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

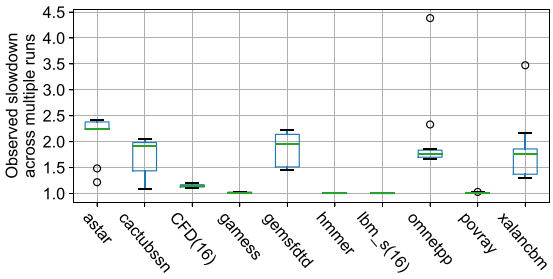


Fig. 1. Distribution of per-application slowdown across 10 runs of the same multi-program workload under the Linux default scheduler. The numbers in parentheses by the names of parallel programs indicate the number of threads they run with.

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

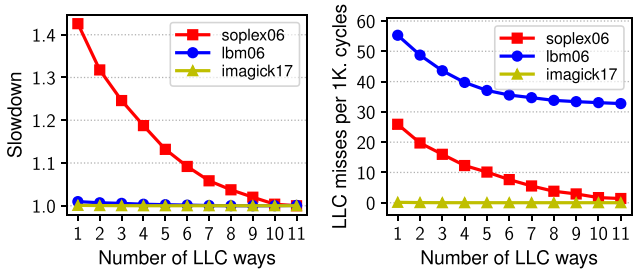


Fig. 2. Slowdown and LLCMPKC for different applications and ways.

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

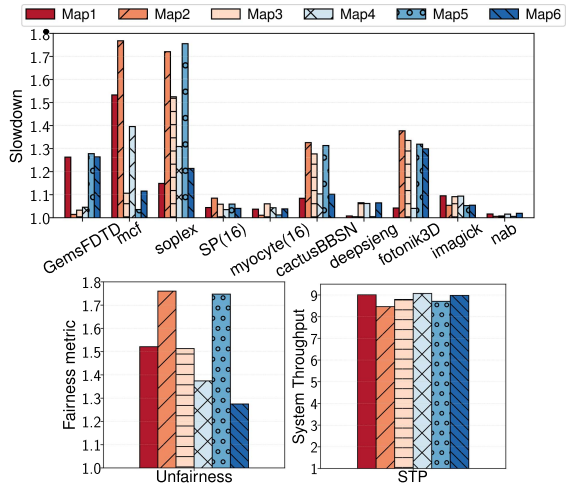


Fig. 3. Per-application slowdown (top), unfairness and throughput figures (bottom) registered for 6 different thread-to-LLC mappings of a 10-application workload under the LFOC+ LLC-partitioning policy. The numbers in parentheses by the names of the `SP` and `myocyte` (parallel) programs indicate the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

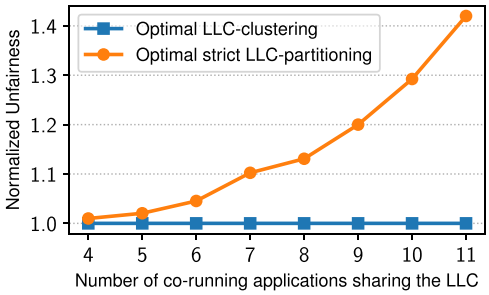


Fig. 4. Comparison between optimal strict LLC-partitioning and optimal cache-clustering for different application counts [32].

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).

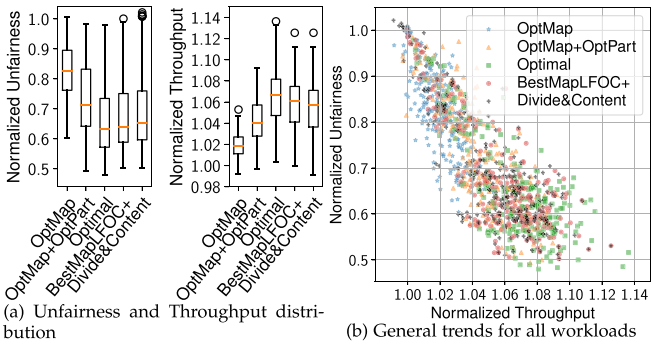
TABLE I
NUMBER OF POSSIBLE SOLUTIONS IN THE SEARCH SPACE FOR THE OPTMAP AND OPTIMAL SOLUTIONS

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.

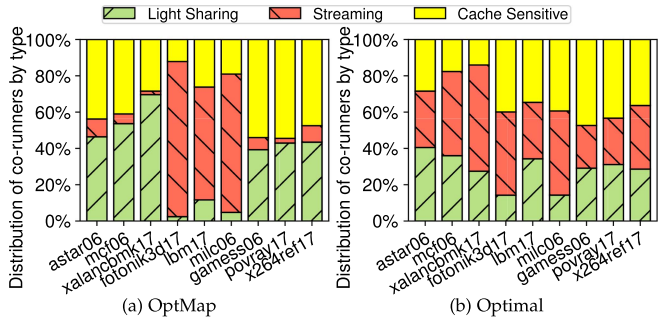
TABLE II
CACHE-SENSITIVITY CLASS OF THE DIFFERENT SPEC BENCHMARKS

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.

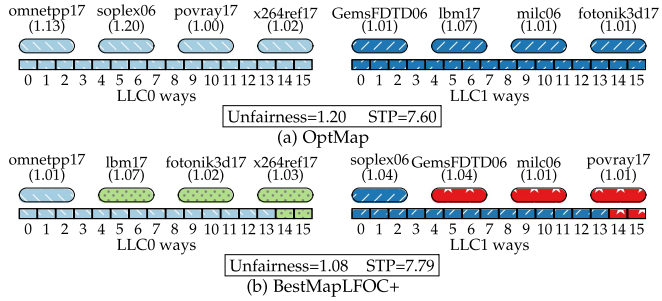


Fig. 7. Application-to-group mapping and LLC-space distribution within each group made by (a) OptMap and (b) BestMapLFOC+, for a sample workload.

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

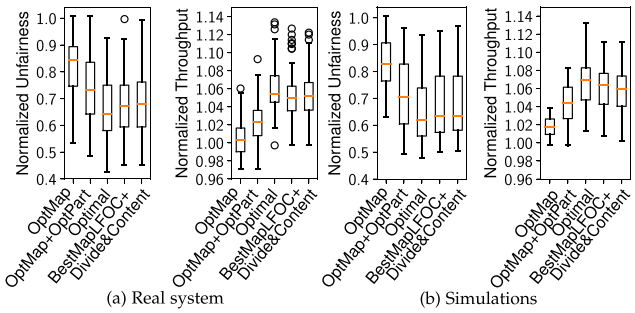


Fig. 8. Validation results for the simulations. Throughput (STP) and Unfairness values have been normalized to the results provided by a random application-to-group mapping and using no LLC-partitioning.

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

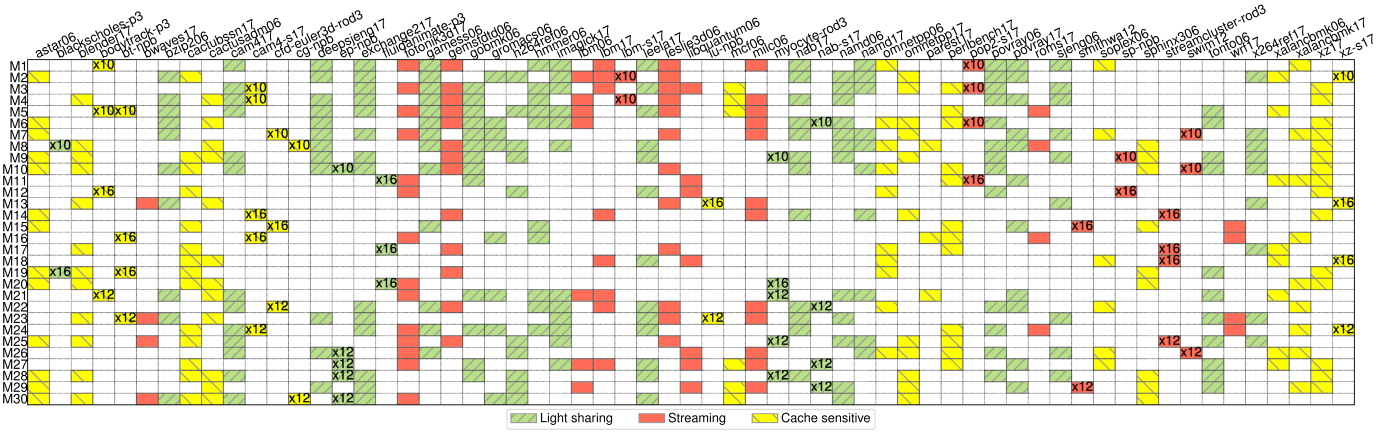


Fig. 9. Each table row M_i depicts the composition of the i-th workload used in our experiments. For each benchmark, the dominant application class observed during the execution is displayed in the associated column. When a program is not included in a workload a blank cell is used in the corresponding 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).

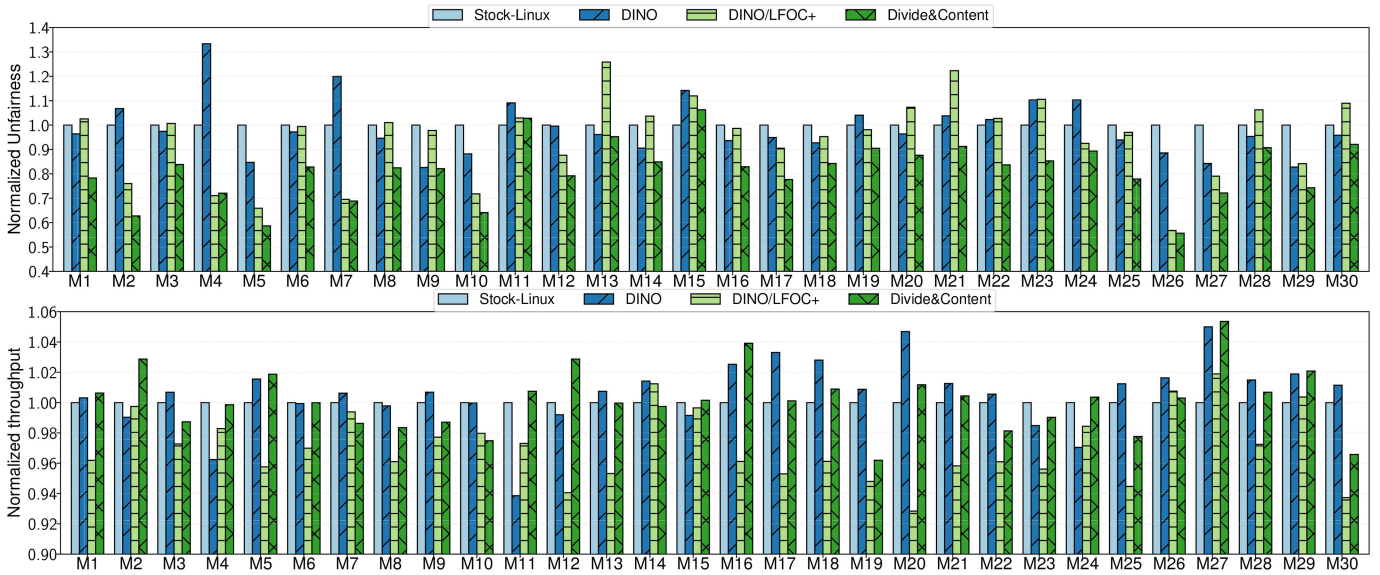


Fig. 10. Normalized unfairness and STP values obtained by the various strategies.

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

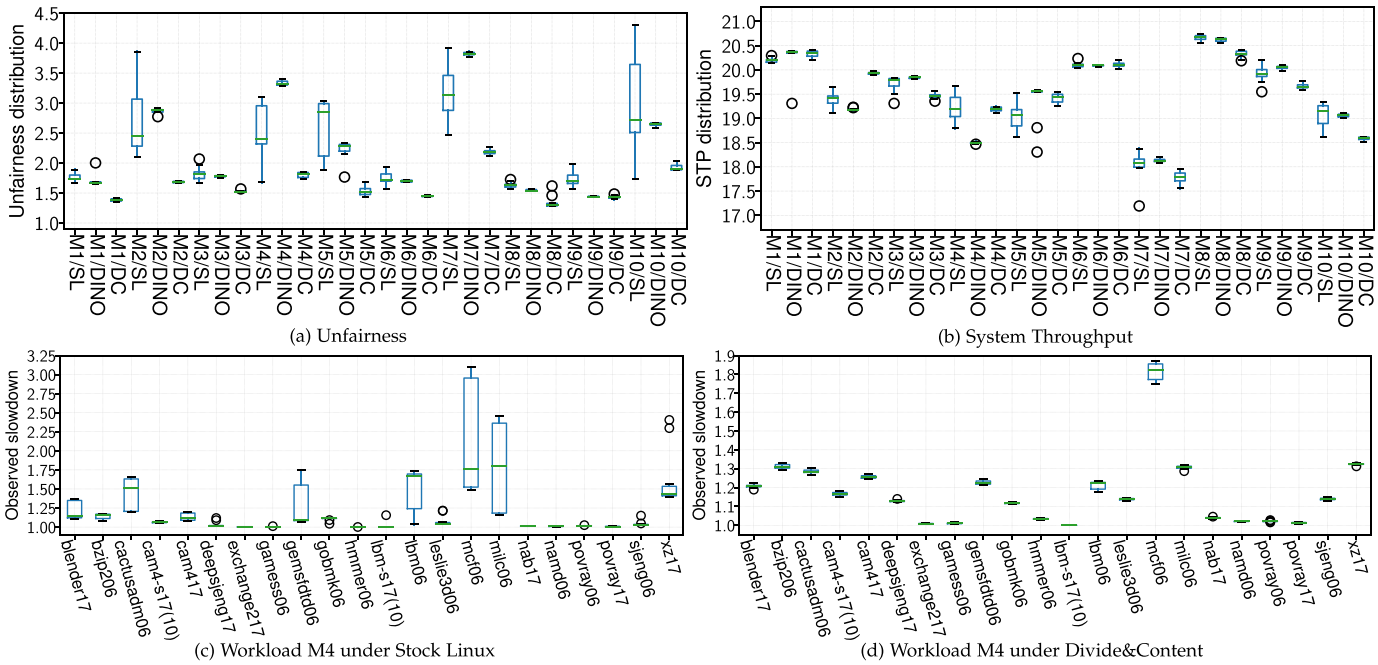


Fig. 11. Variability of the unfairness -Fig.(a)- and the STP metric -Fig.(b)- across multiple runs of workloads M1-M10 with Stock-Linux (SC), DINO and Divide&Content (DC). Figs (c) and (d) show the distribution of the per-application slowdown values observed in different runs of the M4 workload with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