Towards Hybrid Isolation for Shared Multicore Systems

Yoonsung Nam[1], Byeonghun Yoo[1], Yongjun Choi[1], Yongseok Son[2], and Hyeonsang Eom[1]

[1] Department of Computer Science and Engineering, Seoul National University
1 Gwanak-ro, Gwanak-gu, Seoul, Korea
{yoonsung.nam, isac322, drgnjoon, hseom}@snu.ac.kr
[2] Department of Computer Science and Engineering, Chung-Ang University
84 Heukseok-ro, Dongjak-gu, Seoul, Korea
sysganda@cau.ac.kr

Abstract. Co-locating and running multiple applications on a multicore system is inevitable for data centers to achieve high resource efficiency. However, it causes performance degradation due to the contention for shared resources, such as cache and memory bandwidth. Several approaches use software or hardware isolation techniques to mitigate resource contentions. Nevertheless, the existing approaches have not fully exploited differences in isolation techniques by the characteristics of applications to maximize the performance. Software techniques bring more flexibility than hardware ones in terms of performance while sacrificing strictness and responsiveness. In contrast, hardware techniques provide more strict and faster isolations compared to software ones. In this paper, we illustrate the trade-offs between software and hardware isolation techniques and also show the benefit of coordinated enforcement of multiple isolation techniques. Also, we propose *HIS*, a hybrid isolation system that dynamically uses either the software or hardware isolation technique. Our preliminary results show that *HIS* can improve the performance of foreground applications by from $1.7-2.14\times$ compared with static isolations for the selected benchmarks.

1 Introduction

A variety of applications from the simple web server to the complicated machine learning are running in the modern data centers. In the data centers, these applications are typically running on the multicore servers, sharing the computing resources such as CPUs and memory to improve resource efficiency. Sharing resources on a machine is essential to reduce the total cost of ownership (TCO) of the data center; however, it causes contentions for the shared resources leading to performance degradation [13]. The performance degradation may result in user complaints and tremendous revenue loss [14]. To meet the service level objectives (SLOs) of multiple applications while improving resource efficiency in a machine, it is necessary to enforce isolation techniques appropriately to mitigate resource contentions.

There are two types of isolation techniques for multicore systems, that is, software and hardware ones. Software techniques are isolation techniques that allocate resources such as CPU and memory by controlling interactions among threads and resources in a software manner. They are broadly used in various platforms because it is relatively easy to adopt software isolation techniques [8,21]. Moreover, software techniques are flexible in terms of performance, allowing multiple configurations for maximizing performance [6,17]. On the other hand, software techniques are relatively loose isolation than hardware ones since they do not directly segregate or manipulate resources contrary to hardware ones. It makes software isolations less strict and less responsive than hardware ones, which may result in relatively slow isolation enforcement and high-performance variations [23]. Further, compared with hardware isolation, software one may have a larger search space for configurations due to considerable available combinations. For example, hardware cache partitioning provides strict isolation for last-level cache, and per-core dynamic voltage frequency scaling (DVFS) is useful when boosting latency-critical operations [23,9].

Several research works have utilized software and hardware isolation techniques. First, some works use software techniques, such as core allocation, cycle throttling, and thread placement [21,17,6,20]. Software approaches focus on efficient, portable, and flexible isolation. However, their approach is less strict in terms of providing predictable performance, and less responsive in that latency to isolation may be relatively high. Second, a few works utilize hardware techniques, such as hardware cache partitioning and per-core DVFS [23,9]. Hardware approaches are strict and fast because they directly control the hardware feature for performance isolation. Their approach allows stable performance for workloads by segregating resources completely or quick response time for rapid changes in workloads. However, the approach may use a few hardware configurations that may not be enough for achieving maximum performance. Third, some research works use both hardware and software techniques for the isolation of multiple resources [15,4]. Their works are in line with ours in terms of using multiple types of isolation techniques. However, we focus on the tradeoffs in hardware and software techniques, which they have not fully explored.

In this paper, we investigate the characteristics of isolation techniques in terms of strictness, responsiveness, and flexibility. We explore the tradeoffs lying between hardware and software techniques and further evaluate a prototype that combines software and hardware isolation techniques to overcome the shortcomings of each isolation technique. The proposed scheme considers the tradeoffs mainly caused by the isolation mechanism which is either strict and low-latency hardware techniques or loose but flexible software ones to mitigate the contentions dynamically according to the workloads' resource demands and execution patterns. To realize the hybrid isolation scheme, we developed a profiler and a user-level scheduler that uses four isolation techniques. It uses two hardware isolations, which are hardware cache partitioning and per-core DVFS, and two software isolations that allocate cores and perform thread placement. Using these techniques, the scheduler can perform isolation strictly, fast, and flexibly to consolidated workloads. We have evaluated our prototype with the two types of foreground workloads, such as latency-sensitive and batch one, while the batch workload runs in the background. Our preliminary results show that the proposed scheduler can improve the performance of foreground workloads by from 1.7-2.4× compared with static software isolations.

The contributions of our work as follows:

- We have explored the tradeoffs between hardware isolation techniques and software ones in terms of the strictness, responsiveness, and flexibility.
- We have designed and implemented a hybrid isolation system which adaptively isolates workloads considering the characteristics of workloads and tradeoffs in the isolation techniques.
- We have evaluated preliminarily that our system can improve the performance compared with static isolations for the selected benchmarks.

The rest of this paper is organized as follows: Section 2 briefly presents the background for isolation techniques. Section 3 describes the tradeoff between hardware and software techniques, and Section 4 shows problem of ineffective isolations. Section 5 describes the design and implementation of our prototype. Section 6 shows the preliminary evaluation. Section 7 covers the related work. Finally, Section 8 concludes this paper.

2 Background

This section briefly describes the existing software and hardware isolation techniques and illustrates tradeoffs between these isolation techniques.

2.1 Existing Isolation Techniques

Table 1 shows the existing hardware and software isolation techniques. Most schedulers utilizes the software isolation techniques and hardware isolation techniques in the table. All isolation techniques can be categorized by three types; `Throttling`, `Scheduling`, and `Partitioning`.

2.1.1 Software Isolation Techniques Software techniques reduce contentions among workloads by using software interfaces. `Throttling` and `Scheduling` are the representative types of software isolation techniques. `Throttling` is a broadly used to minimize performance interference by controlling the execution rates of contentious workloads among co-located ones. For example, Google CPI^2 throttles CPUs of background workloads to protect the performance of co-located production workloads [21]. Memguard restricts the memory accesses of the memory-intensive workloads based on assigned memory budget throttling CPU cycles [20]. Limiting CPU cycles is an efficient software isolation technique which throttle the execution of specific workloads [20,21,8]. The technique mitigates the contention for shared resources by limiting the number of cycles to

Table 1: Comparison of the existing hardware and software isolation techniques

	Hardware isolation techniques		Software isolation techniques		
	Intel CAT [10]	Per-core DVFS [19]	CPU Cycle Limit [18]	CPU Allocation [7]	Thread Migration [7]
Type	Partitioning	Throttling	Throttling	Scheduling	Scheduling
Latency (ms)	3	2	40~50	3	90
Configurations (Xeon E5-2683v4)	# of ways (20 per LLC)	# of available freq. (10 per core)	quota / period (100)	# of cores (16)	# of sockets (2)
Strictness	High	High	Medium	Medium	Low
Responsiveness	High	High	Medium	High	Low
Flexibility	Low	Low	High	High	High

quota within the configured *periods*. If the assigned cycles are exhausted during a period, the core will remain idle until the new period begins.

Another technique is mitigating contentions via `Scheduling`. Two techniques are mostly used for `Scheduling`. One is *CPU allocation*, and the other is *thread migration*. *CPU allocation* is simple, yet the effective software technique to isolate workloads. It works purely in software manner, and easily reduces the contention of shared resources. It allocates dedicated CPU cores to each workload to minimize resource contention among workloads. When allocating cores to workloads, it is critical to consider which workloads will be colocated with each other [24,6,11,16,17]. Because resource contention among workloads can grow or not depending on which workloads are co-located. When resource contentions can not be resolved by other isolations in a socket, *thread migration* can be helpful by migrating the most suffered workload to the less contentious socket (or machine). This can be helpful where exist severe contentions that `Throttling` can not mitigate. In contrast, in the cases of all the possible schedule pairs can not relax the contention, `Scheduling` may result in poor performance due to the unnecessary overheads as it would fail to find better workload pairs.

2.1.2 Hardware Isolation Techniques

Hardware techniques physically allocate resources to mitigate contentions among workloads or exploits specific hardware features equipped on recent multicore machines. Hardware techniques can provide fast and strict isolation compared with the software ones, because they directly control hardware interfaces. Besides, hardware techniques have lower latency than software ones. Because they have a fewer number of available configurations, which makes configuration search faster when enforcing isolations. There are two types of hardware isolation techniques; `Partitioning` and `Throttling`. `Partitioning` is a representative hardware isolation technique which strictly segregates resources for multiple workloads. For hardware partitioning, there are Intel Cache Allocation Technology (Intel CAT) for LLC way-partitioning [10] and Intel Running Average Power Limits (Intel RAPL) for limiting power consumption [5].

Another hardware isolation technique is `Throttling`-type one using dynamic voltage frequency scaling (DVFS). DVFS is originally designed to perform power management, however, owing to the advance of DVFS, voltage regulators on recent CPUs can adjust a voltage of each core in the CPUs. This enables faster and low-overhead controls for specific operations [9], thus this can enable fine-grained isolation for latency-sensitive workloads [23].

3 Trade-offs between Hardware and Software Techniques

In this section, we describe the trade-offs between hardware and software isolation techniques. Also, we present the effects of isolation techniques by the characteristics of workloads such as resource demands.

To describe the trade-offs, we ran two workloads, each is a multi-threaded process and ran on a single socket while enforcing performance isolation. The test machine has 32GB of RAM, and its CPU is a Xeon E5-2683v4 (2.1GHz, 16-cores). We turned off the hyper-threading feature. For baseline, we used static software isolation (i.e., *Core Allocation*). We used `cgroups cpuset` [7] to allocate 8 cores (16 cores) of one socket equally to each workload and allocate local memory. We chose several benchmarks for foreground workloads that show a diverse range of memory and LLC access pattern; `streamcluster` and `canneal` of PARSEC [2], and `kmeans` and `nn` of Rodinia [3], and Apache benchmark (`ab`). For background workloads, we used `SP` of the NASA parallel benchmark [1] because it shows high LLC and memory bandwidth usage enough to stress memory subsystem.

Strictness. To show the strictness of hardware techniques and software ones, we compared *hardware cache-partitioning* and *software cycle-limiting* by running two workloads concurrently on a socket. We ran `canneal` as a foreground and `SP` as a background by allocating the equal number of dedicated cores. For hardware isolation, we allocated the equal amount of LLC to each workload, and for software isolation, we limited the CPU cycles of a background workload to use only 50% of assigned CPU cycles to restrict LLC accesses to its half.

Figure 1 shows the changes in LLC usage and instructions per cycle (IPC) of foreground and background workloads. As shown in Figure 1a and 1b, when using the hardware isolation technique, the LLC allocations are equally divided all the time due to the direct and strict segregation of hardware isolation. On the other hand, when using the software isolation technique, the LLC allocations are changed dramatically over times, because the software isolation does not guarantee the physical segregation of resources.

The difference between isolation techniques makes the performance of workloads unpredictable. Figure 1c and 1d show the performance variations of the software isolation. Software CPU cycle limiting shows a larger variation compared with the hardware cache partitioning in the case of foreground workload. Even worse, in the case of background workload, those variations are getting much bigger, showing more unpredictable IPCs when software cycle limiting. As a result, we find that the hardware isolation technique provides better predictable performance than the software isolation technique by enforcing strict isolation.

Responsiveness. We also compared responsiveness of the hardware and soft-

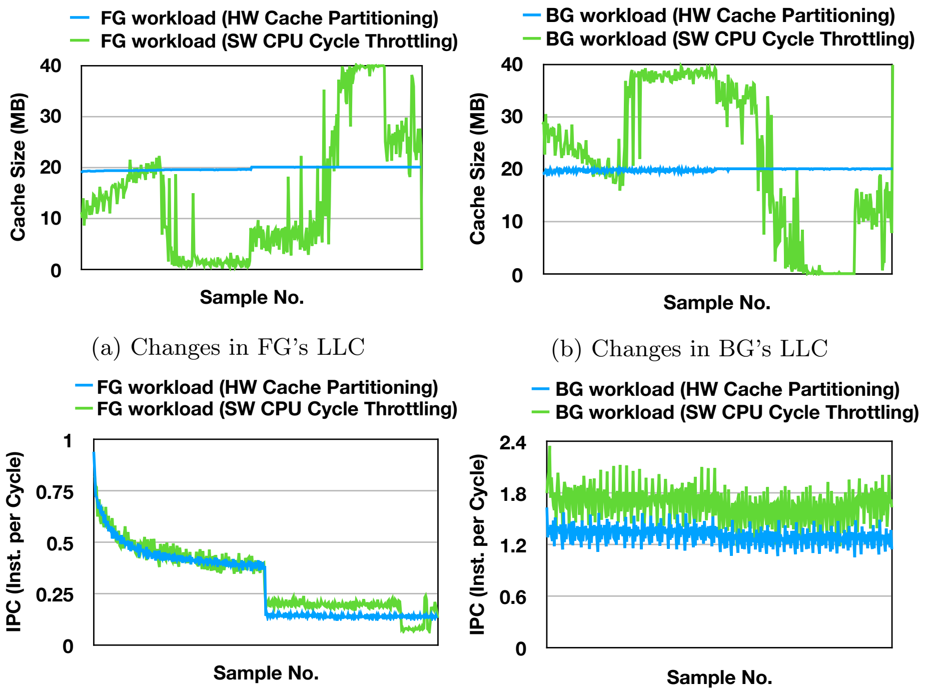


Fig. 1: Comparing the strictness of the hardware and software isolations, necessary for predictable performance, the software one shows high variation of performance. Workloads are `canneal` (foreground) and `SP` (background). *x-axis* represents the number of samples and *y-axis* represents LLC allocation and IPC of workloads.

ware technique to find which technique can provide more fine-grained contention control by performing isolation quickly. We define responsiveness of isolation technique as the latency to its effect. We chose per-core DVFS as a hardware isolation technique and CPU cycle limiting as a software one to demonstrate the difference in terms of the responsiveness. Per-core DVFS can adjust the core frequencies at 0.1GHz granularity. On the other hand, CPU cycle limiting can change the cycle at 1% granularity. Even though the control granularity of software is more fine-grained, the speed of enforcing isolation is faster when enforcing hardware isolation. Enforcing core frequency takes a couple of milliseconds. Meanwhile, enforcement of cycle limiting takes 40-50 milliseconds which is 13-25× longer than DVFS as shown in Table 1.

To illustrate the responsiveness of the hardware and software isolation technique, we ran two workloads in a socket, and each runs on the eight dedicated cores; one is apache web server and the other is `SP` that shows high LLC and

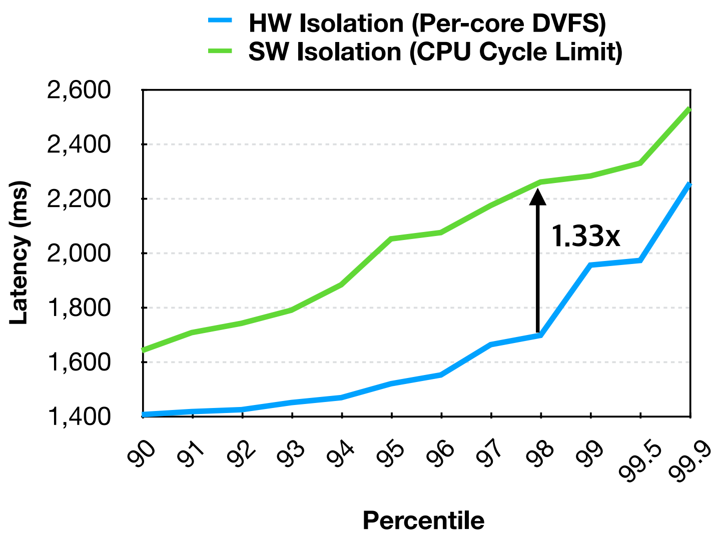


Fig. 2: Comparing the responsiveness of the hardware and software isolations. The graph shows the responsiveness can affect the performance. Workloads are apache web server (`ab`) (foreground) and `SP` (background). *x-axis* represents the percentile of web server request and *y-axis* represents the latency of web server

memory bandwidth demands. We evaluated the responsiveness of isolation techniques by running the apache benchmark (`ab`) which sends the requests to the web server. While running two workloads, we increased the request load of the web server, and also throttled the execution of the background workload. To compare hardware and software techniques, we conducted the experiments twice; first with per-core DVFS, and second with CPU cycle limiting. In both experiments, we throttled the CPU cores of the background workload by increasing the degree of isolation by a step at every 200 milliseconds, and we increased ten steps. For per-core DVFS, we changed the CPU frequency of the background workload from 2.1GHz to 1.2GHz by 0.1GHz. In the same way, for CPU cycle limiting, we also changed the allowed CPU cycle percentage from 100% to 57%, which is the same degree as DVFS. Figure 2 presents how the hardware isolation technique responds more quickly. When performing the software isolation, 98th percentile latency can be 1.33× higher than hardware isolation. This latency difference in tail-latency comes from fast isolation speed thanks to low overhead of hardware technique. The effect of fast isolation may be more important where the resource contention changes frequently or fine-grained control matters. Consequently, we find that the hardware isolation technique is more responsive than software one.

Flexibility. We investigated the flexibility of the hardware and software isolation technique. Flexibility means the ability to choose better scheduling options by mapping threads to resources (e.g., CPU cores and memory nodes) or grouping workloads which minimize the contentions and improve the throughput for the workloads. To describe the effectiveness of flexibility, we grouped four workloads, which shows high LLC-intensity or memory bandwidth intensity, into two groups. And, two workloads are paired in each group, and scheduled each group to the separate sockets. We performed different isolations to the same four workloads; the first with the hardware cache partitioning and the second with scheduling by regrouping the background workloads. Figure 3 shows scheduling is more effective than hardware cache partitioning, so that the performance of `canneal` and `SP` improves by up to 1.6× and 1.3× than the hardware one. Some workloads show performance degradations, but their performance loss is reasonable considering the other workloads' performance benefit. The results indicate that software isolation can be useful when the resource contention can not be reduced by the hardware isolation, which have a few isolation options. In this experiment, hardware cache partitioning can only solve resource contention in a socket. However, software isolations such as migration enable more options for enhancing performance and improving resource efficiency.

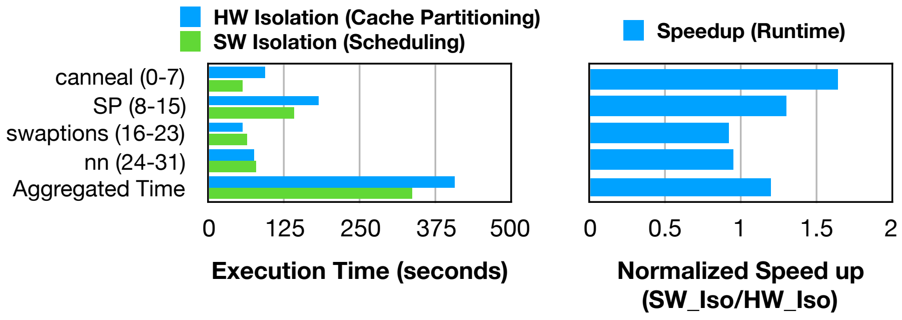
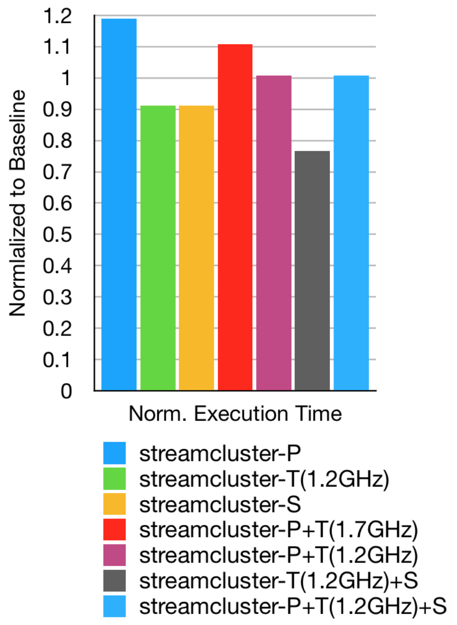


Fig. 3: Benefits of the flexible software isolation. `canneal` and `SP` show high LLC and memory contentions. However, `swaptions` and `nn` are relatively not. In case of performing the hardware isolation, the contention is still high. On the other hand, the software isolation can effectively mitigate the contention significantly. *x-axis* shows the runtime and speedup of workloads and *y-axis* shows their name and CPU affinities. The ranges in parenthesis indicate the range of CPU IDs where workloads runs.

4 Ineffective Isolations

In addition to the trade-offs between isolation techniques, the isolation effects depend on the characteristics of workloads such as resource demands. The same



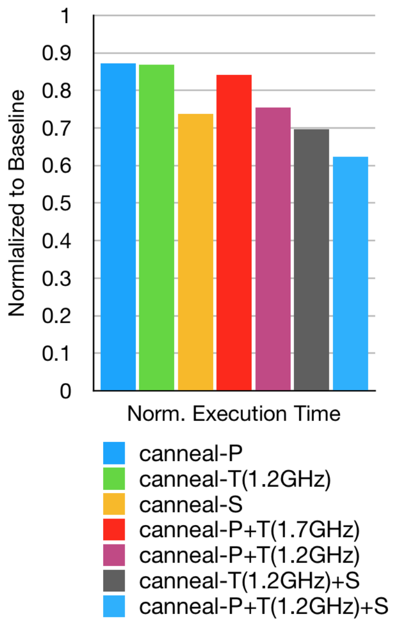


Fig. 4: Enforcing multiple isolations to `streamcluster` and `canneal`, each colocated with `SP`. The execution time is normalized to the performance of a workload running on the dedicated cores on the default system (P: `Partitioning`, T: `Throttling`, and S: `Scheduling`).

isolation technique can deliver different impacts according to the workloads. We present a simple example of multiple isolations are performed on the different foreground under the high LLC and memory contention.

To demonstrate the effectiveness of each isolation, we tested all isolation techniques which are `Partitioning`, `Throttling`, and `Scheduling`. We manually divided LLCs evenly to workloads using Intel CAT for `Partitioning`. We also changed the execution rate of background workload by setting the frequency of core as the highest frequency(2.1 GHz), the middle frequency(1.7 GHz), and the lowest frequency(1.2 GHz) for `Throttling`. Finally, we changed the number of cores of the background workload to the half of allocated cores, which is four cores, to describe the effect of mitigating memory contention via `Scheduling`. The baseline is the case of when two workloads are running on its dedicated cores without performing any isolation.

As shown in Figure 4, the performances of foregrounds vary according to the different isolation techniques. This is because the resource demands of the foregrounds are different from each other, and also isolation effects are different depending on the isolation techniques as well. In the case of `streamcluster`, partitioning LLC increased the execution time by 20% compared with the baseline, but `Throttling` or `Scheduling` reduced the execution time by 10% compared to the baseline. This is because the `streamcluster` is a memory bandwidth intensive workload, so restricting LLC makes its performance worsen. However, `Throttling` or `Scheduling` could increase the memory bandwidth of `streamcluster` by reducing background's memory access. In the case of `canneal`, it uses less memory bandwidth than `streamcluster`, but it is an LLC intensive workload with high LLC hit ratio. For `canneal`, all three isolations can reduce the execution time significantly.

However, in the case of `streamcluster`, when both techniques (`Throttling` and `Scheduling`) are used, the execution time is reduced by 24% compared with the baseline configuration. On the other hand, in case of `canneal`, the execution time is reduced by up to 38%, which is the highest performance improvement. In this way, we find that effective isolation techniques can be different according to the characteristics of the workload. Moreover, we realize that it is necessary and important to enforce appropriate isolation techniques adaptively considering the changed contentions.

5 HIS: Hybrid Isolation System

This section briefly describes the overview of how our proposed system can deal with the trade-offs between multiple isolation techniques depending on the characteristics of workloads. To achieve this goal, we propose *HIS*, a hybrid isolation system that leverages hardware and software isolation techniques to mitigate contentions and improve the performance of workloads.

Figure 5 illustrates our *HIS* architecture. As described in the figure, our system consists of a profiler, isolation techniques, and a scheduler. We divide workloads as foreground workloads and background workloads. The foreground is a latency-critical or high-priority batch workload and the background is the best-effort workload. *HIS* groups these two types of workloads and performs isolations on *workload group*s and places a group on a socket to improve resource efficiency. We also assume there is one foreground workload in the workload group like other clouds do [15].

The profiler collects the performance counters from workloads, and then profiles resource contentions from the collected counters. To profile resource contention online, the profiler performs *solo-run mode*, which enables for a foreground workload to run alone, to obtain the performance counters of each workload when no contention exists. After profiling *solo-run* data, the profiler collects performance counters of consolidated workloads to estimate how contentions affect resource usages. We define the performance counters of workloads when workloads co-executes as *co-run* data. Note that, we currently consider the *solo-run* data for the foreground workloads. We will describe more detail of *solo-run mode* at the Section 5.1.Both the obtained *solo-run* data and the *co-run* data are used to calculate the resource contention. After that, it sends the information of resource contention to the scheduler. Then, the scheduler checks which

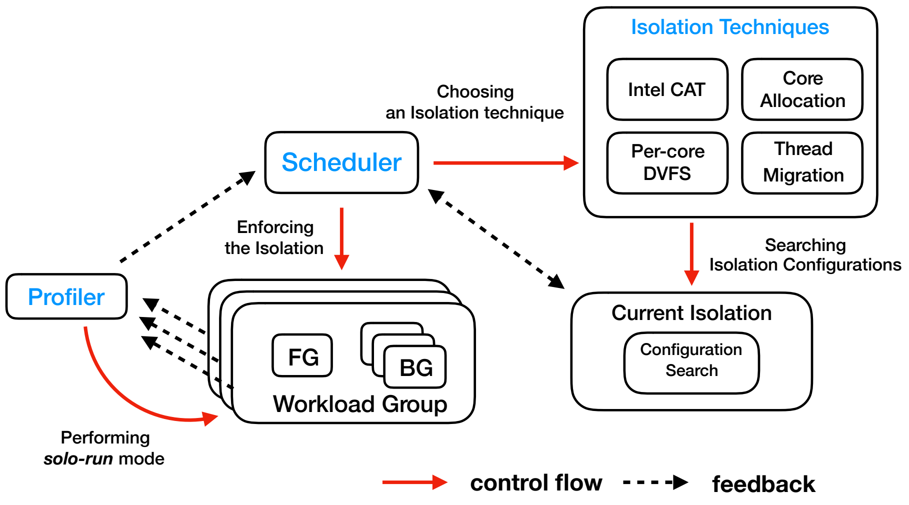


Fig. 5: *HIS* architecture. It consists of a *profiler*, *isolation techniques*, and a *scheduler*. The redline shows control flow and black dotted line shows the feedback of workload profiles, performances, and isolation decisions. The scheduler uses four isolation techniques; two hardware isolations (i.e., Intel CAT and per-core DVFS) and two software isolations (i.e., core allocation and thread migration).

resource contention is the most contentious and decides an isolation technique considering the types of isolation technique and resource contention. Once an isolation technique is selected, the scheduler searches for a configuration of isolation and enforces isolations until the contention is minimized to below the pre-tuned threshold (i.e., 5% for each contention). In other words, the scheduler adjusts isolations to reduce the resource contention for a foreground workload close to when the workload runs alone. The scheduler repeats this procedure until the foreground workload finishes.

For isolations, *HIS* checks which isolation technique is the most appropriate one among multiple isolation ones; *HIS* considers multiple hardware and software isolations, and applies isolation techniques incrementally to improve the performance of a foreground workload while maximizing that of background one. This approach is useful because the scheduler reflects the subsequent resource contention and can enforce the corresponding isolation technique. For enforcing a proper isolation, the scheduler should know the dominant contention and decide appropriate isolations. Following sections will describe how the profiler profiles contention and how scheduler chooses isolation configurations in detail.

5.1 Profiling Contention

Profiling contention is essential for performance isolation. Our scheduler receives the resource usages of workloads from the profiler to estimate the contention on the system. To profile the contention, the profiler measures the per-workload performance counters such as LLC misses and LLC references in every profile interval (i.e., 200ms). It calculates the resource contention by the difference of resource usages between when all workloads run concurrently (*co-run*) and when a workload runs alone (*solo-run*). We used the differences of *co-run* and *solo-run* data, because it presents resource sensitivity of a workload that how much the performance of workloads is degraded by the contention compared with no contention exists [11,23,22].

The profiler maintains the *solo-run* data for each foreground workload to calculate the resource contention, thus the scheduler checks whether the *solo-run* data exists or the execution phase has changed at every scheduling interval using already sampled data. If there is no data to calculate resource contention or the profile sample data is outdated, then the scheduler dictates to collect the new samples for *solo-run* data by stopping other background workloads. We call this procedure *solo-run mode*. We used two signals to enable *solo-run mode*; `SIGSTOP` for stop running workloads and `SIGCONT` for resume stopped workloads. To enable the *solo-run mode*, the profiler stops all current isolations and also pauses other background workloads during the successive profile intervals (e.g., one or two seconds). During the *solo-run mode*, only a foreground workload runs alone, and after finishing, the profiler stores all collected performance counters during the mode and resumes all previously paused isolations and background workloads.

The profiler classifies the workloads by their mostly used resources and also classify them by the type of the workload provided by users (e.g., FG and BG). We focused on the LLC and memory bandwidth to mitigate the contention on the memory subsystem. To measure the LLC contention, we used the `LLC misses` and `LLC references`, obtained by performance counters, and calculate the LLC hit ratio reflecting how much workload reuses the LLC. In addition, `local_mem_bytes`, obtained by Intel *Resctrl*, is used to estimate the memory bandwidth contention. The metrics can be added to consider more contentions and complicated execution patterns. With these metrics, the profiler can determine the dominant resource by comparing them, and also classify a workload as one of which CPU-intensive, LLC-intensive, or memory bandwidth intensive at every scheduling interval.

5.2 Hybrid Isolation

In this section, we will detail the trade-offs of isolation techniques and describe how our scheduler leverages them to mitigate the contention.

5.2.1 Isolation Mechanisms *HIS* considers four isolations to mitigate contentions. In Table 1 of Section 2.1.1, HIS uses four techniques, which are hardware cache partitioning, per-core DVFS, core allocation, and thread migration, in hybrid isolation system.

Hardware Isolations. For hardware isolations, we used the Intel Cache Allocation Technology (Intel CAT) and per-core dynamic voltage frequency scaling

(DVFS). With Intel CAT, *HIS* can allocate a LLC by the unit of a way. In our machine (i.e., Xeon E5-2683v4, 16-cores per socket), a socket has 40MB of LLCs and each consists of 20 ways. Intel CAT provides strict isolation for the LLC in a socket, because it partitions LLCs physically by masking the ways in *Resctrl*. We also used the per-core DVFS to throttle the execution of workload. Per-core DVFS is used to improve power efficiency of processors as well as mitigate the contentions and enable fine-grained control to improve performance of workloads. Using per-core DVFS, the scheduler can rapidly mitigate the memory contention, generated from contentious background workloads by adjusting the frequencies of cores running backgrounds. For enforcing core frequencies, we used the `CPUFreq Governor` of Linux.

The hardware isolations perform strict and quick isolation compared with the software isolations. Hardware cache partitioning provides the strict isolation which affects more predictable performance for the workloads. They generally take few milliseconds to reflect their effects to the workloads' performance. As shown in Table 1 (in Sec. 2.1.1), we observed 2−3ms of latencies, and this low latency is beneficial to meet the SLOs of the latency-sensitive workload when the execution patterns of workloads changed frequently or the load of latency-sensitive workload shows high variation.

Software Isolations. For software isolations, we used the `cgroups cpuset` to allocate CPUs and memory nodes to workloads. To mitigate the contentions on the multicore systems, two software isolations are used in scheduling; core allocation and thread migration. Core allocation performs the allocation of CPU cores for workloads to isolate core resources by their CPU demands. For example, latency-sensitive workloads such as the web server can show high load variation by the user patterns, so the CPU demands can vary by their loads. Therefore, core allocation should be performed according to the CPU demands to improve resource efficiency and meet the SLO of foreground workloads.

Unlike core allocation which manages the contention of a workload group, thread migration detects the performance imbalance between workload groups, then it regroups those workloads by migrating workloads to the other socket. The thread migration is effective when the contention on a workload group is too large to be mitigated by other isolations (e.g., hardware isolations) on the single socket. However, too frequent thread migrations may be harmful to the performance because the cost of the memory migration over the sockets is expensive [12]. Therefore, we designed that thread migration is triggered only (1) if the performance benefit is estimated to exceed the threshold or (2) if the phase changes in a workload group is detected.

The software isolations provide flexibility compared with the hardware isolations. Core allocation treats CPU demands as well as mitigates memory contention according to the type of contention of workloads. They typically take more times than the hardware isolations to reflect isolation impacts on the workloads' performance (e.g., tens to hundreds of milliseconds).

5.2.2 Hybrid Scheduler The hybrid scheduler periodically (1) chooses a proper isolation technique and (2) searches isolation configurations to improve the performance of foreground workloads within the workload groups. Before the hybrid scheduler initiates isolations, the profiler sends the information about current active workload groups, such as pid, workload type (FG or BG), and profiled resource contention to the scheduler. By using the workload group information, the scheduler initiates the isolations for the workload groups in parallel. While performing isolations, the scheduler checks whether the workloads in the group need *solo-run* data to calculate contentions, and if yes, requests for the profiler to perform *solo-run mode* to collect the new *solo-run* data.

Choosing an isolation technique. The hybrid scheduler chooses an isolation technique based on the mostly contentious resource, identified by the profiler. For the resource contention, at first, the scheduler checks whether the hardware isolation is available for the resource or not, and chooses the isolation if the isolation is possible and has not been tried. Between software and hardware isolations, the scheduler prioritizes hardware isolations for the strict and fast isolation. If all hardware isolation has tried before, the scheduler checks whether the software isolation technique are available for the resource or not and if it is possible then chooses the software isolation. If all the hardware and software isolations are used, the scheduler reconsider all techniques to reuse them. We implemented our policy to consider hardware isolation as much as possible. However, the policy for choosing an isolation technique can be changed to meet SLOs of the workloads.

There are two cases that the software isolations are chosen rather than hardware one. The first case is wrong invocation for an isolation technique. The scheduler often fails to search a better configuration due to the a few errors of profile data. For instance, the profiler may identify CPU contention as major factor when the actual contention is LLC contention. In this situation, the scheduler may perform hardware cache partitioning by its profile results. To minimize this case, we may choose isolation techniques more conservatively by not changing techniques until successive contentions are detected. The second case is when the scheduler exploits all hardware techniques, but still fails to reduce the contention because of their lower number of available configurations. For example, while the per-core DVFS may be not enough for mitigating severe memory contention due to its small configuration ranges, restricting the number of cores may be more beneficial to mitigate memory contention.

Enforcing isolation. After choosing the isolation, the scheduler searches isolation configurations that minimize the resource contention by enforcing the various configurations repeatedly and incrementally. Whenever before enforcing isolation, the scheduler decides whether it allocates more resources to the foreground workload or not, based on resource contention. For example, if the dominant resource contention for the foreground workload is LLC contention, and also if the LLC hit ratio of the foreground one during *co-run* is lower than that of *solo-run*, the scheduler allocates more LLC ways to foreground workload.

Because lower LLC hit ratio than the *solo-run* typically means that foreground workload can be improved if the workload is assigned more LLC ways.

Once the isolation is performed, the scheduler waits until the effect of enforcing an isolation is reflected, and then it repeatedly checks the degree of the contention. We empirically find that 200ms is the most effective time to feedback contentions, yet the wait time can be tuned depending the target workloads. The scheduler finds there is no severe contention, or it can not perform the isolations further (e.g., searching all possible configurations), then the configuration search ends. Finally, the scheduler enforces the configuration for chosen isolation.

6 Preliminary Evaluation

This section describes the preliminary experimental setup and results. We evaluated the hybrid isolation system for the batch and latency-sensitive workloads compared with the default Linux system using static software isolations. Here, we define the baseline as the case of *co-run* where the foreground and the background runs together on a socket. Both workloads share memory subsystem such as an LLC and a memory controller, but have their own dedicated CPU cores.

6.1 Experimental Setup

We evaluated the *HIS* on a dual 16-core Intel Xeon E5-2683 v4 server. The LLC size of the server processor is 40MB and can be allocated to the workload in 2MB units (per a way) using Intel CAT. The nominal frequency is 2.1GHz and the configurable core frequencies are 10 steps from 1.2GHz to 2.1GHz. We turned off Turbo-boost and Hyper-threading. Our test machine is equipped 32GB of RAM with each socket. The maximum bandwidth of the socket is measured to 68 GB/s by Intel *VTune* and we used Linux kernel 4.19.0.

We used various benchmark applications from four different suites. For batch foregrounds, we used PARSEC (`bodytrack`, `canneal`, `streamcluster`, `dedup`, `facesim`, `ferret`, `fluidanimate`, `swaptions`, and `vips`) and Rodinia (`cfd`, `nn`, `kmeans`, and `bfs`). For latency-sensitive foregrounds, we used the apache web server and `ab` (apache benchmark). In the case of latency-sensitive foreground, the scheduler should respond quickly to deal with the load spikes of the web server. We chose the `SP` from NPB as the background, because SP shows high memory bandwidth and LLC usage than other benchmarks.

6.2 Preliminary Results

6.2.1 Batch Workloads We show the performance results for the batch workloads running as the foreground in Figure 6. In the figure, *HIS* isolates foreground workload effectively, so that the performance of batch workloads are improved significantly compared to the *co-run*. In case of *canneal*, the performance is improved more than 1.7× than *co-run* with simple core isolation that the workloads run on their dedicated cores, and the scheduler improves the performance of benchmarks on average 1.22× than *co-run*. On the other hand, the performance of the background workload is degraded, because our scheduler restricts the resource usage of the background workload to improve the performance and the responsiveness of foreground.

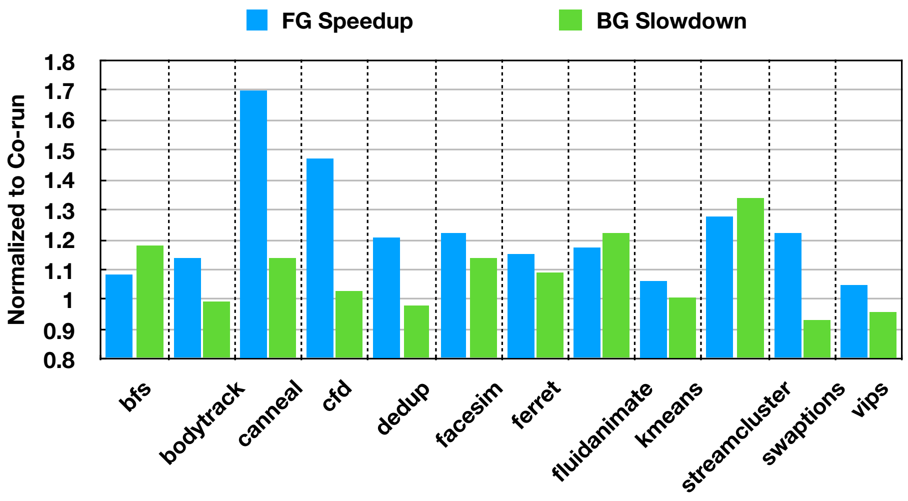


Fig. 6: Performance improvement of the batch foreground workload with *HIS* compared to the *co-run*. Each workload is initially allocated eight dedicated cores (background workload: `SP`).

6.2.2 Latency-sensitive Workloads Figure 7 presents the performance of latency-sensitive workload running as the foreground. In order to evaluate the performance of latency-sensitive workload, we modified the `ab` which uses the *Pareto* distribution to reproduce situations where a few users are connected during most of the time and the connections are bursty. We measured the percentile latencies of requests.

In the figure, *HIS* can reduce the tail-latencies of web server below the performance of *solo-run* (8 cores) until 99.9th percentile, because the scheduler considers changes in dynamic load of the web server as well as the dominant resource contentions, and enforces various isolation techniques according to them. We also plot the tail-latencies of *solo-run* (12 cores) to compare with the proactive approach that reserves CPU cores as much as the maximum CPU cores that *HIS* allocates under the experiment. The latencies of *HIS* are higher than *solo-run* (12-cores), because *HIS* begins by allocating fewer cores to workload and increase the number of cores assigned to the workloads.

Compared with the *co-run*, *HIS* achieves the performance up to 2.14× speedup (for 99.9th percentile latency), while the performance of background workloads is slow down by 1.47×. We observed that the main reason for the performance improvement of foreground is due to fast and strict hardware isolation, core isolation which allocates more cores depending on the CPU demands, and adaptive isolations.

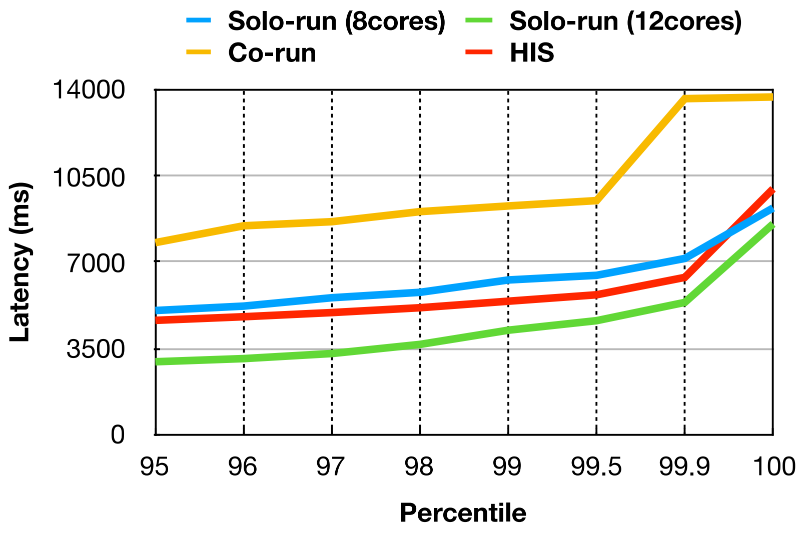


Fig. 7: Performance improvement of the latency-sensitive foreground workload (Apache web server) with *HIS* compared to the *co-run*. Each workload is initially allocated eight dedicated cores (background workload: `SP`).

7 Related Work

There have been many studies on isolation approaches used in multicore systems. Software isolation is widely used in most multicore systems. CPI^2 [21] detects the performance anomaly and identifies the suffered a *victim* workload using statistics of CPI(Cycles Per Instruction), and throttles the CPU usage of the *antagonist* for performance isolation. Their work is inline with ours in terms of throttling background workloads with software isolations. However it only uses software techniques which provide less strict isolation, thereby needs harsh CPU hard-capping for antagoist for strictness (i.e., 0.01 CPU-sec/sec).

Memguard [20] isolates the memory bandwidth contention based on its memory budget. It utilizes a software isolation that throttles memory access of each workload by restricting CPU cycles, thus each workload's memory bandwidth can not exceed the assigned memory bandwidth. Similar to our work, it isolates memory resources by reserving memory bandwidth, but it does not utilize hardware isolation technique, so there is no guarantee for strict isolation. However, our work uses hardware isolation techniques to supplement strictness. Both CPI2 [21] and Memguard [20] isolate workloads by throttling CPU using a software technique, and they can mitigate memory contention easily. However software techniques may result in unintended interferences under the co-location of workloads showing bursty behaviors.

Dirigent [23] is a fine-grained isolation runtime system which partition an LLC and throttle CPUs. Similar to ours, it exploits hardware isolation techniques such as hardware cache partitioning and per-core DVFS to meet the SLOs of a latency-sensitive workload while backfilling batch workloads to improve resource efficiency. Our work is in line with their work [23] in terms of providing fine-grained isolations for considering the characteristics of workloads. However, we focus on the adaptive enforcement of multiple isolation techniques according to the characteristics of workloads, thus we can take more options for better performance isolation.

Quasar [6] utilizes a machine learning algorithm to infer which colocation mostly mitigates the shared resource contention, and uses scheduling and thread migration, which is the software approach, for isolation of consolidated workloads. Their work is inline with ours in terms of multiple isolation techniques In contrast, they only uses software isolation techniques for higher flexibility which can not provide strict and fast isolation.

Heracles [15] and PARTIES [4] isolate workloads by partitioning and throttling resources using both hardware and software isolation schemes to meet SLOs of production workloads while increasing resource efficiency. Similar to ours, their works are inline with ours in terms of using multiple isolation techniques for multicore systems. However, their works do not consider the tradeoffs between isolation techniques which can be harmful for the strictness and flexibility.

8 Conclusion

We developed a hybrid isolation system that utilizes hardware and software isolation techniques in a hybrid manner by the characteristics of the workloads. We have explored the tradeoffs between hardware and software isolation techniques, and illustrated how these properties affect performance of consolidated workloads. We have proposed an algorithm for isolation to use isolation techniques mutually complementary through characteristics analysis of workloads and comparison of each isolation technique. Our experimental results show that our approach can improve the performance of foreground workloads in terms of execution time than the static software isolation by from $1.7\times - 2.14\times$ while improving resource efficiency for the selected benchmarks. For future work, we will evaluate our prototype with more diverse workload combinations. Also, we will investigate isolation techniques for the different micro-architectures such as AMD and ARM to generalize our ideas.

Acknowledgment

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2017R1A2B4004513, 2016M3C4A7952587, 2018R1C1B5085640), the Institute for the Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP) (No. R0190-16-2012), and BK21 Plus for Pioneers in Innovative Computing (Dept. of Computer Science and Engineering, SNU) funded by National Research Foundation of Korea(NRF) (21A20151113068).

References

1. David H Bailey, Eric Barszcz, John T Barton, David S Browning, Robert L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The nas parallel benchmarks. *The International Journal of Supercomputing Applications*, 5(3):63–73, 1991.
2. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.
3. Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.
4. Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.
5. Howard David, Eugene Gorbatov, Ulf R Hanebutte, Rahul Khanna, and Christian Le. Rapl: memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, pages 189–194. ACM, 2010.
6. Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.
7. Simon Derr. *Control Group Cpusets*. BULL SA, 2004. `https://www.kernel.org/doc/Documentation/cgroup-v1/cpusets.txt`.
8. Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. Perfiso: Performance isolation for commercial latency-sensitive services.
9. Chang-Hong Hsu, Yunqi Zhang, Michael A Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, and Ronald G Dreslinski. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 271–282. IEEE, 2015.
10. CAT Intel. Improving real-time performance by utilizing cache allocation technology. *Intel Corporation, April*, 2015.
11. Shin-gyu Kim, Hyeonsang Eom, and Heon Y Yeom. Virtual machine consolidation based on interference modeling. *the journal of Supercomputing*, 66(3):1489–1506, 2013.

12. Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. Thread and memory placement on numa systems: Asymmetry matters. In *USENIX Annual Technical Conference*, pages 277–289, 2015.
13. Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, page 4. ACM, 2014.
14. Greg Linden. Make data useful, 2006.
15. David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 450–462. ACM, 2015.
16. Dongyou Seo, Hyeonsang Eom, and Heon Y Yeom. Mlb: A memory-aware load balancing method for mitigating memory contention. In *Conference on Timely Results in Operating Systems (TRIOS14)*, 2014.
17. Boris Teabe, Alain Tchana, and Daniel Hagimont. Application-specific quantum for multi-core platform scheduler. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 3. ACM, 2016.
18. Paul Turner, Bharata B Rao, and Nikhil Rao. Cpu bandwidth control for cfs. In *Linux Symposium*, volume 10, pages 245–254. Citeseer, 2010.
19. Rafael J. Wysocki. *CPU Performance Scaling*. Intel Corporation, 2017. `https://www.kernel.org/doc/html/v4.12/_sources/admin-guide/pm/cpufreq.rst.txt`.
20. Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th*, pages 55–64. IEEE, 2013.
21. Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi 2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 379–391. ACM, 2013.
22. Yong Zhao, Jia Rao, and Qing Yi. Characterizing and optimizing the performance of multithreaded programs under interference. In *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*, pages 287–297. IEEE, 2016.
23. Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. *ACM SIGARCH Computer Architecture News*, 44(2):33–47, 2016.
24. Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *ACM Sigplan Notices*, volume 45, pages 129–142. ACM, 2010.