The Dissertation Committee for Haishan Zhu
certifies that this is the approved version of the following dissertation:

QoS-Aware Mechanisms for Improving Cost-Efficiency of Datacenters

Committee:

Mattan Erez, Supervisor

Jichuan Chang

Keshav Pingali

Gustavo de Veciana

Mohit Tiwari

QoS-Aware Mechanisms for Improving Cost-Efficiency of Datacenters

by

Haishan Zhu

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2018

Dedicated to my wife, Cathy Chen

and my parents, Zhiliang Zhu and Meifang Cui

Acknowledgments

First and foremost I want to thank my adviser Mattan Erez. This dissertation would not have been possible without his mentorship throughout my PhD career. He provided the much needed rigorous and incisive technical critiques on each and every piece of my work. His wide research interests and fruitful work have also inspired me to keep expanding my knowledge and sharpening my skills. Perhaps more importantly, Mattan gave me the precious intellectual freedom to pursue projects that I am interested in, to take inevitable detours in research work, and to learn from those experiences. When I encountered challenges and obstacles in numerous occasions, Mattan has always provided me with patient guidance and selfless support. I am deeply grateful to have him as my adviser and I am certain that I will continue to be inspired by his teaching even beyond research work.

I am also privileged to have collaborated with many outstanding computer architects at Google Platform Team. I would like to especially thank Jichuan Chang, David Lo, Liqun Cheng, Rama Govindaraju, and Parthasarathy Ranganathan for working with me for two consecutive summers on challenging and impactful projects. Thanks to their knowledge, experience, and resourcefulness, I was able to grow out of an inexperienced graduate student and gain insights in the state-of-art datacenter architectures. My collaboration with

them also played a fundamental role to the constitution of this dissertation.

I would like to give special thanks to my mentors and professors at Michigan: Jason Clemons, Todd Austin, Mark Brehob, Peter Chen, and Thomas Wenisch. While I only spent two years there, I learned a lot about computer architecture from either taking their classes or working on research projects with them. Their diligent teaching and research work led me to continue to explore this field in graduate school.

I am glad to get to know many other graduate students during my PhD. I want to thank Yuhao Zhu, Jingwen Leng, Jungrae Kim, and Minsoo Rhu for selflessly sharing their experience and knowledge with me and inspiring me to keep improving my work. I was also fortunate to receive much academic and career advice from Ikhwan Lee, Milad Hashemi, Min Kyu Jeong, Doe Hyun Yoon, and Dan Zhang. I also thank all the other members of our research group: Jinsuk Chung, Mike Sullivan, Song Zhang, Dong Wan Kim, Seong-Lyong Gong, Tianhao Zheng, Derong Liu, Kyushick Lee, Nick Kelly, Esha Choukse, Benjamin Cho, Sangkug Lym, Yongkee Kwon, Chun-Kai Chang, Majid Jalili, and Wenqi Yin. And I am grateful for the friendship with many others: Hao Xin, Rohit Pandey, Andrea Lottarini, Junwhan Ahn, Grant Ayers, Yuhuan Du, Zhuoran Zhao, Siavash Kamali, Ben Lin, Stephen Pruett, Ali Fakhrzadehgan, and Matt Halpern. It has been much fun discussing work, drinking coffee, having pizza, and occasionally slacking off together.

Last but not the least, I can never overstate how much I appreciate my family. I am grateful to have my incredible wife Cathy Chen with me through

all the adventures and endeavors during graduate school. Her love and caring made it easy for me to keep my eyes on the target. I deeply thank my parents, Zhiliang Zhu and Meifang Cui, for their unconditional love and support while I pursue my goals thousands of miles away from them. My love for my family is what gives meaning to working on this dissertation and pushing through all the obstacles.

Haishan Zhu

February 2018, Austin, TX

QoS-Aware Mechanisms for Improving Cost-Efficiency of Datacenters

Publication No. _____

Haishan Zhu, Ph.D.
The University of Texas at Austin, 2018

Supervisor: Mattan Erez

Warehouse Scale Computers (WSCs) promise high cost-efficiency by amortizing power, cooling, and management overheads. WSCs today host a large variety of jobs with two broad performance requirements categories: latency-critical (LC) and best-effort (BE). Ideally, to fully utilize all hardware resources, WSC operators can simply fill all the nodes with computing jobs. Unfortunately, because colocated jobs contend for shared resources, systems with high loads often experience performance degradation, which negatively impacts the Quality of Service (QoS) for LC jobs. In fact, service providers usually over-provision resources to avoid any interference with LC jobs, leading to significant resource inefficiencies. In this dissertation, I explore opportunities across different system-abstraction layers to improve the cost-efficiency of dataceters by increasing resource utilization of WSCs with little or no impact on the performance of LC jobs. The dissertation has three main components.

First, I explore opportunities to improve the throughput of multicore systems by reducing the performance variation of LC jobs. The main insight is that by reshaping the latency distribution curve, performance headroom of LC jobs can be effectively converted to improved BE throughput. I develop, implement, and evaluate a runtime system that achieves this goal with existing hardware. I leverage the cache partitioning, per-core frequency scaling, and thread masking of server processors. Evaluation results show the proposed solution enables 30% higher system throughput compared to solutions proposed in prior works while maintaining at least as good QoS for LC jobs.

Second, I study resource contention in near-future heterogeneous memory architectures (HMA). This study is motivated by recent developments in non-volatile memory (NVM) technologies, which enable higher storage density at the cost of same performance. To understand the performance and QoS impact of HMAs, I design and implement a performance emulator in the Linux kernel that runs unmodified workloads with high accuracy, low overhead, and complete transparency. I further propose and evaluate multiple data and resource management QoS mechanisms, such as locality-aware page admission, occupancy management, and write buffer jailing.

Third, I focus on accelerated machine learning (ML) systems. By profiling the performance of production workloads and accelerators, I show that accelerated ML tasks are highly sensitive to main memory interference due to fine-grained interaction between CPU and accelerator tasks. As a result, memory resource contention can significantly decreases the performance and

efficiency gains of accelerators. I propose a runtime system that leverages existing hardware capabilities and show 17% higher system efficiency compared to previous approaches. This study further exposes opportunities for future processor architectures.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Billions of dollars are invested every year in building warehouse scale computers (WSCs). By amortizing power, cooling, and management overheads, WSCs promise significantly higher cost-efficiencies compared to private datacenters, and attract applications with a large range of performance characteristics and requirements. One important category of WSC workloads includes a combination of latency-critical and latency-noncritical tasks. Ideally, noncritical tasks are used to "backfill" compute resources to fully utilize the WSC. Unfortunately, this is often difficult to achieve while still maintaining the performance goals of the latency-critical tasks because their performance degrades from resource interference. In fact, hardware is often intentionally over-provisioned to ensure quality-of-service (QoS) goals for latency-critical tasks, and the resulting under-utilization translates into huge wastes of system capacity and capital investment. As a result, performance interference bottlenecks system utilization and causes significant loss in cost-efficiencies of datacenters. This dissertation focuses on mitigating the fundamental conflicts between stringent QoS requirements for latency-critical tasks and inflated in-

frastructure Total Cost of Ownership (TCO)[1].

1.1 Warehouse Scale Computers and Workloads

Warehouse Scale Computers (WSCs) power the Internet as we know it today [15]. With the increasing adoption and deployment of machine learning services, WSC operators continue to evolve their infrastructure and workloads to maintain competitive service cost-efficiency.

From the hardware infrastructure perspective, a WSC usually consists of thousands of computing nodes that are interconnected through a network subsystem. While WSC operators update hardware over time, the infrastructure is relatively homogeneous in that there are relatively few configurations in a given WSC (instead of hundreds of configurations in the consumer electronics market). The small number of configurations significantly reduces overhead in managing the system and allows the WSC operator to easily test and deploy new services across the WSC.

From the workload perspective, WSC applications typically rely on distributed storage systems to process huge amounts of data. The scale of these services dictates that applications have to leverage parallelisms over multiple

[1]Content in this chapter is published in the following article: 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. Authorship contributions: conception and design of study: Haishan Zhu and Mattan Erez; data acquisition and analysis: Haishan Zhu and Mattan Erez; drafting manuscript: Haishan Zhu and Mattan Erez; critical manuscript revision: Mattan Erez.

levels of system abstraction (computation, storage, and network). Another interesting characteristic of the workload is the increasing variety of applications that run on WSCs [97]. A wide range of services and applications are attracted to WSCs because of immediate availability of large-scale computation resources and competitive cost-efficiency. For example, with constant advances in data analytic techniques (e.g., machine learning), more entities that handle large amounts of data depend on WSCs to extract information and build services.

Finally, the recent advancement in machine learning applications leads to increasingly wide adoption of accelerators in WSC environments. GPUs have been widely used for ML training and inference [5, 94, 33, 59]. Other solutions that are application specific (e.g., FPGAs [143] and ASICs [95]) can achieve even higher efficiency and performance and have also been deployed in production. Accelerated nodes can further scale-out to accommodate larger ML models. As the heavy computation is carried out by the accelerators, CPUs in these systems are often responsible for various supporting tasks to sustain the high accelerator performance. This new computation paradigm brings new challenges in resource management and CPU design.

1.2 WSC Workload Management

Workloads in the datacenter can be broadly classified into two categories: The first includes tasks that are best-effort (BE), for which throughput is the primary concern and that can be freely scheduled in the background

when resources are available. The second category includes latency-critical (LC) user-facing tasks, which often have a short execution time with strict completion time targets to achieve certain quality-of-service (QoS) goals. This LC category can be further divided into three application types:

1. Applications like key-value stores usually have extremely short delay targets (a few milliseconds) in order to achieve good user experience. [16, 78, 119] Caching services that built on top of these applications often rely on sharding of data to improve system capacity and throughput, which can cause high sensitivity to tail latency on each node for large fan-out services [40].

2. Offloaded tasks from mobile devices is an emerging class of workloads. Such tasks are computationally intensive and have relatively long execution times (e.g., hundreds of milliseconds) [114, 72, 182]. Examples of this class of workloads include online video processing, online stream data analysis, and recognition tasks.

3. Accelerated machine learning workloads, which include both training and inference. These tasks have high priority because accelerators are often capable of higher computational throughput and efficiency compared to CPUs, and customers are usually charged more for using these resources [66, 8]. Note that while training applications typically have long execution time, latency is still critical for ML model design iterations and service time to market.

LC tasks on their own cannot be used to fully utilize a node. It is often undesirable to collocate LC tasks because of aspects relating to how data is sharded across cloud servers and sizes of application state [15, 119, 40]. WSC applications often scale out to thousands of nodes, each of which works on a portion of the total dataset to exploit data-level parallelism. Ideally, best-effort tasks are used to "backfill" compute resources to fully utilize all available resources in the WSC, thus providing optimal cost-efficiency for both the datacenter operators and WSC clients. Unfortunately, this is often difficult to achieve while still maintaining the performance goals of the latency-critical tasks because their performance degrades from resource interference, as we discuss in more detail next.

1.3 Performance Interference and Utilization Bottleneck

"Backfilling" BE tasks to better utilize nodes is not always possible because of the detrimental impact it has on LC performance. Specifically, collocating BE and LC tasks can cause performance degradation because of the contention between LC and BE tasks for limited shared resources, such as CPU time, cache and main memory occupancy, and I/O bandwidth. Due to the scale-out design of WSC applications, the response time of each request is ultimately determined by the delay of the slowest node. As a result, even a small increase in the probability of a QoS violation at the node level can be significantly amplified at the service level [40]. One of the most common techniques to minimize QoS violations it to over-provision hardware resource

to guarantee quality-of-service (QoS) for latency-critical tasks. The resulting under-utilization translates into wasting system capacity and capital investment. As a result, performance interference bottlenecks system utilization and causes significant loss in the cost-efficiency of datacenters.

1.4 WSC Workload Statistical Behavior

WSC workloads exhibit various statistical behaviors in terms of workload intensity and resource requirements. As a result, it is often difficult for programmers and WSC operators to accurately estimate and allocate resources for production workloads. Resource allocation is often conservative to guarantee the performance of high-priority latency critical tasks, leading to low utilization and loss of efficiency.

There are two main factors that contribute to the statistical behavior of WSC workloads. First, user behavior is non-deterministic, which causes the workload to vary over time. Specifically, user behavior varies on two time scales. On the coarse time scale, user behavior varies in a diurnal pattern [119], in addition to spikes events and holidays. On the fine time scale, user behavior also oscillates over hours and even seconds [167]. Second, WSC application behavior is non-deterministic. For example, master-slave database architectures are commonly used in data hosting. The slaves are mostly responsible for backing up data from the master and exhibit low load. However, in cases when the master fails, a slave will be promoted and its load will increase rapidly [138]. Other factors that contribute to the variability of WSC workloads include sys-

tem daemons, queueing at various layers, and maintenance activity [40, 122]. The statistical behavior of WSC workloads places unique challenges and opportunities in WSC resource provisioning. In this dissertation, I exploit this observation to improve utilization of hardware resources.

1.5 Thesis Statement

Internet service providers need to further improve cost-efficiency of Warehouse Scale Computing beyond simple resource aggregation. However, increasing hardware utilization without managing resource interference can cause performance and quality-of-service degradation for latency-critical tasks. As a result, system operators often over-provision hardware resources, causing significant loss of total cost of ownership. My thesis is that the resource utilization, hence cost efficiency, of a datacenter can be drastically improved with a set of cross-layer techniques, which leverage the statistical behavior of latency-critical tasks and maximize total system throughput without compromising the quality-of-service for latency-critical tasks.

1.6 Contributions

In this dissertation, I study the performance and QoS issues caused by task colocation in the WSC environment. I choose three representative workload and system architecture combinations and tailor solutions based on application requirements and architecture performance isolation capabilities. These studies show that significant opportunities exist in increasing WSC re-

source utilization without sacrificing QoS for latency-critical tasks. I propose a set of runtime solutions to take advantage of these opportunities by measuring workload performance characteristics and leveraging low-level hardware capabilities. I evaluate these solutions on existing hardware when the capabilities are available and use simulators when they are not. The main contributions of this dissertation are summarized as follows:

1. To improve cost-efficiency of multicore systems that target emerging offloaded workloads, I propose to leverage the statistical behavior of LC workloads to improve total system throughput. This is achieved by isolating tasks to limit performance interference and guarantee QoS for LC tasks, while converting latency headroom of the LC tasks to improve total system throughput. I demonstrate the benefits of this approach with Dirigent, a lightweight resource management runtime system that reduces the execution time variation of LC tasks to improve the throughput of BE tasks. Dirigent coordinates all contending processes with the knowledge of expected completion times and deadlines. Dirigent is thus able to improve background-task performance by enabling foreground tasks to yield resources when they are expected to finish faster than their required latency.

2. To improve the cost-efficiency of the memory subsystem, I identify an opportunity to replace a significant portion of DRAM with slower but cheaper Non-Volatile Memory (NVM). To quantify the performance im-

pact of such a Heterogeneous Memory Architecture (HMA), I develop and build ONSim, an OS-level NVM simulator. ONSim integrates within the Linux kernel and virtualize multiple PTE bits to dynamically collect and act on runtime memory access information. ONSim is transparent to userspace applications. ONSim enables users to easily modify HMA parameters, page migration policies, and various resource-allocation priorities without changing the underlying functional kernel code. I perform rigorous evaluation on the capability of ONSim, and show that it has both low overhead and high fidelity for its target workloads.

3. I study the performance of memcached, a popular key-value store application, on systems with heterogeneous memory using ONSim. I analyzed the performance of memcached in both standalone and colocated execution. I then propose three techniques to reduce NVM performance impact by exploiting the performance characteristics of the target workloads, such as the strong locality in memory accesses, low write-to-read ratio, and load variation over time. I evaluate these techniques with ONSim and show that these techniques can maintain high performance and QoS for the popular and important key-value store application on HMA systems.

4. To improve the cost-efficiency of accelerated systems for machine learning (ML) applications, I first profile a set of production ML workloads on various accelerated systems. Through a detailed sensitivity study, I

show that the performance of these workloads can be significantly impacted by host memory bandwidth pressure, which prevents colocation of low priority tasks and leads to wasted resources. To tackle this problem, I propose Kelp, a lightweight runtime system that leverages existing hardware features to mitigate performance interference. Evaluation results show that Kelp is effective in isolating accelerator performance from memory bandwidth interference while sustaining high system throughput. This study also shows that high-performance accelerators pose new system architecture challenges, and motivates the need for fast and low-overhead fine-grained memory performance isolation mechanisms

1.7 Dissertation Organization

The rest of my dissertation is organized as follows. Chapter 2 discusses mechanisms that improve cost-efficiency of multicore systems by leveraging latency headroom to achieve better system performance. Chapter 3 details the design and evaluation of ONSim, which simulates HMA systems with high fidelity and low overhead. I also describe the work on performance and QoS of HMA systems in this chapter. Chapter 4 presents the study on performance interference in accelerated ML systems and discusses mechanisms to mitigate it on real systems. Finally, Chapter 5 discusses future work and concludes this dissertation.

Chapter 2

Improving QoS and Utilization on Multicore Sysmtes

Latency-critical applications suffer from both average performance degradation and reduced completion time predictability when collocated with batch tasks. Such variation forces the system to overprovision resources to ensure Quality of Service (QoS) for latency-critical tasks, degrading overall system throughput. We explore the causes of this variation and exploit the opportunities of mitigating variation directly to simultaneously improve both QoS and utilization. We develop, implement, and evaluate Dirigent, a lightweight performance-management runtime system that accurately controls the QoS of latency-critical applications at fine time scales, leveraging existing architecture mechanisms. We evaluate Dirigent on a real machine and show that it is significantly more effective than configurations representative of prior schemes[1].

[1]Content in this chapter is published in the following article: 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. Authorship contributions: conception and design of study: Haishan Zhu and Mattan Erez; data acquisition and analysis: Haishan Zhu and Mattan Erez; drafting manuscript: Haishan Zhu and Mattan Erez; critical manuscript revision: Mattan Erez.

2.1 Background

2.1.1 Target Workloads and Metrics

Based on prior publications, we classify cloud workloads into three categories [15, 109, 72, 27, 31, 114]. The first includes tasks that are not user-facing, for which throughput is the primary concern, and that can be freely scheduled in the background when resources are available. The second class includes short latency-critical user-facing tasks, such as responding to web search requests and content caching. This class of workloads is characterized by short deadlines in the order of tens of milliseconds. The third is an emerging class of workloads that correspond to offloading of work from user devices to the cloud. Examples include recognition tasks. Such tasks are user facing and latency critical, yet are also computationally intensive and have relatively long execution times. compared to some other user-facing cloud These tasks can take hundreds of milliseconds or more to finish and therefore can benefit from being offloaded to the cloud [114, 72, 27].

Both the second and third classes are typically user-generated tasks, of which those arising from sophisticated data and sensor processing applications often belong to the third class. Both classes pose challenges for efficient cloud resource usage. In particular, their performance is expressed both in terms of throughput (number of tasks processed per unit time) and in terms of strict latency constraints where only a small fraction of tasks can violate the constraints without severe penalties. Two common and useful measures corresponding to these performance goals are *average execution time* for through-

put and *95-percentile execution time* (or other percentile goals) for latency constraints. Note that in evaluating Dirigent in this paper, we use workloads comprised of tasks from the first and the third categories.

2.1.2 Contention Control Mechanisms

Dirigent relies on existing mechanisms to manage interference and provide good QoS for the LC jobs. Dirigent uses per-core DVFS and cache partitioning, which we summarize below. We also discuss additional hardware mechanisms that have recently been proposed.

Per-Core dynamic voltage and frequency scaling (DVFS) is a common mechanism for controlling performance and improving processor energy efficiency [130, 104, 86]. Per-core frequency management enables performance adjustments at fine time scales [147, 78], and it can also be used as a throttling mechanism to manage contention and resource usage [74, 80].

Cache partitioning is another well-studied mechanism that provides performance isolation of processes that are collocated and that is used to limit variation and contention [145, 115, 100, 150, 30, 35, 81]. Prior work established the effectiveness of cache partitioning as a QoS mechanism and it has recently been implemented in commercial processors. However, because of the large capacity of the last level cache, changes to cache partitions take significant time to have an impact on execution, an effect termed *cache inertia* [100]. Thus, cache partitioning is effective at relatively long time scales.

While we do not currently use these mechanisms in Dirigent as they

are not yet available in commercial processors, there is also a large body of work on QoS mechanisms for managing memory bandwidth and latency resources. Yun et al. studied the performance benefits of memory bandwidth reservation for latency-sensitive applications [173]. Mutlu et al. show different levels of QoS goals and performance benefits that can be achieved by making memory scheduling QoS aware [133, 134, 105, 135]. Usui et al. presented QoS aware memory scheduler that handles different priority levels in heterogeneous systems [164]. In other related work, Ebrahimi et al. proposed source throttling, which is a hardware-based mechanism controls the rate at which cores generate requests to shared memory system [51]. Jeong et al. studied the row-buffer locality interference in multicore processors [92]. Zhou et al. proposed an architecture that allows fine-grain micro-architecture resource partitioning among threads [180]. Ma et al. designed a mechanism to control queueing delay of requests in different architecture structures [121]. While other contention sources exist in warehouse scale datacenters, there is a large body of related work that show how intelligent task schedulers can identify and avoid such resource conflicts [96, 110, 112, 46, 47].

2.2 Dirigent Principles

An important insight that Dirigent leverages is that minimizing variation can simultaneously meet the performance targets of latency-critical tasks and improve system utilization. Large variations in the completion time of LC tasks cause inefficiencies in the shared system because they lead to over-

provisioned resources for LC tasks and system underutilization for latency targets to be met. To understand these inefficiencies, consider an LC task that can meet its throughput and latency targets when running alone, but suffers from large variation in execution time when under contention.

Figure 2.1 shows an example of the execution time probability density function of such a program, where the blue curve represents standalone execution, the red curve shows the behavior under contention, and an ideal curve is shown in green. In standalone execution, LC tasks often complete significantly ahead of the deadline and the throughput achieved (average execution time) is higher than required. As a result hardware is not fully utilized because there is large performance headroom in most LC executions that could be used for other tasks (shown as the highlighted region). However, when a BE task is introduced, too many deadlines are missed despite the average throughput target being maintained.

Previous work focused on keeping core occupancy high while meeting latency goals [126, 119, 176]. However, by explicitly addressing task-to-task variation at fine time scales within a single task, the ideal (green curve) can be achieved. In this ideal case, both throughput and latency targets are met precisely and core and frequency resources are maximally utilized. Dirigent achieves this ideal curve through unique fine time scale monitoring and control mechanisms.

Furthermore, high variation reduces resource utilization because of task scheduling policies designed for meeting LC task latency constraints. Consider

Figure 2.1: Example LC completion time probability density functions when run alone, under contention, and in an ideal scenario; the shaded region points to underutilized resources when run alone.

a common reservation-based scheduling policy that ensures on-time completion of LC tasks by reserving sufficient resources to guarantee a 95% latency target [10]. ¡Figd:fig:scheduler shows how this scheduler operates on two different sets of tasks: tasks of type A, which have high execution time variance, and type B which have low variance. With high variance, the scheduler reserves too much time for the first task (shown in green) because allocation is forced to expand due to the long tail of execution time probability distribution curve, leading to poor system utilization. This does not happen with the low-variance tasks. Significant prior work on scheduling has shown that scheduling tasks with deadlines can be done more aggressively and with higher system resource utilization when variance is low [154].

Dirigent controls fine-grained scheduling and frequency to maximize utilization while meeting QoS goals. In this way overall utility per unit energy

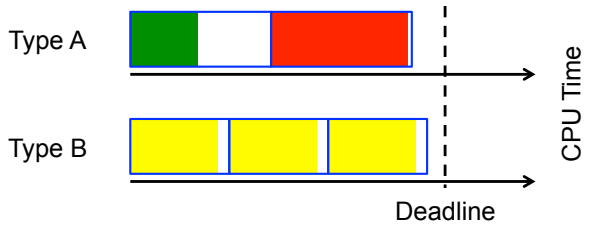


Figure 2.2: Reservation-based scheduler efficiency with two different task types: type A with high execution time variance and type B with low variance.

is maximized. This is in contrast to a line of prior work that reduces the impact of variation by rapidly adjusting processor clock frequency [118, 78, 165, 99, 117, 25]. Matching frequency to LC compute needs reduces processor energy consumption, but falls short of maximizing efficiency because the processor itself consumes just $25\% - 35\%$ of total system power [127, 118, 54].

2.3 Dirigent Design and Implementation

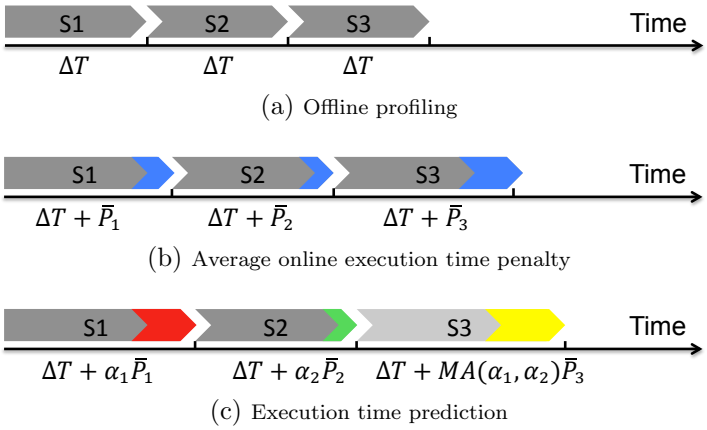
Dirigent is composed of three main components: profiler, predictor, and controller. The profiler examines and records the execution of an LC applications offline and in isolation. The profiling information is then used online to predict the expected execution time of the LC application. The controller then partitions resources and throttles tasks to minimize the execution time variation of LC tasks while providing as much resources as possible to BE tasks, thus achieving the goal of simultaneously meeting latency-critical requirements and allowing high processor utilization. The rest of this section discusses in detail the design and initial implementation of Dirigent. Again, we point out that Dirigent is unique in the fine time scales on which it operates.

We implement Dirigent in C++ and evaluate it using a 6-core Intel Xeon E5 2618L-v3 processor, which supports per-core frequency settings and cache partitioning [81].

2.3.1 Offline Execution Profiler

Dirigent profiles the execution of an LC application when running alone offline. The profiler records progress information that is then used to make accurate online predictions of completion time under contention while a task is still executing. The Dirigent profiler periodically samples the execution progress of the LC task being profiled by measuring and recording a series of (*time*, *progress*) pairs when the LC is running alone without any contention (see Figure 2.3a). The LC program in the example of Figure 2.3a has 3 profiled segments and takes $3\Delta T$ time units to execute, where ΔT is the sampling period. We measure *progress* by counting the number of retired instructions using the processor's model-specific performance counter monitors [86], but more abstract metrics can also be used. Note that progress can significantly differ between segments even though the sampling frequency is constant. This is because progress depends on the instruction mix, data access pattern, data set size, and other factors.

Dirigent requires minimal profiling. In our current implementation, the profiler requires no extra hardware as both execution time and instruction count are readily available in most architectures today. Dirigent uses performance counters and `sleep` method for periodic sampling with negligible



(a) Offline profiling

(b) Average online execution time penalty

(c) Execution time prediction

Figure 2.3: Execution time predictor example.

impact on the running application. Although performance overhead is not a great concern in offline profiling, this low overhead ensures the accuracy of these measurements. Furthermore, note that while our current implementation of Dirigent relies on offline profiling, it is possible to perform online profiling instead, which requires the system to run the LC task a few times while all other tasks are paused to record a stable profiling record.

2.3.2 Execution Time Predictor

To predict LC completion time and control resources, Dirigent periodically samples progress during contended online execution. Dirigent predicts LC completion time by tracking actual progress, comparing this progress to the profiled data, and computing a *time penalty* experienced by the LC process, which is then projected forward to completion as explained below. We pin the Dirigent runtime thread to a core that runs a BE task and again use the

`sleep` method to periodically interrupt the BE task and execute the Dirigent runtime task.

The time penalty of a specific segment is computed assuming a fixed rate of progress within each segment. The penalty is the difference between the expected time to make the amount of progress within the profiled segment at the rate of progress experienced in the online segment vs. the profiled time for this segment. This is summarized in Equation 2.1 where P_i is the penalty for the i^{th} segment and ΔT_i is the duration of that segment. Note that ΔT_i can be slightly different than ΔT in the real implementation because of factors such as errors in timers. We account for that difference in Dirigent to ensure the accuracy of the prediction. Also, we introduce the symbol α_i shown in Equation 2.1 as shorthand for the ratio of measured vs. expected progress rates for the i^{th} segment.

$$P_i = \frac{Profiled\_progress_i}{Measured\_progress_i}\Delta T_i - \Delta T_i = (\alpha_i - 1)\,\Delta T_i \tag{2.1}$$

Frequent samples increase the execution time prediction accuracy and the opportunities for performance management, but each prediction and control segment has overhead. We measured this overhead using a subset of our workload mixes. Results show the runtime overhead is minimal and each Dirigent invocation requires on average less than $100\mu s$ (including predictor and throttler). We therefore chose a sampling period $\Delta T = 5ms$ to balance the overhead and effectiveness of online prediction and control. This sampling

period provides 100 or more segments in all the LC applications we test with only negligible runtime overhead.

To increase prediction accuracy, Dirigent maintains an exponential moving average (with weight 0.2) of the penalty within each segment across multiple executions of the LC task, which we denote $\overline{P_i} = 0.2P_i + 0.8\overline{P_i}$. Figure 2.3b shows the average penalty in blue. Note that just like progress, the average penalty can differ significantly across segments. The moving average smooths occasional outlier executions. At a given point in time during a single LC task's execution, Dirigent's predictor uses the penalties observed so far for each segment, the total elapsed time from the start of the process, and the average penalties of the segments yet to execute to compute the expected execution time of the task. The formula is shown in Equation 2.2, where k is the segment corresponding to current time T, N are the total number of segments in the profiled execution, and $MA\left(\{\alpha_i\}_{i=1}^{k}\right)$ denotes an exponential moving average over the rate factors measured so far in the current execution. This moving average is used as the expected penalty scaling factor for the remainder of the current execution.

$$T_{est,k} = T + \sum_{i=k+1}^{N} \left(MA\left(\{\alpha_i\}_{i=1}^{k}\right) \overline{P_i} + \Delta T_i \right) \tag{2.2}$$

Figure 2.3c illustrates how the prediction calculation of Equation 2.2 is performed. When the example program finishes executing the second segment, the penalty scaling factor α_2 is much smaller than the factor α_1 of the first

segment. This difference arises from factors such as OS noise, phase changes of the BE applications, and context switches. The moving average across executions and of executions within the segment smooths out the difference of the scaling, which is then used as the predictive factor for the remaining segment.

In our experiments with Dirigent, we arbitrarily chose a weight of 0.2 for the exponential moving averages and a 5ms sampling interval. As we show in Section 2.4.2, the predictor is highly accurate with these parameters and is able to predict the expected execution time to within 2%, typically, across multiple applications and different levels of contention. We tested the sensitivity of Dirigent to weight factors in the range of $0.1 - 0.3$ and conclude that Dirigent is robust. We also evaluate Dirigent's sensitivity to sampling periods. We conclude that even 40 samples per execution of the LC task tested provide for accurate completion-time predictions. However, the low performance overhead ($< 100\mu s$ per invocation) enables high sampling frequency for accurate prediction and tight QoS control for LC tasks that widely differ in execution time.

2.3.3 Performance Controller

Dirigent monitors the performance of LC applications online and uses the predictor to determine whether these applications are progressing faster or slower than necessary to meet their latency goals. Recall that Dirigent does not strive to minimize the execution time of LC tasks, but rather to

minimize their execution time variation while meeting their latency targets. As a result, resources for the LC tasks are not over-provisioned. If a LC task is expected to complete before its target time, it is deprioritized and BE tasks can achieve higher throughput. On the other hand, if a LC task is lagging, Dirigent prioritizes resources toward that LC task and away from BE tasks.

In our current implementation of Dirigent, we allocate resource by controlling the frequency at which each core operates, by partitioning the last-level cache (LLC), and by pausing BE tasks when necessary. We chose these mechanisms from the possible ones described in Section 2.1.2 because they are both effective and available in current systems. We use frequency and task-pausing to control LC progress at fine time scales and cache partitioning at a coarser ones; with large caches, *cache inertia* means significant time passes before the impact of adjusting partitions takes effect [100].

Fine time scale control: The goal of the fine time scale controller is to quickly respond to changes in contention and LC task progress to ensure deadlines are met and minimize performance variance. The controller observes the execution time predictions and decides whether LC tasks can yield resources or whether BE tasks must be throttled and to what extent. A simplified version of the fine time scale controller policy for a single LC task is designed as follows.

At each decision point, the controller determines if the LC task is ahead or behind. If ahead, the controller will check the following three options in

order. First, if any best-effort jobs are paused, the control decision is to continue them. Second, if no tasks are paused but some are throttled, the decision is to speed up any throttled BE processes by one speed grade (using existing per-core DVFS mechanisms). Third, if all BE tasks are already running at their maximum frequency, the decision is to throttle the LC task frequency. Similarly, if the LC task is behind schedule, the decision is to speed up to maximum frequency. If it is already at maximum frequency, the decision is to immediately throttle the frequency of the BE tasks. If the BE tasks are already at the minimum frequency, the most intrusive active BE is paused; we define intrusiveness as the number of LLC load misses a task generates, which we obtain from existing performance counters. The throttler controls each core using the `CPUFreq Governor` of Linux [18].

While the Dirigent runtime is very lightweight, the impact of control decisions is not instantaneous. We therefore only make control decisions every some small number of prediction segments (5 in our experiments, arbitrarily). Furthermore, we only take control actions if the expected LC execution time is more than 2% ahead the target deadline and only pause BE tasks if the LC task is expected to complete more than 10% behind its deadline. We chose 2% because it corresponds to the typical error of the predictor and is thus a good safety margin that prevents prematurely slowing down or interfering with a LC task. We chose a larger threshold for pausing because its overhead is greater (again, the value of 10% was arbitrarily chosen within a reasonable value range; sensitivity studies reveal that Dirigent is not sensitive to this

choice).

The decision making process is slightly more complicated when there are multiple collocated LC processes along with BE tasks. Each LC task may exhibit different levels of performance degradation even when the interference level is the same for all of them. Furthermore, any action taken on BE tasks will impact all collocated LC tasks. As a result, when all LC tasks show the same performance tendency, we use the same policy described before for a single LC process. Otherwise, BE tasks are throttled based on the performance of the slowest concurrent LC task, and any other LC tasks that are expected to finish sooner than the deadline are throttled down individually.

Coarse time scale control: Dirigent uses cache partitioning for coarse-grain control over the expected execution time of LC tasks, specifically the Cache Allocation Technology recently introduced by Intel [81], which can be used to specify which cache ways may be used by each processor. Because of cache inertia the system's response time to partition changes is fairly slow when compared to the typical short durations of LC tasks. We therefore use statistics collected over multiple executions of a LC task to guide adjustments to cache partitioning. Specifically, our current implementation of Dirigent tracks three measures: (1) the correlation between a LC tasks' execution time and the LLC misses it generates (over multiple executions); (2) a history of the absolute number of LLC misses over executions; and (3) a history of the Dirigent decisions states over time. In our current implementation, we use the

history of 10 last executions to compute the measures above. We construct three heuristics to determine whether LC tasks benefit from greater isolation and more dedicated cache ways or whether BE tasks are allowed to utilize a greater portion of the LLC.

First, if there is strong correlation between the execution time of LC tasks and their LLC misses, it indicates that growing the LC partition is likely to improve LC performance. Therefore, if correlation is strong *and* LC tasks have recently missed deadlines, we increase isolation and add one LLC way to the LC partition (removing it from the list of ways utilized by BE tasks). We somewhat arbitrarily chose a correlation coefficient of 0.75 as the threshold determining strong correlation.

Second, Dirigent observes the LLC hit-rate history and if growing the LC partition does not lower LC tasks LLC misses, Dirigent shrinks the LC partition. This heuristic coupled with the coarse time scale and averaging performed prevents the LC partitions from continuously growing due to anomalous executions.

It is also possible for the correlation between misses and performance to not be strong yet for partitioning to still help: when LC performance is bottlenecked by high memory latency caused by contention from BE tasks. Because Dirigent's fine time scale controller throttles BE tasks when they heavily contend for resources, correlation would not detect the need for stricter partitioning of LC and BE tasks. Therefore, our *third heuristic* grows the LC partition when the controller history indicates that BE tasks are heavily

throttled and their utilization of core resources is low. The second heuristic then differentiates between scenarios where BE tasks should be throttled from those where partitioning is more beneficial by shrinking the LC partition back if hit-rate does not improve. We show the effectiveness of our method in Section 2.4.3.

2.4 Evaluation

We evaluate Dirigent on a real machine with a range of workloads representative of a wide range of LC and BE behaviors. We first introduce the evaluation infrastructure and workloads. We then discuss a set of experiments that demonstrate the accuracy of Dirigent's completion-time predictor, the effectiveness of the coarse time scale partitioning heuristic, Dirigent's performance benefits compared to a baseline configuration and configurations that roughly correspond to prior work, and the new tradeoff between BE task throughput and LC deadline target that Dirigent enables.

2.4.1 Workloads and Evaluation Infrastructure

System: We evaluate Dirigent on a 6-core Intel Xeon E5-2618L v3 server processor. The nominal per-core maximum frequency is 2GHz and 9 frequency steps are available for throttling ($1.2 - 2.0$GHz, though Dirigent uses just 5 equi-spaced frequencies). Turbo Boost is enabled in all experiments. The processor has a 15MB L3 LLC and supports Intel's Cache Allocation Technology, which enables us to partition the cache between LC and BE tasks to provide

Table 2.1: LC and BE Benchmarks

Type	Name	Description
LC	bodytrack	Body tracking of a person
	ferret	Content similarity search
	fluidanimate	Fluid dynamic for animation
	raytrace	Real-time raytracing
	streamcluster	Online clustering of an input stream
Single BE	bwaves	Simulation of blast waves in 3D
	PCA	Principal Component Analysis
	RS	Range Search
Rotate BE	namd	Biomolecular system simulation
	soplex	Linear program solver
	libquantum	Simulation of quantum computer
	lbm	Simulation of fluids with free surfaces

additional isolation. The system is configured with 4 2133MHz DDR4 channels and has a total of 16GiB. We run a Linux 3.13.0 kernel at runlevel S, which provides an environment with little OS interference and good facilities for implementing both the Dirigent predictor and controller, with an efficient `sleep` implementation and the built-in `CPUFreq Governer`, respectively. We pin all tasks to individual cores and Dirigent is pinned to a core that is shared with a BE task. We set the BE processes to have higher process niceness than the Dirigent runtime and LC processes to have the lowest niceness. We use the configuration above with no explicit resource management as the *baseline configuration*.

Workloads: Table 2.1 lists the benchmarks we use in the evaluation. We select the subset of PARSEC applications that represent latency-sensitive applications as LC tasks [17]. We chose PARSEC as it is designed to represent

emerging and user-facing workloads of the type that are being offloaded to cloud systems. We use a single run of each benchmark with sim-medium inputs as a single LC task. As shown in Figure 2.4, these tasks span a range of completion times ($0.5 - 1.6$s) and LLC miss rates. The figure shows the behavior of the LC benchmarks both when running alone and under contention. For this figure, we use 1 LC for all LC tasks and 5 BE cores all running *bwaves*, which falls in the middle of contention range. The LC workloads are all fairly compute-intensive, making them good offload candidates. While they are compute-intensive, they still offer a range of sensitivity to interference from BE tasks both because of LLC and memory access contention. This can be seen by the different correlation levels between LLC miss rate increase and execution time degradation between the different benchmarks. To ensure accurate measurements of these tasks, execution time is measured inside the LC processes using PARSEC's Region of Interests (ROI) interface.

We use two kinds of BE workloads to represent different interference types: BE phase changes and context switches. We use three standalone BE workloads that exhibit strong phase change behavior: the scientific simulation *bwaves* from SPEC 2006 [73] and the machine learning applications Principal Component Analysis (*PCA*) and Range Search (*RS*) from MLPack [39]. All other benchmarks we examined did not provide strong phase behavior, at least with respect to impact on interference, and we omit them from the evaluation—such workloads do not pose significant challenges to the Dirigent predictor.

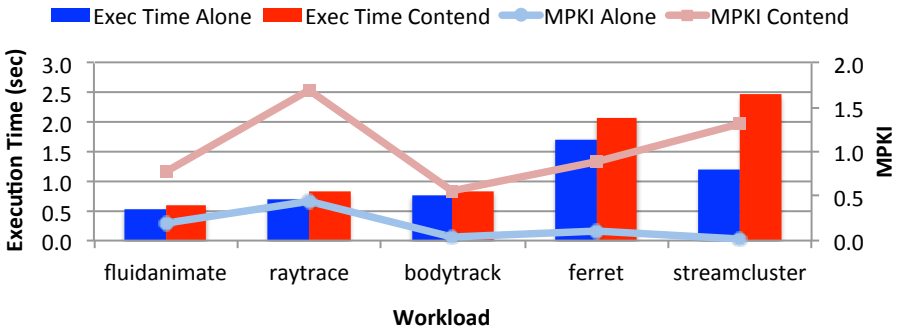


Figure 2.4: Overview of LC Workloads.

To mimic varying interference caused by context switches, we select four applications from SPEC 2006 that exhibit a range of memory intensiveness [73], though we arbitrarily chose between all benchmarks that exhibit similar intensity. We then form two-benchmark workloads and randomly switch between the two paired benchmarks each time a LC task completes. The pairs we use are (*lbm+namd*), (*lib+namd*), (*lbm+soplex*), and (*lib+soplex*). We refer these BE workloads as Rotate BE workloads. Figure 2.5 summarizes the different behaviors of the BE workloads while using a single core running *ferret* as a representative LC workload. The blue bars show the total number of L3 load misses per thousand LC instructions generated by all 6 cores. The red curve shows the fraction of misses generated by LC tasks, which can be interpreted as the ratio between LC task and total memory bandwidth consumption. As can be seen, the BE workloads cover a wide spectrum of behaviors and contention pressure. Similar trends are observed when mixing these BE tasks with other LC workloads.

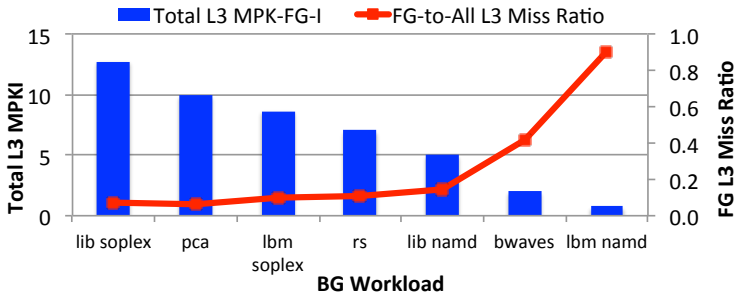


Figure 2.5: Overview of BE Workloads.

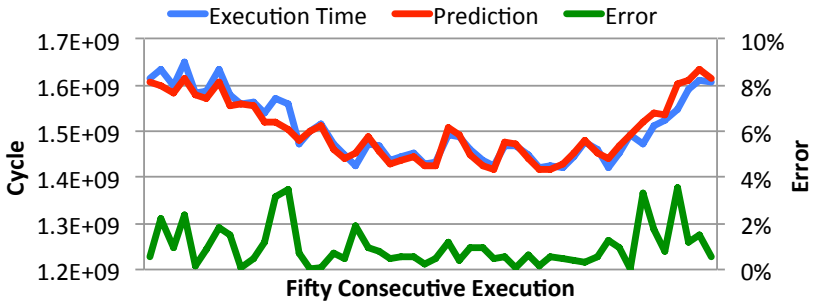


Figure 2.6: Prediction Trace for *Raytrace* with *RS*.

2.4.2 Predictor Accuracy

Since the throttling actions of Dirigent are guided by the execution time predictor, it is important to validate its accuracy. Figure 2.6 shows the execution time, prediction results, and prediction error for 50 consecutive executions of *raytrace* and *RS* workload in the baseline configuration (no explicit resource management). The results shown correspond to a completion-time prediction that is made about half-way through a LC task's execution. Predicted completion closely tracks the actual completion time. Figure 2.7 shows the average predictor accuracy and the completion time standard deviation normalized to the mean of each workload for all 35 workload combinations we use (combinations of one of the 5 LC benchmarks with each of the 7 BE workloads). Average error (ϵ) is computed as shown in Equation 2.3 at the midpoint of each task over 100 consecutive task executions. The predictor is highly accurate across all these workload combinations with an overall average error of just 2.4%. As expected, higher execution time variation poses greater challenges and predictions tend to be less accurate in such cases; the 5 points with average error of $> 4\%$ all use streamcluster as the LC, with each of the BE we included in the workload. Among those, *RS* gives the highest error rate (12.5%) and the mix between *libquantum* and *namd* gives the lowest error rate (4.4%). Note that the standard deviation of execution time in these cases is

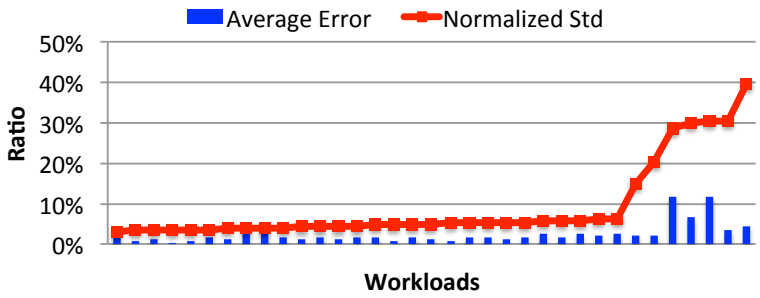


Figure 2.7: Prediction Accuracy for all LC-BE mixes.

significantly larger than predictor errors.

$$\epsilon = \frac{1}{N} \sum_{i=1}^{N} \frac{|predict_i - measure_i|}{measure_i} \tag{2.3}$$

2.4.3 Coarse Time Scale QoS Control

To verify the effectiveness of the heuristics used in the coarse time scale QoS controller in Dirigent, we conducted an exhaustive search on cache partitions for 5 arbitrarily chosen workload mixes and show one of the results, for *streamcluster* as the LC task and *PCA* as the BE task in Figure 2.8. We chose this combination to present because it is one of the few workloads in which LC tasks require a larger partition and therefore stresses the heuristic. As the partition dedicated to the LC tasks grows, the performance of the LC task improves. The knee of the curve representing this exhaustive search is at 5 ways. Dirigent's coarse time scale controller converges to this same partition after just 32 LC task executions (5 coarse time scale controller invocations).

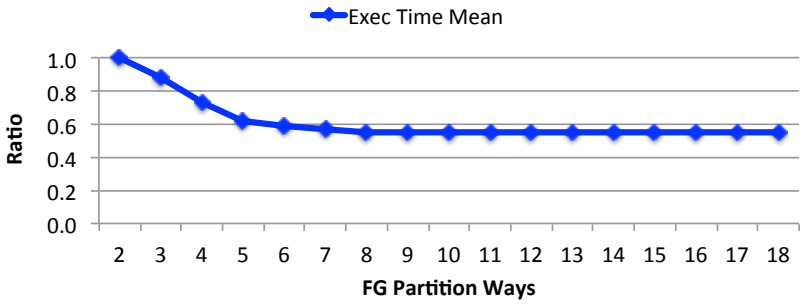


Figure 2.8: Exhaustive Search on Partition Size.

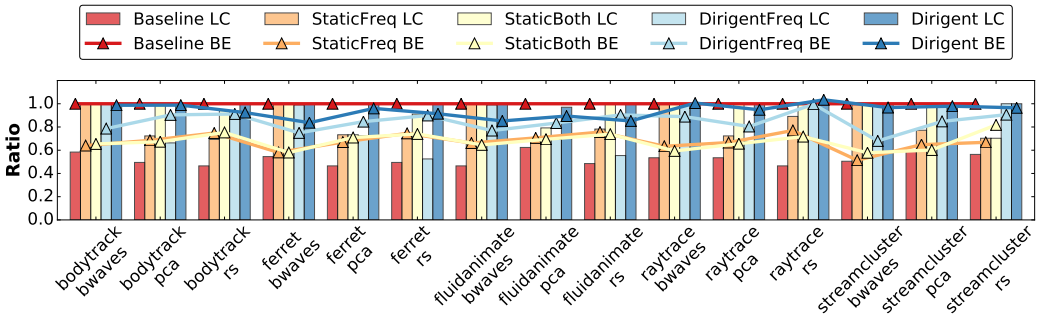
2.4.4 Dirigent Performance

We use five configurations to evaluate the effectiveness and benefits of Dirigent. In the *Baseline* configuration all cores run at the highest frequency and freely contend for resources. *StaticFreq* sets the LC cores to run at the highest allowed frequencies (2GHz) and BE cores to run at the slowest speed (1.2GHz), giving more resources to LC tasks. *StaticBoth* sets the best static cache partitioning (corresponding to Dirigent's heuristic, which we verified is near-optimal) as well as the best frequency for the BE cores. *DirigentFreq* uses Dirigent's fine time scale control only and does not use cache partitioning. *Dirigent* is the full Dirigent implementation that combines coarse time scale cache partitioning with fine time scale frequency control. Note that we omit the coarse time scale-only configuration of Dirigent because it performs just slightly worse than StaticBoth because both use the same partition. Our understanding is that the *StaticBoth* configuration is very similar to the behavior of Heracles [119] in our scenario because the execution time of LC tasks in our experiments are much shorter than polling intervals and controller's

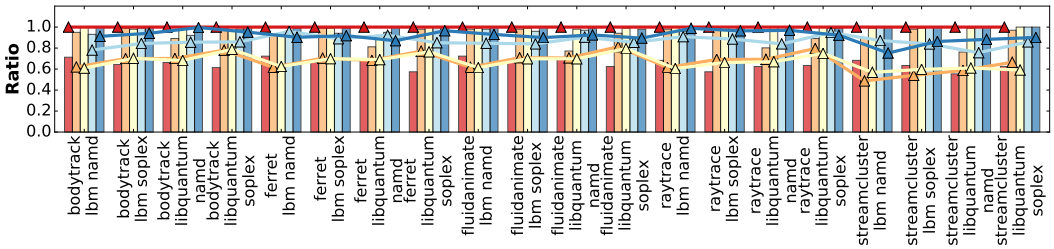
optimization convergence time in [119]; our workloads also do not exercise the network.

To quantify the benefits of reduced variation in LC task execution time, we define the deadline for each LC task to be $\mu_{Baseline} + 0.3\sigma_{Baseline}$, where $\mu_{Baseline}$ and $\sigma_{Baseline}$ are the average and standard deviation of LC completion time in the *Baseline* configuration; PARSEC does not define latency goals even though the applications do represent potential user-facing tasks. We set the deadline for each benchmark to be slightly larger than the uncontended run time of a task but still far smaller than when contention is unmanaged. In this way Dirigent has the LC run time slack to allow BE jobs to run. The tradeoff between deadline tightness and system throughput is demonstrated later in Section 2.4.5. For LC tasks, we focus on the *LC success ratio*, which is computed as the fraction of LC task executions that complete within the deadline defined above. For BE tasks, we report *BE performance*, which is the total number of instructions executed during the experiment (across all cores) normalized to *Baseline*. Results are normalized to *Baseline* since running with no constraints results in the highest BE throughput.

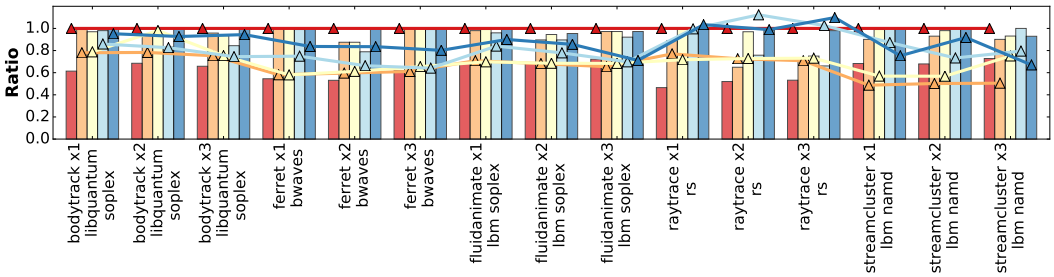
Single LC process: Figure 2.9a and Figure 2.9b report the performance of both LC and BE tasks in all the workload mixes with one LC process and five BE processes, respectively. The results are summarized in Figure 2.10 with arithmetic mean of LC success rate and harmonic mean of relative BE throughput.



(a) Single BE Workload Mixes



(b) Rotate BE Workload Mixes



(c) Multiple FGs Workload Mixes

Figure 2.9: Comparison of LC and BE Performance.

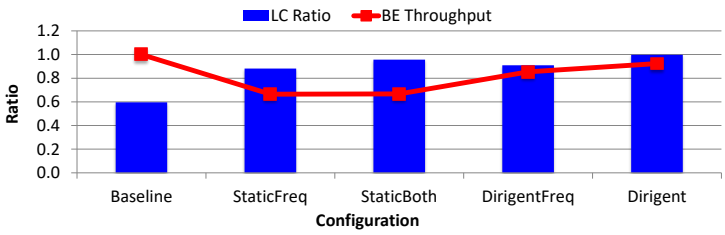


Figure 2.10: Summary of All Single LC Workload Mixes.

We make three key observations about these results. First, while BE performance is high with *Basline*, the LC success rate is very poor, averaging just under 60%. Second, while the (semi-)static mechanisms significantly improve LC completion rate (to nearly 100% in the case of *StaticBoth*, BE performance is severely degraded. On average, BE throughput is reduced to $\sim$ 60% that of *Baseline*. Not shown in the graphs is that the LC throughput is improved by 7.5% on average with *StaticFreq*; this is because more BE tasks are throttled more than with the other configurations. Third, the importance of fine time scale control is clearly demonstrated by both *DirigentFreq* and *Dirigent*. Even without cache partitioning, *DirigentFreq* is able to meet the 95% completion target for all but a few workloads and is able to consistently deliver better BE performance than the static schemes (85% of *Baseline* on average). Finally, *Dirigent*, which combines both fine and coarse time scale control, is able to consistently match or exceed both the LC success rate ($>$ 99% on average and 97% in worst success) and BE throughput of all other managed schemes simultaneously; the BE throughput of *Dirigent* comes very close to the throughput of unconstrained execution, averaging 92% and never dropping below 75% of *Baseline*.

We further look at the execution of one of the workload mixes, a *ferret* LC task collocated with five *RS* BE tasks. Figure 2.11 shows the execution time probability density function curves for the five configurations. Results show that the curves for *Baseline* and *StaticFreq* stretch wide horizontally. Comparing to the *StaticBoth*, the *DirigentFreq* is able to significantly reduce execution time variation by moving the two peaks in *StaticBoth*'s curve closer. *Dirigent* is able to further reshape the curve and merge the two peaks together, achieving even more predictable performance and better BE job throughput. Figure 2.12 demonstrates the distribution of frequencies that *DirigenttFreq* and *Dirigent* use for cores running BE tasks, and show that partitioning the cache significantly reduces the performance contention on LC performance, allowing BE process to run safely at much higher frequency on average. Overall, *Dirigent* can achieve 85% reduction in the standard deviation of execution time of LC tasks at the cost of only 9% of BE performance loss across all the workload mixes we tested. *DirigentFreq* captures part of the benefits in reducing performance variation, achieving 70% reduction in standard deviation of execution time, but suffers from higher BE performance loss at 15%.

Concurrent LC Processes: Figure 2.9c uses the same metrics but with workload mixes that have multiple LC processes. These detailed per-workload results are summarized in Figure 2.13. Due to the large number of possible combinations, it is impractical to exhaustively experiment with all of them. We therefore select five combinations that cover a low to high performance

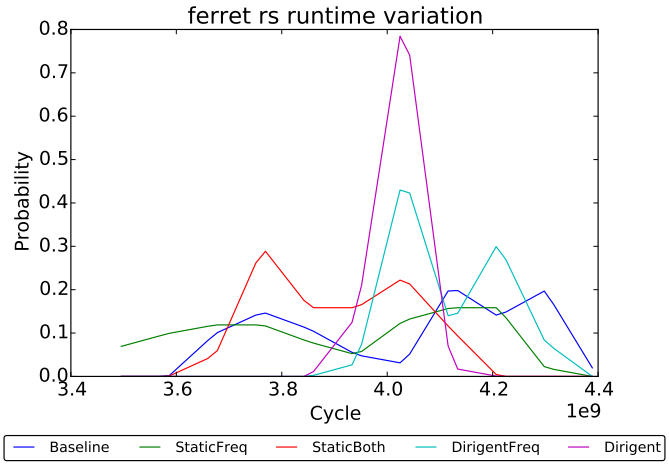


Figure 2.11: Execution Time Probability Density Function Curve.

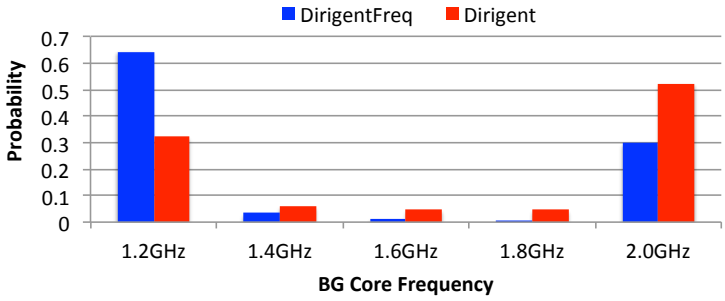


Figure 2.12: BE Core Frequency Distribution.

variation range in *Baseline* and measure the performance of these mixes using a varying number of concurrent LC tasks. The workloads in Figure 2.9c are sorted in ascending order of number of concurrent LC processes within each pair; the total number of LC and BE processes is always 6 (the number of cores).

Overall, the results show similar trends and observations to those exhibited by single-LC process workloads. We discuss two additional insights. First, each LC task may experience different levels of contention due to different LC-BE phase interleavings. This forces the fine-grain controller to use conservative BE performance settings to try and allow even the slowest LC task to complete on target. As seen in Figure 2.9c, within each LC-BE workload mix with *DirigentFreq*, the general trend is that BE throughput decreases with each additional LC task. This problem is alleviated by the introduction of cache partitioning, as it effectively isolates most of the performance interference between LC and BE tasks. Second, since all LC tasks share the same partition, increasing the number of LC tasks also increases their performance variation. This is shown in Figure 2.14, where standard deviation for each configuration is normalized to the value in the baseline. While the results show increased variance with more LC processes within each workload mix, *Dirigent* is still able to effectively reduce the performance variation with low BE task performance overhead.

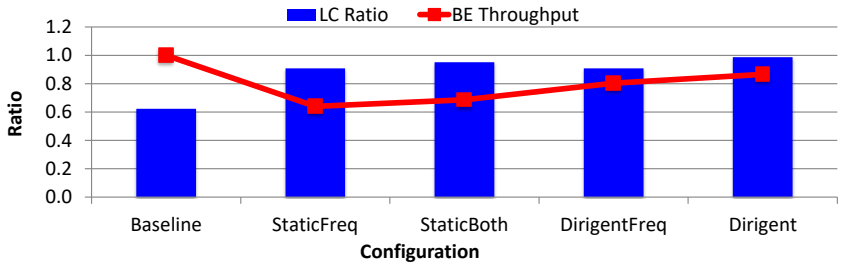


Figure 2.13: Summary of All Multiple LC Workload Mixes.

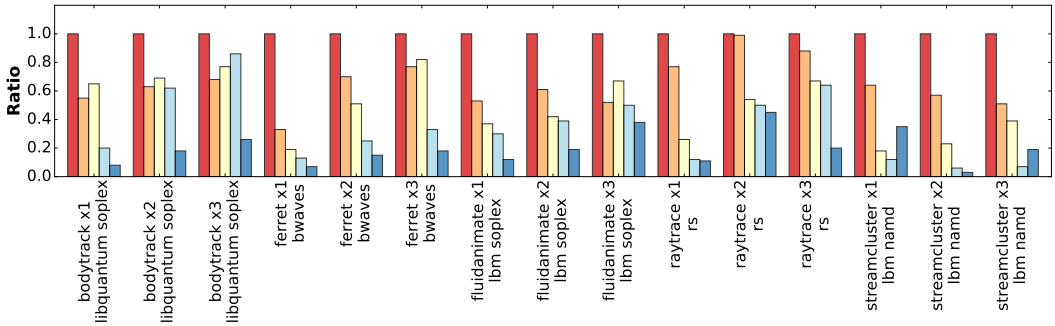


Figure 2.14: Normalized Standard Variation of Multiple LC Workload Mixes.

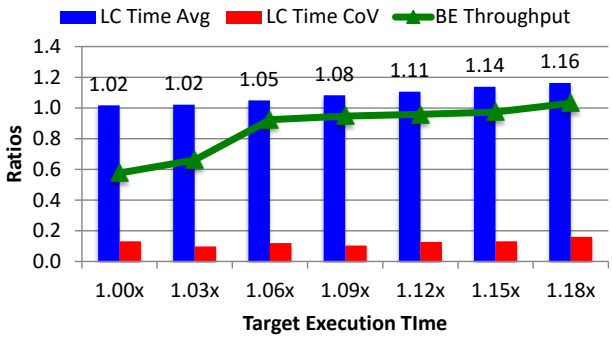


Figure 2.15: Tradeoff Between LC throughput and BE performance

2.4.5 LC Throughput and BE Performance Tradeoffs

So far we evaluated Dirigent with a fixed LC target execution time that reflects the baseline LC throughput. We now show the tradeoff that Dirigent enables between LC task throughput and BE task performance with precise QoS control. Figure 2.15 shows one arbitrarily chosen workload: a single *raytrace* LC process and 5 BE *bwaves* processes. We gradually increase the target completion time from the average completion time in standalone execution until it is larger than the average *Baseline* execution time. The blue bars show the average execution time normalized to the standalone execution time, the red bars show the standard deviation at each target normalized to that of *Baseline*, and the green curves show the BE task throughput normalized to *Baseline*.

Dirigent is able to accurately control the execution time across the entire range of target deadlines; the only exception is when the target is set at the standalone execution time because there is no opportunity for collocation without violating QoS. When the deadline is set higher, Dirigent exploits

opportunities when LC tasks are running faster than necessary and converts them into BE performance. Importantly, Dirigent effectively enables the trade-off between LC throughput and BE performance at high system utilization by reducing the variance of LC task execution time. Note that we do not plot the success rates because Dirigent consistently achieves them at the desired $> 99\%$ rate, except for when the target is set at the standalone execution time.

2.5 Related Work

Online QoS Management: Mars et al. proposed an interference characterization methodology and QoS management schemes that target data center applications [126, 172]. Heracles is a performance management runtime that uses multiple control modules leveraging software and hardware mechanisms to enforce QoS for latency sensitive tasks that are collocated with batch jobs [119]. Zhang et al. proposed to identify interference using cycle-per-instruction data, and the results can be used to enforce QoS by static and manual throttling [176]. most closely related to Dirgient, however, these mechanisms use only coarse time scale statistics and performance variation is not discussed or addressed. As a result the system lacks the ability to adjust contention at fine time granularity, which we demonstrate is crucial to maximizing utilization.

Other related work studies fine-granularity QoS with a focus on saving the energy of executing only LC tasks. PEGASUS improves the energy efficiency of datacenters by dynamically adjusting the power limit of proces-

sors [118]. Adrenaline categorizes queries for target applications and only speeds up ones that are likely to fail QoS goals [78]. TimeTrader and Rubik exploit request queueing latency variation and apply any available slack from queueing delay to LC computation to reduce energy consumption [165, 99]. Suh et al. propose to use various execution time prediction mechanisms to guide frequency scaling to save energy [117, 25]. In contrast, Dirigent converts variation in the run time of LC tasks into improved system throughput.

To the best of our knowledge, Dirigent is the first to trade off the performance of latency-critical jobs that finish sooner than required with higher system throughput for BE tasks. A related mechanism proposed by Min et al. tackles fine time granularity QoS problems for GPUs in heterogeneous platforms [91]. However, the progress heuristics used for the GPU were not general and the mechanism proposed is limited to managing main memory bandwidth contention between the CPU and GPU.

Interference Analysis: Interference is a well studied problem and many models and heuristics have been proposed. A good example is the sophisticated model for predicting multicore interference proposed by Zhao et al. [178]. However, this and other prior models do not address deadline-oriented applications because the prediction is made on coarse time scales. Further, these analysis techniques require complex computation that are not suitable for a dynamic lightweight runtime such as Dirigent [178, 176]. Application Heartbeats is a general progress report framework [77], our profiler uses similar

concept but on a millisecond scale.

QoS-Aware Scheduling: In addition to the QoS control mechanisms discussed in Section 2.1.2, task scheduling across multiple nodes in a cloud server or cluster can also be used to manage contention and performance. Kambadur et al. proposed a sample-based interference prediction methodology for identifying application pairs whose collocation should be avoided [96]. SMiTe predicts the SMT-level interference by profiling interference and sensitivity on each kind of shared resources to guide cluster scheduler [177]. Lee et al. designed and implemented a scheduler in the virtual machine hypervisor for soft realtime applications [110]. Leverich et al. analyzed the source of QoS degradations of latency-critical workloads, and devised a new scheduler to handle these issues [112]. Paragon and Quasar are two task schedulers that classify applications and schedule them into data centers by performing resource allocation and assignment to minimizing interference [46, 47]. Q-Clouds and DeepDive are two QoS-aware scheduler that handle contentions and interference in virtual environment [136, 137]. These works are orthogonal to Dirigent and Dirigent can be integrated with these schemes to manage performance on each node.

2.6 Summary

In this work, we expose the problem of performance variation for latency-critical tasks when collocated with batch jobs and the associated challenges and

opportunities. We explain how such variation leads to low hardware utilization and resource over-provisioning. Our main insight is that minimizing task-to-task variation offers significant opportunities for improving system utilization without compromising QoS goals. We present the design, implementation, and evaluation of the Dirigent lightweight contention management runtime to exploit these opportunities. Dirigent is effective because it can accurately predict the completion time of a running task at very fine time scales. It can thus control resources during a task's execution to meet deadlines and maximize batch throughput.

We show that Dirigent is particularly well suited for cloud-oriented applications where it is common to run just one latency-critical process per node and use batch-oriented tasks to improve utilization. In this case Dirigent exerts precise control over completion time and resources to boost utilization by $\sim 30\%$ compared to configurations that are similar to previously proposed schemes while providing higher QoS for the latency-critical tasks. We further show that even with multiple concurrent latency-critical tasks, Dirigent is still effective, much more so than alternative techniques, and always achieves very high deadline success rates ($> 98\%$).

We opted to implement Dirigent using existing hardware mechanisms and evaluate it on a real machine. Even without new hardware techniques, Dirigent has low runtime overhead ($< 100\mu s$ per invocation) and we are able to demonstrate the effectiveness and benefits of our techniques. However, for tasks with very tight latency constraints ($\lesssim$ 10ms) additional hardware sup-

port may be required. One limitation of the current Dirigent implementation is its dependency on profiling. In this paper we assumed offline profiling but because of the short profiling duration it can be performed online, though it will require pausing all BE tasks while profiling. In future work, we plan to improve the Dirigent prediction and control algorithms to allow concurrent profiling by adding interference offsets into the baseline execution time. A second limitation is that in this first effort with Dirigent, we limited our evaluation to performance variation caused by external interference. Accurate predictions of execution times in the presence of strong input dependence may require interfaces that extend Application Heartbeats [77] or program slicing [117]. Finally, we intend to evaluate Dirigent for parallel applications, where precise control over completion time reduces the overheads associated with synchronization and load imbalance.

Chapter 3

Mitigating Performance and QoS Impact of NVM on Datacenter Applications

Recent developments in non-volatile memory (NVM) technologies enable datacenter operators to further improve the cost-efficiency of warehouse-scale computing (WSC) by provisioning system memory using both DRAM and NVM. To understand the performance and QoS impact of the resulting heterogeneous memory architecture (HMA), we design and implement ONSim, a performance emulator in the Linux kernel that runs workloads with high accuracy, low overhead, and complete transparency. We evaluate ONSim in terms of simulation accuracy and performance overhead using both synthetic and real-world workloads. We then perform a detailed study of HMA with ONSim using memcached as the target workload. By exploiting the inherent data locality in access streams and statistical behavior of application traffic, we bridge the performance gap between HMAs and traditional DRAM systems with simple yet effective HW/SW co-optimizations. Our results show that the proposed mechanisms can achieve 97.5% of baseline performance while only 60% of the data footprint is contained in DRAM and over 96% of baseline performance while sharing limited DRAM with best-effort tasks. We also show that the proposed mechanisms effectively reduce request tail latency and

improves QoS for heterogeneous memory systems.

3.1 Background

3.1.1 Target Workloads

In this work, we focus on one specific kind of LC task: the in-memory key-value store. This application type is widely used in web services by major service providers such as Google, Facebook, and Twitter. Many recent academic and industry studies focus primarily on memcached [78, 112, 9, 45, 43, 16]. A common approach studied in previous work is to provide large scale data caching service (in both capacity and throughput) is to shard the data across multiple memcached nodes. However, this design can cause high sensitivity to request tail latency for large fan-out services. To study the performance of memcached in production environment, we use an open-source memcached load generator [112] that mimics traffic patterns observed in a production environment [9]. We further augment the load generator to demonstrate different levels of access locality (Section 3.4.1).

3.1.2 Workload Characteristics

Previous work makes three observations about WSC workload characteristics and use cases. First, average memory utilization is low in production clusters today. For example, recent studies report typical memory utilization of 40% to 50% at both Twitter [47] and Google [149], and even lower numbers from other service providers [116, 14]. There are a few reasons for this low

utilization.

(*a*) Memory usage varies over time while resources are provisioned based on worst-case expectations. For example, Google's scheduling system Borg consistently checks the memory usage of each task and tries to reserve extra guard band memory to tolerate accidental usage spikes [167].

(*b*) Low occupancy is sometimes necessary to avoid performance interference among colocated tasks. As a result, total system usage can be bottlenecked by other components such as the processor [182, 119] or the network [93].

(*c*) Customers often intentionally over-subscribe resources to guarantee good performance and QoS in the WSC environment.

Second, there is strong inherent temporal locality in many user-facing WSC applications. For example, Atikoglu et al. present a detailed locality study using access traces from key-value stores at Facebook [9], and find that the hottest 20% of the keys cover over 90% of the accesses. Recently, Agarwal et al. [6] report that up to 50% of application data footprint is accessed infrequently in datacenter workloads. Traditional database workloads also exhibit strong locality in production [101].

Third, many user-facing LC tasks have inherently low write-to-read ratios—in most general cases, write-to-read ratio is approximately 3% [9, 78]. Beaver et al. reach similar conclusions, showing significantly more reads than

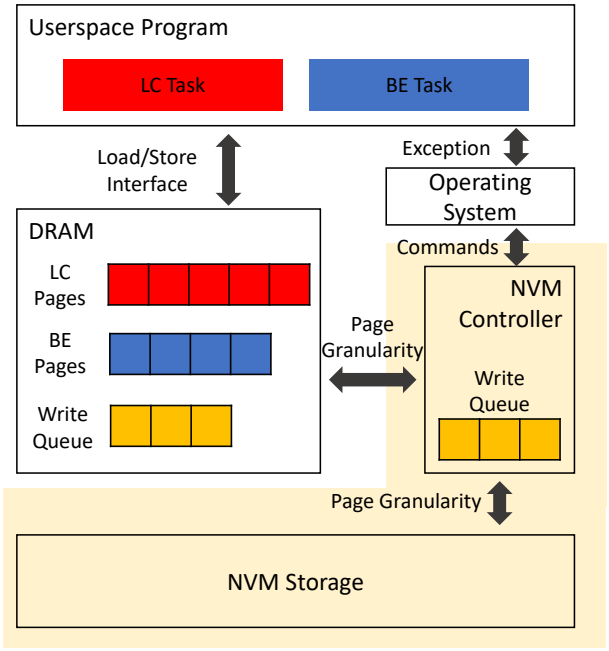


Figure 3.1: Baseline system architecture.

writes at both CDN and storage servers [16]. Traditional datacenter applications show a wider range of behavior, although read-dominant benchmarks have a strong presence [101].

3.2 Target Architecture

We target a node with an HMA that consists of DRAM and a high-performance NVM, where the NVM is used as a block device. We choose this architecture because it offers significant advantages over systems that use DRAM alone, yet does not demand extremely high-performance NVM device that can completely replace DRAM. In addition, treating NVM as a block device simplifies many of the mechanisms required for NVM reliability and

cost-effectiveness. Concrete examples of such a system include low-latency NVM connected over PCIe [3, 161] or in NVDIMM [90] (Figure 3.1). Applications do not directly access the NVM. Rather, the operating system maps a portion of the virtual address space onto NVM and manages access and migrations on behalf of applications. We assume the HMA systems host both latency-critical (LC) and best-effort (BE) tasks, either of which can contend for DRAM resources.

Pages are migrated from NVM frames to DRAM frames transparently. Essentially, the NVM acts as a fast swap device, as assumed by prior related work that targets NVM-based [6, 11, 171] and disaggregate-memory based HMAs [113, 61]. Note that NVDIMM-p specifications have not yet been released and we treat it as caching pages rather than caching at finer granularity. We leave the analysis relating to caching granularity to future work. Unlike prior work, we assume important modifications to improve page migration for QoS of latency-critical tasks and colocation scenarios. We therefore introduce additional explicit components to NVM management. First, because writes to NVM are slow and occur with relatively low bandwidth compared to reads, we add a write queue that may reside in dedicated DRAM or within NVM or HMA modules. We also add mechanisms and policies for provisioning DRAM and NVM resources, and enable task prioritizing as explained below.

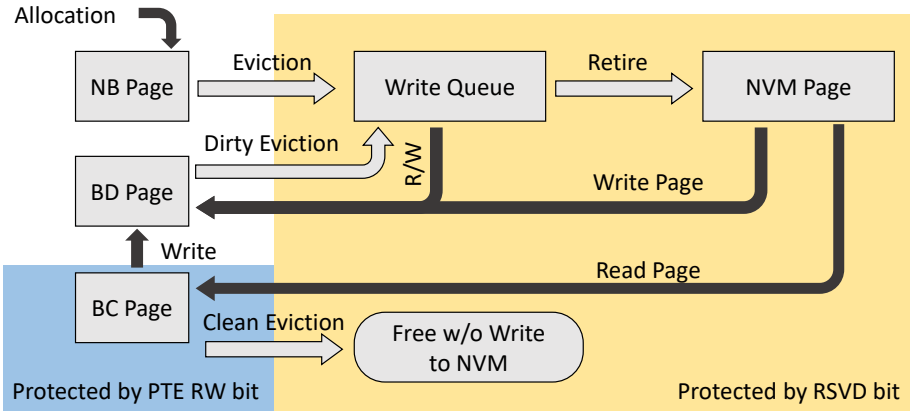


Figure 3.2: Baseline page management policy.

3.2.1 Baseline Page Placement and Migration

Figure 3.2 shows the baseline page migration scheme that corresponds to the baseline system architecture shown in Figure 3.1. Black arrows indicate external events triggered by programs and gray arrows indicate internal OS events. We track three types of pages: ones that are *not backed by NVM* (NB) and which only have a DRAM frame, ones that are *backed by NVM and have clean* copies in DRAM (BC) and have been copied to DRAM to improve performance, and ones that are backed by NVM and are dirty in DRAM (BD) and which have essentially migrated from an NVM to a DRAM frame.

Our baseline system follows prior work, which treats NVM as a swap device for Linux's standard page management mechanisms, but adds an explicit write queue. The baseline page allocator always places pages in DRAM and creates NB pages (i.e., no NVM is allocated); "page allocation" refers to a physical frame being mapped to a virtual address and not a PTE entry

being created. Baseline migration from DRAM to NVM is done following the *CLOCK* algorithm [36], as in Linux's current memory manager. Baseline page migration from NVM to DRAM is done any time a page that is only in an NVM frame is accessed. We assume that NVM is cheap relative to DRAM and do not free the NVM frame after a page is migrated to DRAM, i.e., the page is now *backed by NVM*. Initially the page is marked BC in DRAM, because the copy in DRAM is clean. The first write to a BC page turns the page into a BD page. BD pages are written back to NVM by first placing them in the OS-managed NVM write queue. This queue is maintained as a linked list of frames that need to be written back. The queue is processed in FIFO order in the baseline system. Note that any access to a page that is in the write queue is handled by DRAM without accessing NVM.

Limitations of baseline management This baseline scheme successfully hides NVM from applications, but has three important limitations. First, as also discussed by Agarwal and Wenisch [6], migrating cold pages from NVM to DRAM displaces hot pages in DRAM and causes a significant increase in NVM access. Our evaluation confirms these observations and quantifies the impact of cold pages on performance and task latency statistics. We also identify that NVM write-bandwidth constraints can also severely degrade performance; displacing hot dirty pages by cold pages is thus particularly detrimental.

Second, in colocation scenarios, the baseline scheme does not prioritize pages of high-priority tasks over those of low-priority ones. This can

cause rapid degradation in the performance and latency statistics of high-priority latency-critical tasks. Third, low-priority applications may dominate limited NVM write bandwidth, further exacerbating their impact on high-priority tasks. We develop, and later evaluate, three mechanisms that each address one of the above limitations.

3.2.2 DRAM Admission Control

Our first mechanism targets the degradation caused by cold pages displacing hot pages in DRAM. We enhance the page management scheme with access frequency-based page admission. The main insight is that with a few modifications to the kernel, we can gather enough information to determine the "hotness" of a page with low overhead. We then leverage this information to avoid bringing "cold" pages into DRAM. We thus prevent thrashing and save DRAM resources for pages with higher access frequencies to improve performance and QoS.

The only change from the baseline architecture is the addition of NVM buffer regions that gate admission to "general" DRAM. These regions may reside in dedicated DRAM if NVM is provided over the system bus (e.g., PCIe), or as part of the DRAM within each NVDIMM module. We choose a fixed buffer size that totals 512MB in the system. DRAM size is reduced by the same amount to keep the results comparable across configurations. Pages that are identified as hot are migrated out of the buffer into a "general" DRAM frame, while pages that are identified as "cold" remain in the buffer

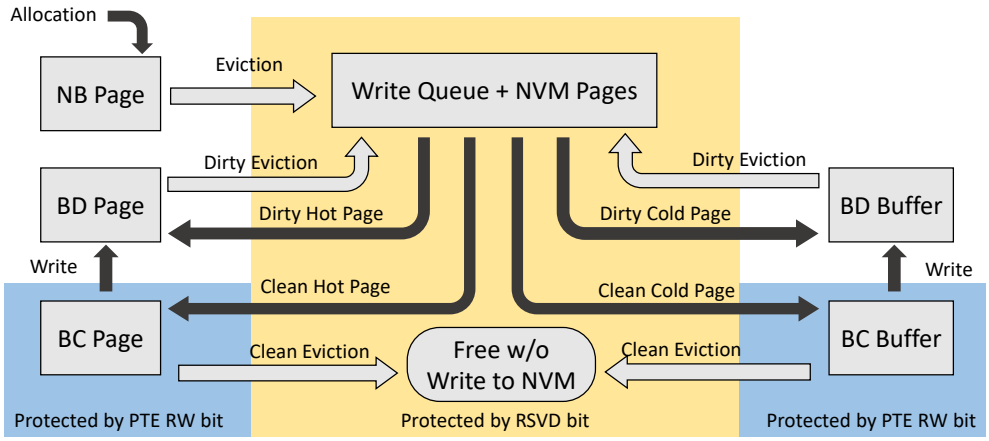


Figure 3.3: Page management policy with NVM buffer.

until they are evicted. Since we expect little locality in the NVM buffer, we use FIFO as the replacement policy. Figure 3.3 depicts how this is done using two additional page states: BC page in buffer and BD page in buffer.

Identifying hot pages To quantify the access frequency or "hotness" of a page, we enhance `struct page` to track the *Idle Distance* of each page. Figure 3.4 shows more details of this mechanism. A page's idle distance is the number of NVM read requests between its most recent access (tracked by `glb_rcnt`) and when it was evicted out of DRAM or NVM buffer last time (tracked by per page `rcnt`). Our implementation updates idle distance using running average (`RA`) to filter out system noises. To better adapt to phase changes quickly, we set the maximum idle distance to be 2× the DRAM size. With this mechanism in place, we avoid thrashing DRAM by keeping "cold" pages (identified by large idle distances) in NVM. We use a customizable

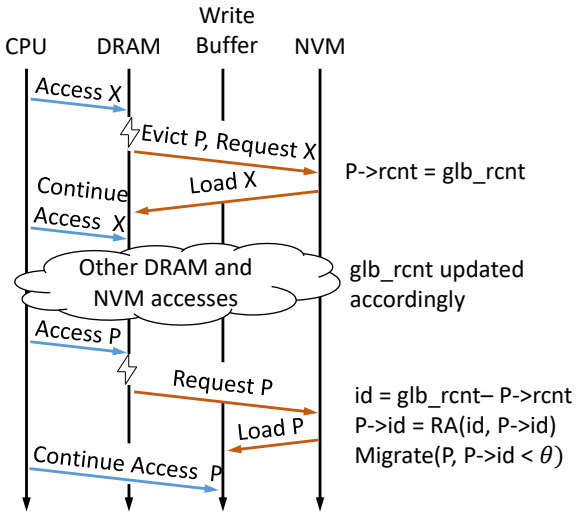


Figure 3.4: Page admission with Idle Distance. The idle distance `id` of page `P` is the number of NVM read requests between its most recent access (tracked by `glb_rcnt`) and when it was evicted last time (tracked by per page `rcnt`). Our implementation updates `id` using running average (`RA`) to filter out system noises.

threshold (θ) to decide whether a page is hot or cold. Maintaining the idle distance has negligible performance overhead because the interrupt delay is included in the simulated NVM latency. It also has negligible storage overhead because it only adds two extra integers to an existing per-page data structure. This allows us to track "hotness" for each page. In contrast, Thermostat [6] lacks this capability, but instead samples accesses to a subset of pages to estimate the hotness of the entire dataset.

Idle distance is similar in concept to stack distance, which may be used to compute cache miss rates. However, stack distance is significantly harder to compute because it requires interception of all accesses to all pages. Stack

distance also counts number of references to distinct pages, which requires additional data structures. There is previous work that enables lightweight reconstruction of miss rate curves using sampling [170], which can provide valuable information to further improve memory management, but this is distinct from our per-page hot/cold mechanism. We leave the exploration of more sophisticated page management schemes to future work.

3.2.3 DRAM Occupancy Control

To address contention for DRAM from colocating tasks, we develop a new technique to balance DRAM usage between tasks under priority goals. Instead of directly managing occupancy, we manage the DRAM miss rates of different applications. We introduce a user-defined knob, the *Miss Ratio Fraction (MRF)*. The idea of MRF is to let users decide the ratio of miss rates (i.e., number of misses per second) from high-priority and lower-priority tasks when occupancy of DRAM reaches *steady state*, (e.g., DRAM resources should be allocated such that BE tasks generate two times DRAM misses as LC tasks). MRF-based page eviction will dynamically adjust DRAM occupancy to reach this priority goal. Hence MRF provides users with a straightforward metric to control DRAM occupancy based on the extent to which the user prioritizes LC tasks over BE tasks. MRF is agnostic to access patterns and can be implemented with very low overhead.

Figure 3.5 shows an example. We assume MRF is set to 1, meaning the rate of misses between LC and BE tasks should be the same. As a result,

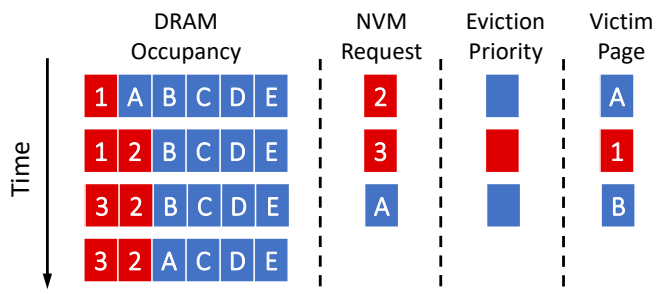


Figure 3.5: Miss Rate Fraction example.

the occupancy control mechanism will evict LC and BE pages from DRAM at the same probability. In our example, LC (red) tasks generates DRAM misses twice as fast as BE (blue), so the mechanism automatically increases the occupancy of LC (red) pages in order to meet the MRF goal. Our implementation of the arbitration mechanism uses only an atomic counter, the remainder of which decides which set of pages to prioritize evicting.

There are several benefits to the MRF approach. First, it fits the QoS requirements for millisecond-level LC tasks because it controls the rate of DRAM misses, which directly affects tail latency. Second, it adapts to user-specified priority without extra information on the working set size or access patterns. Third, it gives users a metric that is easy to reason about. Note that it is possible that a given MRF cannot be met unless BE tasks are given so few pages that they essentially stop. It is better to kill the BE tasks in such cases.

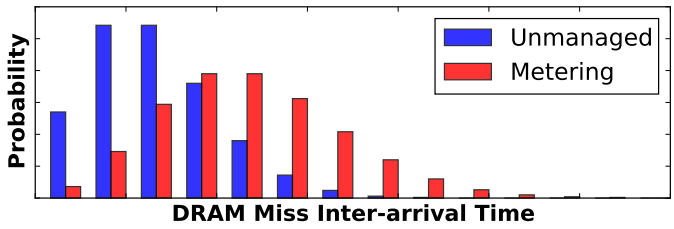


Figure 3.6: Write Bandwidth Metering smoothes out write bursts to reduce interference with LC tasks.

3.2.4 Write Bandwidth Metering

When BE tasks have write bursts, a large number of write requests can temporarily overwhelm the write queue and block LC tasks. We therefore propose write queue "jailing" to meter BE task write bandwidth [181]. We implement jailing by limiting the maximum write queue occupancy of pages from BE tasks. While this approach does not limit the maximum write bandwidth of the BE tasks, it limits the size of the write bursts that the write queue absorbs and throttles BE tasks once their limit is exceeded. To show the effect of this mechanism, Figure 3.6 plots example distributions of DRAM misses with respect to the inter-arrival time. Because BW metering limits the burst size that the HMA will absorb, it reshapes the distribution by throttling bursty tasks. We show later that this mechanism together with MRF effectively limits the performance and QoS impact of colocation.

3.3 ONSim

Evaluating our proposed mechanisms is challenging because it requires running applications with large datasets for long durations. As discussed in Section 3.5.3, prior approaches lack the speed, precision, and flexibility for this purpose. To overcome the limitations of previous performance evaluation techniques, we propose a new simulation infrastructure called the **O**S-level **N**VM **Sim**ulator (ONSim). We implement ONSim by modifying the memory management module of the Linux kernel (2700 LOC). ONSim offers the following benefits:

- **Accurate HMA modeling.** ONSim simulates dynamic data migration and injects appropriate, configurable delay when NVM is accessed. In addition, ONSim collects the necessary information to enable a programmable HMA manager to make informed migration decisions.
- **Transparency.** WSC workloads often have complex dependencies and it is tedious or impossible to change all the source code (including libraries) to use a simulator. ONSim leverages kernel mechanisms and runs unmodified binaries.
- **Low performance overhead** is crucial for latency-critical WSC workloads because the QoS target can be milliseconds or lower. ONSim does not degrade performance when a program is not accessing NVM and hides the overhead of its book-keeping within the NVM access delay.
- **Flexibility.** The ONSim infrastructure enables users to design and exper-

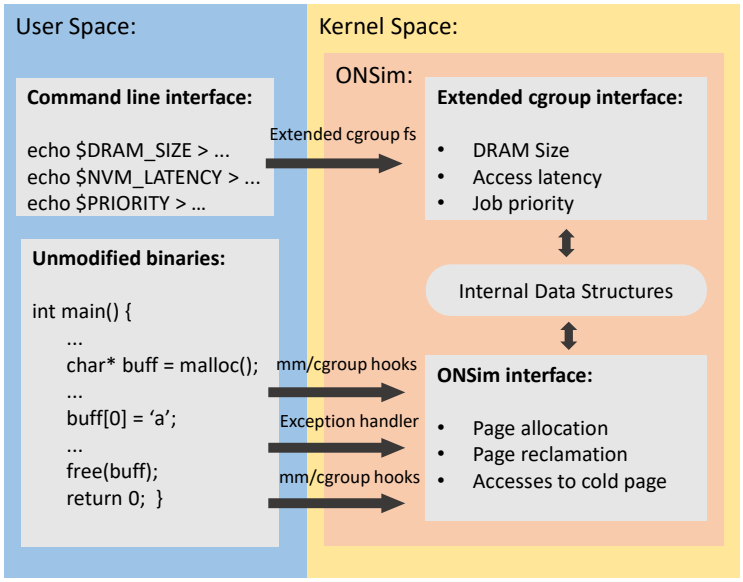


Figure 3.7: Simulator architecture.

iment with HMA systems with flexibility. ONSim provides multiple knobs to enable users to change parameters (e.g., latency, bandwidth, buffer sizes) at runtime. ONSim further enables users to program different system architectures and data management policies to explore the design space of HMAs. Our evaluation of page management policies (Section 3.4.2 and Section 3.4.3) demonstrates this flexibility. ONSim can also be used to simulate other architectures (e.g., disaggregated memory [113]).

3.3.1 Simulator Architecture

Figure 3.7 shows the overall architecture of ONSim. ONSim interfaces with the user through Linux Control Groups. Linux Control Groups (`cgroups` [4]) is a kernel mechanism that accounts for and partitions shared

resources (CPU, memory, network, etc.) among a set of tasks. A pseudo-filesystem (`cgroupfs`) is used to interface with resource controllers. In this work, we extend `cgroupfs` to identify each userspace program being simulated and to configure the simulator and task priorities. We assume two priority levels: latency-critical (LC) and best-effort (BE); finer-grain task management can be easily achieved by adding more priority levels to the simulator. Users configure both global parameters (e.g., read latency, write bandwidth, and DRAM size) and per-cgroup parameters (e.g., job priority and migration policy) through the command line interface (Figure 3.7).

Users run unmodified binaries and ONSim intervenes only when necessary to simulate HMA performance impact. We enhance the `cgroups` hooks that tap into the memory allocation and reclamation process within the kernel to monitor and intercept page allocations and deallocations. Pages that are moved to NVM are *poisoned* [60] so the next access triggers a page fault and ONSim can intercept the access.

We create and extend several data structures within the Linux kernel to implement ONSim. We use several linked lists to track pages from jobs of different priority levels in the simulated DRAM and NVM queues. We also extend `struct page` to track frame location, state, and timing information (see Section 3.3.3). However, the most important extension is to the page table entries (PTEs). Specifically, we enhance the PTE to implement three functionalities: poison NVM pages, track clean pages, and get page access information for page migration policies. Figure 3.8 shows the bits we use in

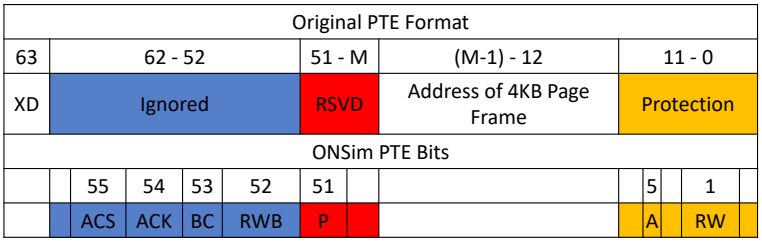


Figure 3.8: ONSim PTE extension bits.

ONSim. This extension is necessary to preserve the original functionality of the kernel.

To accurately simulate the HMA microarchitecture and write bandwidth of NVM, ONSim must only simulate writing back dirty DRAM pages to NVM. ONSim intercepts the first write to a DRAM page that is already mapped to NVM. We use the read/write (*RW*) bit in the PTE, and "virtualize" this bit to not break existing kernel operation. We clear the *RW* bits to force exceptions the first time when clean pages that are backed by NVM are written to and again intercept the resulting page fault. To make sure the rest of the kernel functions normally, we use one bit (*RW* Backup, or *RWB*) from the ignored bits in the PTE to make a copy of the *RW* bit, and another bit to indicate whether the page is "Backed by NVM and Clean" in DRAM (*BC*); more details in Section 3.2.1. Functions that access the original *RW* bit now check the *BC* bit first to determine which bit to operate on. When simulated programs write to a page that does not have system *RW* permission (which is decided by checking $\left(\left(\overline{BC} \wedge \overline{RW}\right) \vee \left(BC \wedge \overline{RWB}\right)\right)$), the kernel follows its original (non-ONSim) code path to handle the fault. If the fault is caused by

accessing a page that has system *RW* permission and *BC* is set, the page is marked with *RW* permission and an ONSim routine is invoked to track the write.

Furthermore, we borrow a technique from the `kstaled` kernel patch [111] to get page access information. The CPU sets the accessed (A) bit of the PTE whenever data in the page is accessed, which a migration algorithm reads and clears to approximate usage information. To ensure that ONSim does not break existing kernel code, we again introduce two new virtual bits in the PTE. "Accessed cleared by kernel" (ACK) is set whenever the A bit is cleared by the kernel for normal system operations, and "accessed cleared by simulator" (ACS) indicates that the A bit was cleared by ONSim. When the simulator clears A to implement the simulated migration algorithm, it also sets ACS and clears ACK; $(A \vee ACK)$ indicates to the simulator whether a page has been accessed. Kernel code does the opposite. The simulator can thus maintain its own access information without perturbing the underlying host system.

3.3.2 Performance and Architecture Modeling

Because of the asymmetric read and write performance of NVM, we treat read and write accesses to NVM differently. ONSim injects a user-specified delay whenever an NVM page is read, unless the access hits in a simulated queue (e.g., the NVM write queue). Because the system architecture we model uses coarse-grained access NVM, threads block on NVM access. ONSim cannot simulate an HMA with fine-grained hardware load/store NVM

accesses. This is because there is no way to accurately model memory-level parallelism within a thread without highly-intrusive instruction-level instrumentation/simulation to track dependencies on data read from NVM.

NVM writes happen when pages retire from the write queue. It is important to accurately enforce write bandwidth because it can easily be the performance bottleneck of an HMA. We buffer write requests and process them the rate corresponding to the user-specified write bandwidth (currently as a FIFO). ONSim uses a linked list to store these requests and tags them with a timestamp. It checks this list asynchronously to pop out any entry that should have finished, assuming write requests are processed serially. As the write queue becomes full, it blocks more pages from migrating out of DRAM and pauses the execution of a program until its oldest entry is freed.

ONSim simulates NVM delay using "busy waiting" on timestamp counters. One accuracy feature in our implementation is that ONSim hides its own latency by recording current time immediately after the interrupt type is determined and uses it as the start time of future injected delay. The current version of ONSim targets NVM with ≥ 5 µs latencies. ONSim can be extended to accurately simulate latencies down to ~ 1 µs by batching the simulation of NVM accesses. In this mode only page state is tracked by the page-fault handler (modifying PTEs), which we measure at 0.77 µs. Delay is then injected for each batch of NVM accesses.

3.3.3 ONSim Accuracy and Overheads

We first quantify the overhead and accuracy of ONSim itself before evaluating an HMA. We use a well-known WSC workload (memcached) and three synthetic workloads, each mimicking a different memory access pattern:

(*a*) *Thrash* iterates over each allocated page and with the default CLOCK replacement, each memory access from the benchmark triggers a page fault;

(*b*) *Random* accesses any of the allocated pages with uniform probability; and

(*c*) *Locality* accesses 20% of the allocated pages with 80% probability.

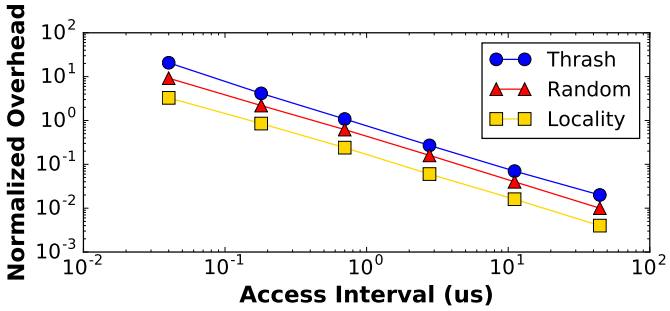
Each synthetic workload has a dataset of 100 pages, while DRAM size is just 50 pages. The working set and capacity are intentionally small (first bytes of each page) to minimize OS overhead and remove any performance bottlenecks (e.g., cache capacity). We use one million accesses in each configuration. Note that *Thrash* is a pathological case and represent the absolute worst-case scenario. The target workload for HMA (which we evaluate later) has much higher locality, and ONSim overhead and error are much smaller.

Performance Overhead We quantify performance overhead by the extra execution time when workloads run under ONSim. We set both read and write latency to zero so the performance overhead can be completely attributed

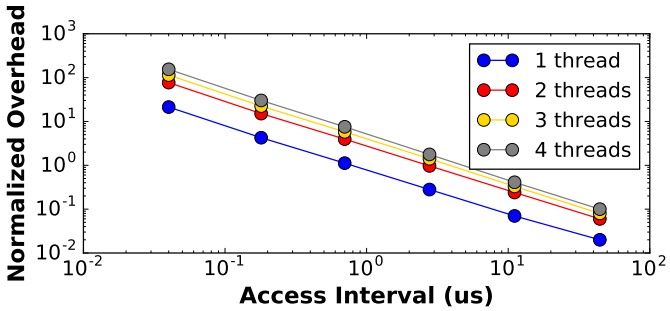
to the simulator. The overhead is normalized to the execution time of each benchmark when running without ONSim.

Figure 3.9a summarizes the performance overhead measurements with the three single-threaded synthetic workloads mentioned above. Different memory access intervals are used in the benchmark to mimic different levels of memory intensity. *Thrash* consistently shows over 3 times more performance overhead than *Locality* and represents the worst case scenario. In all three cases, the average delay for each page fault is consistently ~ 0.77 µs. At about 10 µs (33K CPU cycles) between consecutive page faults, the performance overhead of ONSim is 6.8% with *Thrash*, 4% with *Random*, and 1.7% with *Locality*. The performance of multi-threaded workloads is more complicated to analyze because of lock contention in the operating system and ONSim code, and nondeterministic thread interleaving. Figure 3.9b summarizes the results for multi-threaded *Thrash* synthetic workloads, showing increased lock contention level causing noticeably higher performance overhead. However, the overhead is different from performance error, which will be significantly smaller if the simulated NVM read latency is high compared to ONSim's overhead.

Finally, for a more representative overhead measurement, we use a similar method of injecting zero delay to measure ONSim performance overhead with memcached. We configure memcached with our *worst-case scenario* (low locality, DRAM size is 40% of working set size, two memcached threads, more details in Section 3.4.1), and measure a 2.9% performance degradation. Given that memcached is a very memory intensive application, and ONSim perfor-



(a) Single-threaded benchmarks.



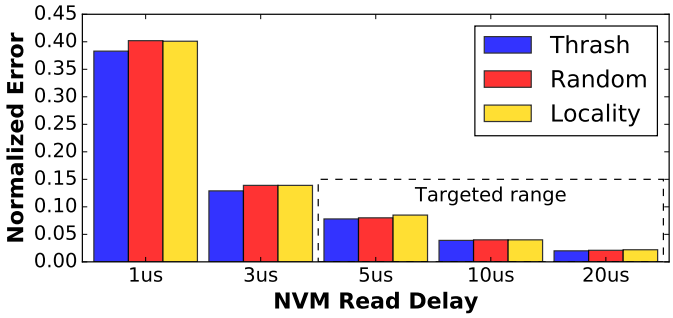
(b) Multi-threaded Thrash (pathological case).

Figure 3.9: ONSim Performance Overhead.

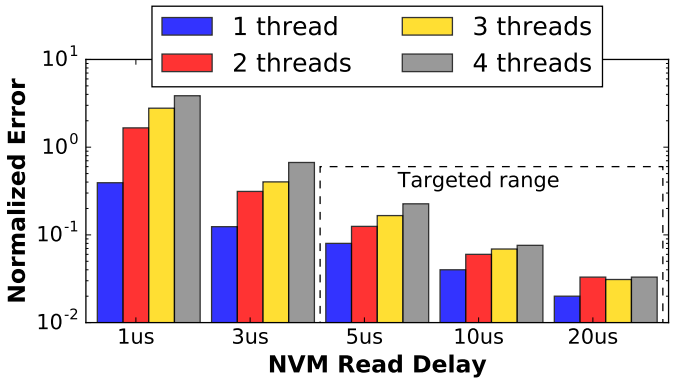
mance overhead is proportional to memory intensity and DRAM miss rate, we believe ONSim has low enough performance overhead for simulating HMA for most applications, including memcached.

Simulation Accuracy We quantify delay accuracy by the *relative delay error*, shown below. We sweep a wide read-latency range, which includes the delay of recently released 3D XPoint devices [3, 2, 161].

$$\left(\frac{T_{ONSim} - T_{Native}}{\#NVM\ Accesses} - Delay\ Target \right) \ / \ Delay\ Target$$



(a) Single-threaded benchmarks.



(b) Multi-threaded Thrash (pathological case).

Figure 3.10: ONSim NVM-latency accuracy.

Figure 3.10a shows the average delay error for each single-threaded benchmark. Our measurements show a consistent 0.39 µs extra latency, which suggests that ONSim successfully hides 49.4% of its total overhead within the NVM latency, given the exception delay observed. Figure 3.10b summarizes the results for multi-threaded *Thrash* at a fixed access interval. Error is significantly higher at low NVM read latency, which is expected because of high lock contention levels. For our target NVM, the overhead of ONSim has negligible impact on simulation results and the relative error plateaus at 3.3%.

Table 3.1: Default workload generator configuration.

Parameter	Value
Key size	30 bytes
Value size	500 bytes
Interarrival distribution	Generalized Pareto
Dataset size	10 GB
Memcached server thread#	2
Default locality level	Medium

Storage Overhead ONSim extends existing Linux data structures to track page locations. Specifically, it adds extra fields into `struct page`, which is a native kernel data structure that describes physical page frames. On a system with 32GB DRAM the storage overhead is less than 1%.

3.4 Evaluation

3.4.1 Methodology

In the rest of this paper, we demonstrate the capabilities of ONSim by using it to study the QoS impact of HMAs on datacenter workloads. We perform various sensitivity studies of different system parameters, quantify performance and QoS challenges in using HMAs for datacenter workloads, and evaluate our proposed DRAM admission, DRAM occupancy, and NVM bandwidth metering mechanisms. We perform experiments when latency-critical tasks run standalone and also when colocated with best-effort tasks.

We use memcached [58] as a representative application with latency-critical tasks. We generate queries to memcached using a published mem-

Table 3.2: Default simulation parameters.

Parameter	Value
DRAM size	6 GB
Write queue size	256 MB
NVM buffer size	512 MB
Maximum write bandwidth	100 MB/s
Read latency	20 us

cached request generator [112], which guarantees that the memcached server is consistently under very high load. Table 3.1 lists the default parameters we use in the rest of the paper. Most of the parameters are derived from observations of real world traffic [9]. We use *query-per-second (QPS)* as memcached's *performance metric*, and *percentile latency* as the *QoS metrics*. Enough client-side threads are used such that performance is not bottlenecked by request generation.

One important enhancement we made to the request generator is the distribution of request indexes. We add a power-law distribution generator to mimic locality of user requests [38]. More specifically, the frequency of the k-th most accessed item out of all N items is C/k^s, where s is the parameter that controls the locality of the distribution, and C is a normalization factor that guarantees the sum of the probability over all N elements is one. We simulate three levels of locality: low, medium, and high, where 40%, 20%, and 10% of the most-accessed records cover 80% of the accesses respectively. We use medium locality by default (sweep analysis in Section 3.4.2). We also use deterministic key and value sizes to prevent changes in "hot" records from

causing performance results to vary across runs.

When running memcached standalone, we limit its DRAM capacity so that the time for the system to reach steady state (e.g., DRAM page migrations and heuristics warm up) is tolerable. We use a relatively small number of threads to reduce the error caused by contention in the kernel (Section 3.3.3). We use a low NVM write bandwidth to match the small number of threads. In the default configuration, we simulate two memcached threads with a 10GB working set and use 6GB of DRAM, where the rest of capacity is provided by NVM. The default workload parameters are summarized in Table 3.1, and Table 3.2 lists the default values for major system parameters.

To understand the performance impact of task colocation, we run three workloads as best-effort tasks and study the performance interference. In each case, we configure memcached to use the same amount of memory as the BE task and provision only half of the total memory as DRAM. Table 3.3 summarizes the BE tasks we use and their memory usage. Graph500 is a well-known workload that represents graph algorithms, which often constitute a core step in many analytics workloads [1]. We use the breadth-first search (BFS) phase of the algorithm. GUPS (Giga Updates Per Second) is characterized by random memory accesses and consistently generates high write bandwidth [120]. Block Tridiagonal solver (BT) from NPB suite represents computational fluid dynamics applications [12]. Notably, BT shows strong phase changes and will demonstrate the effectiveness of the performance isolation schemes we discuss later.

Table 3.3: Best-Effort Tasks Used in This Study.

Workload	Description	Memory Usage
Graph500	Breadth-first search	4GB
GUPS	Random memory access and updates	8GB
BT	Strong phases changes	10GB

3.4.2 Standalone Execution

We first evaluate the impact of an HMA on the performance of memcached when running alone. We follow the methodology described above and explicitly identify any parameters we vary. We explore the impact of DRAM size (constant 10GB memcached footprint), locality of the workload, latency of NVM, and NVM write bandwidth.

Performance and locality Figure 3.11 shows the impact of DRAM capacity on memcached performance (QPS) normalized to the performance when running with sufficient DRAM (all-DRAM). Each bar group represents a different capacity of DRAM, bars within each group represent different NVM latencies (5/10/20 µs), and the stacks within each bar show different levels of locality. This experiment uses the default Linux *CLOCK* migration policy. Two important observations can be made from the results. First, workloads with higher data locality are more tolerant to NVM latency and smaller DRAM capacity because their small "hot" data footprint can be more easily contained in the DRAM.

Second, for the same locality level, as DRAM size increases, perfor-

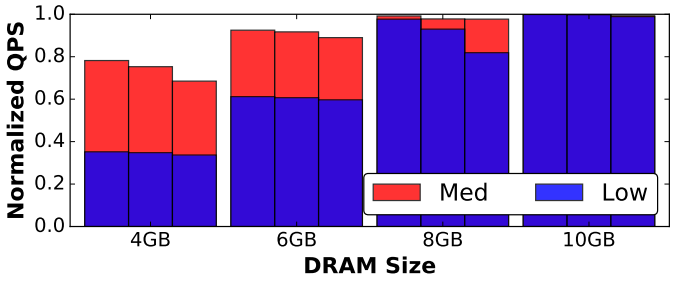


Figure 3.11: Performance sensitivity to DRAM size. QPS is normalized to all-DRAM configuration. Bars with in each group represent 5/10/20 µs NVM latency. Results show larger performance improvements when bottleneck shifts from write bandwidth to read latency.

mance changes at two separate rates. When DRAM is small and misses are frequent, performance is bottlenecked by NVM write bandwidth. However, when DRAM size crosses a threshold (6 GB for low locality, 4 GB for medium locality), performance bottleneck shift to read latency, which leads to higher overall performance. The reduction in miss rates also improves bandwidth usage. In Baseline configuration, the admission control mechanism reduces read and write bandwidth usage by 23% and 26% respectively compared to *CLOCK*. Other results are omitted due to page limitation.

Next we compare the caching efficiency of admission control compared to *CLOCK* and an oracle policy. Figure 3.12 plots the ratio between NVM reads and the number of memcached requests; assuming all important metadata (e.g., hash tables) are always cached in DRAM, this ratio approximates the frequency of migrating pages from NVM to DRAM (the "miss rate"). With 4GB of DRAM with low and medium locality, and at 6GB with low locality, the miss rate is significantly higher than when more DRAM or locality

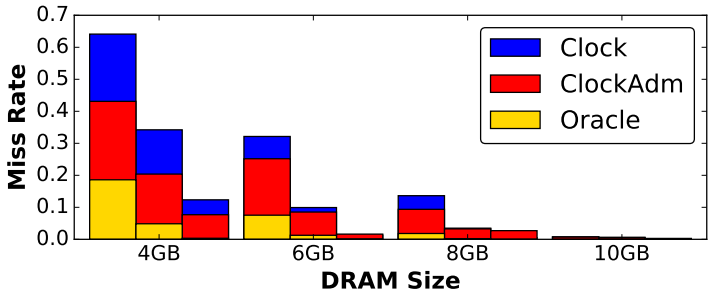


Figure 3.12: DRAM Caching Efficiency Sweep Analysis. Each group of bars show results of low, medium, and high locality from left to right.

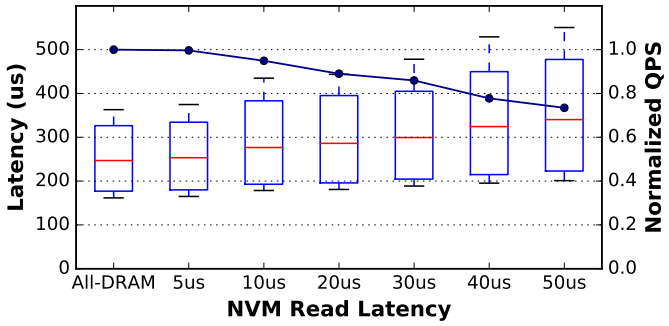
is available. The stacks in Figure 3.12 also show that admission control is still far from a theoretical optimal migration policy. The oracle policy is simply an estimate of the theoretical miss rate curve for each locality level when DRAM covers different percentage of the most-accessed blocks. We factor in the segmentation of record size with respect to page size and assume the oracular replacement policy always keeps the pages with the highest access frequency in DRAM. ONSim enables future studies of better migration policies than our initial admission control mechanism.

Latency statistics Figure 3.13a shows the impact of NVM read latency with the baseline *CLOCK* page migration policy. Other than latency, the parameters follow those in Table 3.2. The line shows performance relative to an all-DRAM system and the box plot shows average latency and the 5, 10, 90 and 95 percentiles for each read latency for the medium-locality workload. At a low latency of 5 µs, average performance (normalized QPS) of medium-locality workload is not affected when only 60% of the footprint fits in DRAM.

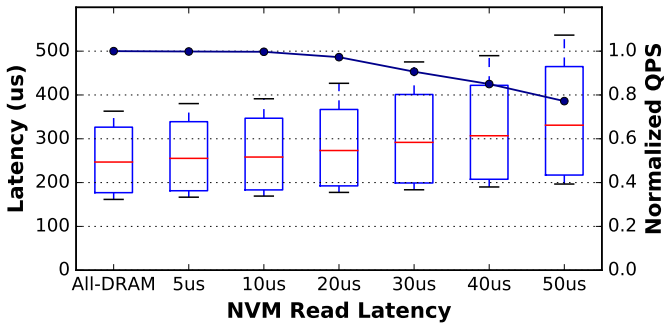
However, even a fast device cannot provide satisfactory performance when locality level is low. While results are not plotted, low locality configurations suffer almost 30% performance degradation at 5 µs NVM read latency. At a currently more-reasonable read latency of 20 µs, QPS degrades by 14% with medium locality.

Figure 3.13b repeats the experiment using our DRAM admission policy. With the enhancement of page admission control, we can further reduce DRAM miss rate by 2.8%, which is achieved by setting the idle distance threshold to be 20% of the total DRAM size. In doing so, we not only reduce thrashing of DRAM capacity, but also filter out victim pages that are not worth caching. Overall, 63.8% of the evicted pages are from the 512 MB NVM buffer. Remember we use a FIFO replacement policy in this buffer because there is little locality. Overall, we are able to achieve 97.3% of the all-DRAM QPS. Compared to baseline *CLOCK*, our DRAM admission policy improves QPS by 11.3%.

To conclude, our systematic study demonstrates that, for memcached with medium locality level, an HMA with even a 20 µs-latency NVM and DRAM provisioned for just 60% of data footprint can deliver performance with negligible degradation, while maintaining tail latency within 2.7% of that when DRAM is overprovisioned.

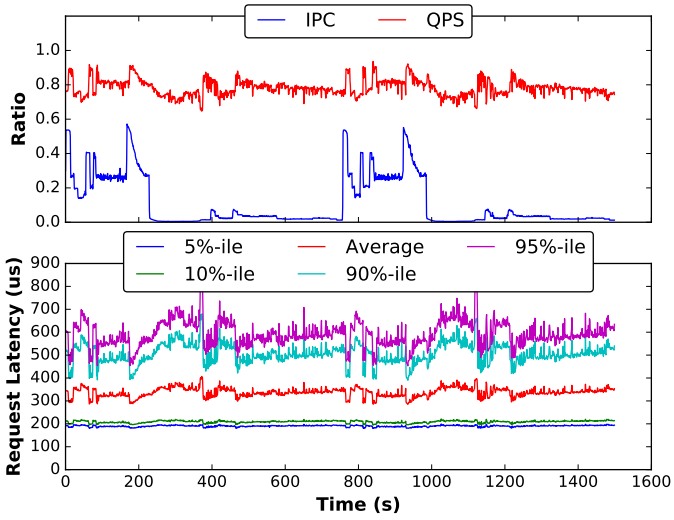


(a) CLOCK

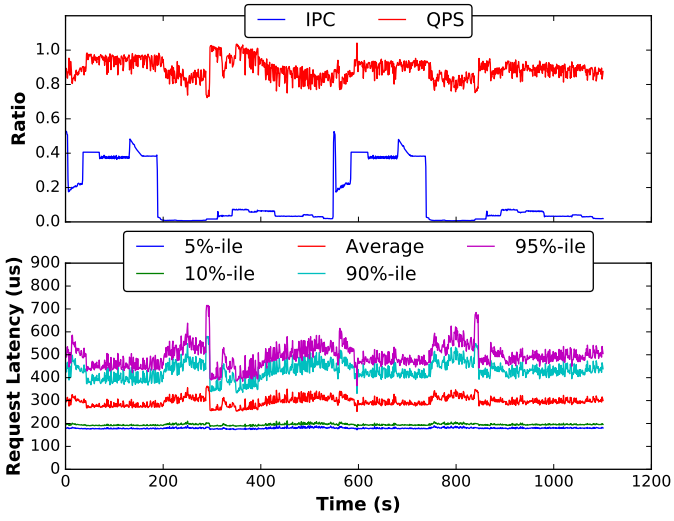


(b) CLOCK with Admission Control

Figure 3.13: NVM latency sweep. Curve is normalized QPS; the whiskers are 5%-ile and 95%-ile latency; box bounds are 10%-ile and 90%-ile latency; line is average latency.

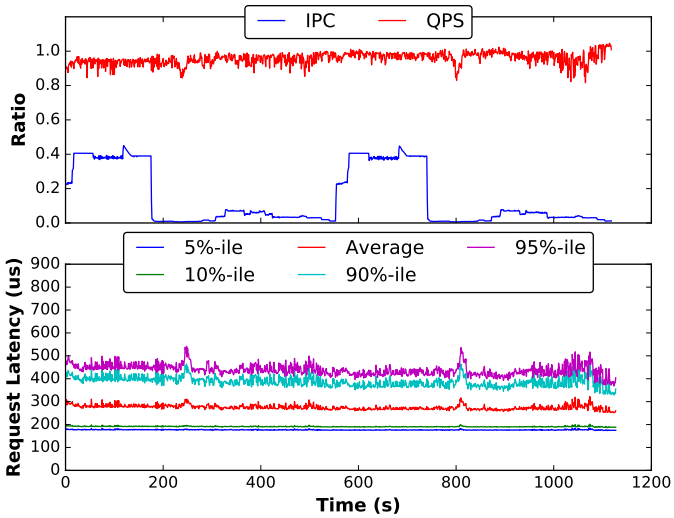


(a) CLOCK and ADM



(b) Occupancy control

Figure 3.14: Runtime statistics of memcached colocated with BT.



(c) Bandwidth metering

Figure 3.14: Runtime statistics of memcached colocated with BT (cont.).

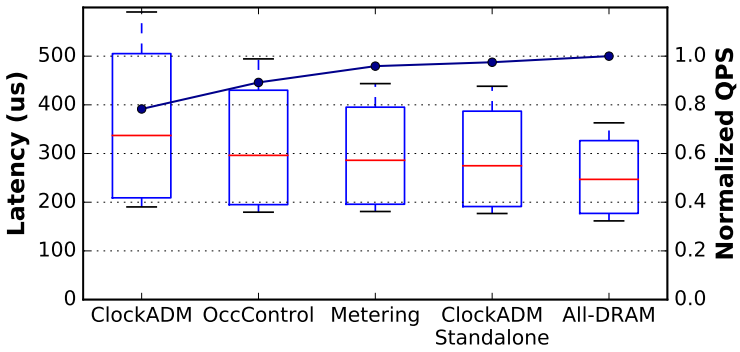


Figure 3.15: Latency distribution colocated memcached.

3.4.3 Colocation Scenarios

A major appeal of WSC is to maximize resource utilization. This can be done by colocating best-effort tasks in the same nodes running latency-critical high-priority tasks. We show detailed evaluation results of the colocation of memcached with NAS BT [12]. BT exhibits strong interference and large differences between multiple phases of execution. We also show summarized results when colocating memcached with Graph500 and GUPS. In all experiments, we use the default system and workload parameters (medium locality and 20 µs latency). We configure the memcached workload to have the same footprint as the interfering application. The system has enough DRAM for just half of the combined memcached + BE task footprint.

Figure 3.14 shows how the relative performance (QPS for memcached and IPC for BT, normalized to all-DRAM) and percentiles of memcached query completion-times vary over time when memcached is colocated with BT. Figure 3.14a shows a system with our admission control that is oblivious to task priorities. Performance of memcached peaks at about 90% of all-DRAM. This degradation is caused by increase in migrations as BT contends for DRAM resources with memcached. As BT goes into its memory-intensive phase, its IPC drops significantly because it saturates the NVM write bandwidth. Memcached QPS degrades by another 10% in these periods. A similar trend is observed for percentile request delays. Figure 3.14b shows the benefit of our occupancy control QoS mechanism (Section 3.2.3). We set MRF to 2 (BT DRAM-to-NVM migrations occur twice as frequently as those of mem-

cached). This prioritizes the DRAM occupancy of memcached at a coarse time scale and increases average QPS by 7.1%. However, memcached performance is still susceptible to BT write bursts and hence suffers from poor QoS and large performance swings. These swings are mitigated by the write bandwidth metering QoS mechanism (Section 3.2.4). Specifically, we limit BT to one third of the 256MB write queue. Figure 3.14c shows that combining the two QoS mechanisms effectively removes nearly all performance swings and achieves an average of 95.7% of standalone QPS.

The completion-time statistics and performance are summarized in Figure 3.15). Thee QoS mechanisms are able to maintain memcached targets even at the extreme conditions of this experiment, which mimic worst-case peak usage scenarios on a system with 20 µs-latency NVM. This is achieved without killing the BT best-effort task, which exhibits an IPC that is 14.3% of its standalone performance. This is slightly higher than the 12.3% observed without the Qos mechanisms in the baseline system. While this seems like low performance for BT, it saves significant datacenter resources because the alternative is to put BE tasks in dedicated BE nodes. Graph500 and GUPS do not show strong phase behavior and colocation results for these two BE tasks, together with that for BT, are summarized in Figure 3.16. Graph500 exerts the least amount of pressure on memcached because it accesses cold data at a much lower frequency. As a result, memcached is able to keep most of its hot pages in DRAM. Graph500 also writes to few pages and does not saturate the write bandwidth. When colocated with GUPS, memcached sees similar

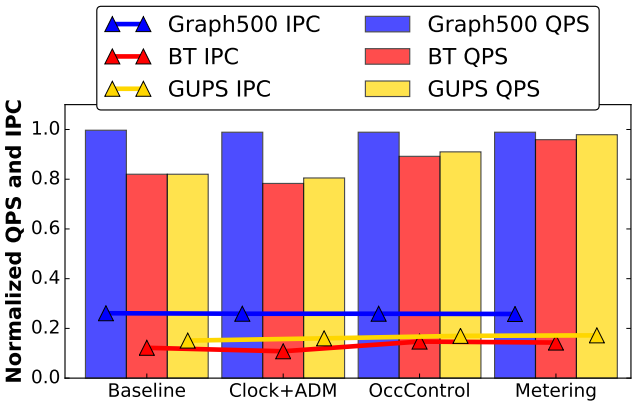


Figure 3.16: Performance results of task colocation.

performance to BT. However, performance of GUPS itself is completely bottlenecked by NVM write bandwidth and doesn't change significantly across configurations.

3.5 Related Work

3.5.1 Heterogeneous Memory Architectures

Loosely, a heterogeneous memory architectures (HMA) is one that combines multiple memory types in a single memory system, where data can be accessed with fairly-low latency even from the slower memory device (e.g., under $10 - 20$ µs). Prior HMA studies can broadly be classified based on the exposure of heterogeneity to the software.

Some prior work completely hides the HMA from software. Examples include: an HMA with faster on-package, and slower off-package DRAM [156, 158, 179, 129, 157]; an HMA with a small DRAM acting as a cache for a

separate slower NVM [146, 148, 144]; and possibly future memory modules that offer DRAM-cached access to NVM on the same module [62]. At the other end of the spectrum are architectures that fully expose memory heterogeneity to applications. Examples include: library interfaces for explicit memory allocation within slower NVM [11, 171]; explicit use of on- vs. off-package DRAM [128, 158, 34, 139]; and also architectures that directly target fine-grained NVM (e.g., [32]) are out of scope of this paper. In addition, there is work on application-specific use of an HMA. Debnath et al. study a tiered-memory architecture optimized for key-value stores and propose to use flash memory as a cache for disks, storing only hashing data in DRAM to reduce DRAM footprint [43, 44]. Dulloor et al. design a system to place data between DRAM and NVM using offline profiling and change the application source code [50].

A third approach is to assume a heterogeneous memory architecture (HMA) that is exposed to the operating system but transparent to userspace, so userspace programs can benefit from it without modification to source code or even the binary. Huang et al. optimize flash memory performance by reducing the overhead associated with translations and indirections across system layers [79]. Memory 1 from Diablo is an NVDIMM-f device that relies on kernel and BIOS changes to communicate with the processor [163]. Kannan et al. propose to improve HMA performance in virtualized environment with both host and guest OS support [98].

Our goal is to balance QoS and system cost instead of solely minimizing

DRAM provisioning, so we assume 50%-60% of system memory is provisioned by DRAM. While this assumption matches the observations made in previous work [6], we also perform a sensitivity study on DRAM size in Section 3.4.2. Such an architecture is especially interesting in a WSC environment because it effectively exploits the important WSC workload characteristics observed above. We show in this paper that HMAs can offer new tradeoffs to system design by exploiting the high density of NVM with little-to-none performance or QoS degradation.

3.5.2 Cache Replacement Policies

Our work is within the third category of HMAs described above. As such, it essentially improves the page migration and replacement mechanisms of the OS in a way that is aware of the HMA, aware of task priorities, and aware of write-bandwidth constraints. The modifications we propose fall within the large body of work on page replacement policies, whose surveying is beyond what can fit within this paper. Our specific mechanisms relate to the classic *CLOCK* algorithm [36], admission-control based page replacement [13, 53, 152], ideas related to the fair sharing of cache resources [103, 22, 145, 89], and bandwidth metering [181, 106, 174, 133]. We note that ONSim makes implementing and evaluating replacement policies for an HMA simple and opens the door to much future work that more carefully evaluates prior ideas in this new context.

3.5.3 HMA Performance Simulation

Accurate simulation of HMA performance is challenging. Traditional cycle-accurate simulators (e.g., Gem5 and MARSSx86) are unfit for this purpose because of the slow simulation speed and limited support for running complex workloads. One alternative is to statically map applications' memory address spaces into a custom ramdisk-like character device and inject delay whenever the programs access these data from the device [166, 19, 162]. Another approach is to model performance of NVM by repeatedly injecting delay during a program's execution based on estimating the number of NVM accesses using performance counters [49, 169]. Moneta [20], HASTE [19], and FAME [125] instead use FPGAs to capture NVM performance events. Thermostat uses PTE manipulation techniques; it samples a small them [6]. It hence can only estimate percentage of hot pages in application's dataset, and lacks the capability to simulate architectural and performance events in the HMA. HMEP simulate NVM performance using proprietary BIOS firmware and special CPU microcodes [124]. FlashVM [151], NVMalloc [171], NV-Heaps [32], and SSDAlloc [11] use off-the-shelf devices to approximate the performance of NVM.

Unfortunately, none of the previous approaches can effectively simulate HMA performance. The custom ramdisk approach assumes static memory mapping. As a result, it is not capable of gathering access frequency information or simulate dynamic data migration between DRAM and NVM. Delay injection per epoch omits many important architectural details, such as data

location, DRAM replacement, bandwidth limitation, etc. All but FPGA- and PTE-based approaches require changes to existing code, which significantly limits them in running complex workloads. FPGA-based approaches are difficult to set up and are less flexible in modeling complex page management algorithms compared to software approaches. PTE manipulation techniques are ideal, but the capability of the sampling approach [6] is limited and cannot accurately simulate architectural events.

3.6 Summary

In this work, we enable accurate and low-overhead evaluation of heterogeneous memory architectures (HMA) to study cost-efficiency improvements for warehouse-scale computing. We design and implement ONSim, an OS-level NVM Simulator that is transparent to all workloads and runs unmodified binaries and libraries. We demonstrate that ONSim is highly accurate ($\leq$3.3% relative delay error) and has very small performance overhead ($\leq$3%) when simulating a range of interesting and relevant NVM access latencies (which corresponds to realistic devices [3, 2, 161]). We use ONSim to quantify the performance impact of NVM on memcached, a representative WSC workload. We show that while memcached performance and QoS degrade with naive data management schemes, simple mechanisms such as *CLOCK* replacement algorithm and *Idle Distance* based page admission are able to achieve 97.5% of the baseline memcached throughput with 40% of memory provisioned using NVM. We further investigate the cause of performance degradation with task colo-

cation and propose *Miss Ratio Fraction* based DRAM occupancy control and *Write Bandwidth Metering*. Results show that these techniques are effective in mitigating performance interference from colocated tasks, achieving 97.6% of the baseline memcached throughput with 50% of memory provisioned using NVM. Finally, we identify additional architectural and microarchitectural enhancement opportunities to further improve HMA performance and QoS.

Chapter 4

Enforcing QoS for Accelerated Machine Learning Systems

Development and deployment of machine learning (ML) accelerators in Warehouse Scale Computers (WSCs) demand significant capital investments and engineering efforts. However, even though heavy computation can be offloaded to the accelerators, applications often depend on the host system for various supporting tasks. As a result, contention on host resources, such as memory bandwidth, can significantly discount the performance and efficiency gains of accelerators. The impact of performance interference is further amplified in distributed learning for large models.

In this work, we study the performance of four production machine learning workloads on three accelerator platforms. Our experiments show that these workloads are highly sensitive to host memory bandwidth contention, which can cause 40% average performance degradation when left unmanaged. To tackle this problem, we design and implement Kelp, a software runtime that isolates high priority accelerated ML tasks from memory resource interference. We evaluate Kelp with both production and artificial aggressor workloads, and compare its effectiveness with previously proposed solutions. Our evaluation

shows that Kelp is effective in mitigating performance degradation of the accelerated tasks, and improves performance by 24% on average. Compared to previous work, Kelp reduces performance degradation of ML tasks by 7% and improves system efficiency by 17%. Our results further expose opportunities in future architecture designs.

4.1 Background

4.1.1 Target Accelerator Use Case

We focus on the use case in which the accelerator is used by a single application while the CPU is shared by multiple applications. This is in contrast to previous work that assumes multiple workloads share the same accelerator at the same time [24, 23]. While our assumption is different to that of previous work, we find this use case to be very common in production environments. Two main factors lead to our usage model. First, we observe in our measurements that performance of the accelerated workloads is mostly bottlenecked by accelerator memory BW. This is also confirmed by Jouppi et al. [95], who arrived at a similar conclusion through detailed roofline analysis. As a result, time-multiplexing accelerators is unlikely to improve performance. Second, given the large datasets of many production ML workloads, accelerator memory is often not large enough to fit the data of multiple workloads. Time-multiplexing is hence infeasible due to the large overhead of data spilling. However, we show in this work that, even in the absence of accelerator resource interference, colocation of accelerated tasks and CPU tasks can still lead to

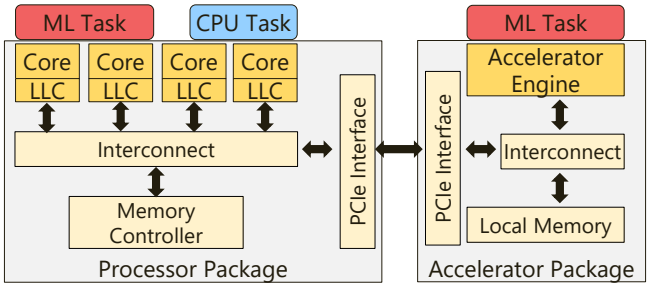


Figure 4.1: Architecture of an accelerated platform.

large performance degradation.

4.1.2 Accelerator-CPU Interaction

We assume that each machine is shared by a *high priority ML task* and multiple *low priority CPU tasks*. Accelerated tasks typically have high priority because accelerators are often capable of higher computational throughput and efficiency compared to CPUs, and customers are usually charged more for using these resources ([66, 8]). Figure 4.1 shows the general architecture of an accelerated platform. While the ML workload offloads its heavy computation to the accelerator, part of the computation still runs on the CPU. These tasks are often memory-intensive due to their large datasets. As a result, under heavy memory BW contention, the CPU tasks in the accelerated ML workload can easily become the bottleneck of the entire ML application.

One example of such dependence on the CPU is the *parameter server* in distributed machine learning. In this configuration, a cluster of worker tasks executes the TensorFlow graph using different training data, while the shared parameters are hosted by multiple parameter server instances spanning

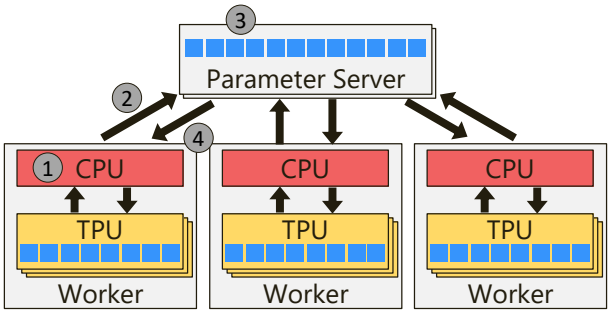


Figure 4.2: Example workflow of distributed TensorFlow training with parameter servers.

across nodes [65, 41]. Figure 4.2 shows an example workflow. Each worker first computes the gradients of the variables by gathering results from all local accelerators ① and sends them to the parameter servers ②. Each parameter server then aggregates the gradients and computes the updated training variables using a pre-defined optimizer ③ [67]. Finally, updated variables are copied to each worker at the end of the iteration ④. The parameter server tasks are memory intensive and can easily become the performance bottleneck of the entire service, as we show later in the paper. Furthermore, due to the distributed nature of the computation model, performance degradation on any parameter server instance is amplified at the service-level [40]. Another example of CPU assistance is the *in-feed operation*, in which the host processor is responsible for interpreting and reshaping the input data before it is consumed by the accelerators [68]. It is also common for CPUs to handle miscellaneous computation tasks, which take advantage of CPUs' capability to handle irregular and complex instruction streams. For example, *beam search* is a commonly used algorithm to reduce the search space in machine transla-

tion programs [107]. Instead of greedily following the best candidate in each iteration, beam search sorts partial solutions and expands on a subset of best candidates [107]. Slowdown of the above tasks can starve the accelerators and significantly degrade performance of the application. The machine learning workloads used in this work include all three types of interaction discussed above (Section 4.2).

The supporting role of CPUs in accelerator platforms brings new challenges in system design and resource management. Specifically, host memory bandwidth (BW) interference can cause significant performance and Total Cost of Ownership (TCO) loss for the entire accelerated application. To demonstrate the impact of such interference, we show the execution timeline of a production RNN inference server running on the TPU platform [95]. Each query to the server in this workload is broken down into multiple iterations and Figure 4.3 shows one such iteration. We further break down the execution time into different phases, which include CPU-assist tasks, CPU-TPU communication, and TPU computation. We then show the execution timeline with and without a DRAM aggressor. Note that for this illustrative example, we generate requests serially to simplify the presentation of the trace.

The results show that, while the CPU-accelerator interaction is not sensitive to the DRAM BW aggressor, the CPU-intensive phases are highly sensitive to memory BW interference. Execution time for CPU-intensive phases increases significantly by up to 51%. As a result, service-level tail latency increases by over 70%. In this work, we first use synthetic workloads to confirm

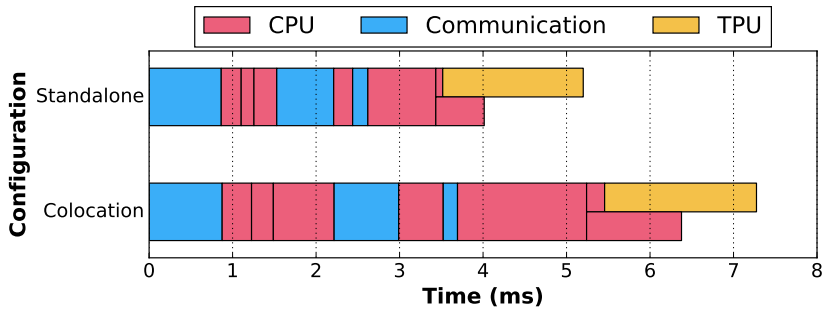


Figure 4.3: RNN inference server execution timeline on a TPU platform. Execution time for CPU-intensive phases increases by 51% under heavy contention. The interleaving among different phases in the execution timeline is on the order of sub-milliseconds to millisecond.

memory BW interference to be the dominant factor that causes performance degradation. We further observe similar performance degradation with benchmarks across platforms. If left unmanaged, the resulting performance degradation can discount the efficiency gain of accelerators and cause significant loss of the capital investments in accelerator development and deployment.

Such contention also highlights the needs for robust and efficient performance isolation mechanisms. As shown in Figure 4.3, the interleaving among different steps in the execution timeline is on the order of sub-milliseconds to millisecond, which is too fine-grained for effective polling-based reactive core throttling, such as the approaches proposed in previous work [119, 182, 176]. Request pipelining (as used in production environments and evaluation in Section 4.4) further exacerbates this issue because the timeline becomes more complicated due to phase interleaving and overlapping. Because of the above reasons, hardware-based solutions, such as fine-grained memory BW QoS (e.g.,

request-level prioritization), are much better suited to provide the much needed BW interference protection in future system architectures.

4.1.3 Managing Interference at WSC Scale

Contention for host resources between the ML tasks and low priority CPU tasks causes significant performance degradation and efficiency loss for the accelerated workload. The resource contention problem is further exacerbated when deploying accelerators at the WSC scale by the following factors.

(a) Accelerated workloads can span multiple nodes and cross-node synchronization is often necessary for each iteration of variable computation [5]. As a result, service-level performance of distributed workloads is even more susceptible to interference due to "tail amplification" [40].

(b) ASIC accelerators are largely programmable, and different accelerated workloads can have different levels of sensitivity to resource contention and different requirements on host resources. As a result, an ideal solution needs to handle different application behaviors (e.g., compute and memory intensity, interaction time granularity, etc.) at runtime.

(c) There is a large number of CPU workloads with drastically different performance characteristics in WSC production environments [97]. Performance of any colocated accelerated ML tasks can be severely impacted without effective and adaptive isolation mechanisms.

Workload	Platform	Description	CPU-Accelerator Interaction	CPU Intensity	Host Memory Intensity
RNN1 Inference	TPU	Natural language processing	Beam search	Medium	Low
CNN1 Training	Cloud TPU	Image recognition	Data in-feed	Low	Low
CNN2 Training	Cloud TPU	Image recognition	Data in-feed	High	Medium
CNN3 Training	GPU	Image recognition	Parameter server	Low	High

Table 4.1: Accelerated ML platforms and production workloads. Detailed measurements are not publishable due to confidentiality concerns.

4.2 Accelerated Machine Learning Workloads

We use four ML workloads that run on three accelerated platforms in this study (see Table 4.1; details of the workloads are, unfortunately, confidential because they are used in production). In this section we first describe the platforms and workloads. We then analyze their sensitivity to different types of shared resources.

4.2.1 Platforms and Workloads

The TPU platform is equipped with the first generation *Tensor Processing Unit*. The TPU is a PCI-e based accelerator that targets inference workloads. The execution engine is a Matrix Accumulation unit with peak throughput of 92 TFLOPS [95]. We run an RNN-based natural language processing inference workload (**RNN1**) on the TPU platform. Requests are generated in a pipelined fashion to ensure high utilization of all computing resources. Specifically, we sweep the query throughput (measured in queries-per-second or QPS) and analyze the tail latency. The target throughput we use in the paper is at the knee of the tail latency curve. The sweep plot is omitted for brevity.

Cloud TPU is the second-generation *Tensor Processing Unit*. A Cloud TPU device has a peak throughput of 180 TFLOPS and 64 GB of high-bandwidth on-chip memory [42]. Compared to the first generation TPU, the Cloud TPU can also execute both training and inference workloads in the cloud. We include two CNN training benchmarks (**CNN1** and **CNN2**), which have different CPU and memory intensities in the workload mix.

Finally, we also study GPU platforms which are widely used for training ML models (**CNN3**). While **CNN3** is based on a distributed TensorFlow architecture, we only use one GPU worker in the experiment in order to reduce noise caused by the network. The training steps of this benchmark are processed in lock-step among all distributed workers and parameter servers, and latency of the slowest parameter server can bottleneck the service-level throughput [40]. As a result, performance degradation caused by resource interference measured in production configurations is mostly similar to what is observed in our evaluation.

4.2.2 Interference Sensitivity

As we discussed in Section 4.1.1, we focus on the use case in which one high priority application has exclusive access to the accelerators. However, low priority CPU tasks can still interfere with the accelerated task by contending for shared resources, including in-pipeline resources and private caches shared through simultaneous multi-threading (SMT), the last-level cache, and main memory BW. To identify the performance bottlenecks and quantify sensitivity,

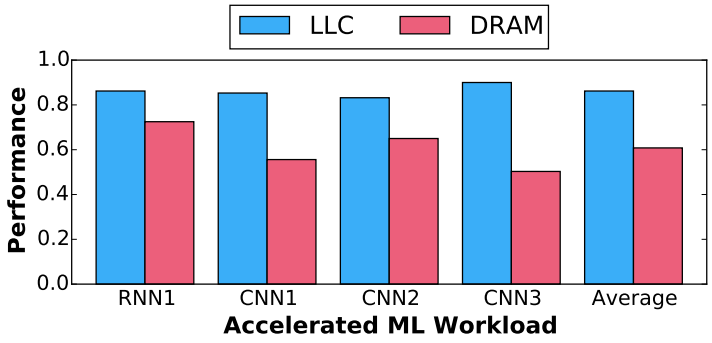


Figure 4.4: Workload sensitivity to shared resource interference. Performance is normalized to no interference.

we use the following two synthetic workloads (we will introduce more production workloads in Section 4.4). **LLC** contends for the last-level cache, all caches closer to the CPUs, and in-pipeline resources (SMT is enabled in all experiments). Its dataset size is just small enough to fit in the LLC on the CPU of each platform. **DRAM** contends for DRAM BW. It traverses a large array that doesn't fit in the LLC. On our multi-socket platforms, we use core affinity and control the NUMA policy (`numactl`) to ensure that all threads and data reside in the same socket as the accelerated workload.

Figure 4.4 summarizes the results of the sensitivity study. On average, LLC resource contention causes a noticeable performance degradation of 14%. However, colocation with the DRAM aggressor causes a dramatic 40% performance loss on average. This result is surprising because the accelerators are already designed to minimize interactions with the host CPUs [95]. While not shown in our evaluation, we performed a sweep analysis of the ratio of computation and communication between accelerator and host CPU for CNN1 and

CNN2. The same level of sensitivity is observed across the spectrum for both workloads.

In the rest of this paper, we focus on mitigating performance interference caused by DRAM BW contention because it dominates the performance degradation. We use a combination of production and synthetic batch workloads in our evaluation. LLC interference is addressed by dedicating an LLC partition to accelerated tasks using Intel's Cache Allocation Technology (CAT) [86]. We also identify another performance bottleneck in memory traffic that crosses socket boundaries. Related experiments are discussed in Section 4.5.

4.3 Kelp Design and Implementation

The goal of Kelp is to mitigate performance impact of DRAM BW interference as observed in the previous section. While much work has been done on various performance isolation mechanisms (Section 4.6), we are constrained to techniques that are already implemented today. A commonly used approach explored by previous work is core throttling [119, 182, 176]. It relies on a software runtime to detect performance interference and throttles tasks at core granularity. While effective, core throttling is not efficient to fully exploit the available bandwidth due to the coarse throttling granularity and time granularity.

To further improve system efficiency, we leverage NUMA Subdomain mechanisms to separate each socket into two NUMA subdomains, enabling

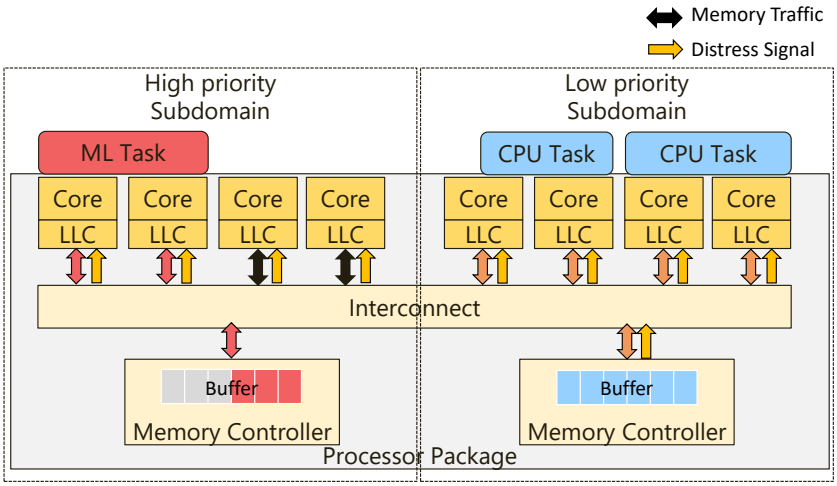


Figure 4.5: NUMA subdomain and memory backpressure.

low priority tasks to take full advantage of memory BW resources. We also discover and address challenges in shared memory backpressure [83] and further enhance system throughput with subdomain backfilling. The rest of this section describes the design of Kelp in more detail. Our exploration of existing CPU capabilities also exposes challenges and opportunities for future system architectures (Section 4.5).

4.3.1 NUMA Subdomain Performance Isolation

One of the key CPU mechanisms that Kelp relies on is *NUMA subdomain* performance isolation. Figure 4.5 provides a high-level overview of this feature. The technique splits a physical socket (cores, LLC, interconnect, and memory controllers) into two *NUMA subdomains*, which are exposed to the operating system as two NUMA domains. Memory requests within each NUMA subdomain are handled by the corresponding memory controller. While chan-

nel partitioning has been discussed before for CPU workloads [133], we evaluate it on real accelerated platforms with Intel processors that implement this techniques as *sub-NUMA Clustering* (SNC) [132] and *Cluster-on-Die* (CoD) [131].

As a result of the partition, local memory requests enjoy both lower LLC and memory latency compared to when NUMA subdomain is disabled, although latency for memory accesses to the remote NUMA subdomain will be worse. Because memory traffic for each NUMA subdomain is handled by a different memory controller, we can largely isolate contention for memory resources between the accelerated tasks and low priority CPU tasks by assigning them to two different NUMA subdomains.

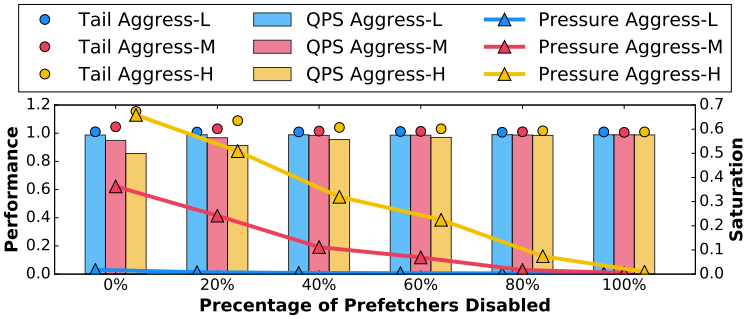
4.3.2 Shared Memory Backpressure

While the NUMA subdomain ideally should provide almost perfect memory isolation, in our experiments with synthetic benchmarks, we still observe noticeable performance degradation. After further investigation, we identify the source of cross subdomain interference to be the shared backpressure mechanism, which is also shown in Figure 4.5. When CPU tasks in the low priority subdomain generate a large amount of memory traffic, requests queue up at the corresponding memory controller and saturate its bandwidth. In such cases, that memory controller broadcasts a distress signal to all CPU cores across the entire socket. After receiving the distress signal, the CPU cores are throttled in order to avoid congesting the interconnection network. Such throttling is essential in most cases to prevent unnecessary delay of other

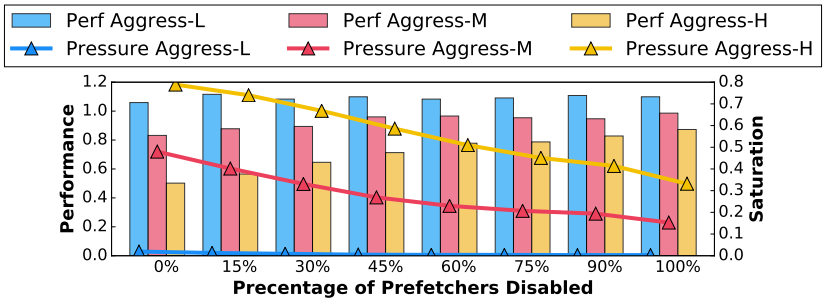
communication over the network. Unfortunately, this mechanism is detrimental in our use case because most of the memory traffic is routed within each NUMA subdomain. Instead, this mechanism actually reduces the effectiveness of the memory interference protection that NUMA subdomains can potentially provide.

Fortunately, system software can measure the level of memory saturation on our hardware using existing hardware performance monitoring infrastructure. Specifically, we use measurements from the performance event `FAST_ASSERTED` from the Intel uncore LLC coherence engine [83]. This event reports the number of cycles in which the distress signal is asserted. We can then quantify *memory saturation* by dividing this cycle count by the number of elapsed cycles between two measurements. To control the memory pressure generated by the low priority CPU cores, we resort to disabling L2 prefetchers for the low priority CPU tasks [168], which significantly reduces memory traffic at the cost of performance loss of low priority CPU tasks.

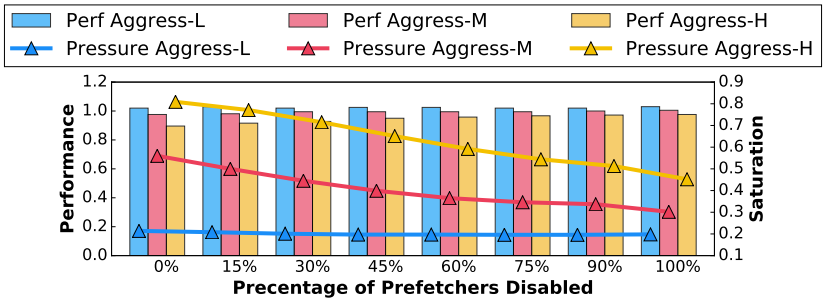
To demonstrate the impact of the global throttling caused by memory backpressure and the effectiveness of toggling prefetchers, we run three accelerated workloads (RNN1, CNN1, and CNN2) with synthetic DRAM aggressors. The aggressors are configured to produce three levels of memory pressure (low, medium, and high). Accelerated tasks and CPU tasks run in separate NUMA subdomains. Performance of the accelerated tasks is normalized to that of a standalone configuration. For each workload mix, we gradually change the number of prefetchers disabled and plot the performance of the accelerated



(a) RNN1



(b) CNN1



(c) CNN2

Figure 4.6: Performance impact of shared memory backpressure and effectiveness of backpressure management with prefetchers toggling. Three levels of aggressiveness of the antagonists (L, M, and H) are experimented with.

task and measured memory saturation. For the RNN1 inference workload, we further plot the 95%-ile tail latency. Results are summarized in Figure 4.6. Each cluster of bars plots ML task performance (and tail latency for RNN1) for the given prefetcher configuration across three levels of memory pressure. Measured memory pressure is plotted as lines using the right axes.

Several observations can be made from the results. First, NUMA subdomains alone cannot provide enough protection. When no prefetchers are turned off, RNN1 QPS decreases by 14% and tail latency increases by 16%. CNN1 and CNN2 suffer a performance degradation of 50% and 10% respectively. Second, turning off prefetchers successfully reduces the performance degradation caused by shared backpressure for most of the workload mixes. Finally, when memory pressure is low, performance of the accelerated tasks may be slightly better than standalone due to the lower LLC and memory access latency associated with enabling NUMA subdomains (e.g., CNN1 and CNN2 performance is 9% and 2% higher than standalone in best cases).

4.3.3 Improving System Throughput

One significant limitation of using NUMA subdomains alone to provide performance isolation is that this degrades total system throughput. Due to the coarse granularity of NUMA subdomains (SNC and CoD), we can only achieve memory BW interference isolation between two subdomains. As a result, this approach can suffer from significant fragmentation of resources (core, cache, and memory). We quantify this performance loss in Section 4.4.

To regain the lost throughput due to fragmentation, we backfill the high priority subdomain with CPU tasks. We show in Section 4.4 that combining backfilling with subdomains can improve system efficiency by 17%. This is largely because CPU tasks in the low priority NUMA subdomain can have high utilization of memory resources with relatively small performance penalty on the colocated ML task. Furthermore, backfilling CPU tasks in the high priority NUMA subdomain enables the system to leverage most of the fragmented resources.

With task backfilling, it is crucial to tightly manage the BW interference within the high priority subdomain. Since the DRAM BW is lower than when NUMA subdomains are not enabled, a similar level of BW interference can potentially cause even higher performance degradation. In this work, we measure the BW consumed by the memory channels that correspond to the high priority subdomain, and throttle CPU tasks when necessary by reducing the number of cores available to the low priority tasks using CPU masks.

4.3.4 Kelp Workflow and Implementation

To put everything together, Figure 4.7 summarizes the architecture of Kelp. Kelp is designed to run with the node-level scheduler runtime (e.g. Borglet [167]) in order to gather necessary task information such as job priority and profile. When applications are first scheduled onto the server, the corresponding profile is loaded by Kelp, which includes high and low watermarks for each measurement. Kelp assigns both accelerated ML tasks and low pri-

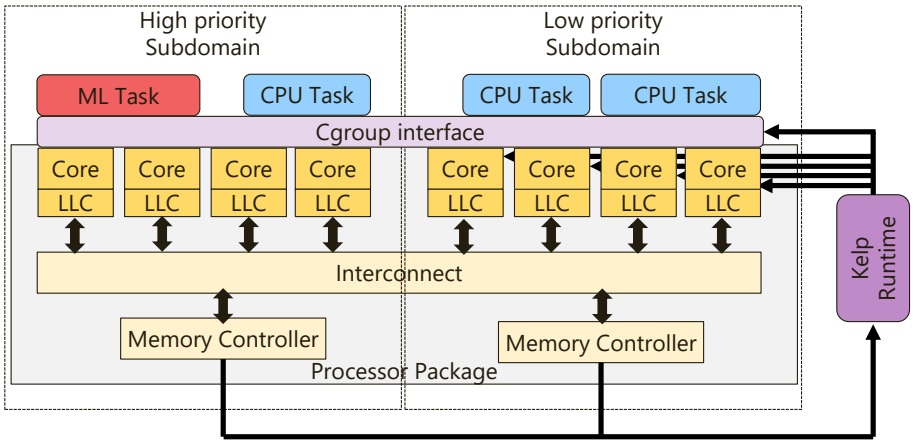


Figure 4.7: Kelp architecture.

Algorithm 1 Kelp Resource Management Algorithm

```
1:  procedure KelpResourceMngt
2:      bw_s, lat_s, sat_s = MeasureSocket()
3:      bw_h = MeasureHiPriority()
4:
5:      if HiBW_h(bw_h) or HiLat_s(lat_s) then
6:          action_h = THROTTLE
7:      else if LoBW_h(bw_h) and LoLat_s(lat_s) then
8:          action_h BOOST
9:      else
10:         action_h = NOP
11:
12:     if HiBW_s(bw_s) or HiLat_s(lat_s) or HiSat_s(sat_s) then
13:         action_l = THROTTLE
14:     else if LoBW_s(bw_s) and LoLat_s(lat_s) and LoSat_s(sat_s) then
15:         action_l = BOOST
16:     else
17:         action_l = NOP
18:
19:     ConfigHiPriority(action_h)
20:     ConfigLoPriority(action_l)
21:     EnforceConfig()
```

ority CPU tasks to the designated subdomains. CPU tasks are prioritized to be assigned to the low priority subdomain. At runtime Kelp makes four types of measurements from the processor: socket-level memory bandwidth, memory latency, memory saturation (Section 4.3.2), and high-priority subdomain bandwidth. Kelp samples system performance every 10 seconds and has negligible performance overhead. The effectiveness of Kelp is not sensitive to the sampling frequency.

Algorithm 1 describes the node level resource management algorithm used by Kelp. Subscripts denote the scope (subdomain or socket) that the corresponding value describes. By comparing measurements from performance counters with the watermarks specified in the application profile, Kelp chooses to boost, throttle, or keep the resource configuration for low priority CPU tasks in each subdomain. Algorithm 2 details the approach we use to configure resources within each subdomain. When throttling the low priority subdomain, we are more aggressive in disabling prefetchers in order to prioritize ML task performance.

4.4 Evaluation

4.4.1 Methodology

We evaluate Kelp with four production ML workloads across three accelerator platforms as listed in Table 4.1. For the colocated CPU tasks, we use a combination of synthetic and production workloads:

- **Stream** traverses a large array that does not fit in the last-level cache of

Algorithm 2 Resource Configuration Algorithms

```
1: procedure ConfigHiPriority(action_h)
2:     if action_h = THROTTLE then
3:         if coreNum_h > minCoreNum_h then
4:             coreNum_h -=1
5:     else if action_h = BOOST then
6:         if coreNum_h < maxCoreNum_h then
7:             coreNum_h +=1
8:
9: procedure ConfigLoPriority(action_l)
10:    if action_l = THROTTLE then
11:        if prefetcherNum_l > 0 then
12:            prefetcherNum_l /=2
13:        else if coreNum_l > minCoreNum_l then
14:            coreNum_l -=1
15:    else if action_l = BOOST then
16:        if prefetcherNum_l < coreNum_l then
17:            prefetcherNum_l +=1
18:        else if coreNum_l < maxCoreNum_l then
19:            coreNum_l +=1
```

any of the platforms.

- **Stitch** is a production batch job that stitches images to form the panoramas for Google Street View.
- **CPUML** is a production CNN training workload based on TensorFlow-Slim [69]. CPU-based training as a low priority task is common because of high resource availability.

To better demonstrate the effectiveness of Kelp, we compare evaluation results of four system configurations:

- **Baseline (BL)**: Task priority is specified through the Borg [167] interface; resource contention is unmanaged.
- **CoreThrottle (CT)**: A competitive resource management configuration that closely mimics mechanisms from previous work[119, 182, 176]. Memory BW interference is managed by limiting the number of cores available to the low priority CPU tasks, while LLC interference is managed by using dedicated LLC partitions to the accelerated tasks.
- **Kelp Subdomain (KP-SD)**: A simplified Kelp implementation that uses only NUMA subdomains (SNC and CoD) and manages global throttling due to memory backpressure by toggling L2 prefetchers for CPU tasks [168].
- **Kelp (KP)**: The full Kelp implementation which further improves system throughput by backfilling CPU tasks.

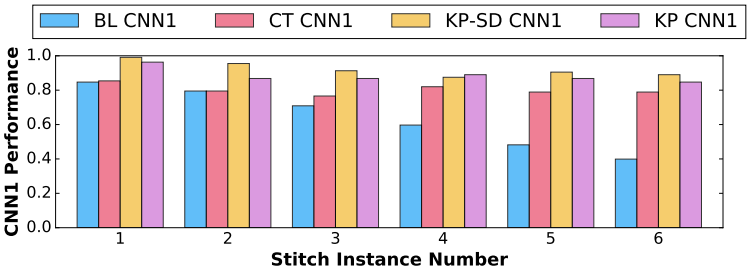
We evaluate the workload mixes on real hardware. Experiment for each configuration and workload mix combination is repeated multiple times and

the median is reported. Performance results are captured from the applications using application-specific metrics. Hardware measurements are made through performance counters. Requests for RNN1 are generated in a parallel and pipelined fashion. While the results are not included in the paper, we sweep load generation configurations for RNN1 and choose a target rate at the knee of the throughput-latency curve. Application profiles for each configuration are derived from sweep analysis with synthetic aggressors.

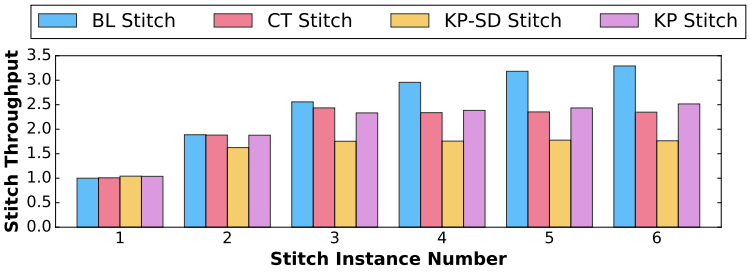
4.4.2 Benchmark Case Studies

To understand the effectiveness of Kelp, we perform a configuration sweep analysis that compares the performance of the four configurations for all workload mixes. In this section, we discuss the results for two representative cases.

In the first mix, we run CNN1 with Stitch. This workload mix is interesting because CNN1 is highly sensitive to BW contention and Stitch aggressively contends for BW resources. Figure 4.8a plots CNN1 performance and Figure 4.8b plots Stitch throughput. As the number of Stitch instance increases, Baseline performance of CNN1 decreases by up to 60%. In the mean time, throughput of Stitch keeps increasing as more instances are added. CoreThrottle improves the average performance of CNN1 by 16% while decreasing the harmonic mean of throughput of the low priority Stitch by 11%. Subdomain further improves CNN1 average performance from CoreThrottle by 12%, but Stitch suffers from a significant 25% average throughput degra-



(a) CNN1 performance normalized to standalone performance.



(b) Stitch throughput normalized to Baseline with one instance.

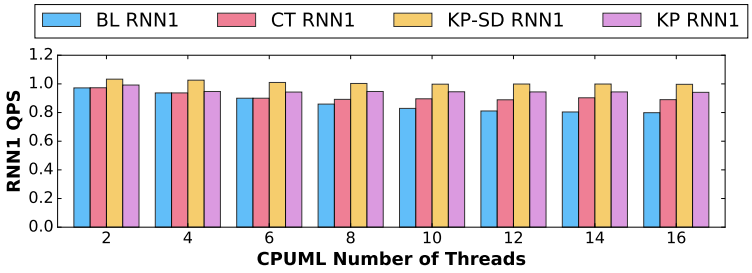
Figure 4.8: Memory pressure sweep CNN1 + Stitch.

dation. In comparison, Kelp improves the average performance of CNN1 from CoreThrottle by 8%, while only reduces Stitch throughput by 9%. We conclude that Kelp achieves higher efficiency for this challenging workload mix compared to previous work because it achieves 8% higher CNN1 performance *and* 2% higher Stitch throughput.

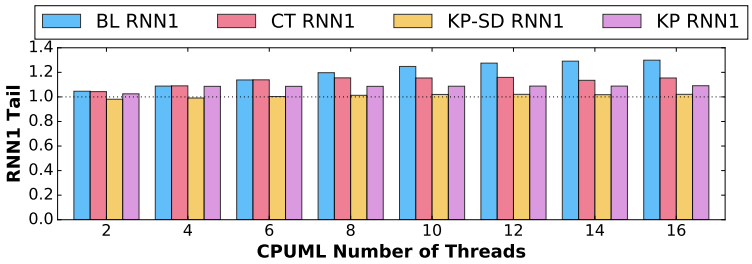
For the second workload mix, we run RNN1 with CPUML. Compared to the previous workload mix, RNN1 is less sensitive to memory BW interference and CPUML is also less aggressive. Figure 4.9a plots the QPS of RNN1, Figure 4.9b plots the tail latency of RNN1, and Figure 4.9c plots the normalized throughput of CPUML. In this workload mix, Baseline performance of RNN1

(both QPS and tail latency) gradually plateaus as more CPUML instances are added and system resources saturate. Comparing with the other three configurations, on average CoreThrottle manages 9% average QPS loss, 13% average tail latency increase, and 5% decrease in CPUML throughput. Subdomain achieves almost no performance degradation for RNN1 at the cost of 33% average CPUML throughput degradation. In comparison, Kelp achieves the best of both worlds with 5% QPS loss, 8% tail latency increase, and 13% average CPUML throughput degradation.

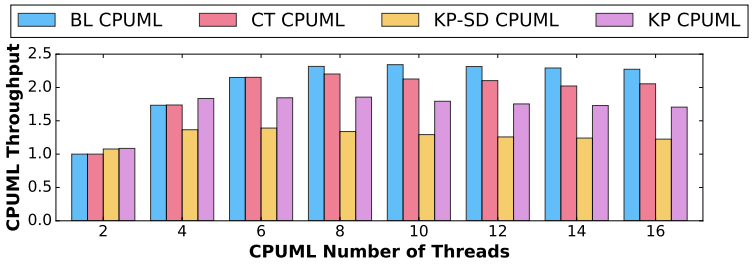
Figure 4.10 and Figure 4.11 show the key parameters that each of the three performance isolation mechanisms use at runtime for the two workload mixes. As a general trend, each mechanism becomes more aggressive in throttling CPU tasks to enforce performance isolation as more memory BW contention is introduced to the system. Overall, the second workload mix exerts less stress on memory BW and the system is throttled less; in Figure 4.11b the vanilla Subdomain configuration is able to achieve enough isolation without having to toggle any prefetchers off. However, Kelp is able to consistently achieve better performance isolation despite different levels of BW sensitivity and interference. Comparing the number of cores allocated for CPU tasks between CoreThrottle and Kelp (Figure 4.10a and Figure 4.10c, Figure 4.11a and Figure 4.11c), Kelp enables the CPU tasks to use more resources and achieves higher system efficiency.



(a) RNN1 QPS normalized to standalone performance.

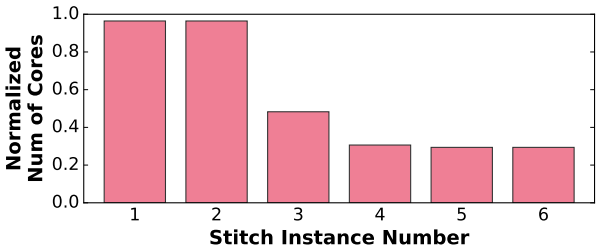


(b) RNN1 95%-ile tail latency normalized to standalone.

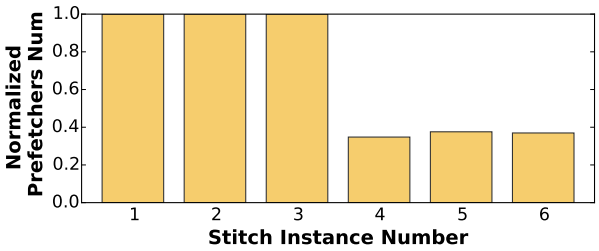


(c) CPUML throughput normalized to Baseline with two threads.

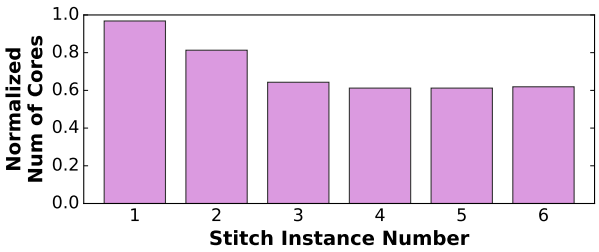
Figure 4.9: Memory pressure sweep RNN1 + CPUML.



(a) CT cores allocated for CPU tasks.

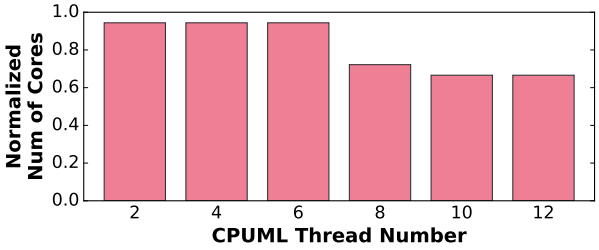


(b) KP-SD prefetchers for CPU tasks.

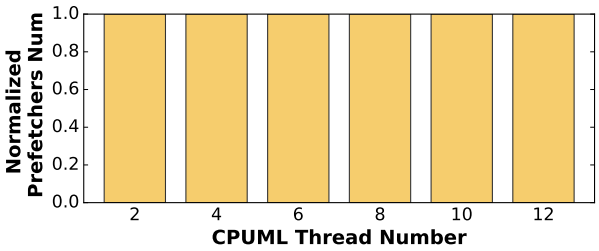


(c) KP cores allocated for CPU tasks.

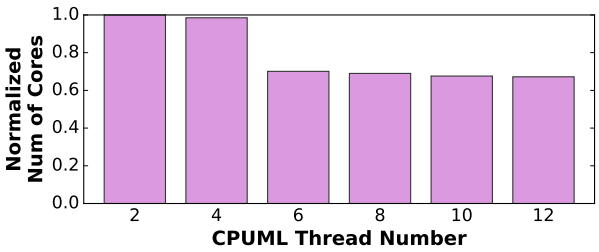
Figure 4.10: Parameters for three performance isolation configurations for CNN1 + Stitch.



(a) CT cores allocated for CPU tasks.



(b) KP-SD prefetchers for CPU tasks.



(c) KP cores allocated for CPU tasks.

Figure 4.11: Parameters for three performance isolation configurations for RNN1 + CPUML.

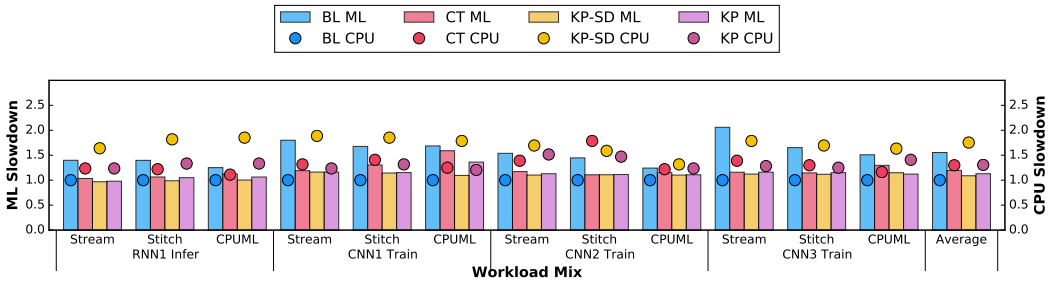


Figure 4.12: ML and CPU task performance results.

4.4.3 Overall Results

Figure 4.12 summarizes the evaluation results for all workload mixes. We plot ML workload slowdown on the left axis (average computed as arithmetic mean) and CPU workload slowdown on the right axis (average computed as harmonic mean). Compared to Baseline, Kelp reduces the slowdown of accelerated ML tasks by 43%, at the cost of 24% CPU task throughput loss. Compared to previous work, as represented by CoreThrottle, Kelp reduces the slowdown of ML tasks by 7% while achieving the same CPU throughput. Compared to Subdomain, Kelp increases ML task slowdown by 4%, but achieves 19% higher CPU task throughput.

To quantify the efficiency achieved by each runtime solution, we define a new metric that represents the tradeoff between ML and CPU task performance. Specifically, we define the efficiency of a runtime to be the ratio of *performance gain of high priority ML tasks compared to Baseline*, and *throughput loss of CPU tasks compared to Baseline*. This metric can also be interpreted as the ML task performance gain per unit of CPU task throughput

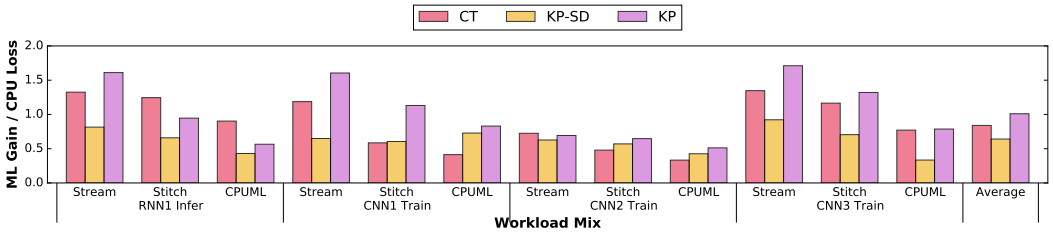


Figure 4.13: Performance tradeoff comparison between CT, KP-SD, and KP.

loss (so higher is better). Note that this metric does not account for tail latency changes (e.g., RNN1 + CPUML as shown above). Figure 4.13 summarizes the results for all workload mixes. Overall, Subdomain has the lowest efficiency because of the fragmentation of resources at coarse granularity. Kelp has higher efficiency compared to CoreThrottle for almost all the workload mixes we tested. While Kelp is less efficient than CoreThrottle for two RNN1 workload mixes, Kelp reduces the tail latency of RNN1 in both cases as we demonstrated with the second case study above. On average, Kelp achieves 17% higher efficiency compared to CoreThrottle, and 37% higher efficiency compared to Subdomain.

4.5 CPU Design Challenges and Opportunities

Our exploration of the capabilities of existing hardware also exposes several challenges in CPU designs for accelerated platforms. We summarize these challenges in this section and provide suggestions for future architectures.

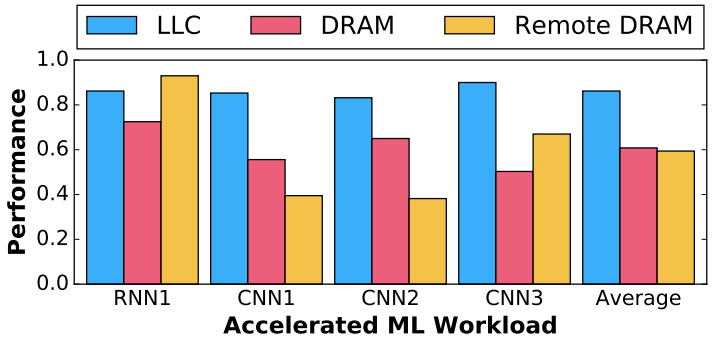


Figure 4.14: Workload sensitivity to remote memory interference compared to LLC and local DRAM.

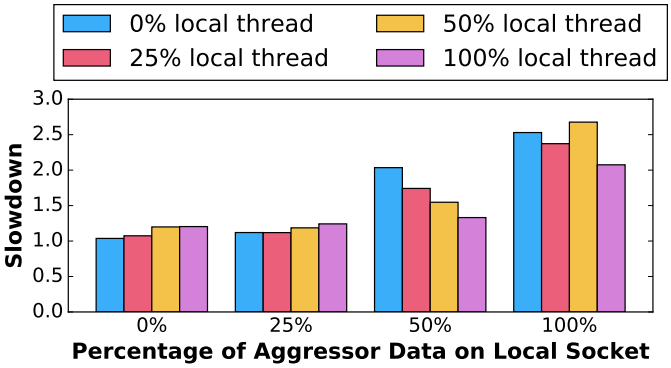
4.5.1 Remote Memory Interference

So far we have studied memory BW interference when both accelerated ML tasks and low priority CPU tasks reside on the same socket. However, on some of the platforms we tested, we notice remote memory traffic that crosses socket boundaries causing exceptionally large performance degradation. To focus on this issue, we use an additional synthetic workload **Remote DRAM**. **Remote DRAM** is similar to **DRAM** with the exception that only some of the data and threads are resident on the local memory socket (where the accelerated ML task resides), while the rest reside in the remote socket. This exercises the inter-processor interface (i.e., UPI [85] and QPI [82]). We observe that, compared to TPU and GPU platforms, Cloud TPU platform (CNN1 and CNN2) are more sensitive to **Remote DRAM** traffic that crosses socket boundaries. Figure 4.14 summarizes the results. Compared to the performance degradation caused by **DRAM**, **Remote DRAM** causes an additional 16% and 27% performance loss for CNN1 and CNN2.

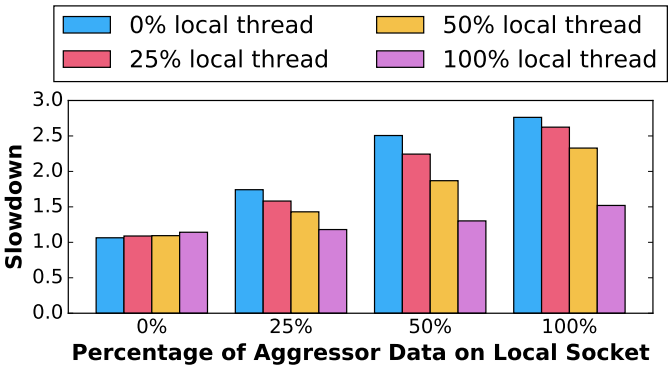
To further understand the performance impact of remote memory traffic on the Cloud TPU platform, we perform a sweep analysis in which we gradually change the percentage of the aggressor's dataset on the socket local to the ML tasks. Within each dataset percentage configuration, we sweep the percentage of threads that reside on the ML tasks' local socket. Results of this experiment are shown in Figure 4.15. The figure shows that remote memory traffic causes even higher slowdown than local memory interference. While the high sensitivity can be an artifact caused by processor-specific implementation choices (e.g., overhead associated with the coherence protocol), future scaling of multi-processor multi-core systems is likely to encounter similar issues in the memory subsystem. These architectural and micro-architectural decisions can have significant system-level performance and utilization implications.

4.5.2 QoS-Aware Prefetching

We show in Section 4.3.2 that prefetching requests can cause high memory pressure. This is shown by the restored accelerated task performance when L2 prefetchers for low priority CPU task cores are partially turned off. While it is well understood that prefetcher requests should not impact the performance of demand memory requests [52], this problem still exists in our scenario because it involves interactions between the memory subsystem and NUMA subdomains. Although Kelp solves the issue by managing prefetcher pressure in system software, this functionality can be integrated into hardware. A hardware-based solution has the advantage of being able to adapt to fast-



(a) CNN1 Sensitivity to Remote Memory Traffic



(b) CNN2 Sensitivity to Remote Memory Traffic

Figure 4.15: Cloud TPU Platform Remote Memory Sweep.

changing system behavior with little performance overhead. It can also guide the aggressiveness of prefetchers based on the immediately-available information of memory resources. Srinath et al. proposed similar ideas for adaptive prefetching but, importantly, miss the key notion of priority across tasks [159].

4.5.3 Global Memory BW Backpressure

We show in Figure 4.6 that NUMA subdomains alone cannot provide performance isolation because of the global memory backpressure mechanism. The slowdown caused by global memory backpressure is an example of how the on-chip interconnect and memory subsystems can together cause unexpected QoS issues. Specifically, the backpressure-based throttling mechanisms do not differentiate requests coming from different subdomains. Ideally, memory backpressure should be sent to the offending hardware thread in order to avoid unnecessary performance loss. Exposing the capability of throttling individual threads to system software (through interfaces such as machine specific registers [86]) can help further improve system utilization. One example is to let users annotate priority of hardware threads so that low priority offending tasks can be throttled first in cases of memory BW contention.

4.5.4 Fine-Grained Memory Isolation

While we show in Section 4.4 that Kelp is able to successfully isolate performance interference and improve system efficiency compared to previous work, it has the limitation of depending on SNC and CoD to achieve most

of the benefits. Therefore, Kelp operates on a relatively coarse granularity. Industry has made significant progress in enhancing the QoS capabilities of server products. For example, Intel introduced the Memory Bandwidth Allocation (MBA) feature in recent architectures that uses a hardware request rate controller [86]. Users can de-prioritize memory-intensive jobs by throttling these jobs' memory requests [86]. Unfortunately, this rate controller appears to throttle traffic from the core to the interconnect, last-level cache, and memory controllers. As a result, any throttling decisions also impact last-level cache BW in addition to main memory BW.

In future work, we will explore the possibility to further improve system efficiency when enforcing QoS for accelerated tasks by hardware-based fine-grained memory performance isolation [164, 91, 174, 181]. Compared to software solutions (e.g., Kelp), hardware techniques can differentiate memory requests from different tasks and handle them with different policies. By exposing the hardware QoS capability to software, system software can further control the tradeoff between ML-task QoS and system throughput.

To estimate the effectiveness of such fine-grained mechanisms, our evaluation results on Subdomain and Kelp in Figure 4.12 approximate an upper bound of what can potentially be achieved. Specifically, fine-grained isolation can achieve ML performance better than Subdomain (at least 4% higher than Kelp), because Subdomain methods increase memory access latency at high BW due to decreased channel interleaving. In the meantime, low priority performance can still be higher than CoreThrottle or Kelp because the hardware

mechanism can achieve higher total memory BW utilization. More importantly, hardware solutions can provide more robust performance by adapting to program behavior changes faster (we demonstrate examples of such opportunities in Figure 4.3), which is critical for high priority tasks that have short execution time deadlines.

4.6 Related Work

4.6.1 Wide Adoption of Accelerators

Recent years have seen the increasingly wide adoption of accelerators, especially for ML applications due to their inherently high computation and data intensity. Notably, GPUs are frequently used in various ML frameworks (e.g., [5, 94, 33, 59]) and applications (e.g., [108, 76, 160]). Jouppi et al. describe the first generation TensorFlow Processing Unit (TPU) that targets inference for neural network applications [95]. The recent release of the Cloud TPU further expands the capability of Google's ML accelerator to training and to scale out using high-speed interconnection [42]. Many other ASIC neural network accelerator designs (e.g., [21, 26, 29, 28]) and optimizations (e.g., [70, 71, 7, 175]) have been proposed while other work explores the option of using FPGA-based solutions (e.g., [141, 142, 56, 55]).

Many other disciplines have adopted accelerators to improve performance and energy efficiency for mission-critical tasks. Putnam et al. explore the option of using FPGAs to accelerate Bing's ranking stack [143]. Khazraee et al. study designs of ASIC-based acceleration schemes for video transcod-

ing and cryptocurrency systems [123, 102]. Baidu and Xilinx develop FPGA acceleration services to accelerate encryption and decryption, in addition to deep learning applications, for public Cloud services [63]. Intel also announced Programmable Acceleration Card product that uses FPGAs to improve both performance and efficiency of WSC workloads [84]. Custom accelerators are also being adopted in the super-computing community. The Cray XT5H integrated CPUs and FPGAs with AMD's HyperTransport interconnect [37]. Tianhe-2A uses a proprietary accelerator to improve performance and energy efficiency [57]. The wide adoption of accelerators further emphasizes the importance of enforcing QoS for accelerated workloads.

4.6.2 Accelerator QoS and Utilization

Accelerator QoS and utilization have been studied with different assumptions in the past. Chen et al. study QoS for accelerators, assuming that the accelerators are time-multiplexed between several tasks that are categorized into two priority groups [24, 23]. *Baymax* focuses on performance interference caused by queuing delay and PCI-e BW contention [24]. It predicts task durations using linear regression and KNN, and re-orders tasks based on the prediction results to enforce QoS targets. *Prophet* further considers intra-accelerator memory BW and profiles each application with testing inputs to estimate runtime memory BW usage [23]. Shen et al. focus on the utilization of arithmetic units for FPGA-based CNN accelerators and propose to partition FPGA resources to improve utilization [155].

However, we assume in this work that each accelerator device can be subscribed by only one application at a given time. As shown in the roofline model analysis in [95], performance for production workloads is almost always bound by accelerator memory BW instead of computational throughput, so there is little motivation to enable the time sharing of accelerators across applications. Doing so will also increase the complexity of on-chip memory management and potentially data management overhead. Also, while not discussed in the evaluation, we do not observe PCI-e BW constraining performance of the profiled workloads. On the other hand, we demonstrate that resource interference of host memory can cause significant performance degradation across various production workloads and accelerator types.

4.6.3 System Performance Isolation

There is a large body of work on performance interference and isolation for tasks of different priorities, the most related of which monitor and manage shared CPU [96, 126, 172, 176, 119, 182, 112, 78, 100, 87, 88, 91] and memory [133, 164, 91, 174, 181] resources at runtime through various mechanisms.

Kambadur et al. conduct a study of application interference using Google's production datacenter workloads [96]. Mars et al. use microbenchmarks to measure workload sensitivity and stress for the memory subsystem and schedule tasks accordingly [126, 172]. Zhang et al. propose to use CPI data to identify interference issue and throttled the interfering tasks with CPU capping [176]. Heracles is a feedback-based controller that uses architectural tech-

niques to guarantee that high priority tasks meet their latency targets [119]. Zhu et al. propose to convert latency headroom for high priority tasks to improved system performance [182]. Jacob et al. and Hsu at al. study the tail latency of memcached and address interference problems using an improved kernel scheduler [112] and fine-grained voltage boosting [78]. Kasture et al. propose a cache partitioning technique to balance the tail latency of high priority tasks and system throughput [100].

Muralidhara et al. propose to reduce memory interference through channel partitioning [133]. Usui et al. [164] and Jeong et al. [91] focus on memory request scheduling policies for SoC memory controllers. Yun et al. implement a memory BW reservation scheme to provide memory isolation [174]. Zhou et al. study memory inter-arrival time traffic shaping and proposed to avoid congestion caused by bursty traffic through BW metering [181].

Kelp builds on many of the ideas proposed by previous work (e.g., cache partitioning [100], core throttling [119, 182, 176], and channel partitioning [133]). However, we identify the new problem of performance interference in accelerated machine learning platforms due to fine-grained interaction between CPUs and accelerators. We successfully apply a combination of the above techniques in this new context. We show that these techniques, when enabled by hardware, can effectively tackle the problem of accelerator QoS which was not present before. Our detailed profiling of production workloads also shows additional opportunities to further improve system utilization and QoS for accelerated platforms through hardware-based performance isolation.

4.7 Summary

In this work, we study the performance interference between high priority accelerated ML tasks and low priority CPU tasks. We use three accelerated platforms and experiment with four production ML workloads. Our experiments with synthetic workloads show that these production ML workloads are highly sensitive to memory BW contention. Specifically, while core resource contention causes a noticeable performance degradation of 14%, resource contention for DRAM BW causes a 40% performance loss on average. To address the resource interference problem, we design and implement Kelp, a software runtime that isolates high priority accelerated ML tasks from memory resource interference. Kelp uses existing hardware features such as cache partitioning, NUMA subdomains, and memory pressure management by toggling prefetchers. We evaluate Kelp with both production workloads and synthetic aggressors and compare its effectiveness with a previously proposed solution. Results show that Kelp is effective in mitigating performance degradation of the accelerated tasks and improves their performance by 24% on average. Compared to previous work, Kelp reduces performance degradation of ML tasks by 7% while achieving the same throughput from low priority CPU tasks, and increases system efficiency by 17%.

The wide adoption of accelerators creates exciting opportunities to evolve traditional system architectures. Our work focuses on node-level runtime mechanisms and demonstrates multiple challenges posed by high-performance accelerators. Specifically, we show that further exploring the design space of

fine-grained memory performance isolation can potentially enable better trade-off between performance and QoS of high priority accelerated tasks and total system throughput. In future work, we would like to continue exploring the architectural and micro-architectural opportunities to improve efficiency and performance for accelerated systems. We would also like to study the QoS impact of cluster-level task scheduling in the WSC environment.

Chapter 5

Conclusion

This dissertation presents a set of cross-layer mechanisms that improves the utilization and cost-efficiency of datacenter resources, which are driven by the key observation on the statistical behavior of WSC workloads. Specifically, I design and implement mechanisms that leverage this observation by carefully exploiting spare hardware resources at runtime through cross-layer techniques. I evaluate these mechanisms with real-world workloads and show that they can achieve better tradeoff between performance and QoS of latency-critical tasks and throughput of best-effort tasks. Overall, this dissertation has three main components:

Multicore Systems. I explore opportunities to improve the efficiency of task colocation on multicore systems in Chapter 2. The main insight is that by reshaping the latency distribution curve, performance headroom of LC jobs can be effectively converted to improved BE throughput. I develop, implement, and evaluate a runtime system called Dirigent that achieves this goal with existing hardware. Specifically, Dirigent predicts application performance with a lightweight and accurate predictor, and reconfigures hardware resources at runtime to adapt to resource interference.

Heterogeneous Memory Architecture. To improve the cost-efficiency of the memory subsystem, I explored opportunities enabled by Heterogeneous Memory Architecture (HMA) in Chapter 3. I develop and build ONSim, an OS-level NVM simulator, which transparently executes unchanged binaries. I further propose a set of mechanisms that exploits the performance characteristics of the target workloads and evaluated them with ONSim. These mechanisms, while having negligible overheads, significantly improve DRAM caching efficiency, automatically adjust DRAM occupancy, and avoid NVM bandwidth interference.

Accelerated Machine Learning Systems. In Chapter 4, I study the performance of production machine learning workloads on accelerated systems in WSC environment. I show that accelerated ML tasks are highly sensitive to memory interference due to fine-grained interaction between CPUs and accelerators. I propose a runtime solution called Kelp to isolate performance interference using a set of low-level hardware features. This study also motivates the need for fast and low-overhead fine-grained memory performance isolation mechanisms.

While this dissertation studies opportunities across system components, there is still a large design space to explore in WSC efficiency. The rest of this chapter outlines other opportunities and challenges in future work.

Priority-Aware Microarchitectures. Microarchitectures for CPUs are traditionally designed with performance and power efficiency as the main targeting metrics. However, to further drive the cost-efficiency of WSC, future microarchitectures also need to address the needs of priority-aware performance isolation when tasks are colocated. Performance isolation at hardware level has significant advantages over software-based solutions on both effectiveness and robustness. As discussed in Chapter 4, computation in WSC is becoming more heterogeneous and distributed, and the rapid maturity and wide adoption of accelerators further highlight this challenge. While much work has been done in studying and enforcing job fairness of specific system components (e.g., [133, 164, 91, 174, 100, 145, 150, 81]), no holistic solution has been proposed to enable truly safe task colocation for high-priority tasks at low overhead. Many open research opportunities branch off from this topic, such as holistic microarchitectural isolation mechanisms, SW/HW interface for isolation capabilities, and designs of runtime system that leverage these features.

QoS-Aware WSC Scheduler and Interaction with Runtime. Task scheduling is critical to achieving high utilization and low contention. While there is a large body of previous work on WSC scheduler [167, 153, 75, 64, 140, 48, 126], the interaction between scheduler and runtime is often not considered. In most cases, the QoS runtime can effectively mitigate performance interference and reduce QoS impact on latency-critical tasks. A QoS-aware

scheduler that cooperate with the node-level runtime will have more freedom in terms of number of available node candidates to schedule a task on. This is likely to help the scheduler to both improve schedule quality and achieve higher resource utilization. A potential approach to this problem is to analyze the tradeoff between WSC performance benefits and scheduler overhead across different extent of scheduler-runtime cooperation. There are also significant challenges in designing scheduling algorithms that can explicitly describe and accommodate the resource requirements and interactions of tasks at low latency and high throughput.

Bibliography

[1] Graph500. http://www.graph500.org.

[2] Intel optane memory series. http://ark.intel.com/products/97544.

[3] Intel optane ssd dc p4800x series. http://http://www.intel.com/content/www/us/en/\solid-state-drives/optane-solid-state-drives-dc-p4800x-series.html.

[4] Linux control groups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[5] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[6] Neha Agarwal and Thomas F Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. 2017.

[7] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, 2016.

[8] Amazon. Amazon EC2 pricing. `https://aws.amazon.com/ec2/pricing/on-demand/`, 2017.

[9] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*. ACM, 2012.

[10] Alia Atlas and Azer Bestavros. Statistical rate monotonic scheduling. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*. IEEE, 1998.

[11] Anirudh Badam and Vivek S. Pai. Ssdalloc: Hybrid ssd/ram memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[12] 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. *International Journal of High Performance Computing Applications*, 1991.

[13] Sorav Bansal and Dharmendra S Modha. Car: Clock with adaptive replacement. In *FAST*, 2004.

[14] Luiz Andres Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 2007.

[15] Luiz Andres Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan Claypool, 2009.

[16] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[17] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[18] Dominik Brodowski and Nico Golde. CPU Frequency and Voltage Scaling Code in the Linux kernel.

[19] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[20] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[21] Lukas Cavigelli, David Gschwend, Christoph Mayer, Samuel Willi, Beat Muheim, and Luca Benini. Origami: A convolutional network accelerator. In *Proceedings of the 25th edition on Great Lakes Symposium on VLSI*, 2015.

[22] Jichuan Chang and Gurindar S Sohi. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 402–412. ACM, 2014.

[23] Quan Chen, Hailong Yang, Minyi Guo, Ram Srivatsa Kannan, Jason Mars, and Lingjia Tang. Prophet: Precise QoS prediction on non-preemptive accelerators to improve utilization in warehouse-scale computers. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[24] Quan Chen, Hailong Yang, Jason Mars, and Lingjia Tang. Baymax: QoS awareness and increased utilization for non-preemptive accelerators in warehouse scale computers. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[25] Tao Chen, Alexander Rucker, and G Edward Suh. Execution time prediction for energy-efficient hardware accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[26] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ACM Sigplan Notices*, 2014.

[27] Yixin Chen and Li Tu. Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2007.

[28] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016.

[29] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, et al. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[30] Derek Chiou, Prabhat Jain, Srinivas Devadas, and Larry Rudolph. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference*. Citeseer, 2000.

[31] Jason Clemons, Haishan Zhu, Silvio Savarese, and Todd Austin. Mevbench: A mobile computer vision benchmarking suite. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*. IEEE, 2011.

[32] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[33] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[34] Henry Cook, Krste Asanovic, and David A Patterson. Virtual local stores: Enabling software-managed memory hierarchies in mainstream computing environments. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-131*, 2009.

[35] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *ACM SIGARCH Computer Architecture News*. ACM, 2013.

[36] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968.

[37] Cray. Cray introduces next-generation supercomputers. `http://`

investors.cray.com/phoenix.zhtml?c=98390&p=irol-newsArticle&ID=1073071, November 2007.

[38] Carlos Cunha, Azer Bestavros, and Mark Crovella. Characteristics of www client-based traces. Technical report, Technical Report BU-CS-95-010, Computer Science Department, Boston University, 1995.

[39] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram, Nishant A. Mehta, and Alexander G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 2013.

[40] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 2013.

[41] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. Large scale distributed deep networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, 2012.

[42] Jeffrey Dean and Urs Hölzle. Google cloud tpus. https://www.blog.google/topics/google-cloud/google-cloud-offer-tpus-machine-learning/, August 2017.

[43] Biplob Debnath, Sudipta Sengupta, and Jin Li. Flashstore: High throughput persistent key-value store. *Proc. VLDB Endow.*, 2010.

[44] Biplob Debnath, Sudipta Sengupta, and Jin Li. Skimpystash: Ram space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, 2011.

[45] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, 2007.

[46] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGARCH Computer Architecture News*, 2013.

[47] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 2014.

[48] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[49] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, 2014.

[50] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[51] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. In *ACM Sigplan Notices*. ACM, 2010.

[52] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Prefetch-aware shared resource management for multi-core systems. *ACM SIGARCH Computer Architecture News*, 2011.

[53] Gil Einziger and Roy Friedman. Tinylfu: A highly efficient cache admission policy. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2014.

[54] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *ACM SIGARCH Computer Architecture News*. ACM, 2007.

[55] Clément Farabet, Yann LeCun, Koray Kavukcuoglu, Eugenio Culurciello, Berin Martini, Polina Akselrod, and Selcuk Talay. Large-scale FPGA-based convolutional networks. *Scaling up Machine Learning: Parallel and Distributed Approaches*, 2011.

[56] Clément Farabet, Cyril Poulet, Jefferson Y Han, and Yann LeCun. Cnp: An fpga-based processor for convolutional networks. In *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, 2009.

[57] Michael Feldman. Tianhe-2 supercomputer being upgraded to 95 petaflops. `https://www.top500.org/news/tianhe-2-supercomputer-being-upgraded-to-95-petaflops`, September 2017.

[58] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004.

[59] The Apache Software Foundation. Apache mahout. `http://mahout.apache.org/`, 2017.

[60] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. Badgertrap: A tool to instrument x86-64 tlb misses. *SIGARCH Comput. Archit. News*, 2014.

[61] Peter Xiang Gao, Akshay Narayan, Sagar Karandikar, Joao Carreira, Sangjin Han, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. Network requirements for resource disaggregation. In *Proceedings of OSDI'16: 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[62] Bill Gervasi. NVDIMM-P: A new hybrid architecture. `http://www.discobolusdesigns.com/personal/20160414_gervasi_NVDIMM-P.pdf`,

April 2016.

[63] Silvia Gianelli. Baidu deploys xilinx FPGAs in new public cloud acceleration services. https://www.xilinx.com/news/press/2017/baidu-deploys-xilinx-fpgas-in-new-public-cloud-acceleration-services.html, 2017.

[64] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert NM Watson, and Steven Hand. Firmament: fast, centralized cluster scheduling at scale. In *Proceedings of OSDI16: 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[65] Google. Distributed tensorflow. https://www.tensorflow.org/deploy/distributed, August 2017.

[66] Google. Google cloud platform pricing calculator. https://cloud.google.com/products/calculator/, 2017.

[67] Google. High-performance models. https://www.tensorflow.org/performance/performance_models, August 2017.

[68] Google. Xla: The tensorflow compiler framework. https://www.tensorflow.org/versions/r0.12/resources/xla_prerelease, February 2017.

[69] Sergio Guadarrama and Nathan Silberman. Tensorflow-slim. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/contrib/slim.

[70] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[71] Song Han, Jeff Pool, John Tran, and William Dally. Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems*, 2015.

[72] Johann Hauswald, Michael A Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, et al. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[73] John L. Henning. Spec cpu2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 2006.

[74] Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Don Newell, Vineet Chadha, and Jaideep Moses. Rate-based qos techniques for cache/memory in cmp platforms. In *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009.

[75] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos:

A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011.

[76] Geoffrey Hinton, Li Deng, Dong Yu, George E Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara N Sainath, et al. Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 2012.

[77] Henry Hoffmann, Jonathan Eastep, Marco D Santambrogio, Jason E Miller, and Anant Agarwal. Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments. In *Proceedings of the 7th international conference on Autonomic computing*. ACM, 2010.

[78] Chang-Hong Hsu, Yunqi Zhang, Michael Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, Ronald G Dreslinski, et al. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 2015.

[79] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. Unified address translation for memory-mapped ssds with flashmap. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015.

[80] Ramesh Illikkal, Vineet Chadha, Andrew Herdrich, Ravi Iyer, and Donald Newell. PIRATE: QoS and performance management in CMP architectures. *ACM SIGMETRICS Performance Evaluation Review*, 2010.

[81] Intel. Cache Monitoring Technology and Cache Allocation Technology.

[82] Intel. An introduction to the intel quickpath interconnect, January 2009.

[83] Intel. Intel Xeon processor e5 and e7 v3 family uncore performance monitoring reference manual, June 2015.

[84] Intel. Intel programmable acceleration card with intel arria 10 gx FPGA. `https://www.altera.com/pac`, October 2017.

[85] Intel. Intel Xeon processor scalable family datasheet, volumn one: Electrical, July 2017.

[86] Intel. Intel 64 and ia-32 architectures software developer manuals. `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`, 2018.

[87] Ravi Iyer. Cqos: a framework for enabling qos in shared caches of cmp platforms. In *Proceedings of the 18th annual international conference on Supercomputing*. ACM, 2004.

[88] Ravi Iyer, Li Zhao, Fei Guo, Ramesh Illikkal, Srihari Makineni, Don Newell, Yan Solihin, Lisa Hsu, and Steve Reinhardt. Qos policies and

architecture for cache/memory in cmp platforms. *ACM SIGMETRICS Performance Evaluation Review*, 2007.

[89] Aamer Jaleel, Hashem H. Najaf-abadi, Samantika Subramaniam, Simon C. Steely, and Joel Emer. Cruise: Cache replacement and utility-aware scheduling. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[90] JEDEC. Jedec announces support for nvdimm hybrid memory modules. `https://www.jedec.org/news/pressreleases/jedec-announces-support-nvdimm-hybrid-memory-modules`, 2015.

[91] Min Kyu Jeong, Mattan Erez, Chander Sudanthi, and Nigel Paver. A qos-aware memory controller for dynamically balancing gpu and cpu bandwidth use in an mpsoc. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, 2012.

[92] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Mike Sullivan, Ikhwan Lee, and Mattan Erez. Balancing dram locality and parallelism in shared memory cmp systems. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012.

[93] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Albert Greenberg, and Changhoon Kim. Eyeq: practical

network performance isolation at the edge. In *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[94] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, 2014.

[95] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang,

Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, 2017.

[96] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A. Kim. Measuring interference between live datacenter applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

[97] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Y. Wei, and D. Brooks. Profiling a warehouse-scale computer. In *ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2015.

[98] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[99] Harshad Kasture, Davide B Bartolini Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[100] Harshad Kasture and Daniel Sanchez. Ubik: efficient cache sharing with strict qos for latency-critical workloads. *ACM SIGARCH Computer Architecture News*, 2014.

[101] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *2008 IEEE International Symposium on Workload Characterization*, 2008.

[102] Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor. Moonwalk: Nre optimization in ASIC clouds. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[103] Seongbeom Kim, Dhruba Chandra, and Yan Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, pages 111–122. IEEE Computer Society, 2004.

[104] Wonyoung Kim, Meeta S Gupta, Gu-Yeon Wei, and David Brooks. System level analysis of fast, per-core dvfs using on-chip switching regulators. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. IEEE, 2008.

[105] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010.

[106] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in

memory access behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[107] Philipp Koehn, Franz Josef Och, and Daniel Marcu. Statistical phrase-based translation. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*. Association for Computational Linguistics, 2003.

[108] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[109] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 2013.

[110] Min Lee, AS Krishnakumar, Parameshwaran Krishnan, Navjot Singh, and Shalini Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *ACM Sigplan Notices*. ACM, 2010.

[111] Michel Lespinasse. idle page tracking / working set estimation, http://lkml.iu.edu/hypermai

[112] 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*, EuroSys '14, 2014.

[113] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[114] Chit-Kwan Lin and H. T. Kung. Mobile app acceleration via fine-grain offloading to the cloud. In *6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*. USENIX Association, 2014.

[115] Fang Liu, Xiaowei Jiang, and Yan Solihin. Understanding how off-chip memory bandwidth partitioning in chip multiprocessors affects system performance. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010.

[116] Huan Liu. A measurement study of server utilization in public clouds. 2012.

[117] Daniel Lo, Taejoon Song, and G Edward Suh. Prediction-guided performance-energy trade-off for interactive applications. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[118] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st annual international symposium on Computer architecuture*. IEEE Press, 2014.

[119] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015.

[120] Piotr Luszczek, Jack J Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi. Introduction to the hpc challenge benchmark suite. *Lawrence Berkeley National Laboratory*, 2005.

[121] Jiuyue Ma, Xiufeng Sui, Ninghui Sun, Yupeng Li, Zihao Yu, Bowen Huang, Tianni Xu, Zhicheng Yao, Yun Chen, Haibin Wang, et al. Supporting differentiated services in computers via programmable architecture for resourcing-on-demand (pard). In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2015.

[122] Martin Maas, Tim Harris, Krste Asanovic, and John Kubiatowicz. Trash day: Coordinating garbage collection in distributed systems. In *HotOS*, 2015.

[123] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. ASIC clouds: specializing the datacenter. In *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016.

[124] Jasmina Malicevic, Subramanya Dulloor, Narayanan Sundaram, Nadathur Satish, Jeff Jackson, and Willy Zwaenepoel. Exploiting nvm

in large-scale graph analytics. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, 2015.

[125] Krishna T. Malladi, Mu-Tien Chang, John Ping, and Hongzhong Zheng. Fame: A fast and accurate memory emulator for new memory system architecture exploration. In *2015 IEEE 23rd International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2015.

[126] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011.

[127] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV. ACM, 2009.

[128] Mitesh R. Meswani, Sergey Blagodurov, David Roberts, John Slice, Mike Ignatowski, and Gabriel H. Loh. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[129] Justine Meza, Jichuan Chang, HanBin Yoon, Onur Mutlu, and Parthasarathy Ranganathan. Enabling efficient and scalable hybrid memories using fine-granularity dram cache management. *IEEE Computer Architecture Letters*, 2012.

[130] Rustam Miftakhutdinov, Eiman Ebrahimi, and Yale N Patt. Predicting performance impact of dvfs for realistic memory systems. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2012.

[131] David Mulnix. Intel Xeon processor e5-2600 v4 product family technical overview. `https://software.intel.com/en-us/articles/intel-xeon-processor-e5-2600-v4-product-family-technical-overview`, April 2016.

[132] David Mulnix. Intel Xeon processor scalable family technical overview. `https://software.intel.com/en-us/articles/intel-xeon-processor-scalable-family-technical-overview`, September 2017.

[133] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011.

[134] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access

scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2007.

[135] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *ACM SIGARCH Computer Architecture News*. IEEE Computer Society, 2008.

[136] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*. ACM, 2010.

[137] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. Technical Report 183449, EPFL, 2013.

[138] Oracle. Mysql 5.7 reference manual, replication. `https://dev.mysql.com/doc/refman/5.7/en/replication.html`, 2018.

[139] Mark Oskin and Gabriel H Loh. A software-managed approach to die-stacked dram. In *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 2015.

[140] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-*

Fourth ACM Symposium on Operating Systems Principles, 2013.

[141] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Accelerating deep convolutional neural networks using specialized hardware. *Microsoft Research Whitepaper*, 2015.

[142] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S Chung. Toward accelerating deep learning at scale using specialized hardware in the datacenter. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, 2015.

[143] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, 2014.

[144] Moinuddin K Qureshi, Michele M Franceschini, Luis A Lastras-Montaño, and John P Karidis. Morphable memory system: A robust architecture for exploiting multi-level phase change memories. In *ACM SIGARCH Computer Architecture News*. ACM, 2010.

[145] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2006.

[146] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[147] Arun Raghavan, Yixin Luo, Anuj Chandawalla, Marios Papaefthymiou, Kevin P Pipe, Thomas F Wenisch, and Milo MK Martin. Computational sprinting. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*. IEEE, 2012.

[148] Luiz E Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, pages 85–95. ACM, 2011.

[149] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[150] Daniel Sanchez and Christos Kozyrakis. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*. ACM, 2011.

[151] Mohit Saxena and Michael M Swift. Flashvm: Virtual memory management on flash. In *USENIX Annual Technical Conference*, 2010.

[152] Peter Scheuermann, Junho Shim, and Radek Vingralek. Watchman: A data warehouse intelligent cache manager. 1996.

[153] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 2013.

[154] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 2004.

[155] Yongming Shen, Michael Ferdman, and Peter Milder. Maximizing cnn accelerator efficiency through resource partitioning. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017.

[156] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. Transparent hardware management of stacked dram as part of memory. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014.

[157] Jaewoong Sim, Gabriel H Loh, Hyesoon Kim, Mike O'Connor, and Mithuna Thottethodi. A mostly-clean dram cache for effective hit speculation and self-balancing dispatch. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[158] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *Hot Chips 27 Symposium (HCS), 2015 IEEE*, 2015.

[159] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, 2007.

[160] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[161] Billy Tallis. Samsung at flash memory summit: Mlc z-nand. `http://www.anandtech.com/show/11703/samsung-at-flash-memory-summit-96layer-vnand-mlc-znand-new-interfaces`, August 2017.

[162] NVM Library team at Intel. Persistent memory programming. `http://pmem.io`.

[163] Diablo Technologies. Memory 1, http://www.diablo-technologies.com/memory1/.

[164] Hiroyuki Usui, Lavanya Subramanian, Kevin Chang, and Onur Mutlu. Squash: Simple qos-aware high-performance memory scheduler for heterogeneous systems with hardware accelerators. *arXiv preprint arXiv:1505.07502*, 2015.

[165] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and TN Vijaykumar. Timetrader: Exploiting latency tail to save datacenter energy for

online search. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 2015.

[166] Brian Van Essen, Roger Pearce, Sasha Ames, and Maya Gokhale. On the role of nvram in data-intensive architectures: an evaluation. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012.

[167] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015.

[168] Vish Viswanathan. Disclousure of hardware prefetcher control on some intel processors. `https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors`, September 2014.

[169] Haris Volos, Guilherme Magalhaes, Ludmila Cherkasova, and Jun Li. Quartz: A lightweight performance emulator for persistent memory software. In *Proceedings of the 16th Annual Middleware Conference*, 2015.

[170] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient mrc construction with shards. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, 2015.

[171] Chao Wang, Sudharshan S. Vazhkudai, Xiaosong Ma, Fei Meng, Youngjae Kim, and Christian Engelmann. Nvmalloc: Exposing an aggregate ssd store as a memory partition in extreme-scale machines. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012.

[172] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*. ACM, 2013.

[173] 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*. IEEE, 2013.

[174] 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*, 2013.

[175] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep con-

volutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2015.

[176] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. Cpi2: Cpu performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, New York, NY, USA, 2013. ACM.

[177] Yunqi Zhang, Michael A. Laurenzano, Jason Mars, and Lingjia Tang. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, 2014.

[178] Jiacheng Zhao, Huimin Cui, Jingling Xue, Xiaobing Feng, Youliang Yan, and Wensen Yang. An empirical model for predicting cross-core performance interference on multicore processors. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, 2013.

[179] Li Zhao, Ravi Iyer, Ramesh Illikkal, and Don Newell. Exploring dram cache architectures for cmp server platforms. In *2007 25th International Conference on Computer Design*, 2007.

[180] Yanqi Zhou and David Wentzlaff. The sharing architecture: sub-core configurability for iaas clouds. In *ACM SIGARCH Computer Architecture News*. ACM, 2014.

[181] Yanqi Zhou and David Wentzlaff. Mitts: Memory inter-arrival time traffic shaping. In *Proceedings of the 43rd International Symposium on Computer Architecture. ISCA*, 2016.

[182] Haishan Zhu and Mattan Erez. 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*. ACM, 2016.

Vita

Haishan Zhu was born in Shanghai, China. He started his undergraduate studies in Shanghai Jiao Tong University in 2008. He was transferred to the University of Michigan – Ann Arbor in 2010 and received a Bachelor degree in Computer Engineering in 2012. He started his Doctoral studies at The University of Texas at Austin in September 2012. His research is focused on improving the utilization and cost efficiency of datacenter resources. He worked as a research intern at Huawei in Plano, Texas in 2013 and 2014. He also interned with the Platform Team at Google in Sunnyvale, California in 2016 and 2017.

Email address: haishanz@utexas.edu

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.