput [24, 38], and may severely limit scalability [33, 39].

LLC-partitioning constitutes a popular control mechanism to mitigate some of the contention-related effects in multi-application and multi-tenant scenarios [7, 19, 20, 22, 25, 26, 32]. The fairly recent adoption of hardware partitioning extensions in commodity processors [2, 16] has given rise to a growing interest in the design of LLC-partitioning strategies [8, 10, 10, 14, 20, 23–25, 27, 29–31, 36].

Table 1: Throughput and fairness metrics of different works.

Work	Throughput metric	Fairness metric
Selfa et al. [31]	STP	UnfairnessCoV
Xiang et al. [36]	AggregateIPC	FairSlowdown
Park et al. [25]	GmeanIPC	UnfairnessCoV
Garcia-Garcia et al. [14]	STP	Unfairness
Roy et al. [29]	GmeanSpeedup	Jain Fairness Index
Pons et al. [27]	GmeanIPC	(Not reported)

LLC-partitioning policies substantially impact the degree of system throughput [8, 10, 24, 27] and fairness [25, 29–31]. It comes as a surprise, however, that recent works on LLC-partitioning show no clear consensus on the choice of metrics to assess the degree of throughput and fairness. To illustrate this, Table 1 summarizes the throughput and fairness metrics employed in a representative set of works published between 2017 and 2023 (the definitions of the various metrics can be found in Sec. 2). Notably, none of these works use the same pair of throughput-fairness metrics to quantify the impact of the corresponding proposed technique.

This lack of agreement makes it challenging to identify the best-performing technique from the fairness and throughput standpoints. This issue is further aggravated by other factors, such as the lack of quantitative comparisons with preceding approaches in specific articles [10, 23, 27], the scarce comparisons with the optimal solution, or the fact that the experimental platforms used for evaluation in recent works [8, 23, 25, 29–31] largely differ in features, such as core count, cache-hierarchy organization, LLC associativity, etc.

By extending and leveraging an open-source simulator [15], this article analyzes the impact that the choice of different metrics has on the reported relative improvement for cache-partitioning policies. It also highlights the interrelationship between specific metrics, and identifies key aspects that make certain metrics misleading. To this end, our work makes the following contributions:

(1) We identify a wide range of popular fairness and throughput metrics employed in recent articles on LLC-partitioning. For an ample spectrum of multi-program workloads, we determine the cache-partitioning solution that optimizes each individual metric. For this purpose, we use the PBBCache simulation tool [15], which allows us to deal with the complexity of obtaining the per-metric optimal in this context.
(2) After a thorough analysis of the simulation results we provide answers to the following questions: (1) What is the range of improvement with respect to a baseline policy provided by each metric? and (2) Is there any correlation between the

Table 2: Throughput and fairness metrics considered in our study.

Throughput Metric	Articles	Fairness Metric	Articles
$STP = \sum_{i=1}^{N} \left(\frac{1}{Slowdown_{a_i}} \right)$	[10, 14, 30, 31]	$Unfairness = \frac{MAX(Slowdown_{a_1}, \cdots, Slowdown_{a_N})}{MIN(Slowdown_{a_1}, \cdots, Slowdown_{a_N})}$	[9, 14, 30, 37]
$GmeanSpeedup = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{Slowdown_{a_i}}}$	[29]	$UnfairnessCoV = \frac{\sigma_{Slowdown}}{\mu_{Slowdown}}$	[25, 31]
$HmeanSpeedup = \frac{N}{\sum_{i=1}^{N} \frac{1}{Slowdown_{a_i}}}$	[21]	$JainFairness = \frac{1}{1 + \left(\frac{\sigma_{Slowdown}}{\mu_{Slowdown}} \right)^2}$	[29]
$GmeanIPC = \sqrt[N]{\prod_{i=1}^{N} IPC_{policy,a_i}}$	[25, 27]	$M1 = \sum_{i=1}^{N} \sum_{j=i+1}^{N} \lvert Slowdown_{a_i} - Slowdown_{a_j} \rvert$	[23]
$AggregateIPC = \sum_{i=1}^{N} IPC_{policy,a_i}$	[36]	$FairSlowdown = ANTT = \frac{1}{N} \sum_{i=1}^{N} Slowdown_{a_i}$	[4, 36]

different metrics, making it possible to narrow down the set of metrics used for a comprehensive analysis?

We show that while some metrics substantially magnify the improvements, others render only subtle percentage differences among partitioning strategies. This indicates that the effectiveness of different policies cannot be compared just by looking at the raw average improvement w.r.t. a common baseline reported across works (even with a similar experimental platform), as the chosen metric for throughput or fairness largely affects the degree of improvement. We also show that there is a clear correlation between most throughput metrics, whereas the vast majority of popular fairness metrics exhibit a much weaker correlation. Therefore, we conclude that reporting the value of multiple fairness metrics is crucial to facilitate the comparison across policies and works.

This paper is organized as follows. Sec. 2 introduces the metrics considered for our analysis. Sec. 3 discusses prior related work. Sec. 4 presents our simulation-based study, and its main insights. Finally, Sec. 5 concludes the paper.

2 THROUGHPUT AND FAIRNESS METRICS

Our article surveys a number of system-level throughput and fairness metrics for multi-program workloads. Each multi-program workload W consists of N co-running applications ($W = \{a_1, \cdots, a_N\}$). All the metrics considered ultimately depend on the performance of each application a_i, represented via its completion time (*CT*), or Instructions Per Cycle (*IPC*). Notably, most of the metrics considered are defined in terms of the *Slowdown* observed for each application a_i, which quantifies a_i's performance degradation, w.r.t. running the application alone on the system. An application's slowdown can be obtained based on its completion time or IPC as follows:

$$Slowdown_{a_i} = \frac{CT_{policy,a_i}}{CT_{alone,a_i}} = \frac{IPC_{alone,a_i}}{IPC_{policy,a_i}} \quad (1)$$

where CT_{policy,a_i} and IPC_{policy,a_i} are the completion time and IPC, respectively, of application a_i when running together with the other applications in W, under a specific partitioning policy. CT_{alone,a_i} and IPC_{alone,a_i} are a_i's time and IPC running alone on the system.

Table 2 shows the definition of the various metrics employed in our study, along with a reference to specific cache-partitioning articles utilizing each metric. To arrive at this set of metrics, we performed an exhaustive search on cache-partitioning related articles published since 2017 in main conferences in computer architecture, and in top ranked journals. We particularly focused on LLC-partitioning works that specifically consider throughput/fairness optimization. Note that while most metrics in Table 2 have been often used in computer-architecture research in the last decade, a few of them were employed –to the best of our knowledge– by just one recent article on cache-partitioning; this is the case of the *HMeanSPeedup*, *GMeanSpeedup*, *AggregateIPC* and *M1* metrics. Despite the limited use of these metrics in this context, we still opted to consider them in our analysis for the sake of completeness.

Throughput metrics (all of them higher-is-better indicators) can be broadly classified into two categories [12]. The first category encompasses metrics that rely on the per-application weighted speedup or weighted-IPC, accounting for the relative performance in multi-program workloads compared to running each application in isolation. The *STP*, *GmeanSpeedup* and *HmeanSpeedup* metrics shown in Table 2 fall in this category, as they depend on the application *Speedup*, which is the inverse of the *Slowdown*. The second metric category encompasses metrics, such as *GmeanIPC* or *AggregateIPC*, which depend only on the *raw IPC* that each application delivers in the multi-program workload (i.e., IPC_{policy,a_i}).

Eyerman and Eeckhout [12] illustrated that weighted-IPC metrics, like *STP* and *GmeanSpeedup*, are more intuitive and consistent from a system-level standpoint than metrics that use raw IPC values only. Moreover, they clearly showed that raw-IPC metrics can often be misleading, and suggested their avoidance. Despite this, recent work on cache-partitioning [25, 27, 36] still reports throughput gains using raw-IPC metrics, like *GmeanIPC* and *AggregateIPC*. Our analysis corroborates Eyerman and Eeckhout's observations on the misleading nature of *AggregateIPC* [12]. However, we find that *GmeanIPC* shows a strong correlation with the considered weighted-IPC metrics, so it does not constitute a misleading indicator.

In the context of LLC-partitioning, El-Sayed et al. [10] employed a variant of the STP metric. Instead of relying on the *Slowdown* (Eq. 1), this alternative throughput metric considers the per-program performance degradation w.r.t. the scenario where the LLC remains unpartitioned and shared with the other applications in the workload. We opted to discard this metric [10] from our analysis because the base performance defined in this way is workload dependent –rather than just application dependent–, which makes it ambiguous on NUMA and clustered-LLC multicores [1, 3, 23], where multiple

LLCs exist on the system. Note that in these multi-LLC systems, the degree of system throughput when the cache is not partitioned may vary substantially in multiple runs of the same workload, as it depends on the underlying application-to-LLC mapping [3].

Fairness metrics are often reported together with the degree of system throughput when assessing the overall effectiveness of specific policies [25, 30, 31]. Table 2 shows the definition of five fairness metrics employed in recent works on LLC-partitioning. Notably, all of them depend on the per-application Slowdown, and, except for the Jain Fairness Index, they are all lower-is-better metrics.

The first four fairness metrics in Table 2 employ a notion of fairness widely used by prior shared-resource contention works [9, 14, 31, 34, 37, 40]: a policy is considered perfectly fair if equal-priority applications in the workload are subjected to the same slowdown product of sharing the system. The closer the per-application slowdowns are, the fairer the policy. The *Unfairness* metric has been extensively used in computer-architecture research [9, 14, 30, 37]. A shortcoming of the *Unfairness* and other related metrics (e.g., *MaxSlowdown* [36]) is that they ultimately depend on the extreme slowdown values. To address this, other authors employ alternative metrics that either consider the pairwise differences between slowdowns of different applications (such as in the *M1* metric [23]) or rely on the coefficient of variation (CoV) of the per-application slowdowns (such as *UnfairnessCoV* [25, 31] or *Jain's Fairness Index* [29]). In particular, the coefficient of variation (CoV) of per-application slowdowns is defined as the ratio of the standard deviation of slowdowns (denoted as $\sigma_{Slowdown}$) and their arithmetic average (i.e., $\mu_{Slowdown}$).

The lower-is-better *FairSlowdown* metric, also known as *Average Normalized Turnaround Time (ANTT)* [11], employs a slightly different notion of fairness than the rest. However, recent works on LLC-partitioning still consider it as a complementary fairness metric [31, 36], so we opted to include it in our analysis.

3 RELATED WORK

In the last decade, a plethora of LLC-partitioning strategies has been proposed to mitigate shared-resource contention effects [8, 10, 13, 14, 20, 22–24, 26, 27, 29–31]. Putting aside their targeted optimization metrics, these policies can be broadly classified into two groups [30]: those exploiting *strict* cache partitioning, and the ones that leverage *cache-clustering* (also known as *partition-sharing*) algorithms. While strict cache-partitioning policies [8, 25, 26, 28, 29] reserve a separate non-overlapping partition for each application, cache-clustering strategies [10, 14, 24, 27, 30, 31] may map several applications to the same partition. The strategies of the latter kind deliver improved performance and fairness [10, 30, 31] on off-the-shelf processors with hardware LLC-partitioning support, which typically allows the creation of fairly coarse-grained partitions. Moreover, unlike strict cache-partitioning strategies [25, 28, 29], cache-clustering still allows partitioning the LLC when the application count exceeds the number of cache ways [15].

In this work, we do not propose LLC-partitioning policies or analyze the effectiveness of existing ones. Instead we determine the cache-clustering solution that optimizes each metric in Table 2 for different workloads, so as to analyze the interrelationship between these metrics, and capture the degree of improvement that each one reports. No previous work has carried out this analysis.

Earlier articles have also explored the optimal cache partition-sharing problem [6, 14, 30]. None of them conducts metric analyses like ours. Brock et al. [6] show that when the cache supports very fine-grained partitioning, optimal strict cache-partitioning and optimal cache-partition sharing operate in a close range. This, however, is uncommon in most commercial processors with HW cache-partitioning support, which implement a form of coarse-grained way-partitioning [2, 16]. For instance, in our Intel-based experimental system, cache partitions can be no smaller than 2.5MB. Garcia et al. [14] demonstrated that on the same Intel processor we used, leveraging LLC-clustering instead of strict LLC-partitioning leads to higher reductions in Unfairness as the number of co-running applications grows, achieving over a 40% average unfairness reduction when the application count matches the total way count (11). Saez et al. [30] identify recurrent patterns in the optimal fairness-wise cache-clustering solution for different workloads, and use these patterns to guide the design of a partitioning policy.

Other authors have carried out insightful metric studies. Of special attention is the work of Eyerman and Eeckhout [12], which points out that weighted-IPC metrics are more widely used than raw-IPC metrics in computer-architecture research articles. This work also enumerates a number of issues of raw-IPC metrics, that suggest their avoidance. Vandierendonck et al. [35] conducted a study on metrics for fairness assessment on multithreaded processors. As opposed to these works [12, 35], which do not focus on LLC-partitioning and analyze a reduced set of workload scenarios and suboptimal policies, ours determines the cache-clustering solution that optimizes a number of throughput and fairness metrics used by recent works on LLC-partitioning. This allows us to comprehensively study the degree of correlation among metrics as well as to compare their relative improvement range.

Kim et al. [18] proposed five fairness metrics that exclusively depend on the number of LLC misses and the LLC miss rate of the various programs in the workload. To the best of our knowledge, no recent work on LLC-partitioning utilizes any of these metrics. Nevertheless, Navarro et al. [23] recently proposed the M1 fairness metric (see Table 2), which is directly inspired by the work of Kim et al. [18]. Our analysis does consider this metric.

4 SIMULATION ANALYSIS AND DISCUSSION

In this section we begin by describing the methodology and the simulation environment we used for our analysis. We then proceed to discuss the main takeaways of our study.

4.1 Methodology and simulation environment

Goal. We aim to capture the interrelationship among the metrics introduced in Sec. 2, and illustrate the diversity in their relative improvement range. To this end, we employed a set of multi-program workloads consisting of SPEC CPU2006 and CPU2017 benchmarks. Previous works on cache-clustering policies primarily employ SPEC CPU benchmarks for evaluation [10, 14, 24, 27, 31], as these programs exhibit a varying degree of competition for LLC space and memory bandwidth. For each workload, we determine the *LLC-clustering* (partition-sharing) solution that optimizes each individual metric. As the base system for our analysis, we considered a platform equipped with a 20-core Intel Xeon Gold 6138 "Skylake" processor where cores run at 2 GHz. All cores have two private

cache levels (64KB L1 + 1MB L2), and share an 11-way 27.5MiB LLC (L3). The processor supports way-partitioning of the LLC [16], enabling us to create partitions no smaller than 2.5MiB (1 cache way).

Challenge. Partitioning the cache optimally with strict LLC-partitioning for a given workload and optimization goal is an NP-hard problem [15, 22]. Notably, finding the optimal cache-clustering solution (what we pursue) adds a new level of complexity to strict partitioning. Essentially, we must determine how to best group applications into clusters (i.e. set of applications sharing the same LLC partition), and also find the optimal distribution of the total LLC ways in the platform (denoted as W) across clusters. Garcia et al. [15] showed that the number of possible options to organize N applications into clusters when $N \leq W$, matches the N-th Bell number –denoted as $\mathcal{B}_N$–; when $N > W$, the number of choices is upper bounded by $\mathcal{B}_N$ [15]. So for example, for an 11-application workload, there exist $\mathcal{B}_{11}$ (678570) possible clustering options in the aforementioned Intel system. For a 16-application and a 17-application workload the number of possible clustering choices amounts to roughly 10.48 billions and 82.8 billions, respectively. Recall that to find the optimal solution for a given optimization metric, the optimal distribution of LLC ways between clusters still needs to be determined for each possible clustering option. This process requires the extensive exploration of a vast search space, whose size grows exponentially with W and the application count. For that reason, conducting our analysis is unfeasible in a real system, as it requires running a huge amount of experiments, even for a single workload.

Simulation platform. A feasible course of action to navigate these complexities to approximate per-metric optimal cache-clustering solutions is relying on simulations. Specifically, we turned to PBBCache [15], an open-source simulation framework for rapid prototyping of LLC-partitioning policies. PBBCache approximates the performance degradation (aka. slowdown) experienced by each application in a multi-program workload under a given partitioning strategy, taking into account both LLC sharing and memory-bandwidth contention [15]. To determine the per-application slowdown, the simulator relies on offline-collected applications' performance data – such as IPC, memory bandwidth consumption, etc. – gathered beforehand for different LLC sizes in isolation on a target platform (the aforementioned real Intel-based system in our case). Based on the obtained per-application slowdowns, PBBCache calculates the value of the applications' IPC in the multi-program setting, as well as various system-wide metrics, such as those presented in Sec. 2. PBBCache also implements a branch-and-bound based parallel algorithm that finds the optimal cache-clustering solution for a certain workload and chosen optimization metric. We should highlight that to conduct the analysis of this work, we had to extend PBBCache's functionality to fully support all the metrics considered in this article. The major changes in the simulator affected the parallel optimizer used for finding the optimal solution, which requires metric-specific heuristics to prune the search space when possible [15]. Note also that for specific metrics, such as Unfairness, multiple cache-clustering solutions may render the optimal metric value. In this scenario, the parallel optimizer identifies as optimal the solution that yields the maximum STP value [15].

Methodology and workloads. We gathered the simulator's input data for the various SPEC CPU programs considered in our study using the aforementioned Intel-based server system, which constitutes one of the platforms where PBBCache already has been validated (system referred to as *Platform A* in [15]). Completing all the simulations required the exclusive use for 32 days of a 128-core server platform, equipped with 2 AMD EPYC 7742 processors and 512GB DDR4 SDRAM. For our study, we employed 172 randomly generated multi-program workloads, including mixes of 8, 12, 16 and 20 SPEC CPU programs each. In generating the workloads, we adopted the classification method introduced in prior work [30], which divides applications into three categories based on their overall degree of cache sensitivity and contentiousness: (1) highly and mildly cache-sensitive programs whose performance drops substantially under a low LLC occupancy; (2) bandwidth-intensive cache-insensitive programs that pollute the LLC, while not delivering much performance benefit from extra cache space; and (3) programs that are not contentious to others and exhibit a largely LLC-insensitive behavior, as most of their working set fits in the private cache levels. Each workload combines a varying amount of SPEC programs of each category. Our simulations use as input the average value of different performance metrics gathered with hardware counters during the execution of the first 150 billion instructions of each program running alone on the aforementioned Intel processor with different LLC way counts (from 1 to 11).

4.2 Discussion and main insights

In what follows we denote the set of optimal solutions for all workloads and a particular optimization metric as *Opt-<metric-name>*. So for instance, Opt-Unf refers to the theoretical policy that achieves the optimal (minimal) value for the Unfairness metric. In most charts, we report the degree of improvement of each policy/theoretical solution with respect to not partitioning the LLC; we refer to this baseline as *NoPart*, widely used by previous work [10, 14, 27, 30, 31].

After a thorough analysis of the simulation results, the main takeaways of our study are as follows.

First, *the metric choice substantially impacts the relative improvement reported, especially when it comes to fairness metrics.* This is illustrated in Fig. 1, which shows the per-metric distribution of relative improvements for the different workloads and optimal schemes. As a reference, we also report the results associated with two heuristic policies: KPart [10] –a throughput-optimized policy–, and LFOC+ [30] –a fairness-conscious strategy. Note, however, that analyzing suboptimal (heuristic) policies is not the goal of this work. By focusing on the throughput metrics (Figs. 1(a-d)), we observe that HmeanSpeedup (Fig. 1d) is the metric that generally renders the highest relative throughput improvements (of up to 12%), which stands in contrast with the STP and AggregateIPC counterparts (no bigger than 7% and 5%, respectively). The improvement ranges associated with the GmeanIPC and GmeanSpeedup metrics exhibit throughput gains usually no higher than 9%. Among the fairness metrics, UnfairnessCoV is the metric that renders the widest gain range; its maximum improvement (83%) –closely followed by the M1 metric (up 78%)– is substantially higher than that of the Unfairness metric (up to 38.3%), and especially the FairSlowdown metric (gains≤10%). Our results also reveal that the relative gains in Jain's

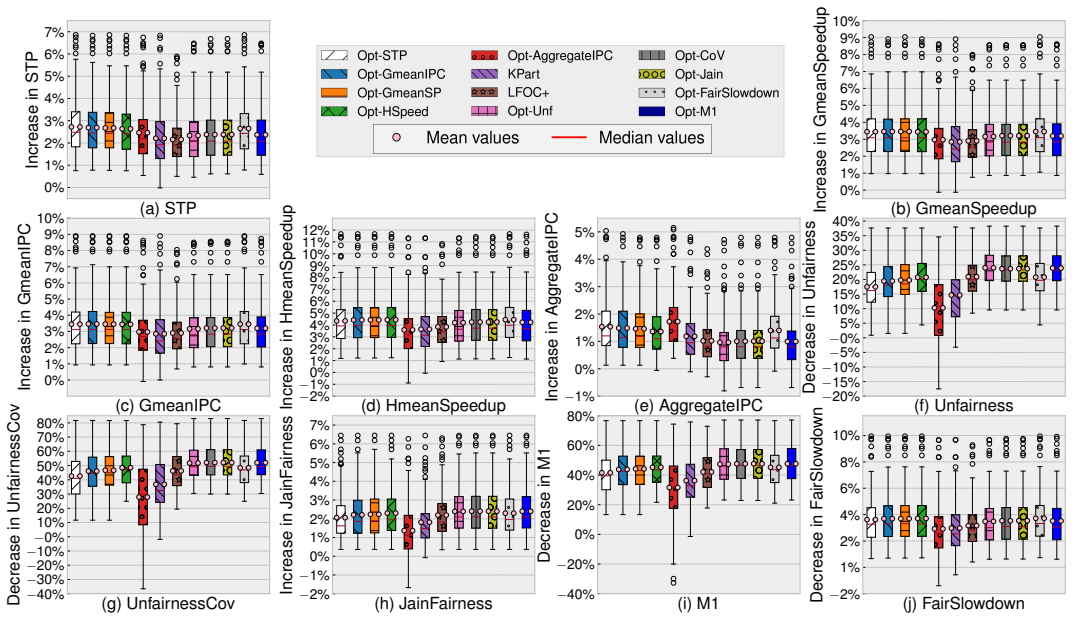


Figure 1: Distribution of improvements for the various metrics with respect to NoPart for the different schemes.

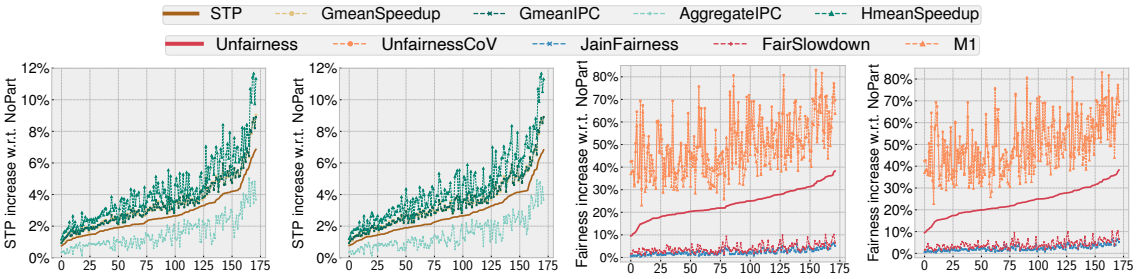


Figure 2: Correlation between throughput and fairness metrics across workloads and optimization objectives, sorted by STP gains –charts (a) and (b)– and Unfairness reductions –charts (c) and (d).

Fairness Index reflect very subtle percentage differences between cache-clustering solutions, with gains of up to 7.3% w.r.t *NoPart*.

Second, *the simulation results showcase a very strong correlation between most of the throughput metrics considered (in terms of their improvements), despite the disparities in range; however, this is not the case for the fairness metrics.* To show this graphically, Fig. 2 depicts the relative improvements for all throughput (Figs. 2a and 2b) and fairness (Figs. 2c and 2d) metrics across all workloads under a few selected optimal schemes. To show the degree of correlation among metrics at a glance, workloads have been sorted in ascending order by their STP gains (in Figs. 2a and 2b) and Unfairness reductions (in Figs. 2c and 2d). Because the trends are very similar across the optimal solutions for the various metrics, we opted to display the figures of just a few optimal schemes. In Figs. 2a and 2b we observe that the differences in throughput gains displayed for two workloads with consecutive IDs in the x axis, do not usually exceed 2% regardless of the metric. This stands in contrast with the gains rendered by the various fairness metrics. Specifically, Figs. 2a and 2b reveal large disparities in UnfairnessCoV gains across workloads with neighboring IDs (in this case, a higher ID comes with a higher Unfairness reduction). This is a clear signal of the weak correlation between the UnfairnessCoV and the Unfairness metrics.

The correlation matrices depicted in Table 3 capture more effectively the generally more modest correlations among fairness metrics vs. throughput metrics. In particular, as Table 3a shows, the correlation factors for every pair of throughput metrics (except for those of AggregateIPC) are greater than 0.945. Later in this section, we will explain the reason of the weaker correlation of AggregateIPC with the remaining metrics, and showcase the main issues of this metric. As for the fairness indicators, their correlation matrix (Table 3b) shows substantially lower factors w.r.t. their throughput counterparts. The only exception is the high correlation between the M1 and UnfairnessCoV metrics (0.96). This does not come as a surprise, considering that both metrics ultimately factor in the

Table 3: Correlation matrices for throughput and fairness metrics. The correlation factors have been calculated taking into account the whole set of simulations with all workloads and optimal schemes.

(a) Correlation matrix for throughput metrics

	STP	GmeanSp.	GmeanIPC	AggregateIPC	HmeanSp.
STP	1.0000	0.9839	0.9842	0.8754	0.9451
GmeanSpeedup	0.9839	1.0000	0.9994	0.8097	0.9873
GmeanIPC	0.9842	0.9994	1.0000	0.8108	0.9875
AggregateIPC	0.8754	0.8097	0.8108	1.0000	0.7381
HmeanSpeedup	0.9451	0.9873	0.9875	0.7381	1.0000

(b) Correlation matrix for fairness metrics

	Unfairness	Unf-CoV	JainFairness	FairSlowdown	M1
Unfairness	1.0000	0.7892	-0.7526	0.5279	0.7131
Unf-CoV	0.7892	1.0000	-0.4916	0.4314	0.9563
JainFairness	-0.7526	-0.4916	1.0000	-0.8351	-0.5493
FairSlowdown	0.5279	0.4314	-0.8351	1.0000	0.5969
M1	0.7131	0.9563	-0.5493	0.5969	1.0000

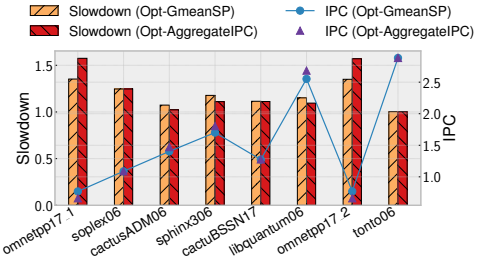


Figure 3: Slowdown and IPC of various co-running programs delivered by Opt-GmeanSP and Opt-AggregateIPC.

differences among per-application slowdowns. For the remaining metric pairs, the absolute value of all correlation coefficients falls below 0.83. Note that JainFairness is a higher-is-better metric –unlike the other metrics–, hence the negative correlation coefficients in the matrix. Surprisingly, JainFairness and FairSlowdown – the two fairness indicators with the lowest improvement potential (Fig. 1) – are those exhibiting the tightest correlation (0.83).

Third, *our experiments reveal that, unlike what previous work claims [12], not all raw-IPC metrics are misleading throughput indicators.* Specifically, the relative gains observed across workloads and schemes for GmeanIPC (raw-IPC metric) [25, 27] and GmeanSpeedup (weighted-IPC metric) are strongly correlated (0.9994 correlation factor). More importantly, GmeanIPC and GmeanSpeedup render almost identical relative gains across the board. By contrast, our simulations also underscore that AggregateIPC is indeed a misleading throughput indicator. The main problem is that in optimizing AggregateIPC, high-IPC applications are systematically favored over low-IPC programs; regrettably, this renders modest improvements in AggregateIPC compared to what is achieved when striving to optimize the other throughput metrics individually. Moreover, in favoring high-IPC programs, the application's slowdown is not directly taken into consideration, which frequently leads to consistent degradation of the remaining throughput metrics. This issue is depicted in Fig. 3, which shows the slowdown and IPC of each application in a workload under Opt-GmeanSP and Opt-AggregateIPC. Notably, Opt-AggregateIPC gives a preferential treatment to the `libquantum` program due to its relatively high IPC; it grants more effective cache space to this program than what Opt-GmeanSP does. This comes at the expense of allocating a smaller cache partition to `omnetpp17` (a low-IPC program, with two instances present in the workload) compared to the situation under Opt-GmeanSP. This causes `omnetpp17`'s slowdown to increase substantially vs. Opt-GmeanSP, in exchange for very small IPC differences between optimal schemes. As a result, Opt-AggregateIPC achieves just a 0.7% AggregateIPC increase vs. Opt-GmeanSP, at the expense of degrading the GmeanSpeedup metric by 2%.

5 CONCLUSIONS

In this paper, we have conducted a simulation-based study to shed light on the impact that the choice of different metrics for throughput and fairness assessment has on the relative gains provided by cache-partitioning policies. Our work is motivated by the lack of consensus on the throughput and fairness metrics used by works on LLC-partitioning in the last few years. Unlike other metric-centric studies [12, 35] ours relies on determining the optimal cache-clustering (partition-sharing) solution that optimizes a number of popular metrics, and explicitly focuses on LLC-partitioning.

We show that the chosen metric profoundly impacts the reported improvement w.r.t. a baseline, as each metric exhibits a very different improvement range. Specifically, the HmeanSpeedup and UnfairnessCoV metrics greatly magnify the throughput and fairness gains, respectively, reporting percentage improvements that often duplicate those of other popular metrics commonly used in the context of LLC-partitioning, such as STP (throughput), and Unfairness. Our study also showcases a strong correlation between the improvements reported by popular throughput metrics, albeit most fairness metrics exhibit a substantially weaker correlation.

In light of the results we propose the following recommendations to facilitate the quantitative comparison among works. First, any of the surveyed throughput metrics except from the AggregateIPC (a fairly misleading throughput indicator) could be used alone to report throughput; however, the disparity in the improvement range of the various metrics, needs to be carefully taken into consideration when drawing conclusions from independent studies. Second, the GmeanIPC metric [25, 27] could still be considered a safe indicator to report throughput in the context of LLC-partitioning, despite being a raw-IPC metric [12]. We show that this metric (unlike AggregateIPC) exhibits a strong correlation with weighted-IPC throughput metrics, which rely on per-application slowdowns. Third, reporting the value of a single fairness metric is not enough to comprehensively assess the degree of fairness of a policy and facilitate the comparison among works. Instead, we propose the utilization of a throughput metric in combination with the three most popular, yet complementary, fairness metrics: Unfairness [30], UnfairnessCoV [31] and FairSlowdown (aka ANTT [31]). Our study revealed that Jain's Fairness Index [29] renders a very tiny improvement range in the context of LLC-partitioning compared to the other fairness metrics. In addition, the M1 metric –only used by one recent work [23]– is largely equivalent to the more popular UnfairnessCov metric, as both metrics are tightly correlated.

ACKNOWLEDGMENTS

This work has been supported by the Spanish MCIN under Grant PID2021-126576NB-I00, funded also by MCIN/AEI/10.13039/501100011033 and by "ERDF A way of making Europe".

REFERENCES

[1] AMD. 2020. High Performance Computing: Tuning Guide for AMD EPYC 7002 Series Processors. https://developer.amd.com/wp-content/resources/56827-1-0.pdf. Accessed: 2021-05-21.

[2] AMD. 2022. AMD64 Technology Platform QoS Extensions. https://developer.amd.com/wp-content/resources/56375.pdf.

[3] C. Bilbao, J. Saez, and M. Prieto-Matias. 2023. Divide&content: A Fair OS-Level Resource Manager for Contention Balancing on NUMA Multicores. *IEEE Transactions on Parallel and Distributed Systems* 01 (aug 2023), 1–17. https://doi.org/10.1109/TPDS.2023.3309999

[4] Carlos Bilbao, Juan Carlos Saez, and Manuel Prieto-Matias. 2023. Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake. *Concurrency and Computation: Practice and Experience* In press (2023), e7814. https://doi.org/10.1002/cpe.7814

[5] Sergey Blagodurov, Sergey Zhuravlev, and Alexandra Fedorova. 2010. Contention-Aware Scheduling on Multicore Systems. *ACM Trans. Comput. Syst.* 28, 4, Article 8 (Dec. 2010), 45 pages.

[6] J. Brock et al. 2015. Optimal Cache Partition-Sharing. In *Proc. of ICPP '15'*. 749–758.

[7] Ruobing Chen et al. 2021. DRLPart: A Deep Reinforcement Learning Framework for Optimally Efficient and Robust Resource Partitioning on Commodity Servers. In *Proc. of HPDC '21*. https://doi.org/10.1145/3431379.3460648

[8] Shuang Chen et al. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proc. of ASPLOS '19* (Providence, RI, USA). 107–120.

[9] E. Ebrahimi et al. 2010. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *Proc. of ASPLOS '10*. 335–346.

[10] N. El-Sayed et al. 2018. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *Proc. of HPCA '18*. 104–117.

[11] S. Eyerman and L. Eeckhout. 2008. System-Level Performance Metrics for Multiprogram Workloads. *IEEE Micro* 28 (2008), 42–53.

[12] Stijn Eyerman and Lieven Eeckhout. 2014. Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance. *IEEE Computer Architecture Letters* 13 (2014), 93–96. Issue 2. https://doi.org/10.1109/L-CA.2013.9

[13] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. 2016. Ginseng: Market-driven LLC Allocation. In *Proc. USENIX ATC '16*. 295–308.

[14] Adrian Garcia-Garcia et al. 2019. LFOC: A Lightweight Fairness-Oriented Cache Clustering Policy for Commodity Multicores. In *Proc. of ICPP'19*. Article 14, 10 pages.

[15] Adrian Garcia-Garcia et al. 2020. PBBCache: an open-source parallel simulator for rapid prototyping and evaluation of cache-partitioning and cache-clustering policies. *J. Computat. Science* (2020), 101102.

[16] Intel. 2022. Intel® 64 and IA-32 Architectures Developer's Manual: Vol. 3B. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[17] D. Kaseridis, M. F. Iqbal, and L. K. John. 2014. Cache Friendliness-Aware Management of Shared Last-Level Caches for High Performance Multi-Core Systems. *IEEE Trans. Comput.* 63, 4 (April 2014), 874–887. https://doi.org/10.1109/TC.2013.18

[18] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. 2004. Fair Cache Sharing and Partitioning in a Chip Multiprocessor Architecture. (2004), 111–122.

[19] Neeraj Kulkarni et al. 2020. CuttleSys: Data-Driven Resource Management for Interactive Services on Reconfigurable Multicores. In *Proc. of MICRO '20*. 650–664. https://doi.org/10.1109/MICRO50266.2020.00060

[20] D. Lo et al. 2015. Heracles: improving resource efficiency at scale. In *Proc. of ISCA '15*. 450–462.

[21] Kun Luo, J. Gummaraju, and M. Franklin. 2001. Balancing thoughput and fairness in SMT processors. In *ISPASS*. 164–171.

[22] S. Mittal. 2017. A Survey of Techniques for Cache Partitioning in Multicore Processors. *ACM Comput. Surv.* 50(2), Article 27 (2017), 27:1–39 pages.

[23] Agustín Navarro-Torres, Jesús Alastruey-Benedé, Pablo Ibáñez, and Víctor Viñals-Yúfera. 2023. BALANCER: bandwidth allocation and cache partitioning for multicore processors. *The Journal of Supercomputing* 79, 9 (2023), 10252–10276.

[24] Konstantinos Nikas et al. 2019. DICER: Diligent Cache Partitioning for Efficient Workload Consolidation. In *Proc. of ICPP '19* (Kyoto, Japan). Article 15, 10 pages.

[25] Jinsu Park et al. 2019. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proc. of EuroSys '19* (Dresden, Germany). Article 10, 16 pages.

[26] Tirthak Patel and Devesh Tiwari. 2020. CLITE: Efficient and QoS-Aware Co-Location of Multiple Latency-Critical Jobs for Warehouse Scale Computers. In *Proc. of HPCA '20)*. 193–206. https://doi.org/10.1109/HPCA47549.2020.00025

[27] Lucia Pons et al. 2022. Cache-Poll: Containing Pollution in Non-Inclusive Caches Through Cache Partitioning. In *Proc. of ICPP' 22* (Bordeaux, France). Article 33, 11 pages.

[28] M.K. Qureshi and Y.N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proc. of MICRO' 06*. 423–432.

[29] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2021. SATORI: Efficient and Fair Resource Partitioning by Sacrificing Short-Term Benefits for Long-Term Gains. In *Proc. of ISCA '21*. 292–305.

[30] Juan Carlos Saez et al. 2021. LFOC+: A Fair OS-level Cache-Clustering Policy for Commodity Multicore Systems. *IEEE Trans. Comp.* (2021).

[31] V. Selfa et al. 2017. Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology. In *Proc. of PACT '17*. 194–205.

[32] Mohammad Shahrad, Sameh Elnikety, and Ricardo Bianchini. 2021. Provisioning Differentiated Last-Level Cache Allocations to VMs in Public Clouds. In *Proc. of SoCC '21*. 319–334.

[33] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. 2008. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, USA) *(ASPLOS XIII)*. 277–286.

[34] K. Van Craeynest et al. 2013. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *22nd Int'l Conf. Parallel Arch. Compilation Techniques (PACT 13)*. 177–187.

[35] Hans Vandierendonck and Andre Seznec. 2011. Fairness Metrics for Multi-Threaded Processors. *IEEE Computer Architecture Letters* 10, 1 (2011), 4–7. https://doi.org/10.1109/L-CA.2011.1

[36] Yaocheng Xiang et al. 2018. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proc. of EuroSys '18* (Porto, Portugal). Article 13, 15 pages.

[37] D. Xu et al. 2012. Providing Fairness on Shared-memory Multiprocessors via Process Scheduling. In *Proc. of SIGMETRICS' 12*. 12 pages.

[38] H. Yun et al. 2016. Memory Bandwidth Management for Efficient Performance Isolation in Multi-Core Platforms. *IEEE Trans. Comput.* 65, 2 (Feb 2016), 562–576.

[39] Felippe Vieira Zacarias et al. 2021. Intelligent colocation of HPC workloads. *Journal of Par. and Distrib. Computing* 151 (2021), 125–137.

[40] S. Zhuravlev et al. 2012. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Comput. Surv.* 45, 1, Article 4 (Dec. 2012), 28 pages.

Appendix C

Thesis publications

Journal articles

- Carlos Bilbao, Juan Carlos Saez, Manuel Prieto-Matias, *Divide&Content: A Fair OS-Level Resource Manager for Contention Balancing on NUMA Multicores*, IEEE Transactions on Parallel and Distributed Systems, 2023. JCR 2023, Impact Factor: 5.6, Q1 in Computer Science, Theory & Methods.

- Carlos Bilbao, Juan Carlos Saez, Manuel Prieto-Matias, *Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case for Intel Alder Lake*. Concurrency and Computation: Practice and Experience, 2023. JCR 2023, Impact Factor: 1.5, Q2 in Computer Science, Theory & Methods.

  - **Important Note:** This article is an extended version of the paper published in Euro-Par 22's Workshop proceedings listed below. Particularly, the workshop article was presented in the HeteroPar '22 [1] workshop, where some authors were invited to submit an extended version to a special issue of *Concurrency and Computation: Practice and Experience*. The journal article was part of this special issue. The contributions of the journal article on top of the workshop paper are described in detail in Appendix A.

Conference papers

- Javier Rubio, Carlos Bilbao, Juan Carlos Saez, Manuel Prieto-Matias, *Exploiting Elasticity via OS-runtime Cooperation to Improve CPU Utilization in Multicore Systems*. 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Dublin, Ireland, 2024. This conference ranked C in CORE 2023 [2].

[1] `https://heteropar2022.inesc-id.pt/`
[2] `https://portal.core.edu.au/conf-ranks/465/`

Workshop papers

- Carlos Bilbao, Juan Carlos Saez, Manuel Prieto-Matias, *Revisiting Fairness and Throughput Metrics for Cache-Partitioning Policy Assessment: Insights and Recommendations*. The 1st Workshop on Computer Architecture Modeling and Simulation (CAMS 2023). Toronto, Canada, 2023.
  - **Important Note:** Contributions to the CAMS '23 workshops were not published; the workshops had no associated proceedings.
- Carlos Bilbao, Juan Carlos Saez, Manuel Prieto-Matias, *Rapid Development of OS Support with PMCSched for Scheduling on Asymmetric Multicore Systems*. Euro-Par 2022: Parallel Processing Workshops. Also part of the book series Lecture Notes in Computer Science (LNCS, volume 13835).

Bibliography

[1] Intel Tick Tock Model — itchasers.com. http://www.itchasers.com/store/news/80/Intel-Tick-Tock-Model.html. [Accessed 27-03-2024].

[2] Intel® 64 and ia-32 architectures developer's manual: Vol. 3b, 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[3] Release v3.0, repository jcsaezal/pmctrack, github.com. https://github.com/jcsaezal/pmctrack/releases/tag/v3.0, January, 2023. [Accessed 17-03-2024].

[4] Blast Benchmark. https://fiehnlab.ucdavis.edu/staff/kind/collector/benchmark/blast-benchmark, n.d. Accessed: 2025-02-19.

[5] Advanced Micro Devices, Inc. (AMD). AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/solution-briefs/amd-secure-encrypted-virtualization-solution-brief.pdf, 2020.

[6] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, 2018.

[7] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Elasticity in cloud computing: State of the art and research challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, 2018.

[8] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, page 63–74, New York, NY, USA, 2008. Association for Computing Machinery.

[9] Alaa R. Alameldeen and David A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4), 2006.

[10] Jose I. Aliaga, Maribel Castillo, Sergio Iserte, Iker Martín-Álvarez, and Rafael Mayo. A survey on malleability solutions for high-performance distributed computing. *Applied Sciences*, (10), 2022.

[11] AMD. AMD64 Technology Platform QoS Extensions. `https://developer.amd.com/wp-content/resources/56375.pdf`.

[12] Murali Annavaram et al. Mitigating Amdahl's Law through EPI Throttling. In *Proc. of ISCA '05*, pages 298–309, 2005.

[13] Murali Annavaram, Ed Grochowski, and John Shen. Mitigating Amdahl's Law through EPI Throttling. In *32nd Ann. Int'l Symp. Computer Architecture (ISCA 05)*, pages 298–309, 2005.

[14] Apple. Apple introduces M4 chip. `https://www.apple.com/newsroom/2024/05/apple-introduces-m4-chip/`, 2024. Accessed: 2024-11-6.

[15] Apple, Inc. Apple M1 Chip. `https://www.apple.com/mac/m1/`, 2020. Accessed: 2020-12-5.

[16] Apple, Inc. Framework Dispatch - Apple Documentation. `https://developer.apple.com/documentation/DISPATCH`, 2025. Accessed: 2025-02-18.

[17] Gomatheeshwari B and J. Selvakumar. Appropriate allocation of workloads on performance asymmetric multicore architectures via deep learning algorithms. *Microprocessors and Microsystems vol. 73*, 2020.

[18] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks—summary and preliminary results. In *Supercomputing '91*, pages 158–165, 1991.

[19] Antonio Barbalace, Robert Lyerly, Christopher Jelesnianski, Anthony Carno, Ho-Ren Chuang, Vincent Legout, and Binoy Ravindran. Breaking the boundaries in heterogeneous-ISA datacenters. In *ACM SIGPLAN Notices*, volume 52, pages 645–659. ACM, 2017.

[20] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behaviors. In *Proceedings of the 2nd USENIX Conference on Large-Scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*, LEET'09, page 8, USA, 2009. USENIX Association.

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

[22] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proc. of PACT'08*, October 2008.

[23] Carlos Bilbao, Juan Carlos Saez, and Manuel Prieto-Matias. Flexible system software scheduling for asymmetric multicore systems with PMCSched: A case

for Intel Alder Lake. *Concurrency and Computation: Practice and Experience*, n/a:e7814.

[24] Carlos Bilbao, Juan Carlos Saez, and Manuel Prieto-Matias. Divide&Content: A Fair OS-Level Resource Manager for Contention Balancing on NUMA Multicores. *IEEE Transactions on Parallel and Distributed Systems*, 34(11):2928–2945, 2023.

[25] Carlos Bilbao, Juan Carlos Saez, and Manuel Prieto-Matias. Rapid Development of OS Support with PMCSched for Scheduling on Asymmetric Multicore Systems. In *Euro-Par 2022: Parallel Processing Workshops*, pages 184–196, 2023.

[26] Sergey Blagodurov, Alexandra Fedorova, Sergey Zhuravlev, and Ali Kamali. A Case for NUMA-Aware Contention Management on Multicore Systems. In *Proc. of USENIX ATC '11*, 2011.

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

[28] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[29] John M. Calandrino, Hennadiy Leontyev, Aaron Block, UmaMaheswari C. Devi, and James H. Anderson. LITMUS-RT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers. In *2006 27th IEEE Int'l Real-Time Systems Symposium (RTSS'06)*, pages 111–126. IEEE, 2006.

[30] Li Cha V., Vinicius Petrucci, and Daniel Mosse. Exploring machine learning for thread characterization on heterogeneous multiprocessors. *SIGOPS Oper. Syst. Rev.*, 51(1):113–123, sep 2017.

[31] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc of IISWC '09*, pages 44–54, 2009.

[32] P.M. Chen and B.D. Noble. When virtual is better than real [operating system relocation to virtual machines]. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 133–138, 2001.

[33] Ruobing Chen, Jinping Wu, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. DRLPart: A Deep Reinforcement Learning Framework for Optimally Efficient and Robust Resource Partitioning on Commodity Servers. In *Proc. of HPDC '21*, 2021.

[34] Shuang Chen, Christina Delimitrou, and José F. Martínez. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *Proc. of ASPLOS '19*, page 107–120, 2019.

[35] Hamidreza Chitsaz et al. A partition function algorithm for interacting nucleic acid strands. *Bioinformatics*, 25(12):i365–i373, 2009.

[36] Younghyun Cho, Camilo A. Celis Guzman, and Bernhard Egger. Maximizing system utilization via parallelism management for co-located parallel applications. In *PACT*, In Proc. of PACT '18, 2018.

[37] Kallia Chronaki, Miquel Moretó, Marc Casas, Alejandro Rico, Rosa M. Badia, Eduard Ayguadé, and Mateo Valero. On the maturity of parallel applications for asymmetric multi-core processors. *J. Par. Distrib. Comput.*, 127:105–115, 2019.

[38] Kallia Chronaki, Alejandro Rico, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero. Criticality-aware dynamic task scheduling for heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS 2015, pages 329–338, 2015.

[39] Confidential Computing Consortium. A Technical Analysis of Confidential Computing v1.3. `https://confidentialcomputing.io/wp-content/uploads/sites/10/2023/03/CCC-A-Technical-Analysis-of-Confidential-Computing-v1.3_unlocked.pdf`, 2022.

[40] Lucian Constantin. MSE false positive detection forces Google to update Chrome. `https://web.archive.org/web/20111004035408/http://www.theinquirer.net/inquirer/news/2113892/mse-false-positive-detection-forces-google-update-chrome`, October 2011. Accessed April 4, 2024.

[41] Ian Cutress. Intel's 'Tick-Tock' Seemingly Dead, Becomes 'Process-Architecture-Optimization'. *AnandTech*, March 22 2016. `https://www.anandtech.com/show/10183/intels-tick-tock-seemingly-dead-becomes-process-architecture-optimization` [Accessed 2024-04-29].

[42] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proc. of ASPLOS '13*, 2013.

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

[44] Nicolas Denoyelle, Brice Goglin, Emmanuel Jeannot, and Thomas Ropars. Data and Thread Placement in NUMA Architectures: A Statistical Learning Approach. In *Proc. of ICPP '19*, 2019.

[45] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *15th Int'l Conf. Architectural Support Programming Lang. and Oper. Syst. (ASPLOS 10)*, pages 335–346, 2010.

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

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

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

[49] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE Micro*, 28(3), 2008.

[50] Stijn Eyerman and Lieven Eeckhout. Restating the case for weighted-IPC metrics to evaluate multiprogram workload performance. *IEEE Computer Architecture Letters*, 13:93–96, 2014.

[51] Stijn Eyerman and Lieven 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 '14, page 591–606, New York, NY, USA, 2014. Association for Computing Machinery.

[52] Josue Feliu, Julio Sahuquillo, Salvador Petit, and Jose Duato. Perf & Fair: a Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores. *IEEE Transactions on Computers*, PP(99), 2016.

[53] Josue Feliu, Julio Sahuquillo, Salvador Petit, and Jose Duato. Perf&Fair: A Progress-Aware Scheduler to Enhance Performance and Fairness in SMT Multicores. *IEEE Trans. Comput.*, 66(5):905–911, May 2017.

[54] Financial Times. Nvidia's revenue soars 262% on record AI chip demand. `https://www.ft.com/content/2ce59a81-61b7-4052-810e-8bdc425367e4`, 2024. Accessed: 2024-07-20.

[55] G Fragkos, O Angelopoulou, and K Xynos. Antivirus false-positive alerts, evading malware detection, and cyber-security issues. *Journal of Information Warfare*, 12(3):26–40, 2013.

[56] Guilherme Galante and Luis Carlos Erpen De Bona. A programming-level approach for elasticizing parallel scientific applications. *Journal of Systems and Software*, 110:239–252, 2015.

[57] Adrian Garcia-Garcia, Juan Carlos Saez, Fernando Castro, and Manuel Prieto-Matias. LFOC: A lightweight fairness-oriented cache clustering policy for commodity multicores. In *Proc. of ICPP'19*, 2019.

[58] Adrian Garcia-Garcia, Juan Carlos Saez, and Manuel Prieto-Matias. Contention-aware fair scheduling for asymmetric single-ISA multicore systems. *IEEE Transactions on Computers*, 67(12):1703–1719, Dec 2018.

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

[60] Adrián García García. *Contention-aware scheduling and resource management for emerging multicore architectures*. PhD thesis, Universidad Complutense de Madrid, 2021.

[61] Saugata Ghose. *General-Purpose Multicore Architectures*, pages 595–643. Springer Nature Singapore, Singapore, 2025.

[62] Matt Gillespie. Preparing for The Second Stage of Multi-Core HW: Asymmetric (Heterogeneous) Cores. *Intel White Paper*, 2008.

[63] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *6th USENIX Security Symposium (USENIX Security 96)*, San Jose, CA, July 1996. USENIX Association.

[64] Vineeth Pillai (Google). Pvsched: Paravirt scheduling framework (RFC), 2024. Accessed: 2024-12-15.

[65] David Gureya, Joao Neto, Reza Karimi, Joao Barreto, Pramod Bhatotia, Vivien Quema, Rodrigo Rodrigues, Paolo Romano, and Vladimir Vlassov. Bandwidth-aware page placement in NUMA. In *Proc. of IPDPS '20*, pages 546–556, 2020.

[66] Md E. Haque, Yuxiong He, Sameh Elnikety, Thu D. Nguyen, Ricardo Bianchini, and Kathryn S. McKinley. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *50th Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO 17)*, pages 625–638, 2017.

[67] Tim Harris, Martin Maas, and Virendra J. Marathe. Callisto: Co-scheduling parallel runtime systems. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

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

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

[70] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Volumes 3A and 3B: System Programming Guide. `http://www.intel.com/products/processor/manuals`. Accessed: 2015-01-15.

[71] Intel. Optimizing software for x86 hybrid architecture. *Intel White Paper*, October 2021.

[72] Intel. 13th Generation Intel Core and Intel Core 14th Generation Processors. `https://edc.intel.com/content/www/us/en/design/products/platforms/details/raptor-lake-s/13th-generation-core-processors-datasheet-volume-1-of-2/`, 2022. Accessed: 2024-11-6.

[73] C. Isci and M. Martonosi. Detecting recurrent phase behavior under real-system variability. In *IEEE International. 2005 Proceedings of the IEEE Workload Characterization Symposium, 2005.*, pages 13–23, 2005.

[74] Canturk Isci, Gilberto Contreras, and Margaret Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 359–370, 2006.

[75] Sian Jin, Guanpeng Li, Shuaiwen Leon Song, and Dingwen Tao. A Novel Memory-Efficient Deep Learning Training Framework via Error-Bounded Lossy Compression. *CoRR*, abs/2011.09017, 2020.

[76] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. `https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf`, 2021.

[77] Vahid Kazempour, Ali Kamali, and Alexandra Fedorova. AASH: an asymmetry-aware scheduler for hypervisors. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, page 85–96, New York, NY, USA, 2010. Association for Computing Machinery.

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

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

[80] Kubilay Ahmet Küçük, Andrew Paverd, Andrew Martin, N. Asokan, Andrew Simpson, and Robin Ankele. Exploring the use of Intel SGX for Secure Many-Party Applications. In *Proceedings of the 1st Workshop on System Software for Trusted Execution*, SysTEX '16, New York, NY, USA, 2016. Association for Computing Machinery.

[81] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Single-ISA Heterogeneous Multi-Core Architectures: the Potential for Processor Power Reduction. In *36th Ann. IEEE/ACM Int'l Symp. on Microarchitecture (MICRO 03)*, 2003.

[82] Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *31st Ann. Int'l Symp. Computer Architecture (ISCA 04)*, pages 64–75, 2004.

[83] Shivam Kundan, Theodoros Marinakis, Iraklis Anagnostopoulos, and Dimitri Kagaris. A Pressure-Aware Policy for Contention Minimization on Multicores. *ACM Trans. Archit. Code Optim.*, 19(3), 2022.

[84] Youngjin Kwon, Changdae Kim, Seungryoul Maeng, and Jaehyuk Huh. Virtualizing performance asymmetric multi-core systems. In *Proceedings of ISCA '11*, 2011.

[85] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. Provable multicore schedulers with Ipanema: Application to work conservation. In *Proc. of Eurosys '20*. ACM, 2020.

[86] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 574–587, New York, NY, USA, 2023. Association for Computing Machinery.

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

[88] Li, Tong and Brett, Paul and Knauerhase, Rob and Koufaty, David and Reddy, Dheeraj and Hahn, Scott. Operating system support for overlapping-ISA heterogeneous multi-core architecture. In *16th Intl. Symp. High-Performance Computer Architecture (HPCA 10)*, pages 1–12, 2010.

[89] Linux. Using the linux kernel tracepoints. `https://www.kernel.org/doc/html/latest/trace/tracepoints.html`. Accessed: 2023-03-14.

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

[91] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, Zihao Wu, Lin Zhao, Dajiang Zhu, Xiang Li, Ning Qiang, Dingang Shen, Tianming Liu, and Bao Ge. Summary of chatgpt-related research and perspective towards the future of large language models. *Meta-Radiology*, 1(2):100017, 2023.

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

[93] Kun Luo, J. Gummaraju, and M. Franklin. Balancing thoughput and fairness in SMT processors. In *ISPASS*, pages 164–171, 2001.

[94] Robert Lyerly, Carlos Bilbao, Changwoo Min, Christopher J. Rossbach, and Binoy Ravindran. An OpenMP Runtime for Transparent Work Sharing across Cache-Incoherent Heterogeneous Nodes. *ACM Trans. Comput. Syst.*, 39(1–4), jul 2022.

[95] Zoltan Majo and Thomas R. Gross. Memory management in NUMA multicore systems: Trapped between cache contention and interconnect overhead. *SIGPLAN Not.*, 46(11):11–20, jun 2011.

[96] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[97] Jim Mauro and Richard McDougall. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Prentice Hall, 2006.

[98] Sergio Mazzola, Gabriele Ara, Thomas Benz, Björn Forsberg, Tommaso Cucinotta, and Luca Benini. Data-Driven Power Modeling and Monitoring via Hardware Performance Counters Tracking, 2024.

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

[100] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim M. Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, Jiwoo Pak, Andy Tong, Kavya Srinivasa, Will Hang, Emre Tuncer, Quoc V. Le, James Laudon, Richard Ho, Roger Carpenter, and Jeff Dean. A graph placement methodology for fast chip design. *Nature*, 604(7906):E24, 2022.

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

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

[103] Tomer Y. Morad, Noam Shalev, Idit Keidar, Avinoam Kolodny, and Uri C. Weiser. Efs: Energy-friendly scheduler for memory bandwidth constrained systems. *Journal of Parallel and Distributed Computing*, 95:3 – 14, 2016.

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

[105] Djob Mvondo, Antonio Barbalace, Jean-Pierre Lozi, and Gilles Muller. Towards user-programmable schedulers in the operating system kernel. In *Proceedings of the 11th Workshop on Systems for Post-Moore Architectures, SPMA 2022*. td-eval, April 2022.

[106] Marta Navarro, Josué Feliu, Salvador Petit, María E. Gómez, and Julio Sahuquillo. SYNPA: SMT Performance Analysis and Allocation of Threads to Cores in ARM Processors. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 705–715, 2024.

[107] Daniel Nemirovsky, Tugberk Arkose, Nikola Markovic, Mario Nemirovsky, Osman Unsal, and Adrian Cristal. A Machine Learning Approach for Performance Prediction and Scheduling on Heterogeneous CPUs. In *2017 29th Int'l Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, pages 121–128. IEEE, 2017.

[108] Ricardo Neri. Intel kernel patches posted to kernel mailing list. `https://lore.kernel.org/lkml/20220909231205.14009-1-ricardo.neri-calderon@linux.intel.com/`. Accessed: 2022-12-19.

[109] Marco A. S. Netto, Rodrigo N. Calheiros, Eduardo R. Rodrigues, Renato L. F. Cunha, and Rajkumar Buyya. HPC cloud for scientific and business applications: Taxonomy, vision, and research challenges. *ACM Comput. Surv.*, 51(1), jan 2019.

[110] K. Nguyen. Introduction to cache allocation technology in the intel xeon processor e5 v4 family. `https://software.intel.com/en-us/articles/`

introduction-to-cache-allocation-technology, 2016. Accessed: 2019-03-20.

[111] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable Parallel Programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? *Queue*, 6(2):40–53, mar 2008.

[112] Konstantinos Nikas, Nikela Papadopoulou, Dimitra Giantsidi, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. Dicer: Diligent cache partitioning for efficient workload consolidation. In *Proc. of ICPP '19*, pages 15:1–15:10, 2019.

[113] Jinsu Park, Seongbeom Park, and Woongki Baek. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Proc. of EuroSys '19*, pages 10:1–10:16, 2019.

[114] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proc. of PACT '18*, pages 5:1–5:14, 2018.

[115] D. Parker and Scott Radvan. Red Hat Enterprise Linux 7. Virtualization Tuning and Optimization Guide. https://linux.web.cern.ch/centos7/docs/rhel/Red_Hat_Enterprise_Linux-7-Virtualization_Tuning_and_Optimization_Guide-en-US.pdf, 2014.

[116] Tirthak Patel and Devesh Tiwari. Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *Proc. of HPCA '20)*, pages 193–206, 2020.

[117] Vinicius Petrucci, Orlando Loques, and Daniel Mossé. Lucky scheduling for energy-efficient heterogeneous multi-core systems. In *Proceedings of USENIX HotPower 12*, pages 7–7, Hollywood, CA, 7 October 2012. USENIX Association Berkeley, CA, USA.

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

[119] Vinicius Petrucci, Eishan Mirakhur, Rita Gupta, Makesh Wagh, Nikesh Agarwal, Su Wei Lim, and Vishal Tanna. CXL memory expansion: A closer look on an actual platform. *White Paper from Micron and AMD*, 2023.

[120] PMCTrack. project official website. https://pmctrack-linux.github.io/getting-started/. Accessed: 2024-11-07.

[121] Lucia Pons, Julio Sahuquillo, Salvador Petit, and Julio Pons. Cache-Poll: Containing Pollution in Non-Inclusive Caches Through Cache Partitioning. In *Proc. of ICPP' 22*, pages 33:1 – 33:11, 2022.

[122] Lucia Pons, Julio Sahuquillo, Vicent Selfa, Salvador Petit, and Julio Pons. Phase-aware cache partitioning to target both turnaround time and system performance. *IEEE Transactions on Parallel and Distributed Systems*, 31(11):2556–2568, 2020.

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

[124] Adrian Pousa, Juan Carlos Saez, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matias. Theoretical Study on the Performance of an Asymmetry-Aware Round-Robin Scheduler. *TR - 5028A. Dept. of Computer Architecture. UCM*, 2012.

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

[126] Emil Protalinski. AVG incorrectly flags user32.dll in Windows XP SP2/SP3. `https://arstechnica.com/information-technology/2008/11/avg-incorrectly-flags-user32-dll-in-windows-xp-sp2sp3/`, November 2008. Published at Ars Technica. Accessed April 4, 2024.

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

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

[129] Aleix Roca, Samuel Rodríguez, Albert Segura, Kevin Marquet, and Vicenç Beltran. A Linux Kernel Scheduler Extension for Multi-core Systems. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 353–362, 2019.

[130] Steven Rostedt. ftrace: Where modifying a running kernel all started. `https://kernel-recipes.org/en/2019/talks/ftrace-where-modifying-a-running-kernel-all-started/`. Accessed: 2024-12-15.

[131] Efraim Rotem, Adi Yoaz, Lihu Rappoport, Stephen J. Robinson, Julius Yuli Mandelblat, Arik Gihon, Eliezer Weissmann, Rajshree Chabukswar, Vadim Basin, Russell Fenger, Monica Gupta, and Ahmad Yasin. Intel alder lake cpu architectures. *IEEE Micro*, 2022.

[132] Michael Roth. Add AMD Secure Nested Paging (SEV-SNP) Hypervisor Support v12. `https://lwn.net/Articles/967459/`, 2024.

[133] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Satori: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains. In *Proc. of ISCA '21*, pages 292–305, 2021.

[134] Javier Rubio, Carlos Bilbao, Juan Carlos Saez, and Manuel Prieto-Matias. Exploiting elasticity via os-runtime cooperation to improve cpu utilization in multicore systems. In *2024 32nd Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 35–43, 2024.

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

[136] Juan Carlos Saez, Fernando Castro, Graziano Fanizzi, and Manuel Prieto-Matias. LFOC+: A Fair OS-Level Cache-Clustering Policy for Commodity Multicore Systems. *IEEE Transactions on Computers*, 71(8):1952–1967, 2022.

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

[138] Juan Carlos Saez, Fernando Castro, and Manuel Prieto-Matias. Enabling Performance Portability of Data-Parallel OpenMP Applications on Asymmetric Multicore Processors. In *49th Int'l Conference on Parallel Processing*, ICPP '20. ACM, 2020.

[139] Juan Carlos Saez, Alexandra Fedorova, David Koufaty, and Manuel Prieto-Matias. Leveraging core specialization via OS scheduling to improve performance on asymmetric multicore systems. *ACM Trans. Comput. Syst.*, 30(2):6:1–6:38, April 2012.

[140] Juan Carlos Saez, Alexandra Fedorova, Manuel Prieto-Matias, and Hugo Vegas. Operating System Support for Mitigating Software Scalability Bottlenecks on Asymmetric Multicore Processors. In *7th Int'l Conf. Computing Frontiers*, pages 31–40. ACM, 2010.

[141] Juan Carlos Saez, Jose Ignacio Gomez, and Manuel Prieto-Matias. Improving priority enforcement via non-work-conserving scheduling. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 99–106, 2008.

[142] Juan Carlos Saez, Adrian Pousa, Fernando Castro, Daniel Chaver, and Manuel Prieto-Matias. ACFS: A completely fair scheduler for asymmetric single-ISA multicore systems. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 2027–2032, New York, NY, USA, 2015. ACM.

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

[144] Juan Carlos Saez, Adrian Pousa, Armando De Giusti, and Manuel Prieto-Matias. On the Interplay Between Throughput, Fairness and Energy Efficiency on Asymmetric Multicore Processors. *The Computer Journal*, 61(1):74–94, 2018.

[145] Juan Carlos Saez, Adrian Pousa, Roberto Rodriguez, Fernando Castro, and Manuel Prieto-Matias. Delivering performance counter monitoring support to the OS scheduler. *The Computer Journal*, (To Appear), 2016.

[146] Juan Carlos Saez and Manuel Prieto-Matias. Evaluation of the Intel thread director technology on an Alder Lake processor. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '22, page 61–67, New York, NY, USA, 2022. Association for Computing Machinery.

[147] Juan Carlos Saez, Manuel Prieto-Matias, Alexandra Fedorova, and Sergey Blagodurov. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 139–152, New York, NY, USA, 2010. ACM.

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

[149] Ravi Sahita, Atish Patra, Vedvyas Shanbhogue, Samuel Ortiz, Andrew Bresticker, Dylan Reid, Atul Khare, and Rajnesh Kanwal. CoVE: Towards Confidential Computing on RISC-V Platforms, 2023.

[150] Bagher Salami, Hamid Noori, and Mahmoud Naghibzadeh. Online energy-efficient fair scheduling for heterogeneous multi-cores considering shared resource contention. *J. Supercomput.*, 78(6):7729–7748, apr 2022.

[151] Rohit Sehgal, Vinicius Petrucci, and Anil Godbole. Optimizing System Memory Bandwidth with Micron CXL Memory Expansion Modules on Intel Xeon6 Processors. *Intel White paper*, 2024. `https://cdrdv2-public.intel.com/842211/White-Paper-Micron-Intel-HPC-AI-Workloads.pdf`.

[152] Vicent Selfa, Julio Sahuquillo, Lieven Eeckhout, Salvador Petit, and María E. Gómez. Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 194–205, Sep. 2017.

[153] Alex Settle, Joshua Kihm, Andrew Janiszewski, and Dan Connors. Architectural Support for Enhanced SMT Job Scheduling. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, page 63–73, USA, 2004. IEEE Computer Society.

[154] Mohammad Shahrad, Sameh Elnikety, and Ricardo Bianchini. Provisioning differentiated Last-Level Cache allocations to VMs in public clouds. In *Proc. of the Symposium on Cloud Computing (SoCC)*, November 2021.

[155] Daniel Shelepov, Juan Carlos Saez Alcaide, Stacey Jeffery, Alexandra Fedorova, Nestor Perez, Zhi Feng Huang, Sergey Blagodurov, and Viren Kumar. HASS: a Scheduler for Heterogeneous Multicore Systems. *Oper. Syst. Review*, 43(2):66–75, 2009.

[156] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *ASPLOS*, 2000.

[157] Srboljub Stepanovic, Georgios Georgakarakos, Simon Holmbacka, and Johan Lilius. An efficient model for quantifying the interaction between structural properties of software and hardware in the ARM big.LITTLE architecture. *Concurrency and Computation: Practice and Experience*, 2019.

[158] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. Bliss: Balancing performance, fairness and complexity in memory access scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):3071–3087, 2016.

[159] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs. *SIGARCH CAM*, 36(1):277–286, 2008.

[160] Ashraf Suyyagh and Zeljko Zilic. Energy and Task-Aware Partitioning on Single-ISA Clustered Heterogeneous Processors. *IEEE Transactions on Parallel and Distributed Systems*, 31:306–3017, 02 2020.

[161] Lingjia Tang, Jason Mars, and Mary Lou Soffa. Contentiousness vs. sensitivity: Improving contention aware runtime systems on multicore architectures. In *1st Int'l Workshop on Adaptive Self-Tuning Comput. Syst. for the Exaflop Era*, pages 12–21, 2011.

[162] Ahmad Tarraf, Martin Schreiber, Alberto Cascajo, Jean-Baptiste Besnard, Marc-André Vef, Dominik Huber, Sonja Happ, André Brinkmann, David E. Singh, Hans-Christian Hoppe, Alberto Miranda, Antonio J. Peña, Rui Machado, Marta Garcia-Gasulla, Martin Schulz, Paul Carpenter, Simon

Pickartz, Tiberiu Rotaru, Sergio Iserte, Victor Lopez, Jorge Ejarque, Heena Sirwani, Jesus Carretero, and Felix Wolf. Malleability in Modern HPC Systems: Current Experiences, Challenges, and Future Opportunities. *IEEE Transactions on Parallel & Distributed Systems*, 35(09):1551–1564, sep 2024.

[163] Christopher Torng, Moyang Wang, and Christopher Batten. Asymmetry-aware work-stealing runtimes. In *Proc. of ISCA '16*, pages 40–52. IEEE, 2016.

[164] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. LLaMA: Open and Efficient Foundation Language Models, 2023.

[165] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. Fairness-aware scheduling on single-ISA heterogeneous multi-cores. In *22nd Int'l Conf. Parallel Arch. Compilation Techniques (PACT 13)*, pages 177–187, 2013.

[166] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *39th Ann. Int'l Symp. Computer Arch. (ISCA 12)*, pages 213–224, 9-13 June 2012.

[167] Felippe Vieira Zacarias, Vinicius Petrucci, Rajiv Nishtala, Paul Carpenter, and Daniel Mosse. Intelligent colocation of HPC workloads. *Journal of Par. and Distrib. Computing*, 151:125–137, 2021.

[168] Jianxiong Wan, Yanduo Duan, Xiang Gui, Chuyi Liu, Leixiao Li, and Zhiqiang Ma. SafeCool: Safe and Energy-Efficient Cooling Management in Data Centers With Model-Based Reinforcement Learning. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 7(6):1621–1635, 2023.

[169] Christian Wressnegger, Kevin Freeman, Fabian Yamaguchi, and Konrad Rieck. Automatically inferring malware signatures for anti-virus assisted attacks. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, page 587–598, New York, NY, USA, 2017. Association for Computing Machinery.

[170] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proc. of EuroSys '18*, pages 13:1–13:15, 2018.

[171] Di Xu, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Zhenjiang Wang. Providing fairness on shared-memory multiprocessors via process scheduling. In *Proc. ACM Int'l Conf. Measurement and Modeling Comp. Syst. (SIGMETRICS 12)*, pages 295–306, 2012.

[172] Hao Xu, Shasha Wen, Alfredo Gimenez, Todd Gamblin, and Xu Liu. DR-BW: Identifying Bandwidth Contention in NUMA Architectures with Supervised Learning. In *Proc. of IPDPS '17*, 2017.

[173] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93, 2009.

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

[175] Laiping Zhao, Yushuai Cui, Yanan Yang, Xiaobo Zhou, Tie Qiu, Keqiu Li, and Yungang Bao. Component-distinguishable co-location and resource reclamation for high-throughput computing. *ACM Trans. Comput. Syst.*, 42(1–2), feb 2024.

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

[177] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Akula: A toolset for experimenting and developing thread placement algorithms on multicore systems. In *Proceedings of PACT 10*, pages 249–260, Vienna, Austria, 11-15 September 2010. ACM, New York.

[178] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto-Matias. Survey of scheduling techniques for addressing shared resources in multicore processors. *ACM Comput. Surv.*, 45(1):4:1–4:28, December 2012.

[179] David Álvarez, Kevin Sala, and Vicenç Beltran. nOS-V: Co-Executing HPC Applications Using System-Wide Task Scheduling. `https://arxiv.org/abs/2204.10768`, 2022.