CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers

Jinsu Park, Seongbeom Park, and Woongki Baek*
School of ECE, UNIST, Republic of Korea
{jinsupark,amita90,wbaek}@unist.ac.kr

Abstract

Workload consolidation is a widely-used technique to maximize server resource utilization in cloud and datacenter computing. Recent commodity CPUs support last-level cache (LLC) and memory bandwidth partitioning functionalities that can be used to ensure the fairness of the consolidated workloads. While prior work has proposed a variety of resource partitioning techniques, it still remains unexplored to characterize the impact of LLC and memory bandwidth partitioning on the fairness of the consolidated workloads and investigate system software support to dynamically control LLC and memory bandwidth partitioning in a coordinated manner.

To bridge this gap, we present an in-depth performance and fairness characterization of LLC and memory bandwidth partitioning. Guided by the characterization results, we propose CoPart, coordinated partitioning of LLC and memory bandwidth for fairness-aware workload consolidation on commodity servers. CoPart dynamically analyzes the characteristics of the consolidated applications and allocates the LLC and memory bandwidth across the applications in a coordinated manner to improve the overall fairness. Our quantitative evaluation shows that CoPart significantly improves the fairness of the consolidated applications (e.g., 57.3% higher fairness on average than the resource allocation policy that equally allocates the resources to the consolidated applications), robustly provides high fairness across various application and system configurations, and incurs small performance overhead.

*Corresponding author

EuroSys '19, March 25–28, 2019, Dresden, Germany

ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303963

CCS Concepts • **Software and its engineering** → *Operating systems.*

Keywords Coordinated partitioning, last-level cache, memory bandwidth, fairness, workload consolidation

ACM Reference Format:
Jinsu Park, Seongbeom Park, and Woongki Baek. 2019. CoPart: Coordinated Partitioning of Last-Level Cache and Memory Bandwidth for Fairness-Aware Workload Consolidation on Commodity Servers. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3302424.3303963

1 Introduction

Workload consolidation is a practical and widely-used technique to improve the resource efficiency of cloud computing systems and datacenters. With workload consolidation, multiple workloads are consolidated on the same physical servers. The main design objective of the resource manager for workload consolidation is to maximize the server resource utilization while providing the fairness of the consolidated workloads.

Despite its advantages, workload consolidation poses critical challenges. One of its major challenges is the performance interference among the consolidated workloads, which is mainly caused by the contention over shared hardware resources on the underlying server system. Prior work has shown that the last-level cache (LLC) and memory bandwidth are highly performance-critical shared hardware resources for consolidated workloads [24, 48].

Recent commodity CPUs have begun to provide hardware support for LLC [17, 42] and memory bandwidth [2] partitioning. They provide basic hardware mechanisms for resource partitioning such as setting the amount of resources allocated to each core or process. It is then the responsibility of the system software to implement a robust policy to achieve the overall fairness by effectively employing the basic resource partitioning mechanisms provided by the hardware.

Prior work has extensively explored architectural and system software techniques for LLC [7, 9, 17, 21, 34, 36, 37, 42, 44, 45] and memory bandwidth [16, 18, 22, 23, 32] partitioning. While insightful, the prior work investigates resource partitioning techniques for either the LLC or memory bandwidth,

lacking the capability to simultaneously partition the two performance-critical hardware resources in a coordinated manner. As quantified in this work, the fairness of workload consolidation can be significantly degraded if LLC or memory bandwidth partitioning is solely used in an isolated manner.

To overcome this limitation, we first present an in-depth performance and fairness characterization of LLC and memory bandwidth partitioning on a widely-used commodity server system. Guided by the findings from the characterization studies, we propose CoPart, coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers.

Specifically, this paper makes the following contributions:

- We present an in-depth characterization of the impact of LLC and memory bandwidth partitioning on the performance and fairness of the consolidated applications. Our characterization results show that each application requires different amounts of LLC and memory bandwidth and exhibits widely-different performance sensitivity to the allocated resources. Further, the sole use of either LLC or memory bandwidth partitioning often achieves suboptimal fairness results.
- Guided by the characterization results, we propose CoPart, coordinated partitioning of LLC and memory bandwidth for fairness-aware workload consolidation on commodity servers. CoPart dynamically analyzes the characteristics of the consolidated applications and partitions the LLC and memory bandwidth in a coordinated manner to maximize the overall fairness of the applications. In particular, we formulate our resource allocation problem as the Hospitals/Residents (HR) problem [12], which is one of the most extensively-studied and widely-applied problems in economics.
- We design and implement a prototype of CoPart. We implement the prototype of CoPart as a user-level runtime system on Linux, requiring no modifications to the underlying operating system.
- We quantify the effectiveness of CoPart using full software and hardware stacks on a 16-core commodity server system. Our quantitative evaluation shows the effectiveness of CoPart in that it significantly improves the fairness of the consolidated applications (e.g., 57.3% higher fairness than a resource allocation policy that equally allocates the resources to the consolidated applications), consistently achieves high fairness across various application and system configurations, and incurs small performance overhead.

2 Background and Terminology

2.1 Container-Based Virtualization

Container-based operating-system virtualization is widely employed in cloud computing systems and datacenters [4]. One of its main advantages over hardware virtualization based on a virtual machine manager is that it is lightweight and efficient. This is mainly because it requires none of the complex system software stacks associated with hardware virtualization such as virtual CPUs and second-level address translation [4].

Container-based OS virtualization provides two major functionalities to containers. First, it provides namespace virtualization, with which each container on the same host OS can have its own root file system. Namespace virtualization is effective for program development and deployment as each container can maintain its own well-tested tools, libraries, and directory structures without modifying the root file system on the host OS.

Second, container-based OS virtualization provides a basic resource partitioning mechanism for the containers on the same host OS through the well-established Linux control group (i.e., Cgroups) functionality. With this functionality, each container can be allocated its own dedicated hardware resources such as cores and memory nodes.

2.2 LLC and Memory Bandwidth Partitioning

Some recent commodity CPUs support LLC [17, 42] and memory bandwidth [2] partitioning functionalities to enhance the fairness of the consolidated applications. While hardware support for resource partitioning is provided by various architectures (e.g., the ARM architecture [42]), we discuss LLC and memory bandwidth partitioning techniques in the context of the `x86-64` architecture. This is done so because it is the only commercial architecture (to the best of our knowledge) that provides architectural support for both LLC and memory bandwidth partitioning.

The `x86-64` architecture partitions hardware resources across the classes of services (CLOSes) [2]. Each CLOS consists of a group of cores or processes that shares the hardware resources allocated to the CLOS.

The `x86-64` architecture provides Intel Cache Allocation Technology (CAT), which is the hardware support for LLC partitioning based on the way partitioning technique [42]. Each CLOS can be dynamically allocated one or more LLC ways. Each LLC way can be allocated to one or more CLOSes.

The `x86-64` architecture provides Intel Memory Bandwidth Allocation (MBA) technology, which is a hardware technique for memory bandwidth partitioning [2]. MBA is a per-core mechanism that controls the traffic between the L2 cache and the LLC, which is generated by the core associated with the corresponding CLOS. The MBA level allocated to a CLOS can be dynamically adjusted.

2.3 Terminology

Without a loss of generality, we assume that N_A applications are consolidated on the same physical server and that the CPU equipped in the server provides the LLC with L ways, the maximum memory bandwidth of B, and the maximum MBA level of 100%. The *resource allocation state* (i.e., s_i) of application i ($i \in [0, N_A - 1]$) is defined as (l_i, m_i), where l_i

Table 1. System configuration

Component	Description
Processor	Intel Xeon Gold 6130 Processor CPU @ 2.1GHz, 16 cores
L1 I-cache	Private, 32KB, 8 ways
L1 D-cache	Private, 32KB, 8 ways
L2 cache	Private, 1MB, 16 ways
L3 cache	Shared, 22MB, 11 ways
Memory	32GB (2 × 16GB DDR4)
OS	Ubuntu 16.04, Linux Kernel 4.13.0

and m_i denote the LLC ways and the MBA level allocated to the application. In addition, the *system state* (i.e., S) is defined as $\{s_0, s_1, \cdots, s_{N_A-1}\}$.

The slowdown of an application i with the resource allocation state s_i is defined in Equation 1, where IPS_{full} and IPS_{s_i} denote the instructions per second when the application is allocated full resources and the resources specified in s_i, respectively. The use of IPS data enables dynamic optimization and eliminates the need for measuring the execution time of each application.

$$\text{Slowdown}_i = \frac{IPS_{i,\text{full}}}{IPS_{i,s_i}} \quad (1)$$

In line with prior work [37], we define the unfairness (i.e., lower is better) of the consolidated applications using Equation 2, where μ denotes the average slowdown across the consolidated applications and σ denotes the standard deviation of the slowdowns of the applications [37].

$$\text{Unfairness} = \frac{\sigma}{\mu} \quad (2)$$

3 Experimental Methodology

3.1 System Configuration

To investigate the performance impact of LLC and memory bandwidth partitioning, we employ a 16-core commodity server system, whose specifications are summarized in Table 1. For all the experiments, the Hyper Threading and Turbo Boost features of the evaluated CPU are disabled. The server system is equipped with two memory DIMMs, providing a total bandwidth of ~28GB/s.

The evaluated CPU supports Cache Allocation Technology (CAT) [3]. The LLC partition assigned to each class of service (CLOS) is represented as a bit vector, in which each bit indicates whether the corresponding LLC way belongs to the CLOS. The LLC partition assigned to each CLOS can be dynamically adjusted through the `Resctrl` interface of Linux [3].

The evaluated CPU supports Memory Bandwidth Allocation (MBA) technology [3]. The MBA level can be changed from 100% (i.e., no throttling) to 10% in steps of 10% through the `Resctrl` interface of Linux [3]. The traffic between the L2 cache and the LLC is further throttled as the MBA level decreases.

3.2 Data Collection

We collect and analyze various performance data based on the Performance Monitoring Counters (PMCs) available in the evaluated CPU. Specifically, we collect the dynamically executed instructions, LLC accesses, and LLC misses from the PMCs using the Performance Application Programming Interface (PAPI) [27]. The use of LLC miss data enables the robust and efficient design and implementation of CoPart as the LLC misses incurred by the target application exhibit a strong correlation with the memory bandwidth consumed by the applications. Moreover, the collected LLC miss data can be effectively used to identify the LLC and memory bandwidth characteristics of the applications.

3.3 Benchmarks

We use 11 multithreaded benchmarks selected from the PARSEC, SPLASH, and NPB benchmark suites [5, 6, 38, 43]. In all the experiments, each benchmark is allocated its dedicated cores. In addition, the threads of each benchmark are pinned on the cores allocated to the benchmark.

Each benchmark is categorized into one of four categories: (1) LLC-sensitive, (2) memory bandwidth-sensitive, (3) LLC- and memory bandwidth-sensitive, and (4) insensitive. Table 2 summarizes all the evaluated benchmarks. The performance data of each benchmark such as LLC misses per second is collected by executing it with four threads and full hardware resources (i.e., 11 LLC ways and 100% MBA level).

The LLC-sensitive benchmarks are the benchmarks that show high performance sensitivity to the allocated LLC capacity. Specifically, we classify a benchmark as an LLC-sensitive benchmark if its performance degradation is 15% or more when the allocated LLC way count decreases from 11 to 1 at the 100% MBA level. The LLC-sensitive benchmarks tend to exhibit relatively high memory intensity (i.e., the ratio of the memory-related instructions to the total dynamically-executed instructions), show relatively high locality, and have the working-set sizes smaller than the LLC capacity.

The memory bandwidth-sensitive benchmarks are the benchmarks whose performance is sensitive to the allocated memory bandwidth. Specifically, we classify a benchmark as a memory bandwidth-sensitive benchmark if its performance degradation is 15% or more when the MBA level decreases from 100% to 10% (but still with 11 LLC ways). The memory bandwidth-sensitive benchmarks tend to exhibit high memory intensity. In addition, the memory bandwidth-sensitive benchmarks tend to show low locality, have the working-set sizes that exceed the LLC capacity, or both.

The LLC- and memory bandwidth-sensitive benchmarks are the benchmarks that exhibit high performance sensitivity to the allocated LLC capacity and memory bandwidth. Specifically, we classify a benchmark as an LLC- and memory bandwidth-sensitive benchmark if its performance degradation is 15% or more when the allocated LLC way count decreases from 11 to 1 at the 100% MBA level *and* 15% or

Table 2. Evaluated benchmarks and their characteristics

Benchmarks	Category	LLC accesses per second	LLC misses per second
water_nsquared (WN) [43]	LLC-sensitive	6.91×10^7	2.58×10^4
water_spatial (WS) [43]	LLC-sensitive	4.32×10^7	9.12×10^5
raytrace (RT) [43]	LLC-sensitive	3.76×10^7	2.16×10^4
ocean_cp (OC) [43]	Memory bandwidth-sensitive	5.19×10^7	4.88×10^7
CG [5, 38]	Memory bandwidth-sensitive	3.10×10^8	1.12×10^8
FT [5, 38]	Memory bandwidth-sensitive	2.45×10^7	2.00×10^7
SP [5, 38]	LLC- & memory BW-sensitive	1.69×10^8	9.21×10^7
ocean_ncp (ON) [43]	LLC- & memory BW-sensitive	9.49×10^7	7.89×10^7
FMM [43]	LLC- & memory BW-sensitive	6.12×10^6	3.47×10^6
swaptions (SW) [6]	Insensitive	1.08×10^4	7.98×10^2
EP [5, 38]	Insensitive	7.34×10^5	1.79×10^4

more when the MBA level decreases from 100% to 10% (but still with 11 LLC ways). The LLC- and memory bandwidth-sensitive benchmarks tend to exhibit high memory intensity. In addition, a large portion of the memory accesses performed by the LLC- and memory bandwidth-sensitive benchmarks exhibit high locality and their working-set sizes are smaller than the LLC capacity. Another significant portion of the memory accesses performed by the LLC- and memory bandwidth-sensitive benchmarks exhibit low locality and/or their working-set sizes exceed the LLC capacity.

The insensitive benchmarks are the benchmarks whose performance is insensitive to the allocated LLC capacity and memory bandwidth. Specifically, we classify a benchmark as an insensitive benchmark if its performance degradation is less than 1% when the allocated LLC way count decreases from 11 to 1 at the 100% MBA level *and* 1% when the MBA level decreases from 100% to 10% (but still with 11 LLC ways).

We also use the STREAM benchmark [26] to determine empirically the maximum memory traffic on the evaluated server system. Given that STREAM intensively executes memory-related instructions without temporal cache locality, we use it as a representative of the most memory bandwidth-intensive application.

We execute multiple benchmarks together as a *workload mix*. We run each benchmark in a workload mix in a separate Linux container. For each workload mix, we execute it for 50 seconds and collect the performance data including the number of instructions that are executed by each benchmark in the workload mix.

As a case study for dynamic server consolidation, we use the memcached benchmark from CloudSuite [10, 29] as the latency-critical application. It is adapted from a production-quality in-memory key-value store [11] and uses the Twitter dataset with a scaling factor of 30 [10, 29]. As the batch workloads, we use Word Count and Kmeans, which are the Spark versions [1, 47] of in-memory big-data workloads in the BigDataBench benchmark suite [41]. We use the 64GB and 4GB input datasets for Word Count and Kmeans.

4 Characterization

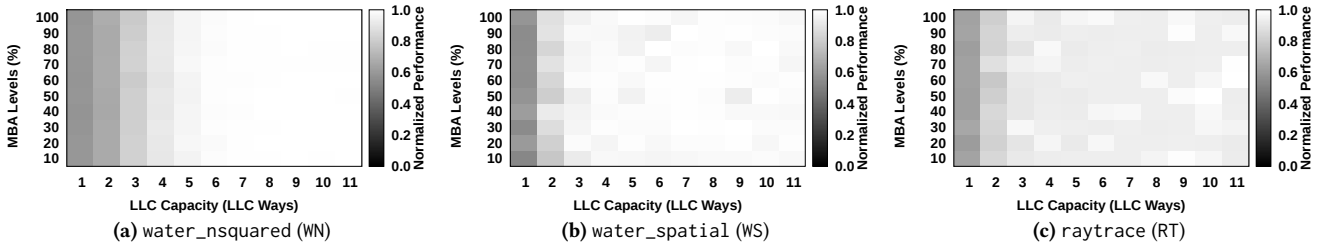
4.1 Performance Characterization

We investigate the performance impact of LLC and memory bandwidth partitioning on multithreaded benchmarks with various characteristics. Specifically, we characterize the performance sensitivity of the LLC-sensitive, memory bandwidth-sensitive, and LLC- and memory bandwidth-sensitive benchmarks (Table 2) to the allocated LLC capacity and memory bandwidth. We collect the performance of each benchmark by executing it with four threads and no co-running benchmark.

Figure 1 shows the performance of the LLC-sensitive benchmarks (i.e., WN, WS, RT) when they are allocated different amounts of LLC and memory bandwidth. Each tile shows the performance (i.e., instructions per second) of the corresponding system state, which is normalized to that of the system state that exhibits the highest performance.

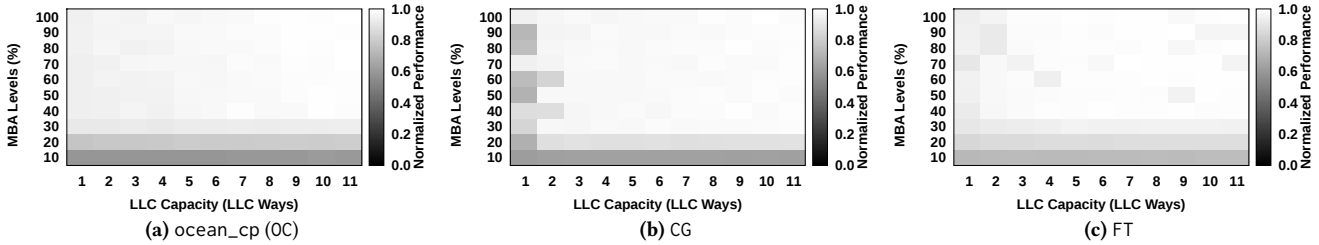
First, the performance of the LLC-sensitive benchmarks is primarily dependent on the allocated LLC capacity and relatively insensitive to the allocated memory bandwidth even when the MBA level is set to small values, although there are some data points that exhibit small performance variations between adjacent system states due to the sophisticated performance implications of the allocated resources (e.g., the performance impact of the allocated ways to the cache replacement policy). This overall performance data trend shows that excessive memory bandwidth allocated to the LLC-sensitive benchmarks can be effectively reclaimed to the consolidated memory bandwidth-sensitive applications to improve the overall fairness.

Second, the LLC-sensitive benchmarks require different amounts of LLC to maintain reasonable performance. Specifically, WN, WS, and RT require 4, 3, and 2 LLC ways to achieve 90% of the performance that can be achieved with the full LLC capacity. This performance data trend shows that the resource manager for fairness-aware workload consolidation should robustly determine the LLC capacity required for each consolidated application and accordingly allocate the LLC to each application to improve the overall fairness.



(a) water_nsquared (WN) **(b)** water_spatial (WS) **(c)** raytrace (RT)

Figure 1. Performance impact of LLC and memory bandwidth partitioning on LLC-sensitive benchmarks



(a) ocean_cp (OC) **(b)** CG **(c)** FT

Figure 2. Performance impact of LLC and memory bandwidth partitioning on memory bandwidth-sensitive benchmarks

Figure 2 shows the performance impact of LLC and memory bandwidth partitioning on the memory bandwidth-sensitive benchmarks (i.e., OC, CG, FT). First, the performance of the memory bandwidth-sensitive benchmarks is largely dependent on the allocated memory bandwidth and relatively insensitive to the allocated LLC capacity. This overall performance trend shows that the resource manager for fairness-aware workload consolidation can effectively reclaim excessive LLC ways allocated to the memory bandwidth-sensitive applications and allocate them to the consolidated LLC-sensitive applications to achieve high fairness.

Second, the memory bandwidth-sensitive benchmarks require different MBA levels to maintain reasonable performance. Specifically, OC, CG, and FT require MBA levels of 30, 20, and 30 to achieve 90% of the performance that can be achieved at the 100% MBA level. Therefore, it is highly crucial for the resource manager to accurately determine the required memory bandwidth for each memory bandwidth-sensitive application and allocate the memory bandwidth to each application in order to improve the overall fairness.

Figure 3 shows the performance impact of LLC and memory bandwidth partitioning on the benchmarks (i.e., SP, ON, FMM) that are sensitive to both the LLC and memory bandwidth. First, the performance of the LLC- and memory bandwidth-sensitive benchmarks is highly dependent on both the allocated LLC capacity and memory bandwidth, which motivates the need for coordinated LLC and memory bandwidth partitioning.

Second, the LLC- and memory bandwidth-sensitive benchmarks achieve similar performance with different system states. For instance, SP achieves similar performance when it is allocated 8 LLC ways and the 20% MBA level and 3 LLC ways and the 40% MBA level. This data trend shows that the resource manager for workload consolidation should be capable of determining the efficient system state among the various candidate system states to achieve high fairness.

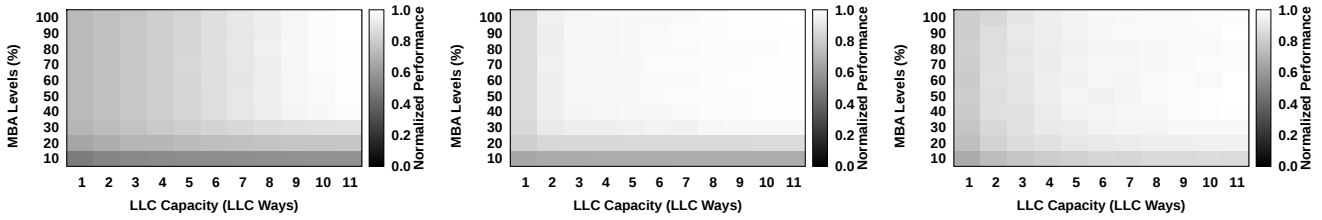
4.2 Fairness Characterization

We investigate the fairness impact of LLC and memory bandwidth partitioning when co-running multiple benchmarks. Specifically, we characterize the fairness of three workload mixes: (1) the LLC-sensitive workload mix, which consists of three LLC-sensitive benchmarks and one insensitive benchmark, (2) the memory bandwidth-sensitive workload mix, consisting of three memory bandwidth-sensitive benchmarks and one insensitive benchmark, and (3) the LLC- and memory bandwidth-sensitive (LM) workload mix, which is composed of three LM benchmarks and one insensitive benchmark.

Figure 4 shows the unfairness results when executing the LLC-sensitive workload mix (i.e., WN, WS, RT, SW). Each tile shows the unfairness (i.e., lower is better) of the corresponding system state, which is normalized to the unfairness of the system state where all the benchmarks are executed without any resource partitioning. For instance, memory bandwidth partitioning of (20, 10, 100, 10) indicates that the MBA level for WN, WS, RT, and SW is set to 20%, 10%, 100%, and 10%, respectively. For another example, LLC partitioning of (5, 3, 2, 1) indicates that 5, 3, 2, and 1 LLC ways are allocated to WN, WS, RT, and SW, respectively.

First, the fairness of the LLC-sensitive workload mix is primarily determined by the LLC capacity allocated to each LLC-sensitive benchmark. Specifically, the fairness is significantly degraded when an insufficient amount of LLC is allocated to one of the LLC-sensitive benchmarks. For instance, when WN is allocated 2 LLC ways, low fairness is observed. This is due to the drastic performance degradation of WN when allocated 2 or fewer LLC ways.

When each LLC-sensitive benchmark is allocated LLC ways with which it can achieve reasonable performance, high fairness can be achieved. For example, when WN, WS, and RT are allocated 5, 3, and 2 ways, high fairness is achieved than



(a) SP **(b)** ocean_ncp (ON) **(c)** FMM

Figure 3. Performance impact of LLC and memory bandwidth partitioning on LLC- and memory BW-sensitive benchmarks

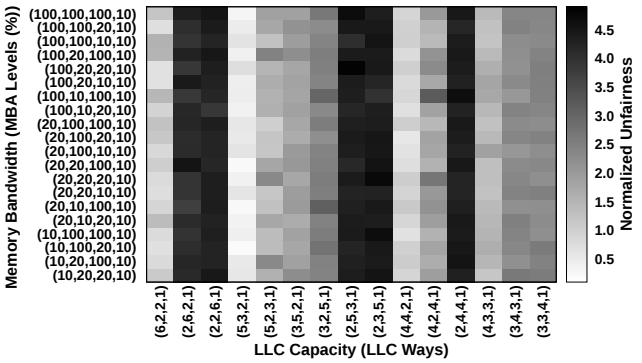


Figure 4. Fairness impact of LLC and memory bandwidth partitioning with LLC-sensitive applications

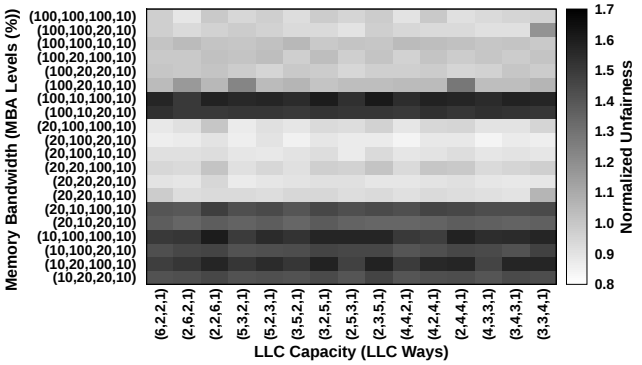


Figure 5. Fairness impact of LLC and memory bandwidth partitioning with memory bandwidth-sensitive applications

other LLC partitioning settings. This is mainly because each LLC-sensitive benchmark achieves reasonable performance when allocated these amounts of LLC capacity.

Second, for the given LLC partitioning, the fairness of the LLC-sensitive workload mix significantly varies with different memory bandwidth partitioning settings. For instance, when WN, WS, and RT are allocated 5, 3, and 2 ways, the fairness of the LLC-sensitive workload mix significantly varies across the memory bandwidth settings. This arises mainly because LLC-sensitive benchmarks compete for memory bandwidth by generating larger amounts of memory traffic due to the increased LLC misses when they are allocated relatively small amounts of LLC. This fairness data trend demonstrates the need for a coordinated approach for LLC and memory bandwidth partitioning to achieve high fairness.

Figure 5 shows the unfairness results when executing the memory bandwidth-sensitive workload mix (i.e., OC, CG, FT, SW). First, the fairness of the memory bandwidth-sensitive workload mix is largely determined by the memory bandwidth allocated to each memory bandwidth-sensitive benchmark. For example, when the two most memory bandwidth-sensitive benchmarks (i.e., OC, CG) are allocated insufficient memory bandwidth (i.e., MBA level of 10%), the overall fairness is significantly degraded. The main cause of this is the significant performance degradation of the memory bandwidth-sensitive benchmarks, which in turn stems from the insufficient memory bandwidth.

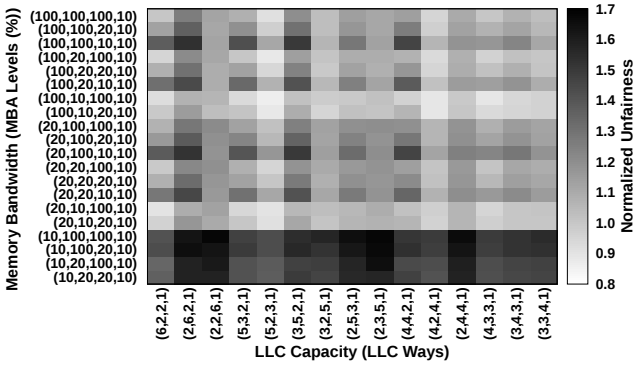


Figure 6. Fairness impact of LLC and memory BW partitioning with LLC- and memory BW-sensitive applications

Second, for the given memory bandwidth partitioning, the fairness of the memory bandwidth-sensitive workload mix shows relatively small variations with different LLC partitioning settings. This is primarily because the performance of each memory bandwidth-sensitive benchmark exhibits no or little performance sensitivity to the allocated LLC capacity. This fairness data trend shows that the excessive LLC capacity allocated to memory bandwidth-sensitive applications can be effectively reallocated to the consolidated LLC-sensitive applications to improve the overall fairness.

Figure 6 shows the unfairness results when executing the LM workload mix (i.e., SP, ON, FMM, SW). Since the performance of the LM benchmarks is sensitive to both the LLC and memory bandwidth, the fairness of the LM workload mix is highly dependent on LLC and memory bandwidth partitioning. Thus, it is highly crucial to precisely analyze the characteristics of the consolidated applications and dynamically allocate the LLC and memory bandwidth in a coordinated manner in order to achieve high fairness.

4.3 Lessons Learned

We summarize the findings learned from the characterization of LLC and memory bandwidth partitioning as follows.

- The performance of the LLC-sensitive benchmarks is highly sensitive to the allocated LLC capacity but relatively insensitive to the allocated memory bandwidth. A similar data trend is observed with memory bandwidth-sensitive benchmarks and their allocated LLC capacity.
- The performance of the LLC- and memory bandwidth-sensitive (LM) benchmarks exhibits high sensitivity to both the allocated LLC capacity and the memory bandwidth. In addition, there are multiple system states that provide the similar performance when executing the LM benchmarks.
- The fairness of the LLC-sensitive applications is primarily dependent on the allocated LLC capacity. The LLC-sensitive applications require different amounts of LLC capacity to achieve reasonable performance. When allocated small LLC capacity, they tend to demand more memory bandwidth in order to achieve reasonable performance with the increased LLC misses.
- The fairness of the memory bandwidth-sensitive applications is largely determined by the allocated memory bandwidth. They require different amounts of memory bandwidth to achieve reasonable performance. Further, they exhibit little performance sensitivity to the allocated LLC capacity even when the allocated memory bandwidth is low.
- The fairness of the LM applications is highly dependent on both LLC and memory bandwidth partitioning. Overall, our characterization results clearly motivate the need for a coordinated approach for LLC and memory bandwidth partitioning to significantly improve the overall fairness of the consolidated workloads.

5 Design and Implementation

5.1 Overview

CoPart aims to dynamically configure the system state in a way that maximizes the overall fairness of the consolidated applications. CoPart is a coordinated technique in that it simultaneously partitions the LLC and memory bandwidth across the consolidated applications. CoPart performs in an application characteristic-aware manner in the sense that it dynamically analyzes the performance characteristics of each consolidated application with the allocated resources.

While we aim to describe the design and implementation of CoPart in an architecture-agnostic manner as much as possible, we discuss some of the techniques implemented in CoPart in the context of the `x86-64` architecture. This is mainly because the current implementation of CoPart uses certain `x86-64` features (e.g., Intel CAT and MBA).

The goal of CoPart is to find an efficient system state that significantly improves the overall fairness of the consolidated applications. CoPart consists of three components: (1) the LLC characteristic classifier, (2) the memory bandwidth characteristic classifier, and (3) the resource manager. Figure 7 shows the overall architecture of CoPart.

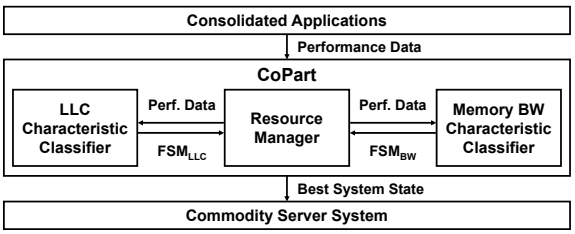


Figure 7. Overall architecture of CoPart

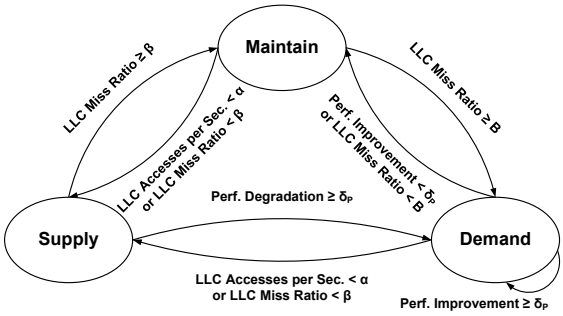


Figure 8. FSM of the LLC characteristic classifier

5.2 LLC Characteristic Classifier

For a given system state S and each consolidated application, the LLC characteristic classifier dynamically determines whether the application can supply one of the allocated LLC ways, needs to maintain the currently allocated LLC ways, or demands more LLC ways. To dynamically track the state of each application in terms of LLC allocation, the LLC characteristic classifier employs the finite state machine (FSM), which is shown in Figure 8.

There are three states (i.e., Supply, Demand, and Maintain) in the FSM of the LLC characteristic classifier. If the application is in the Supply state, it is expected that an LLC way can be reclaimed from the application without significantly degrading its performance. If the application is in the Demand state, it is expected that the performance of the application can be significantly improved if an additional LLC way is allocated to the application. If the application is in the Maintain state, it is expected that allocating an addition LLC way provides marginal performance gains *whereas* reclaiming an LLC way from the application significantly degrades its performance.

Figure 8 shows all the possible transitions among the states and their conditions in the FSM of the LLC characteristic classifier. In this work, α, β, B, and δ_P in Figure 8 are set to 1.5×10^6, 1%, 3%, and 5%.[1] The resource manager, which will be subsequently discussed in Section 5.4, selects one of the three states as the initial state for each application based

[1]The design parameter values of CoPart have been determined through design space exploration, which is discussed in Section 5.5.3.

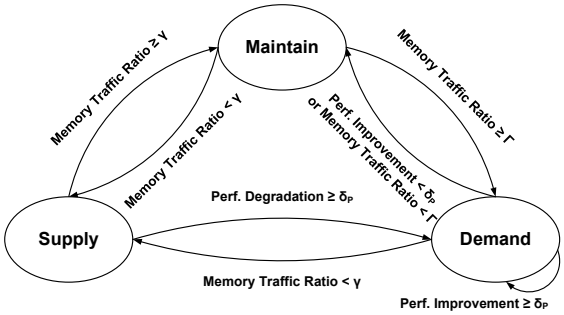


Figure 9. FSM of the memory BW characteristic classifier

on its characteristics identified from the profile data. For instance, let us suppose the DEMAND state is selected as the initial state for an application based on its characteristics. If the performance of the application is considerably improved when an additional LLC way is allocated, the application continues to stay in the DEMAND state. If the LLC access rate (i.e., accesses per second) or the LLC miss ratio is sufficiently low with an additional LLC way, the application transitions to the SUPPLY state. If the performance improvement with an additional LLC way is small, the application transitions to the MAINTAIN state.

5.3 Memory Bandwidth Characteristic Classifier

For a given system state S and each consolidated application, the memory bandwidth characteristic classifier determines whether the application can supply its allocated memory bandwidth, must maintain the currently allocated memory bandwidth, or demands more memory bandwidth. Similarly to the LLC characteristic classifier, the memory bandwidth characteristic classifier uses the finite state machine (FSM) to dynamically track the state of each application in terms of memory bandwidth allocation.

Figure 9 shows the FSM of the memory bandwidth characteristic classifier, which is maintained for each consolidated application. The states and state transitions in the FSM of the memory bandwidth characteristic classifier are designed similarly to the LLC characteristic classifier. The term "memory traffic ratio" in Figure 9 denotes the ratio of the LLC miss rate of each application to the LLC miss rate of the `STREAM` benchmark (Section 3.3) at the MBA level of M. In this work, γ, Γ, and δ_P in Figure 9 are set to 10%, 30%, and 5%, respectively. The resource manager selects one of the three states (i.e., SUPPLY, DEMAND, and MAINTAIN) as the initial state for each application based on its performance characteristics identified from the profile data.

We have judiciously made a design decision to maintain separate FSMs for the LLC and memory bandwidth characteristic classifiers. This is mainly to keep the design and implementation complexity of the classifiers low by having smaller and simpler FSMs instead of a larger and more complicated single FSM (i.e., with more states and transitions). However, each FSM is designed in awareness of the interaction between the allocated resources. For instance, we have designed the FSM of the memory bandwidth characteristic classifier in a way that the application remains in the DEMAND state even if the performance improvement is small, but the recently allocated resource is an LLC way. This is because the marginal performance improvement stems from the low performance sensitivity to the LLC, but not to the memory bandwidth.

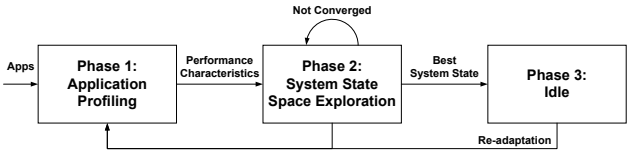


Figure 10. Execution flow of the resource manager

5.4 Resource Manager

The resource manager of CoPart dynamically explores the system state space to find an efficient system state that significantly improves the overall fairness of the consolidated applications. The resource manager consists of the following three phases: (1) the application profiling phase, (2) the system state space exploration phase, and (3) the idle phase. Figure 10 shows the execution flow of the resource manager.

5.4.1 Application Profiling Phase

The first phase of the resource manager is the application profiling phase. The main goal of the application profiling phase is to efficiently profile each of the consolidated applications and identify its performance characteristics without the need for extensive profiling.

Specifically, the resource manager briefly executes each application with all of the LLC ways and memory bandwidth allocated to the application. This is to compute the slowdown of the application when it is subsequently allocated partial amounts of the LLC and memory bandwidth.

In addition, the resource manager briefly executes each application with the resource allocation states of (l_P, 100%) and (L, M_P), where l_P and M_P are the LLC ways and MBA level for profiling the application, which are set to 2 and 20% in this work. This is to determine the performance sensitivity of the application to both the allocated LLC capacity and memory bandwidth. The initial states in the FSMs of the LLC and memory bandwidth characteristic classifiers for each application are determined based on the performance sensitivity of the application. For instance, if the performance degradation of the application exceeds a predefined threshold (i.e., 10%) with the resource allocation state of (l_P, 100%), its initial state in the FSM of the LLC characteristic classifier is set to the DEMAND state. After completing the application profiling phase, the resource manager transitions to the system state space exploration phase.

5.4.2 System State Space Exploration Phase

During the system state space exploration phase, the resource manager explores the system state space in order

Algorithm 1 The exploreSystemStateSpace function

```
1:  procedure EXPLORESYSTEMSTATESPACE
2:      appList ← getAppList()
3:      state ← initializeState(appList)
4:      retryCount ← 0
5:      while true do
6:          setSystemState(state)
7:          sleep(period)
8:          updateFSMs(appList)
9:          previousState ← state
10:         state ← getNextSystemState(state, appList)
11:         if previousState = state then
12:             if retryCount < θ then ▷ In this work, θ is set to 3.
13:                 state ← getNeighborState(state, appList)
14:                 retryCount ← retryCount + 1
15:             else
16:                 transitionToIdlePhase()
```

to find an efficient system state that significantly improves the fairness of the consolidated applications. Algorithm 1 shows the pseudocode for the `exploreSystemStateSpace` function, which is the top-level function of the system state space exploration phase.

In a nutshell, the resource manager explores the system state space as follows. First, it periodically collects the runtime data (e.g., dynamically-executed instructions, LLC accesses, and LLC misses) from each of the consolidated applications and updates the FSMs of the classifiers (Line 8). It chooses an efficient system state that is expected to significantly improve the overall fairness based on the runtime data and the current system state (Line 10). If the generated system state is same as the previous state, it randomly chooses one of the neighbor system states to introduce some randomness into the overall search process (Lines 11–14). It then transitions to the newly selected system state (Line 6). It repeats this process until no further fairness improvements are expected. Finally, it transitions to the idle phase (Line 16).

In each period, the resource manager selects an efficient system state to significantly improve the overall fairness by invoking the `getNextSystemState` function in Algorithm 2. Based on the classification results from the LLC and memory bandwidth characteristic classifiers, each application attempts to *supply* (i.e., producer), *maintain*, or *demand* (i.e., consumer) each of the resources (i.e., LLC and memory bandwidth).[2] The main goal of the system state space exploration phase is to find an efficient match between the producers and consumers.

We formulate our resource allocation problem as the Hospitals/Residents (HR) problem [12] mainly because the HR problem is widely applicable to various resource allocation problems and efficient solutions to the HR problem have been

[2]Following the terminology used in economics, we refer an application that attempts to supply (or demand) a resource to a producer (or consumer).

Algorithm 2 The getNextSystemState function

```
1:  procedure GETNEXTSYSTEMSTATE(state, appList)
2:      producers ← filter(appList, PRODUCER)
3:      resources[LLC].producers ← filter(producers, LLC)
4:      resources[MBA].producers ← filter(producers, MBA)
5:      resources[ANY].producers ← filter(producers, ANY)
6:      consumers ← filter(appList, CONSUMER)
7:      for c in consumers do
8:          consumer ← c
9:          while true do
10:             if consumer.preferences.isEmpty() then
11:                 break
12:             t ← consumer.preferences.delete(preference, MAX)
13:             r ← resources[t]
14:             r.consumers.insert(consumer)
15:             if r.consumers.size() > r.producers.size() then
16:                 consumer ← r.consumers.delete(slowdown, MIN)
17:             else
18:                 break
19:     for t in {LLC, MBA, ANY} do
20:         for c in resources[t].consumers do
21:             if t ≠ ANY then
22:                 resourceType ← t
23:             else if c.resourceType ≠ ANY then
24:                 resourceType ← c.resourceType
25:             else ▷ t = ANY and c.resourceType = ANY
26:                 resourceType ← selectLLCorMBA()
27:             p ← resources[t].producers.delete(slowdown, MIN)
28:             state[p].decrease(resourceType)
29:             state[c].increase(resourceType)
30:     return state
```

investigated as one of the most extensively-studied problems in economics. In the HR problem, there are H hospitals and R medical students. Each hospital admits a certain number of medical students as residents. Each medical student applies to a certain number of hospitals for a resident position. Each hospital has a preference list of the medical students that they want to accept. Each medical student has a preference list of the hospitals to which they want to go.

The objective of the HR problem is to find a match between hospitals and medical students, in which all the pairs in the match are *stable*. A match is stable if it contains no *blocking* pairs. There are blocking pairs if the match has been established against the preferences of the hospitals and medical students. For example, let us suppose that hospitals h_A and h_B prefer medical students s_A and s_B. In addition, the medical students s_A and s_B prefer the hospitals h_A and h_B, respectively. If a match contains pairs of (h_A, s_B) and (h_B, s_A), the pairs are blocking pairs. The match is unstable in that the hospitals and medical students in the blocking pairs would attempt to break the match and find a stable match (i.e., (h_A, s_A) and (h_B, s_B)) that satisfies their preferences.

In our resource allocation problem, we consider the resource types (i.e., LLC, memory bandwidth, and any (i.e., LLC or memory bandwidth)) that some of the consolidated applications are willing to *supply* as the hospitals. The number of the applications that are willing to supply each type of resources is considered as the number of available residency positions at each hospital. The sorted list of the application slowdowns is considered as the preference list of each hospital. In other words, if the slowdown of an application is high, the application is given a higher priority to receive the corresponding resource.

In addition, we consider the applications that demand an additional resource (i.e., an LLC way or a level-up in the MBA setting) as the medical students. If an application demands a single resource type, it prefers producers that are willing to supply the corresponding resource type to other producers that are willing to supply any of the resource types (i.e., any of LLC and memory bandwidth). This is to increase the number of pairs in the match produced by the algorithm. If an application demands any resource type (i.e., LLC and memory bandwidth), it randomly assigns a higher priority to one of the two resource types. This introduces some randomness to the overall search process, reducing the possibility of converging to a local optimal solution.

To address our resource allocation problem, we propose an algorithm based on the instability-chaining algorithm, which finds a stable match with polynomial time complexity [35]. The `getNextSystemState` function shown in Algorithm 2 implements the proposed algorithm. The `getNextSystemState` function takes the current system state and the list of the consolidated applications as the inputs and returns an efficient system state that is expected to significantly improve the fairness of the consolidated applications as the output. It mainly consists of two steps.

During the first step (Lines 7–18 in Algorithm 2), the proposed algorithm determines which consumers (i.e., the applications that demand more resources) can acquire which resources. It iterates each consumer and attempts to allocate the demanded resource based on the preference list of the consumer. If the demanded resource type is oversubscribed, it chooses the consumer with the lowest slowdown among the consumers that have been tentatively allocated the resource type as a victim. This is to improve the overall fairness by favoring the consumers with higher slowdowns when allocating resources. It then continues to decide a resource that can be allocated to the victim based on the preference list of the victim. It repeats this process until it finishes the iteration of all the consumers.

During the second step (Lines 19–29 in Algorithm 2), the proposed algorithm determines which resources need to be reclaimed from which producers (i.e., the applications that are willing to supply their allocated resources). Among the producers that can supply the same resource type, it favors the ones with lower slowdowns to improve the overall fairness.

Considering that the sizes of the application-related lists (e.g., `consumers`, `resources.producers`) in Algorithm 2 are $O(N_A)$, the time complexity of the proposed algorithm is $O(N_A^2)$, where N_A denotes the number of the consolidated applications. As quantified in Section 6, the resource manager incurs small performance overhead as the proposed algorithm has low time complexity.

5.4.3 Idle Phase

During the idle phase, CoPart continues to monitor the consolidated applications and underlying server system without performing any adaptation activities. If a change (e.g., the launch of a new application, the termination of an application) is detected, CoPart terminates the idle phase and triggers the aforementioned adaptation process.

5.5 Discussion

5.5.1 OS-Level Implementation

While the prototype of CoPart is implemented as a user-level runtime system on Linux, CoPart can be also implemented as a part of the OS. The user-level implementation of CoPart significantly improves the portability and applicability across systems and simplifies the overall implementation as it eliminates the need for modifying the underlying OS. The potential advantage of the OS-level implementation of CoPart is that it eliminates the need to use the interfaces (e.g., the `Resctrl` interface) between the OS and user-level applications for dynamically allocating the LLC and memory bandwidth to the consolidated applications.

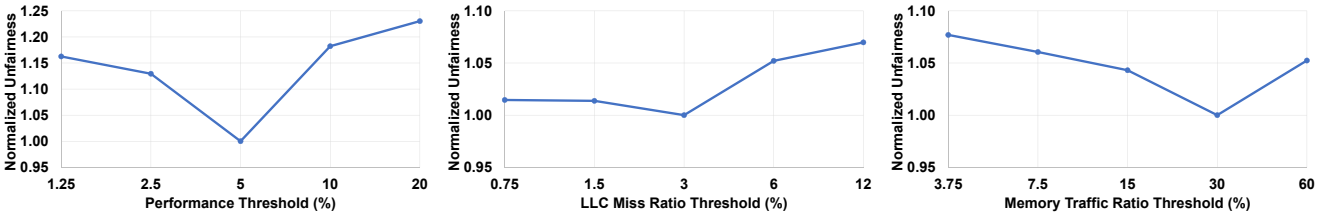
5.5.2 Other OSes and Virtualization Technologies

In this work, we implement CoPart based on the Linux and containers due to their widespread use and efficiency. However, we believe that the mechanisms and techniques proposed in this work can be used to support other widely-used OSes and virtualization technologies (e.g., hypervisors for hardware virtualization) as long as software interfaces for LLC and memory bandwidth partitioning are provided by the underlying system software and hardware stacks.

5.5.3 Design Space Exploration

The design parameter values of CoPart (e.g., the performance threshold parameter) have been determined through design space exploration with various applications that cover a wide range of LLC and memory bandwidth characteristics and system configurations. Specifically, the design parameter values have been set to the ones that consistently achieve high fairness across the various applications and system configurations. We present the sensitivity results of CoPart to the three key design parameters (i.e., the performance, LLC miss ratio, and the memory traffic ratio threshold parameters).

Figure 11a shows the sensitivity of CoPart to the performance threshold parameter (i.e., δ_P in Sections 5.2 and 5.3). Each data point is normalized to the unfairness of CoPart when the performance threshold parameter is set to



(a) Performance threshold parameter **(b)** LLC miss ratio threshold parameter **(c)** Memory traffic ratio threshold parameter

Figure 11. Sensitivity to the design parameters

5%, which is the setting used in this work. We observe that the unfairness decreases, reaches the minimum, and then increases as the value of the performance threshold parameter increases. This is mainly due to the trade-off between increasing and decreasing the value of the performance threshold parameter. With small parameter values, it takes a longer time to converge to an efficient system state as CoPart keeps performing adaptations even with small performance changes. With large parameter values, CoPart may converge to a system state too quickly as adaptations are performed only when large performance changes are observed.

Figure 11b shows the sensitivity of CoPart to the LLC miss ratio threshold parameter (i.e., B in Section 5.2). Each data point is normalized to the unfairness of CoPart when the LLC miss ratio threshold parameter is set to 3%, which is the setting used in this work. Similarly to the case with the performance threshold parameter, the unfairness decreases, exhibits the minimum, and increases as the LLC miss ratio threshold parameter increases. With small parameter values, even applications that exhibit low performance sensitivity to the allocated LLC capacity may demand and compete for the LLC. With large parameter values, applications that exhibit high performance sensitivity to the allocated LLC capacity may fail to demand the LLC, which results in lower fairness.

Figure 11c shows the sensitivity of CoPart to the memory traffic ratio threshold parameter (i.e., Γ in Section 5.3). Each data point is normalized to the unfairness of CoPart when the memory traffic ratio threshold parameter is set to 30%, which is the setting used in this work. We observe the unfairness data trend similar to the performance and LLC miss ratio threshold parameters with respect to the memory traffic ratio threshold parameter. With small parameter values, even applications that incur relatively low memory traffic demand and compete for the memory bandwidth. With large parameter values, applications that incur high memory traffic may fail to demand the memory bandwidth. The configuration of CoPart used in this work balances the trade-off between increasing and decreasing the values of the design parameters and consistently achieves high fairness.

6 Evaluation

This section quantifies the effectiveness of CoPart with respect to the following: (1) the fairness results, (2) the sensitivity to the application and system configurations, (3) the runtime behavior with latency-critical and batch workloads, and (4) the overhead and the performance impact of CoPart.

6.1 Fairness Results

We first investigate the fairness results of CoPart. To quantify the effectiveness of CoPart with workload mixes with various characteristics, we generate the following workload mixes based on the benchmarks in Table 2: (1) `H-LLC`: highly LLC-sensitive with three LLC-sensitive benchmarks and one insensitive benchmark, (2) `H-BW`: highly memory bandwidth-sensitive (i.e., three memory bandwidth-sensitive benchmarks and one insensitive benchmark), (3) `H-Both`: highly LLC- and memory bandwidth-sensitive (i.e., three LLC and memory bandwidth-sensitive benchmarks and one insensitive benchmark), (4) `M-LLC`: moderately LLC-sensitive (i.e., two LLC-sensitive benchmarks and two insensitive benchmarks), (5) `M-BW`: moderately memory bandwidth-sensitive (i.e., two memory bandwidth-sensitive benchmarks and two insensitive benchmarks), (6) `M-Both`: moderately LLC- and memory bandwidth-sensitive (i.e., two LLC- and memory bandwidth-sensitive benchmarks and two insensitive benchmarks), and (7) `IS`: insensitive (i.e., four insensitive benchmarks) workload mixes.

We execute each workload mix with the following resource allocation policies: (1) equal allocation (`EQ`), which allocates equal amounts of LLC and memory bandwidth to each application, (2) static allocation (`ST`), which statically employs the system state that exhibits the highest fairness among the system states that are evaluated through extensive offline experiments, (3) `CAT-only`, which represents techniques that employ dynamic LLC partitioning and equal memory bandwidth partitioning to execute the workload mix, (4) `MBA-only`, which represents techniques that execute the workload mix with equal LLC partitioning and dynamic memory bandwidth partitioning, and (5) CoPart, which executes the workload mix with the LLC and memory bandwidth partitioning performed by CoPart.

Figure 12 shows the unfairness results (i.e., lower is better) of CoPart and the other resource allocation policies. Each bar is normalized to the unfairness of the `EQ` version. The rightmost bars show the average (i.e., geometric mean) of the unfairness of each resource allocation policy across all the workload mixes.

First, CoPart achieves significantly higher fairness than the `EQ`, `CAT-only`, and `MBA-only` versions across the workload mixes. Specifically, CoPart delivers 57.3%, 28.6%, and

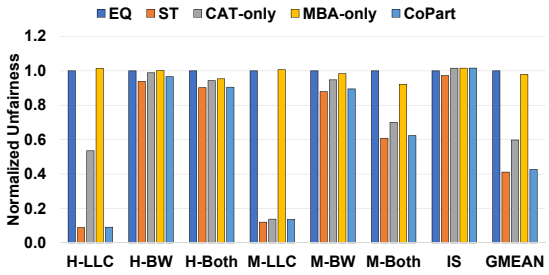


Figure 12. Unfairness results

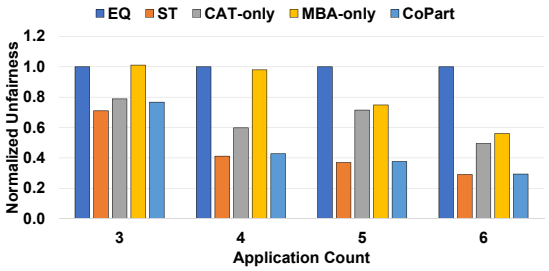


Figure 13. Sensitivity to the application count

56.4% higher fairness (i.e., lower unfairness) than the EQ, CAT-only, and MBA-only versions on average across the workload mixes. This is mainly because CoPart robustly analyzes the characteristics of the consolidated applications and dynamically allocates the LLC and memory bandwidth in a coordinated and fairness-aware manner.

Second, CoPart delivers larger fairness improvements over the EQ version across all the workload mixes except for the insensitive workload mix. Since the EQ version equally allocates the LLC and memory bandwidth to the consolidated applications without considering their characteristics, it significantly degrades the overall fairness. In contrast, CoPart achieves significantly higher fairness by dynamically analyzing the characteristics of the consolidated applications and allocating LLC and memory bandwidth to the applications in awareness of their characteristics.

Third, CoPart achieves considerably higher fairness mainly with the memory bandwidth-sensitive and LLC- and memory bandwidth-sensitive workload mixes than the CAT-only version. Since the CAT-only version lacks the capability of fairness and application characteristic-aware memory bandwidth partitioning, the consolidated applications that are allocated insufficient amounts of memory bandwidth suffer from higher performance degradation, resulting in significantly low fairness.

Interestingly, CoPart achieves significantly higher fairness with the highly LLC-sensitive workload mixes than the CAT-only version. Since highly LLC-sensitive benchmarks are allocated relatively small LLC capacity, they incur frequent LLC misses. Because the CAT-only version lacks the capability of memory bandwidth partitioning, it significantly degrades the overall fairness of the highly LLC-sensitive workload mix. In contrast, CoPart effectively allocates the memory bandwidth across the highly LLC-sensitive benchmarks, achieving significantly higher fairness than the CAT-only version.

Fourth, CoPart delivers significantly larger fairness improvements over the MBA-only version with the LLC-sensitive and LLC- and memory bandwidth-sensitive workload mixes. Since the MBA-only version lacks the capability of fairness- and application characteristic-aware LLC partitioning, the consolidated benchmarks allocated insufficient amounts of the LLC undergo greater performance degradation, significantly degrading the fairness. In contrast, CoPart partitions the LLC across the benchmarks in a fairness- and application characteristic-aware manner, resulting in significant fairness improvements over the MBA-only version.

Fifth, CoPart achieves the fairness similar to the ST version, which requires extensive offline profiling to determine the system state that delivers high fairness empirically. CoPart effectively identifies the characteristics of the consolidated applications and allocates the LLC and memory bandwidth based on their characteristics, achieving high fairness.

6.2 Sensitivity to the Configurations

We investigate the fairness sensitivity of CoPart to the application count. To this end, we sweep the application count of each workload mix from three to six. With regard to the workload mixes with application counts other than four, we generate the seven workload mixes similarly to the case with the workload mixes with four benchmarks, as discussed in Section 6.1.

Figure 13 shows the unfairness results (i.e., lower is better) of CoPart and the other resource allocation policies with various application counts. Each bar shows the average (i.e., geometric mean) unfairness of each version across the workload mixes with the same application count, normalized to that of the EQ version.

First, CoPart achieves considerably higher fairness than the EQ, CAT-only, and MBA-only versions and delivers the fairness comparable with the ST version across all the application counts. For instance, CoPart achieves 70.6%, 40.6%, and 47.6% higher fairness than the EQ, CAT-only, and MBA-only versions when the application count is 6. Regardless of the application count, CoPart dynamically analyzes the characteristics of the consolidated benchmarks and effectively partitions the LLC and memory bandwidth in a coordinated and fairness-aware manner, achieving significantly higher fairness than the EQ, CAT-only, and MBA-only versions.

Second, CoPart tends to achieve larger fairness improvements over the EQ, CAT-only, and MBA-only versions with larger application counts. For instance, CoPart achieves 23.3% and 70.6% higher fairness than the EQ version when the application count is 3 and 6, respectively. The higher contention over the LLC and memory bandwidth, which is caused by a larger number of benchmarks, exacerbates the fairness issues among the consolidated benchmarks. Since the EQ,

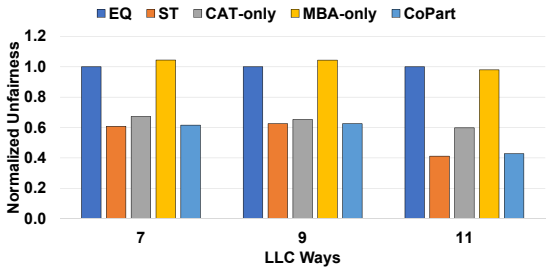


Figure 14. Sensitivity to the total LLC capacity

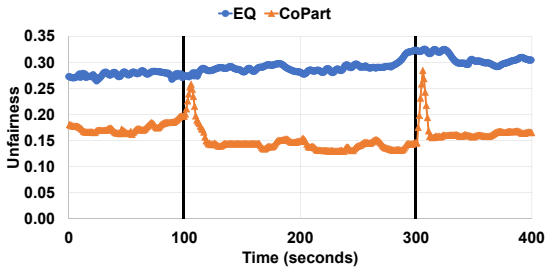


Figure 15. Runtime behavior of CoPart

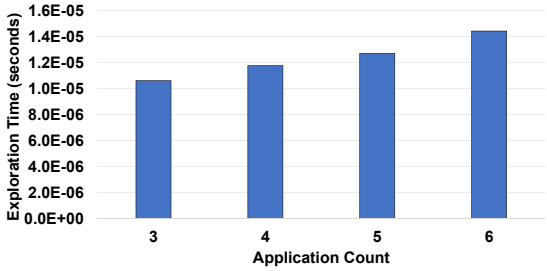


Figure 16. Overhead of CoPart

`CAT-only`, and `MBA-only` versions allocate the resources across the benchmarks in a fairness-oblivious and/or uncoordinated manner, they exhibit low fairness under high contention with a large number of consolidated benchmarks. In contrast, CoPart continues to achieve high fairness by robustly partitioning the LLC and memory bandwidth in a coordinated and fairness-aware manner even when a larger number of benchmarks are consolidated.

Figure 14 shows the unfairness results (i.e., lower is better) of CoPart and the other resource allocation policies as the total LLC capacity is swept from 7 to 11 LLC ways. Each bar shows the average (i.e., geometric mean) unfairness of each version across the workload mixes with the same total LLC capacity, normalized to that of the `EQ` version. We observe that CoPart achieves considerably higher fairness than `EQ`, `CAT-only`, and `MBA-only` versions and delivers the fairness comparable with the `ST` version across all the total LLC sizes, demonstrating its robustness.

6.3 Case Study

We present a case study in which CoPart is used to improve the fairness of batch workloads, which are collocated with a latency-critical (LC) workload on the same physical server. The main goal for workload consolidation is to significantly improve the server utilization by dynamically allocating resources to consolidated batch workloads while satisfying the service-level objective (SLO) for the LC workload and high fairness across the batch workloads.

We employ `memcached` from CloudSuite [10, 29] as the LC application and in-memory big-data applications (i.e., Word Count and Kmeans) from the BigDataBench benchmark suite [41] as the batch workloads. To satisfy the SLO for `memcached`, the 95th percentile latency should be below 1ms [33]. The amounts of the resources that are allocated to the LC and batch workloads are determined by a dynamic server resource manager, which is similar to the systems proposed in [15, 24]. In addition, the resources for the batch workloads are dynamically reallocated across the batch workloads by CoPart.

Figure 15 shows the runtime behavior of CoPart. Initially, the LC workload is applied with a low load (i.e., 75,000 requests per second), which allows for the batch workloads to be allocated more resources. CoPart allocates the LLC and memory bandwidth to the consolidated batch workloads in an application characteristic- and fairness-aware manner, achieving significantly higher fairness than the `EQ` version.

At $t = 99.4$, the load applied to the LC workload abruptly increases (i.e., 150,000 requests per second), which significantly reduces the amounts of the resources allocated to the batch workloads. CoPart robustly detects the change in the resources allocated to the batch workloads and effectively adapts to a new efficient system state to provide high fairness across the batch workloads even when the amounts of the resources allocated to the batch workloads are reduced. Note that CoPart temporarily exhibits slightly low fairness when the resources allocated to the batch workloads change due to the increase in the load applied to the LC workload. This is mainly because CoPart explores less efficient system states during the system state space exploration phase performed as a part of the readaptation process.

At $t = 299.4$, the load applied to the LC workload changes back to the low level. Again, CoPart robustly detects the change in the allocated resources and effectively adapts to an efficient system state to enhance the overall fairness.

6.4 Discussion

6.4.1 Overhead

We investigate the performance overhead of CoPart. Figure 16 shows the time spent for system state space exploration with various application counts. Each bar denotes the average time for system state space exploration across the workload mixes with the same application count.

First, the performance overhead of CoPart is small across the application counts. Specifically, the average system state space exploration time is 10.6, 11.8, 12.7, and 14.4 microseconds when the application count is 3, 4, 5, and 6. In comparison with the application execution time, the system state space exploration time accounts for 9.1×10^{-5}%, 1.0×10^{-4}%,

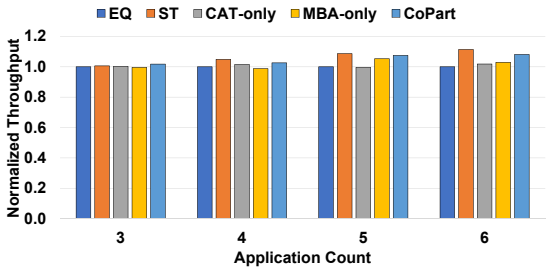


Figure 17. Performance results

1.1×10^{-4}%, and 1.5×10^{-4}% on average when the application count is 3, 4, 5, and 6. Since CoPart employs the efficient algorithm with low time complexity for system state space exploration, it incurs small performance overhead.

Second, the performance overhead of CoPart increases as the application increases. However, since the performance overhead of CoPart remains insignificant even with larger application counts, CoPart can be effectively used as a resource manager on commodity systems.

6.4.2 Performance

While the primary design goal of CoPart is to enhance the overall fairness, we also investigate its performance impact on the consolidated workloads. Figure 17 shows the throughput (i.e., geometric mean of the IPS data across the consolidated applications) results of CoPart and the other resource allocation policies with varying application counts. Each bar shows the average (i.e., geometric mean) throughput of each version across the workload mixes with the same application count, normalized to that of the EQ version.

CoPart achieves comparable or slightly higher performance than the other resource allocation policies across the application counts. While the main optimization metric of CoPart is the fairness, CoPart robustly analyzes the characteristics of the consolidated workloads and dynamically allocates the LLC and memory bandwidth in an application characteristic-aware manner, achieving high efficiency in terms of the overall performance.

7 Related Work

Prior work has extensively presented architectural and system support for LLC [7, 9, 17, 21, 34, 36, 37, 42, 44, 45] and memory bandwidth [16, 18, 19, 22, 23, 32, 39] partitioning. While insightful, the prior work investigates the techniques for partitioning either the LLC or memory bandwidth and/or evaluates the effectiveness of the proposed techniques based on simulation, which lacks the modeling of full hardware and system software stacks. Our work significantly differs in that it presents an in-depth characterization of LLC and memory bandwidth partitioning, proposes coordinated partitioning of LLC and memory bandwidth for fairness-aware workload consolidation, and designs, implements, and evaluates the proposed system using full hardware and software stacks.

An earlier study is closely related to our work [45]. This work has some similarity to ours in that it presents a simple and efficient LLC partitioning technique, which eliminates the need for complex mathematical models and extensive profiling to construct LLC miss curves. However, our work significantly differs as it addresses the broader problem of coordinated partitioning of LLC and memory bandwidth, demonstrates that the coordinated LLC and memory bandwidth partitioning problem can be formulated as one of the most extensively-studied problems in economics, and designs, implements, and evaluates the proposed system on a commodity server system.

Prior work has explored dynamic server resource management techniques for workload consolidation [15, 24, 28, 33, 40, 48]. The prior work lacks the capability of fully employing the hardware support for LLC and/or memory bandwidth partitioning, which are widely supported in recent commodity CPUs. Our work differs in that it presents an in-depth performance and fairness characterization of LLC and memory bandwidth partitioning and proposes a coordinated approach that simultaneously performs LLC and memory bandwidth partitioning on a commodity server system.

Prior work has investigated performance analysis and optimization techniques for non-uniform memory access (NUMA) and/or heterogeneous memory systems [8, 13, 14, 20, 25, 30, 31, 46]. The prior work mainly focuses on the design and implementation of NUMA- and/or heterogeneity-aware memory placement and migration policies to enhance the performance of the target applications. Our work significantly differs from these as it investigates the coordinated LLC and memory bandwidth partitioning technique to significantly improve the fairness of the consolidated workloads.

8 Conclusions

In this work, we present an in-depth characterization of LLC and memory bandwidth partitioning in terms of the performance and fairness of the consolidated applications. Based on the findings from the characterization studies, we present CoPart, coordinated partitioning of LLC and memory bandwidth for fairness-aware workload consolidation on commodity servers. CoPart dynamically analyzes the application characteristics and reallocates the LLC and memory bandwidth across the consolidated applications to significantly improve the overall fairness. Our quantitative evaluation demonstrates the effectiveness of CoPart in that it significantly improves the fairness of the consolidated applications (e.g., 57.3% higher fairness on average than the EQ version), delivers high fairness across various application and system configurations, and incurs small performance overhead.

Acknowledgments

We would like to thank our shepherd Ymir Vigfusson and the anonymous reviewers for their feedback. This research was partly supported by NRF (NRF-2016M3C4A7952587, NRF-2018R1C1B6005961) and IITP (No. 1711080972).

References

[1] Apache Spark™. https://spark.apache.org/.

[2] Intel 64 and IA-32 Architectures Software Developer's Manual.

[3] Intel® Resource Director Technology in Linux. https://01.org/intel-rdt-linux/blogs/fyu1/2017/resource-allocation-intel-resource-director-technology.

[4] Linux Containers. https://linuxcontainers.org/.

[5] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks — Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. ACM, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[6] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT '08)*. ACM, New York, NY, USA, 72–81. https://doi.org/10.1145/1454115.1454128

[7] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. 2013. A Hardware Evaluation of Cache Partitioning to Improve Utilization and Energy-efficiency While Preserving Responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. ACM, New York, NY, USA, 308–319. https://doi.org/10.1145/2485922.2485949

[8] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 381–394. https://doi.org/10.1145/2451116.2451157

[9] N. El-Sayed, A. Mukkara, P. A. Tsai, H. Kasture, X. Ma, and D. Sanchez. 2018. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 104–117. https://doi.org/10.1109/HPCA.2018.00019

[10] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/2150976.2150982

[11] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux J.* 2004, 124 (Aug. 2004), 5–. http://dl.acm.org/citation.cfm?id=1012889.1012894

[12] D. Gale and L. S. Shapley. 1962. College Admissions and the Stability of Marriage. *The American Mathematical Monthly* 69, 1 (Jan. 1962), 9–15. https://doi.org/10.2307/2312726

[13] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 231–242. http://dl.acm.org/citation.cfm?id=2643634.2643659

[14] Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 661–673. https://doi.org/10.1145/2694344.2694361

[15] M. Han, S. Yu, and W. Baek. 2018. Secure and Dynamic Core and Cache Partitioning for Safe and Efficient Server Consolidation. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 311–320. https://doi.org/10.1109/CCGRID.2018.00046

[16] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. 2009. Rate-based QoS Techniques for Cache/Memory in CMP Platforms. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM, New York, NY, USA, 479–488. https://doi.org/10.1145/1542275.1542342

[17] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. 2016. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 657–668. https://doi.org/10.1109/HPCA.2016.7446102

[18] D. R. Hower, H. W. Cain, and C. A. Waldspurger. 2017. PABST: Proportionally Allocated Bandwidth at the Source and Target. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 505–516. https://doi.org/10.1109/HPCA.2017.33

[19] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. 2007. QoS Policies and Architecture for Cache/Memory in CMP Platforms. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/1254882.1254886

[20] Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 277–289. http://dl.acm.org/citation.cfm?id=2813767.2813788

[21] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. 367–378. https://doi.org/10.1109/HPCA.2008.4658653

[22] F. Liu, X. Jiang, and Y. Solihin. 2010. Understanding how off-chip memory bandwidth partitioning in Chip Multiprocessors affects system performance. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. 1–12. https://doi.org/10.1109/HPCA.2010.5416655

[23] Fang Liu and Yan Solihin. 2011. Studying the Impact of Hardware Prefetching and Bandwidth Partitioning in Chip-multiprocessors. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '11)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1993744.1993749

[24] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture (ISCA '15)*. ACM, New York, NY, USA, 450–462. https://doi.org/10.1145/2749469.2749475

[25] Z. Majo and T.R. Gross. 2013. (Mis)understanding the NUMA memory system performance of multithreaded workloads. In *Workload Characterization (IISWC), 2013 IEEE International Symposium on*. 11–22. https://doi.org/10.1109/IISWC.2013.6704666

[26] John D. McCalpin. 1995. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), 19–25.

[27] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A Portable Interface to Hardware Performance Counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference (HPCMP '99)*. 7–10.

[28] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell. 2017. Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads. In *2017 IEEE International Symposium on High Performance Computer*

Architecture (HPCA). 409–420. https://doi.org/10.1109/HPCA.2017.13

[29] T. Palit, Yongming Shen, and M. Ferdman. 2016. Demystifying cloud benchmarking. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 122–132. https://doi.org/10.1109/ISPASS.2016.7482080

[30] J. Park and W. Baek. 2018. Quantifying the Performance and Energy-Efficiency Impact of Hardware Transactional Memory on Scientific Applications on Large-Scale NUMA Systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 804–813. https://doi.org/10.1109/IPDPS.2018.00090

[31] J. Park, M. Han, and W. Baek. 2016. Quantifying the performance impact of large pages on in-memory big-data workloads. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581281

[32] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. 2018. Hypart: A Hybrid Technique for Practical Memory Bandwidth Partitioning on Commodity Servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT '18)*. ACM, New York, NY, USA, 5:1–5:14. https://doi.org/10.1145/3243176.3243211

[33] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mossé, J. Mars, and L. Tang. 2015. Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 246–258. https://doi.org/10.1109/HPCA.2015.7056037

[34] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 423–432. https://doi.org/10.1109/MICRO.2006.49

[35] Alvin E Roth and Elliott Peranson. 1999. *The Redesign of the Matching Market for American Physicians: Some Engineering Aspects of Economic Design*. Working Paper 6963. National Bureau of Economic Research. https://doi.org/10.3386/w6963

[36] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: Scalable and Efficient Fine-grain Cache Partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA '11)*. ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/2000064.2000073

[37] V. Selfa, J. Sahuquillo, L. Eeckhout, S. Petit, and M. E. Gómez. 2017. Application Clustering Policies to Address System Fairness with Intel's Cache Allocation Technology. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 194–205. https://doi.org/10.1109/PACT.2017.19

[38] S. Seo, G. Jo, and J. Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*. 137–148. https://doi.org/10.1109/IISWC.2011.6114174

[39] Kshitij Sudan, Sadagopan Srinivasan, Rajeev Balasubramonian, and Ravi Iyer. 2012. Optimizing Datacenter Power with Memory System Levers for Guaranteed Quality-of-service. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/2370816.2370834

[40] Amin Tootoonchian, Aurojit Panda, Chang Lan, Melvin Walls, Katerina Argyraki, Sylvia Ratnasamy, and Scott Shenker. 2018. ResQ: Enabling SLOs in Network Function Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 283–297. https://www.usenix.org/conference/nsdi18/presentation/tootoonchian

[41] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 488–499. https://doi.org/10.1109/HPCA.2014.6835958

[42] X. Wang, S. Chen, J. Setter, and J. F. Martínez. 2017. SWAP: Effective Fine-Grain Management of Shared Last-Level Caches with Minimum Hardware Support. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 121–132. https://doi.org/10.1109/HPCA.2017.65

[43] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture (ISCA '95)*. ACM, New York, NY, USA, 24–36. https://doi.org/10.1145/223982.223990

[44] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, 13:1–13:15. https://doi.org/10.1145/3190508.3190511

[45] Cong Xu, Karthick Rajamani, Alexandre Ferreira, Wesley Felter, Juan Rubio, and Yang Li. 2018. dCat: Dynamic Cache Management for Efficient, Performance-sensitive Infrastructure-as-a-service. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, 14:1–14:13. https://doi.org/10.1145/3190508.3190555

[46] Seongdae Yu, Seongbeom Park, and Woongki Baek. 2017. Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS '17)*. ACM, New York, NY, USA, 18:1–18:10. https://doi.org/10.1145/3079079.3079092

[47] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=2228298.2228301

[48] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for Latency-Critical Tasks on Shared Multicore Systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 33–47. https://doi.org/10.1145/2872362.2872394