Efficient Utilization of Heterogeneous Compute and Memory Systems

by

Hiwot Tadese Kassa

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2022

Doctoral Committee:

Associate Professor Ronald G. Dreslinski, Chair
Assistant Professor Baris Kasikci
Professor Wei Lu
Professor Trevor N. Mudge

Hiwot Tadese Kassa

hiwot@umich.edu

ORCID iD: 0000-0002-8005-9189

Acknowledgments

Several people contributed to the completion of my thesis. I am incredibly grateful to my advisor Ronald Dreslinski for his mentorship and guidance during my Ph.D. He gave me a chance to explore my research interests and was always available to help me shape my ideas. I would also like to thank Valeria Bertacco and Todd Austin for their help in the earlier times of my Ph.D. work.

Over the years, I was able to meet and work with graduate students and lab mates, who made it somewhat easier for me to navigate my time at umich. I want to thank Salessawi Ferede and Abraham Addisie for helping me discover research areas through the multiple brainstorming sessions we had in the lab. I want to recognize Yichen Yang, Vaibhav Gogte, and Tarunesh Verma for the time and effort they put into various experiments for my projects. I also want to acknowledge my friends Vidushi Goyal, Leul Wuletaw, Nishil Talati, and all my lab mates for easing some difficult times during this challenging process.

In this thesis, I was afforded the chance to work at Meta in my internships. While there, I had the privilege of meeting my mentors, Jason Akers and Mrinmoy Ghosh, who were important figures in helping me gain and maneuver the industry experience that broadened my research area. I would also like to thank Ehsan K. Ardestani, Paul Johnson, Zhichao Cao, and many more people at Meta for their endless contributions to my research work.

The infinite love, encouragement, and support bestowed upon me by my friends and family is the ultimate reason for me to have reached this achievement. First, I would like to express my deepest gratitude to my devoted parents, Netsanet Abera and Tadesse Kassa, for their unconditional love and unwavering support. They nurtured my intrinsic quality of curiosity and created a world where I could always be myself, and allowed me to pursue my interests and realize my potential. I am eternally appreciative for my sister, Tarik Tadese Kassa, who is my motivation for everything I do. Thank you for never getting tired of listening to my countless problems in completing my thesis. I thank my sister Zenebwork Mario for always uplifting and believing in me. My little brother Zelalem Tadese Kassa, for being there to bring wisdom and humor into my life. I want to thank Yemsrach Greenly and Ben Greenly for all the holidays and vacations we spent together and for making me feel at home here in the US. I thank my friends Blen Tshaye, Rebecca Yilima, Winta Kefyalew, and Bethel Kebede for all the ups and downs we went through together and for the lengthy "therapy" phone calls whenever I needed them. I want to thank Helen Hagos for being my Ann Arbor family and for all the paper editing. Ann Arbor wouldn't have been fun without you. I also like to thank my friends Hayat Sultan, Nebiyou Elias, and Blen Ali for their continuous motivation and support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Conventional compute and memory systems scaling to achieve higher performance and lower cost and power have diminished. Concurrently, we have diverse compute and memory-demanding workloads that continue to grow and stress traditional systems with only CPUs and DRAM. Heterogeneous compute and memory systems establish the opportunity to boost performance for these demanding workloads by providing hardware units with specialized characteristics. Specialized compute platforms such as GPUs, FPGAs, and accelerators execute specific tasks faster than CPUs, increasing performance and energy efficiency for the particular task. Heterogeneity in the memory systems, such as incorporating various memory technologies like storage class memories (SCMs) alongside DRAM, allows for denser, low-power, and low-cost memories to accommodate data-intensive applications. However, heterogeneous systems have unique characteristics compared to traditional systems. We must carefully design how workloads utilize these units to harness their full benefits. This dissertation presents software and hardware techniques that maximize the performance, energy, and cost-efficiency of heterogeneous systems based on the compute and memory access patterns of various application domains.

First, this thesis proposes ChipAdvisor, a machine learning-based framework, to identify the best platform for an application in the early steps of systems design. ChipAdvisor considers the intrinsic characteristics of applications such as parallelism, locality, and synchronization patterns and archives 98% and 94% accuracy in predicting the best performant and energy-efficient platform, respectively, for diverse workloads when considering a system with CPU, GPU, and FPGA. Second, we propose a heterogeneous memory-enabled system design with DRAM and storage class memory (SCM) for key-value stores, one of the largest workloads in data centers. We characterize an extensive deployment of key-value stores in a commercial data center and design optimal server configurations with heterogeneous memories. We achieve 80% performance increase compared to a single socket platform while reducing the total cost of ownership (TCO) by 43-48% compared to a two socket platform. Third, this dissertation designs MTrainS, an end-to-end recommendation system trainer that utilizes heterogeneous compute and memory systems. MTrainSefficiently divides recommendation model training tasks between CPUs and GPUs based on the compute patterns. It then hierarchically utilizes various memory types, such as HBM, DRAM, and SCMs, by studying the temporal locality and bandwidth requirements of recommendation system models in data centers. MTrainS reduces the number of hosts used for training by up to 8$\times$, decreasing the power and cost of training. Lastly, this dissertation proposes CoACT, which designs fine-grain cache and memory sharing for collaborative workloads running in integrated CPU-GPU systems. CoACT uses the collaborative pattern of applications to fine-tune cache partitioning and interconnect and memory controller utilization for CPU and GPU, improving performance by 25%.

CHAPTER 1

Introduction

The slowing down of Moore's law and Dennard's scaling have led to diminished performance, cost, and power improvement of general-purpose processors by transistor scaling only. Hence, we require heterogeneous systems with specialized processors, such as GPUs, to increase performance and power efficiency. Similarly, the difficulty of scaling the cost, power, and density of DRAM led to the evolution of new memory technologies, such as storage class memories (SCMs), to operate alongside DRAM. While heterogeneous systems complement the traditional CPUs and DRAM, they have distinct characteristics. These characteristics include differences in processor designs, differences in memory hierarchies, and differences in memory access latencies and bandwidth. Thus, workloads that run on traditional systems might not run or gain performance when ported to heterogeneous platforms. To fully utilize heterogeneous platforms, we first need to understand the design and characteristics of these new devices and how they fit with standard system components. Second, it is critical to study the characteristics of applications that make them suitable for a particular compute or memory platform. This dissertation studies the computation, communication, and memory characteristics of emerging compute and memory components and workloads to effectively design and utilize heterogeneous systems.

1.1 Compute and memory scaling

Moore's law observed that the number of transistors packed per area doubles every two years, providing continuous processor performance improvement. Dennard scaling remarks that as transistors become smaller, the operation voltage and current reduce; hence, we can increase processor frequency and still maintain the same power consumption per area. These scalings have served chip manufacturers to deliver faster and cheaper processors at constant power consumption for decades. However, starting in the 2000s, Dennard scaling ended, and Moore's law is slowing down, leading to dormant single-core processor performance improvement. Multi-core processors were the next best solution to maintain the continuous performance improvement of processors, but power consumption and their difficulties in programming limit their performance enhancement. Currently, the most renowned solutions are specialized architectures or accelerators that are tailored to specific applications.

Scaling for semiconductor memory (DRAM) also closely followed Moore's law for a long time, providing denser and faster memory at a lower cost per bit. However, it is becoming expensive and challenging to scale down DRAM cells because of capacitor reliability and interference, leading to a surge in DRAM cost. These difficulties drove ample research focusing on new memory technologies, such as storage class memory (SCM), phase-change memory (PCM), magnetic random-access memory (MRAM), and resistive-RAM (RRAM). These technologies provide dense memory at low cost and power compared to DRAM, with higher latency and lower bandwidth than DRAM.

The trends of introducing specialized hardware units in the compute and memory systems present heterogeneity in system designs. Heterogeneous compute units provide significant performance and energy consumption improvement for a specific task by utilizing specialized units. Similarly, new memory technologies in the memory system hierarchy deliver very dense and cheap memory, which is impossible with DRAM alone. Nevertheless, such designs come at a cost for hardware and software designers. Specialized compute units (accelerators) tailored to individual applications are challenging to design, manufacture, and test. Especially with the emergence of compute and data-intensive applications such as machine learning, graph analytics, and genomics, we can not continue designing new accelerators for every application. The practical solution, in this case, is to have specialized units that can be repurposed for multiple application domains such as GPUs and FPGAs. Yet, this also comes with its own issues for software and system designers because it requires extensive expertise in every hardware unit to understand the system configurations that best fit each application. Heterogenous memory creates a similar problem because these new technologies have different memory latency and bandwidth characteristics than DRAM. For example, commercially available new memory technology, Intel optane persistent memory, provides 128-512GB of memory per DIMM module in contrast to DRAM, which is usually 4-32GB. However, it has 100-400ns latency and 5$\times$ less bandwidth than DRAM. Given these characteristics, we need to understand if we can replace DRAM with new memory technologies, if we should utilize these memories hierarchically or at the same level as DRAM, and what type of workloads can benefit from them.

1.2 Efficient utilization of heterogeneous systems

This thesis identifies the challenges in effectively implementing heterogeneous compute and memory platforms in different system types, such as integrated and discrete systems. It then studies the characteristics of diverse workloads running in various system types and presents software, system, and hardware mechanisms for efficient heterogeneous compute and memory utilization.

1.2.1 System types

The system type determines the classifications of heterogenous compute and memory units present in the system and how these units interact with one another. We consider two classes of systems; discrete server-grade systems and integrated heterogeneous architecture, primarily present in mobile and embedding systems. In server-grade systems, we have various compute and memory-intensive workloads that demand heterogeneity in both compute and memory. In integrated systems, we have compute units such as CPUs and GPUs sharing memory and cache in the future. In these systems, frequent and fine-grain communication between compute units is possible. Integrated systems are cost-constrained and hence have lower core counts in both CPUs and GPUs and have limited memory availability that does not demand different memory technologies.

1.2.2 Workloads

We took two approaches to characterize and evaluate workloads for heterogeneous systems based on the above system classes. First, we studied various stand-alone and collaborative algorithmic patterns that can represent repeatedly existing patterns in most applications. We use these studies to explore effective heterogeneous system utilization in server and integrated architecture. We use this approach to enable efficiency in general-purpose system configurations with one system running diverse applications. Second, we studied heterogeneity in server-grade systems in data centers. Our studies revealed that database and AI workloads in data centers consume significant resources. In this case, focusing on a single workload and optimizing its performance with heterogeneous systems provides high performance and cost and power improvements.

1.3 Dissertation approaches and contributions

This thesis proposes a series of techniques that improve the performance and utilization of heterogeneous compute and memory systems while still considering energy and cost efficiency for the above systems and workload configurations. We study the compute and memory access characteristics of applications to efficiently map applications to heterogeneous systems. Our solutions and contributions are summarized below.

- In the first work, ChipAdvisor, we build a framework that identifies the best compute platform for an application in early system design steps for discrete and integrated architecture. To perform this study, we first characterize the computation, communication, and data access pattern of a wide range of applications and identify intrinsic properties of applications (properties that are not influenced by how applications run on a particular platform). Based on these properties, we build an ensemble learning algorithm that comprehends representative applications' properties and predicts the best platform for applications when considering CPU, GPU, and FPGA. ChipAdvisor achieves an accuracy of up to 98% in predicting the best performing platform and 94% in predicting the most energy-efficient one, compared to an oracle analysis, that is, one which always selects the best platform for all applications.

- In the second work, we study how to utilize heterogeneous memory systems composed of DRAM and storage-class memory (SCM) for server-grade systems in data centers. In this work, we first analyzed one of the most significant memory-consuming applications in data centers: RocksDB, a key-value store designed for fast storage technologies. We then characterized the memory-consuming components and temporal locality of RocksDB, and we implemented a hybrid DRAM and SCM server configuration that efficiently utilizes these memories hierarchically. We analyzed the benefit of hierarchical heterogeneous memory in a small DRAM single-socket platform with SCM addition and compared it with a large DRAM dual-socket platform. Our results demonstrate that we can achieve up to 80% improvement in throughput and 20% improvement in P95 latency over a small DRAM single-socket platform while maintaining a 43-48% cost improvement over a large DRAM dual-socket platform.

- Further, this dissertation explores the efficient utilization of heterogeneous compute and memory systems for recommendation systems training. Recommendation systems are one of the fastest-growing workloads in data centers recently. These workloads require multiple GPUs and TBs of memory for training. This large memory consumption makes recommendation systems power-hungry, and scaling memory with

DRAM only is expensive. To solve this problem, we design MTrainS, an end-to-end deep learning recommendation model (DLRM) trainer that hierarchically uses HBM, DRAM, and byte and block-addressable SCMs. In our design, we first characterize the locality and the bandwidth requirements of data center deployed DLRM models to achieve a desired query per second (QPS). Based on this study, we implement memory assignment and caching strategies to map the model in these different kinds of memories to take advantage of the large sizes in SCMs without compromising performance. By optimizing the platform memory hierarchy, we are able to reduce the number of nodes for training by up to 8×, saving power and cost of training while meeting our target training performance.

- We then propose lightweight software and hardware co-design, CoACT that manages cache, interconnect, and memory utilization for CPU and GPU integrated on the same die running collaborative workloads. Collaborative workloads have tasks divided between different compute units. CoACT studies the collaborative pattern of application such as workload partitioning types, percentage of data shared between CPU and GPU, and locality, then designs cache partitioning, interconnect, and memory controller sharing mechanisms in the hardware. It then implements an analytical model that chooses the best mechanism for a particular application based on its collaborative pattern. CoACT achieves 25% performance improvements on average across diverse application domains.

1.4 Dissertation organization

The rest of this dissertation is organized as follows. In Chapter 2 we present the vital intrinsic properties across a diverse set of applications and how we can use these properties to build ChipAdvisor [2] and predict the best platform for application in different domains. In Chapter 3 we discuss various server configurations in commercial data centers and how we can build new server variants with heterogeneous memories to improve server deployment efficiency for key-value store workloads [3, 4]. Chapter 4 presents locality and bandwidth-aware heterogenous memory organization for recommendation systems training to reduce the number of hosts used for training. Chapter 5 presents the design of a collaborative pattern-based cache, memory, and interconnect sharing strategies for an integrated heterogeneous system with CPU and GPU. In Chapter 6 we conclude the dissertation.

All the work proposed in the dissertation is done in collaboration with Prof. Valeria

Bertacco, Prof. Todd Austin, Tarunesh Verma, Yichen Yang, Vaibhav Gogte from the University of Michigan, and Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Paul Johnson, and Ehsan K. Ardestani from Meta, inc.

CHAPTER 2

Mapping Applications to Heterogeneous Compute Platforms

While hardware accelerators provide significant performance and energy improvements over general-purpose processors, their limited reusability incurs high design costs. It is thus impractical to have a unique accelerator for each application. Hence, it is critical to develop solutions that can leverage the accelerators available to the best of their capabilities for a wide range of applications. In this paper, we note the common computation, data access, and communication patterns of applications, and based on these patterns, we identify significant intrinsic properties across applications. We then correlate these properties with the unique microarchitectural properties of the compute platforms available and develop a framework, ***ChipAdvisor***, to predict the platform that provides the best performance and energy efficiency for an application. We evaluate ***ChipAdvisor*** for applications from several domains, targeting CPUs, GPUs, and FPGAs as example compute platforms. ***ChipAdvisor*** achieves an accuracy of up to 98% in predicting the best performing platform, and 94% in predicting the most energy-efficient one, compared to an oracle analysis, that is, one which always selects the best platform for all applications.

2.1 Introduction

Moore's Law and Dennard Scaling have served chip designers well in the past to achieve faster and lower energy general-purpose computing. The slowdown of these trends has led to the search for alternative solutions to continue the performance and energy improvement in new computing systems. Currently, the most promising solutions are application-specific architectures (accelerators), which target narrow application domains to achieve performance gains and energy efficiencies through specializations based on applications' characteristics. These explorations have been very successful, delivering high performance over a diverse set

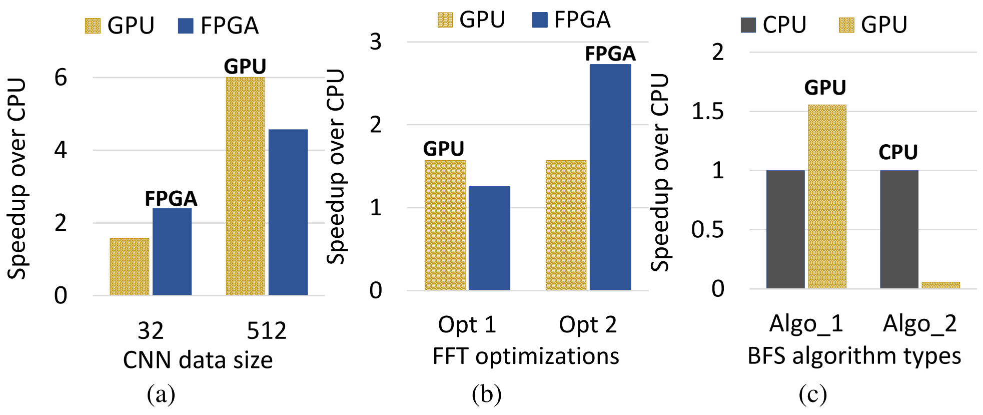


Figure 2.1: (a) CNN's execution time with two data sizes. (b) FFT's execution time with different FPGA optimizations. (c) BFS's execution time with two different algorithms (code structure). The best target platform selections are shown on top of the graph.

of applications. Nevertheless, because these solutions are tailored to specific applications, they cannot be deployed in accelerating other applications, necessitating high design costs. Hence, it is impractical to have a one-to-one mapping of applications to accelerators.

On the other hand, applications, while they seem distinct from each other, share common computation, communication, and data access patterns [5, 6]. A practical approach to accelerator design is to build a small set of accelerators, where each accelerator is optimized for a specific pattern, and together they cover wide application domains. The feasibility of this approach has led accelerator design efforts to shift to programmable/reconfigurable architectures based on the common patterns in applications. Going forward, systems are going to embrace existing compute platforms, such as CPUs and GPUs, and programmable accelerators. In such systems, the benefit of custom architectures cannot be unlocked unless we design solutions that can leverage the compute platforms/accelerators available to the best of their capabilities for multiple applications. This is achieved by providing techniques that can correctly choose where an application should be mapped given a set of platforms/accelerators. This choice is profoundly influenced not just by the type of application but also by the nature of the input, choice of algorithm for a particular application, detailed micro-architecture of the platforms, and implementation of the applications onto the platforms. Figure 2.1 demonstrates how best target selection changes with different parameters. As a result, the process of mapping becomes intricate and time consuming for end users. Hence, it is crucial to design frameworks that can quickly identify the best platform for a given application.

In this work, we undertake the challenge of automatically mapping applications to reusable compute platforms. We first characterize the computation, communication, and data access patterns of common application kernels [5, 7] to discover which computation components can best serve efficient application kernel execution. We then identify key traits that differentiate the execution characteristics of these kernels, and we define quantitative metrics that are intrinsic, platform-independent properties of applications to evaluate each trait. We next map each kernel to three distinct compute units: CPU, GPU, and FPGA. Using the correlation between the kernel's performance and energy usage on each target and our application metrics, we develop a novel tool, ChipAdvisor, to predict where applications should be mapped based on machine learning approaches. Figure 2.2 presents an outline of the mapping process, highlighting our contributions to it, including the analytical predictive model and the optimized mapping to existing platforms. ChipAdvisor aids with design space exploration of an application's performance and energy usage on multiple compute platforms easily and quickly by just measuring its intrinsic properties. It guides users in understanding which platform to use for an application, to examine the benefit we get from a platform, and to check if our performance requirements are met by these platforms.

In summary, we make the following contributions:

- We characterize the intrinsic, platform-independent properties of a range of applications by analyzing the execution of several diverse kernels on CPU, GPU, and FPGA targets.
- We develop a novel predictive model using decision-tree algorithms with AdaBoost. Given a set of compute units and the measured intrinsic properties of an application, the model predicts which platform provides the best performance and energy efficiency, and estimates performance for an application.
- To evaluate ChipAdvisor, we deployed it on several applications, predicted their best target platform, and estimated performance and found the accuracy to be 98% and 94% when targeting performance and energy, respectively (compared to an expert programmer), while remaining agnostic to the domain of applications.

2.2 Background and related works

Mapping applications to heterogeneous systems involves running target applications on all available platforms to find the best fitting one. When doing so, the burden of exploiting efficient mapping techniques and optimizations for an application per compute platform [8,9] falls on the software/hardware developers. This process is time-consuming, even for expert developers, because it requires extensive analysis on all platforms to find the optimal one. For example, as shown in Figure 2.1 (b) for the FFT algorithm, designs with different

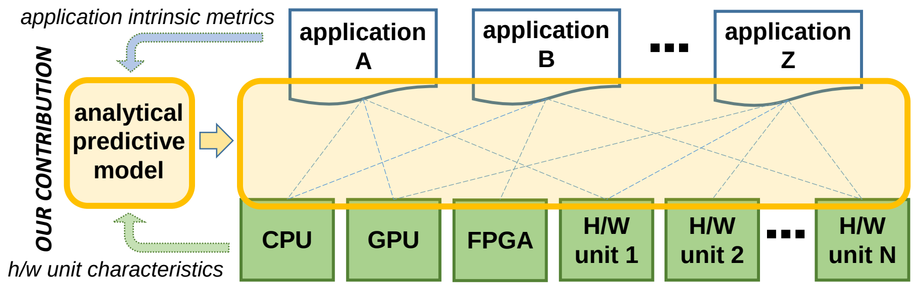


Figure 2.2: Given a set of applications and computation platforms, our proposed solution considers the key characteristics of each application and maps it to the best fitting compute device/accelerator to attain the best possible performance.

optimizations give different speedup on the FPGA. The number of optimizations we can perform per platform to find the best speedup is abundant; accordingly finding the best platform becomes difficult. To help with implementations and optimizations processes, previous works have shown the benefits of specific platform targeted optimizations tools [10, 11], and architectural differences of platforms, such as GPU and FPGA [12, 13]. However, these works are limited to comparing the platforms either for a small set of applications, or targeting only a specific platform. In contrast, we study multiple platforms with different micro-architectural properties and use a diverse set of applications.

Previous works use parallel patterns to efficiently map applications to re-configurable fabrics [14], but focus on efficiently utilizing a single platform. Our model alternatively predicts where to map applications by comparing the benefits of multiple platforms. [15] map applications to a system that is comprised of GPU and FPGA, but their methods rely on the correct annotation of parallel patterns in the code by a programmer, which is error-prone. In contrast, our method measures intrinsic properties from profiling an application and decides what compute unit to use without requiring the programmer to correctly annotate the code. Recent work has shown [16] how to choose components for SOC in early design but focuses more on the accelerators' maximum computation capacity rather than how the properties of applications will influence their performance on the accelerators. Alternatively, ChipAdvisor helps to minimize the effort of deciding the best component for an application in early design space exploration based on the characteristics of the applications.

Intrinsic properties of applications have been studied as important features to design application-specific architectures [7, 17]. [5] shows applications that we use today, even if they are from diverse domains, share common computation, compunction, and data access patterns that can be generalized by a set of unique kernels. In this work, we build on these works and identify the key intrinsic properties that give us an accurate mapping; we then characterize the well-studied common patterns [5] using the intrinsic proprieties and build a

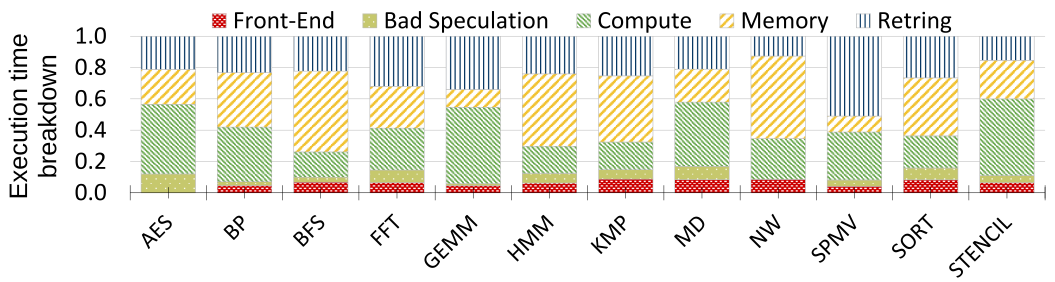


Figure 2.3: TMAM [1] based execution time breakdown for the benchmarks reported in Table 2.4.

model that can cover most application patterns.

Previous work [18, 19] uses predictive models to find inefficiencies within the CPU, to decide where to run applications (CPU or GPU) using hardware-specific properties. In contrast, we use application intrinsic properties that do not depend on how applications run on particular hardware. This enables us to map to more computational targets. Based on this approach we build a general pattern-based design space exploration tool that works for multiple platforms and applications, providing the best target in terms of execution time, energy, and expected speedup/slowdown over CPU, without needing to run the application on all platforms.

2.3 ChipAdvisor design

ChipAdvisor has two main components, **intrinsic properties measurements** and **machine learning-based prediction model**. To develop these two components, we first studied inefficiencies in a CPU-bound code, using representative applications, to identify significant intrinsic properties and strategies of how to quantify them. Later, using these properties and performance analysis on all compute platforms in our system, we build the machine learning-based prediction model. The design process is described below.

2.3.1 Identifying inefficiencies in applications

In developing our framework, we assume a heterogeneous system where CPUs are host units with multiple compute devices/accelerators attached to them. We consider the main kernel of the application where the core part of the computation takes place, and evaluate whether the CPU is the best fit for its execution, or if other platforms can outperform the CPU. As a result, an analysis of inefficiencies based on the CPU-bound execution of an application offers direction for choosing the best device. To this end, we deployed a TMAM analysis [1], using the Intel VTune tool [20], of several applications with different characteristics [7], listed in Table 2.4. A TMAM analysis utilizes micro-architectural performance counters to correlate performance bottlenecks with application code. As seen in Figure 2.3, the bottlenecks for many of the applications evaluated are due to computation and/or memory access inefficiencies (minimal inefficiencies in the front-end parts, such as fetching instructions and branch speculations). Using the TMAM analysis's hardware counter values, the most significant inefficiencies within the compute and/or memory-bound code are shown in Table 2.1. From Table 2.1 we see that *compute bound* applications suffer from insufficient computation capabilities, or they have control/data dependencies that hinder compute unit utilization. Whereas, *memory bound* applications are limited by insufficient memory bandwidth or excessive memory-access latency. These results demonstrate that, while some applications (GEMM, STENCIL, and MD) are dominated by compute inefficiencies, others (BFS and HMM) incline more to memory inefficiencies. The rest of the applications manifest both compute and memory inefficiencies to a varying degree.

Once the critical bottlenecks are identified, to determine the best platform, we considered which architectural properties of the compute platforms support optimizations such as Parallelization, exploiting spatial and temporal data locality, to overcome the above inefficiencies. We then studied the characteristics of applications that make them amenable to the specific optimizations. Here note that the bottlenecks reported above are with respect to the shortcomings of the underlying hardware, as analysis results are based on the performance counters of a specific architecture. In other words, these are architecture-specific manifestations of intrinsic characteristics of an application. To gather measures that help us identify the best fitting compute unit for an application, we took the critical bottlenecks identified above and studied what inherent characteristics of applications would create the opportunity to overcome the bottlenecks. To illustrate with an example, when an application runs in CPU, if the bottleneck is high memory latency due to a low cache hit rate, and if the application has locality, then mapping the application with a platform with a larger cache will solve the issue, whereas if the application does not have locality, then mapping it to a platform that can better handle irregularity will give better performance. Similarly, for the significant inefficiencies identified by TMAM analysis (in Table 2.1) we studied which properties of applications provide an opportunity for acceleration so that we can use it as a guide in selecting the best platform. From these, we derived a set of core properties of both compute platforms and applications, which are responsible for efficient execution.

2.3.2 Intrinsic application properties

We identify a set of intrinsic properties to map an application to compute platforms:

Table 2.1: TMAM's inefficiencies breakdown based on hardware counter

	AES	BP	BFS	FFT	GEMM	HMM	KMP	MD	NW	SPMV	SORT	STENCIL
Comp bound*	✓	✓		✓	✓			✓	✓	✓	✓	✓
Insufficient comp-units*		✓		✓	✓			✓		✓	✓	✓
deps*	✓			✓			✓		✓		✓	
Mem bound+		✓	✓	✓		✓	✓		✓		✓	
High mem latency+			✓	✓		✓	✓		✓			
Insufficient mem bw+		✓	✓								✓	

'*' shows compute related inefficiencies and '+' show memory inefficiencies.

2.3.2.1 Parallelism

To quantify parallelism we considered properties of both regular and irregular applications and measured it based on the number of active elements that can be execute concurrently given unlimited resources (the theoretical maximum parallelism). If we, for example, take an application running a "for" loop, parallelism denotes the maximum amount of time the loop can be unrolled. For an irregular application or nested loops with different unroll factors, we took the average of the unroll factor. This number is collected by running the application on CPU and counting the number of possible independent works.

$$parallelism = \frac{\sum_{i=0}^{i=N} unroll\_factor_i}{N} \tag{2.1}$$

Where N is the number of loops in the application and $unroll\_factor_i$ is the maximum times loop i can be unrolled.

2.3.2.2 Synchronization/data dependencies

Applications that require synchronization should favor platforms that resolve these tasks quickly. We measured these by counting the number of synchronization instructions an application requires, given its theoretical maximum parallelism. For example, for an application running a for loop, synchronization determines the number of times the threads communicate after the loop is unrolled.

$$sync = \#synchronization\_insts \tag{2.2}$$

Table 2.2: Resources available in compute platforms.

Hardware Props	CPU	GPU	FPGA
Parallelism	48	3584	1518
locality	2 MB	0.144 MB	6.62 MB
Sync/data dep.	Atomic insts, fences,barriers,	Barrier, Fences, multi-kernel	Interconnects
Control dep.	Branch prediction	Both branch directions	Longest branch direction

2.3.2.3 Control dependencies

Control dependencies are a source of inefficiency if a significant portion of the single-thread execution is delayed by them. Applications that present a lot of control dependencies are more efficient when running on platforms that resolve dependencies faster. We quantified this metric as the percentage of control instructions in the application's execution. Here we did not consider loop carried dependency instructions because these instructions are not the algorithmic properties of the application:

$$cd = \frac{\#control\_insts}{\#control\_insts + \#memory\_insts + \#compute\_insts} \tag{2.3}$$

2.3.2.4 Temporal locality

Temporal locality [6] can be quantified by the number of memory accesses between two accesses to the same address, which is called *reuse distance*. When an application has a short reuse distance, memory latency can be hidden by leveraging even small on-chip storage. As the reuse distance gets longer, the size of the on-chip storage required increases. For applications with large reuse distance, mapping to a platform with large local storage provides better performance because it can fully exploit the application's temporal locality. We quantified temporal locality as:

$$tl = \sum_{i=0}^{i=log_2(N)-1} \frac{(reuse_{2^{(i+1)}} - reuse_{2^i}) * (log_2 N - 1)}{log_2 N} \tag{2.4}$$

Where N is the size of the local storage in bytes and $reuse_{2^i}$ is the number of memory access with reuse distance 2^i. The larger the value of N, the larger the temporal locality because large local storage can extract high temporal locality. The temporal locality value *tl* ranges between 0 and 1. *tl* gets closer to 1 when the temporal locality is high, and it is 0 when memory accesses follow a streaming pattern or are very irregular. In our analysis, we set N to 2MB, an amount we deemed realistic for modern accelerating hardware.

2.3.2.5 Spatial locality

This determines the predictability of memory accesses, a trait that benefits from prefetching and bringing larger data blocks near compute units. For applications with low spatial locality, an ideal platform would provide a specialized memory system that accommodates irregular memory accesses, so as to best leverage the application's memory bandwidth. Spatial locality depends on the application's data types and its data accesses predictability. To quantify it, we adopted the metric proposed in [6]:

$$sl = \sum_{i=0}^{i=\infty} \frac{stride_i}{i} \tag{2.5}$$

In this equation, we measure strides in the same units as the granularity of data access in the application. The equation generates values ranging from 0 to 1: 1 corresponds to applications presenting contiguous memory accesses and 0 when the memory accesses in the application do not follow any discernible stride pattern.

Resources available in the target platforms. As we see in Table 2.2, the platforms we use exhibit specific hardware characteristics in terms of our mapping metrics. The intrinsic properties values of the applications would prefer one hardware over another based on what the platform can provide, which we incorporate in our model.

Completeness of the intrinsic properties identified. The properties discussed characterize the compute and memory characteristics of most applications. Individual property values and their combination allow us to fully differentiate applications among each other. In a sense, the properties are a set of eigenvalues for the application's traits: in combination, they inform a rich set of characteristics for any application. For example, different combinations of temporal and spatial locality provide varied memory accesses characteristics, such as continuous, stride, streaming, and irregular; they even inform the degree of irregularity. Parallelism, combined with memory characteristic, reveals if an application is more computation- or memory-heavy. Parallelism with synchronization can characterize applications ranging from serial to embarrassingly parallel. In conclusion, the set of properties we identified is capable of distinguishing all the applications we considered from each other. If an application were to arise that we could not distinguish from an existing one, based on the properties described, we would have to identify an additional property to add to our pool.

2.3.3 Prediction model design

When working with a specific CPU's microarchitecture, it is possible to identify performance bottlenecks with straightforward analysis to help increase performance [21]. However, in the context of heterogeneous systems, where there are multiple microarchitectures available, grasping all the relevant traits of hardware units and matching them with the properties of the application to be executed entails a complex relationship among many variables. Each of the above given intrinsic parameters can take a large range of values. The combination of the five intrinsic parameters creates a large space within which to differentiate applications. To identify the best platform for applications in this large space, we deployed a machine learning solution. Machine learning also facilitates the addition of new intrinsic parameters or new compute platforms.

2.3.3.1 Training phase

The training phase of the prediction model is illustrated on the top right of Figure 2.4. We first measured the intrinsic properties (discussed above) of the training kernels (see Table 2.4). We then measured the execution time and energy usage of the kernels on CPU, GPU, and FPGA. Based on these measurements per application, we generated the training data. Using these data, we trained several machine learning models as discussed in Section 2.4.3, with our intrinsic properties' measurements as features; the execution time and energy usage analysis on the three platforms as output prediction targets of the model. We used the *scikit-learn* machine learning library [22] to evaluate several existing machine learning algorithms and chose the one with the best accuracy, as discussed in Section 2.4.3. This effort clearly indicated that the regression decision-tree delivers the best results hence we adopted it to create the prediction model. We then added ensemble learning AdaBoost with decision-tree to increase the accuracy of prediction. The output model from the training process is a decision-tree like the one on the left of Figure 2.4. The output from training includes all the nodes, edges, and thresholds of the decision-tree algorithm. The decision-tree presented in Figure 2.4 is a simplified representation of the overall relationship between the intrinsic properties of applications and the characteristics of our target platforms. In a decision-tree algorithm, selections are made by traversing a tree data structure from root to a leaf. The path is selected based on thresholds at each internal node. In the context of our model design, the internal nodes correspond to the intrinsic properties of the application and the leaves provide the best-fitting platform.

To generate a well representative dataset of the unique application patterns and the architecture specifications of the platforms, we analyzed the performance and energy usage of the kernels on CPU, GPU, and FPGA using target-specific optimizations [8,9] and varied dataset types and sizes for each application. The target-specific optimization revealed the maximum capacity of the platforms per application. We chose to vary the input datasets to limit their impact on our intrinsic properties' measurements. For example, the input

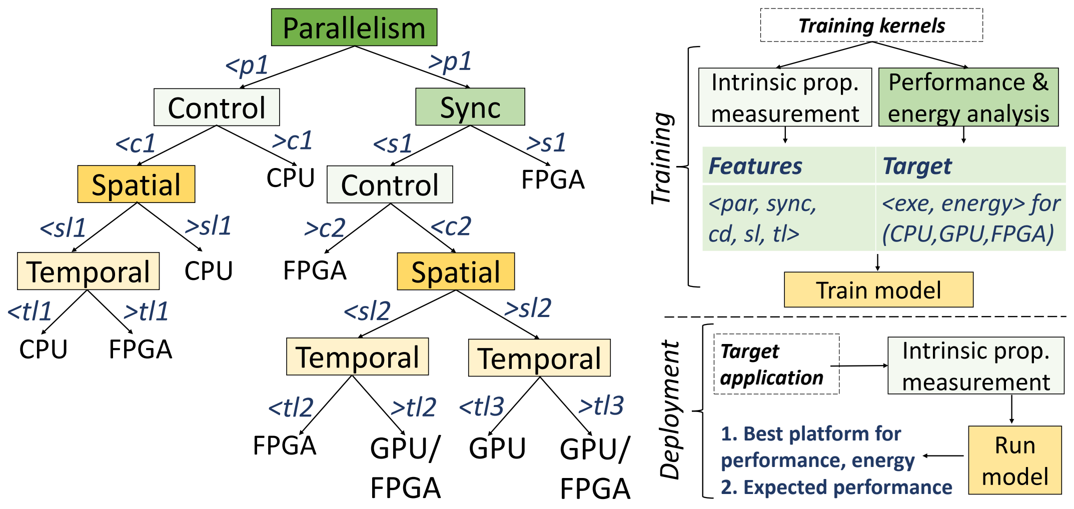


Figure 2.4: On *left* is a simplified decision-tree based model used for mapping applications to CPU/GPU/FPGA. On the *right* is overall ChipAdvisor framework (training process done offline and the online process for using it).

datasets of highly parallel applications, such as matrix multiplication (GEMM), have a strong influence on the parallelism that can be attained, while the memory's temporal and spatial locality traits of irregular applications, such as SPMV and BFS, are directly impacted by the input data's sparsity.

From Figure 2.4 it can be noted that the ideal platform is selected based on different combinations of the intrinsic properties of an application. For instance, breadth-first search (BFS) presents a high level of parallelism and requires frequent synchronization; thus, it provides the best performance when running on FPGA. In contrast, the low level of parallelism and high control dependency level of applications like Knuth-Morris-Pratt (KMP) map best to CPU. Note that the tree shown in Figure 2.4 is trained on a relatively small set of data points (applications). As we included more applications, the tree has become richer and more complex to accommodate the intricate relationships among the properties. We do not show it here for reasons of readability.

2.3.3.2 Deployment phase

After the model is trained, applications are mapped onto the platforms using the process shown on the bottom right of Figure 2.4. The mapping is completed by taking applications as input and then measuring their intrinsic properties using the formulas described in Section 2.3.2; the intrinsic properties are then run in the model as input, traversing the tree until a leaf node is reached. If the leaf node provides one target platform, that platform will be the one that provides the best performance for that application. If more than one platform is

Table 2.3: Compute platforms - hardware specification.

	CPU	GPU	FPGA
Model	Intel Xeon Gold 6126	NVIDIA GTX 1080 TI	Arria 10 GX 1150
Frequency	2.6 GHz	1.582 GHz	0.8 GHz
#cores	24	3584	1518
Memory size	256GB	11GB	8GB
Memory bandwidth	17GB/s	484GB/s	17GB/s

recommended, both are suitable candidates for optimal performance.

In this work we target CPU, GPU, and FPGA; however, we note that it is straightforward to add other current or future application-specific platforms, by using the same methodology we presented. To add a new platform/accelerator, one needs only to carry out the corresponding performance analysis on representative kernels and conduct training to build the new model.

2.4 Experimental evaluation

2.4.1 Evaluation setups

Compute platforms: To evaluate ChipAdvisor, we studied three compute platforms: CPU, GPU, and FPGA, spec details listed in Table 2.3. Our **baseline** for evaluating ChipAdvisor is a manual best platform selection (oracle analysis) based on extensive optimizations on each platform, which is equivalent to an expert programmer. The oracle analysis always selects the best platform.

Training and testing kernels: For **training**, we used **micro-benchmarks** that enabled us to sweep one of the intrinsic properties while keeping others constant; this allowed us to study the impact of varying an intrinsic property. In addition, we trained the model with MachSuite [7] kernels for their varied pattern implementations presented in [5]. For **testing** the validity of our model, we chose applications from different emerging application domains. Note that the testing applications are never used for training the model. From machine learning, we looked at kernels in recommendation systems [23] and reinforcement learning [24], and we implemented a simplified C++ inference algorithm. For genome sequencing, we analyzed the Smith-Waterman algorithm [25]; for n-body simulation, we used the Barnes-Hut algorithm [26]. Finally, we also examined k-means from the Rodinia benchmarks [27]. All these applications are summarized in Tables 2.4 and 2.5. We used two different implementations of BFS for training and testing to evaluate our model's dependence on kernel implementation. For all kernels, we measured the intrinsic properties with varying

Table 2.4: Decision-tree training kernels.

Dwarf name	**Kernel**	**footprint (MB)**
Combinational logic	AES encryption (AES)	256KB-0.64
Graph traversal	Breadth first search (BFS)	0.24-100
Unstructured grid	Backprop (BP)	0.64-135
Spectral method	Fast fourier transform (FFT)	0.06-0.4
Dense linear algebra	Matrix multiplication (GEMM)	0.06-0.4
Graphical model	Hidden markov model (HMM)	0.68-1
Finite state machine	Knuth-Morris-Pratt (KMP)	192KB-25
N-body method	Molecular dynamics (MD)	0.18-43
Map reduce	Merge sort (MergeSort)	256KB-0.13
Dynamic programming	Needleman-Wunsch (NW)	0.18-13
Sparse linear algebra	Sparse matrix/vector mult (SPMV)	0.02-23
Structured grid	STENCIL2D	0.04-20
Micro-benchmarks		
Dense linear algebra	DotProduct/Reducer/Irregular adder	0.09-1
Structured grid	Stencil1D (STENCIL1D)	0.8-1
Finite state machine	Insertion sort (InsertSort)	4KB-0.4

Table 2.5: Decision-tree testing kernels.

Domain	**Kernel**	**Memory FP**
Graph traversal	Breadth-first search queue (BFS_q)	0.24-100
Simulation	Barnes-Hut tree (BHtree)/force (BHforce)	0.02-100
Machine learning	Convolutional neural network (CNN)	0.01-100
Linear algebra	General matrix factorization (GMF)	0.16-30
Data mining	Kmeans (KMEANS)	0.16 - 100
Machine learning	Monte-carlo tree search (MCTS)	0.012-2
Machine learning	Multi-layer preceptron (MLP)	0.16-100
Genomics	Smith-Waterman (SW)	0.1-0.16

dataset types and sizes (see Tables 2.4 and 2.5), since these properties influence the values of the intrinsic properties and target mappings.

We implemented kernels in C++ for CPU using OpenMP, and OpenCL for GPU and FPGA. We followed optimisations provided in [9] for FPGA, and those in [8] for GPU to perform target specific optimizations for the kernels.

Measuring intrinsic properties: We measured a variety of intrinsic properties to identify the most relevant ones. To measure these properties we added counters in our CPU-bound code for the flexibility of measuring any intrinsic property. The same metrics could also be collected from instrumentation and profiling tools, such as pin.

2.4.2 Performance analysis for training kernels

We implemented the training kernels for various data types and sizes to train ChipAdvisor. We show a snapshot of the data with comparable Memory Footprint that are representative of application properties' influence on mappings. Figures 2.5 and 2.6 provide the speedups for GPU/FPGA over CPU and measured intrinsic properties of the training kernels, respectively. For applications like *dot product*, *reducer*, and *GEMM*, that have high parallelism and low temporal locality, as seen in Figure 2.6, which results in higher bandwidth requirements, the GPU has a better speedup because it is best suited to provide these properties. But for *STENCIL* and *MD* (applications which also require high bandwidth), because of their high temporal locality, the FPGA can provide the bandwidth using better on-chip storage; these applications have speed-ups close to those of GPU. Applications like *merge sort*, *FFT*, *HMM*, and *NW*, which have irregular memory accesses, require synchronization, and have control-flow dependencies, the FPGA provides a better speedup. Kernels like *KMP* and *insertion sort* have low inherent parallelism; thus the single-threaded optimized CPU performs best. From these relationships, we observe that there is a correlation between intrinsic properties and the capabilities of the platforms.

2.4.3 Accuracy of the machine learning models

The above analysis shows that different combinations of intrinsic properties indicate different patterns, which influence the best platform selection for kernels. To study the correlation between applications' intrinsic properties and platforms' microarchitectural properties, we used various machine learning models from the *scikit-learn* library [22]. We divided the collected training data, which includes the intrinsic properties measurements and performance analysis of all of the training kernels (see Table 2.4) in CPU, GPU, and FPGA, into training and testing sets. To measure the accuracy of prediction, we then train the model on the training set, which is 50%-70% of the dataset, and tested the model on how accurately it predicts the best platform for the remaining testing set. Despite limiting the training kernels, we used multiple datasets per application. The median results after running with randomly selected training and testing datasets 50 times are shown in Table 2.6. Our analysis shows the algorithms that capture the relationship best is *decision-tree*. This outcome is likely because the data are nonlinear and have correlation within the training features (intrinsic properties), creating a subtle relationship that makes SVM, KNN, and Naive Bayes less accurate. For decision-tree, using an ensemble learning algorithm Adaboost further increased the accuracy. We did not implement any neural nets because we have a satisfactory 98% accuracy with the decision-tree.

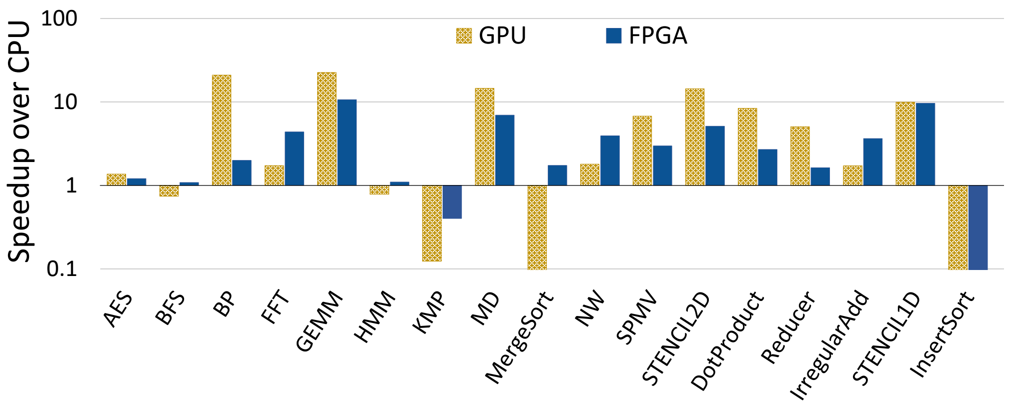


Figure 2.5: Training kernels' speedup over CPU. Y-axis is in log_{10} scale.

	Parallelism	Sync	Control	Temporal	Spatial
AES	512	0	0.004	0.720	0.800
BFS	8192	10	0.282	0.304	0.329
BP	16417	1	0.000	0.619	0.672
FFT	4096	13	0.045	0.752	1.000
GEMM	16384	0	0.000	0.496	0.508
HMM	192	422	0.143	0.632	0.332
KMP	1	1	0.500	0.500	1.000
MD	4096	0	0.000	0.662	0.984
MergeSort	630	13	0.350	0.396	1.000
NW	85	3	0.150	0.418	0.999
SPMV	16384	0	0.000	0.112	0.667
STENCIL2D	32004	0	0.000	0.771	0.834
DotProduct	102400	0	0.000	0.000	1.000
Reducer	256000	0	0.000	0.000	1.000
IrregularAdd	8500	0	0.000	0.571	0.500
STENCIL1D	10000	0	0.000	0.665	0.997
InsertSort	1	1	0.250	0.294	1.000

High

Low

Figure 2.6: Values of intrinsic properties for decision-tree training.

2.4.4 Results for the testing kernels

2.4.4.1 Performance analysis on testing kernels

Figure 2.7 shows a comparison between performance gained of ChipAdvisor and oracle analysis for the testing applications shown in Table 2.5. Note that these workloads are not used for training the model. This figure demonstrates that ChipAdvisor selects the best target platform with accuracy that is close to the oracle best target. These show that using ChipAdvisor will give a higher performance (7x over CPU, oracle is 7.3x) than running all applications on a single target (CPU, GPU, or FPGA), as a result of accurate target platform selection, hence increasing efficient resource utilization in heterogeneous systems.

Table 2.6: ML algorithms accuracy for training based on intrinsic properties as features and compute platforms (CPU/GPU/FPGA) as targets.

Algorithm	NB	SVM	KNN	Decision-t	Random-f	ada(DT)
Accuracy	80	88	92	95	96	98

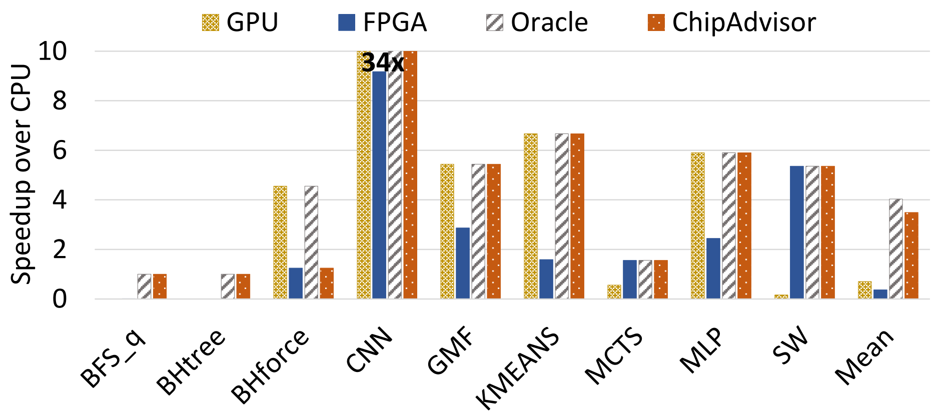
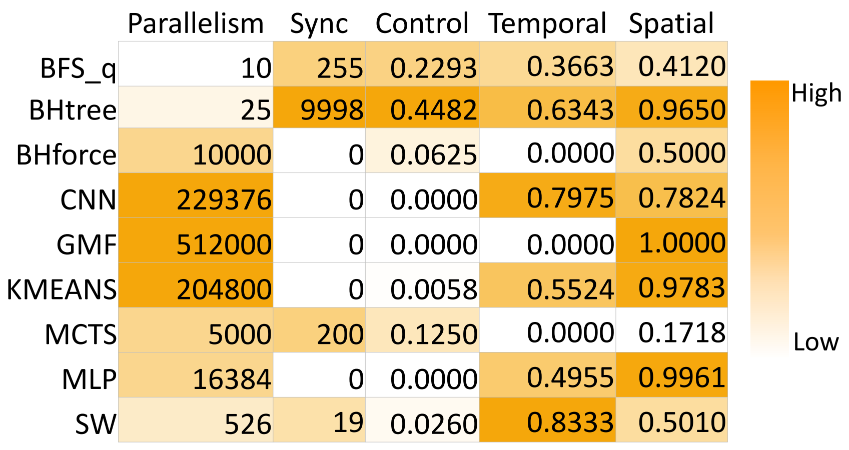


Figure 2.7: Testing kernels' speedup over CPU for GPU, FPGA, oracle (hand optimized best selection) and ChipAdvisor.



	Parallelism	Sync	Control	Temporal	Spatial
BFS_q	10	255	0.2293	0.3663	0.4120
BHtree	25	9998	0.4482	0.6343	0.9650
BHforce	10000	0	0.0625	0.0000	0.5000
CNN	229376	0	0.0000	0.7975	0.7824
GMF	512000	0	0.0000	0.0000	1.0000
KMEANS	204800	0	0.0058	0.5524	0.9783
MCTS	5000	200	0.1250	0.0000	0.1718
MLP	16384	0	0.0000	0.4955	0.9961
SW	526	19	0.0260	0.8333	0.5010

Figure 2.8: Measured values of intrinsic properties for decision-tree testing.

Figure 2.7 also provides the speedup over CPU for GPU/FPGA and Figure 2.8 shows the measured intrinsic properties. Similar to the analysis for the training kernel, we observe a correlation between the best performance achieved on the compute platforms and the intrinsic properties of the kernels. For example, applications like *MCTS* and *SW* have high parallelism, high synchronization, and irregular memory accesses, making FPGA the best platform because it provides high compute capabilities, fast synchronization, and because it allows flexible implementation of local storage that handles irregularities better than the CPU and GPU. Another example is *CNN*, which perform best on GPU because of its high parallelism, regular memory accesses, and high bandwidth requirements, which are satisfied

Table 2.7: Comparison of kernels' best target platform (performance analysis) vs. predictions (ChipAdvisor), incorrect predictions highlighted.

Apps	Oracle best performance	ChipAdvisor best performance	Oracle best energy	ChipAdvisor best energy
BFS_q	cpu	cpu	cpu	fpga
BHtree	cpu	cpu	cpu	cpu
BHforce	gpu	gpu/cpu	fpga	fpga
CNN	gpu	gpu	gpu	gpu
GMF	gpu	gpu	gpu	gpu
KMEANS	gpu	gpu	gpu	gpu
MCTS	fpga	fpga	fpga	fpga
MLP	gpu	gpu	fpga	fpga
SW	fpga	fpga	fpga	fpga

by GPU's architecture. Note that the results shown in Figure 2.7 and 2.8 are not the only data points for the kernels; depending on the dataset, kernels' intrinsic property measurements might change which influence the best platform for it. For example, for *CNN*, when the data size is medium, the bandwidth requirements of the application can be alleviated by the application's high temporal locality; hence the FPGA can perform as good as the GPU.

2.4.4.2 Best performing target prediction

Table 2.7 shows that *ChipAdvisor* accurately predicts the best target for kernels that are not part of training. This is because each kernel we used for training belongs to unique classes of application defined by similarities in computation, communication, and data access patterns. ChipAdvisor predicts *BFS_q* and *BHtree*, with low parallelism and high control flow dependency, run best on the CPU. This pattern is captured by irregular applications like *KMP* and insertion sort. *MCTS* is predicted accurately to perform best on the FPGA because of the high parallelism, high synchronization, and irregular accesses. This kernel's property correlates to the training kernels like *HMM* and *BFS*. *BHforce* is predicted inaccurately to run efficiently on the CPU and GPU for medium dataset because it has moderate parallelism that can be satisfied by both CPU and GPU. But because *BHforce* also has small irregularity, it is incorrectly classified to fit best on CPU and GPU, but *BHforce* requires more compute resources than a platform that handles irregularity; hence GPU is the best platform. This error can be prevented by adding more training data, which we will explore in the future. Prediction is done by first running the applications once on CPU to measure intrinsic properties, followed by running the model. Prediction takes an average of 0.3 ms, therefore this process incurs minimal overhead, making ChipAdvisor a sound tool for early design

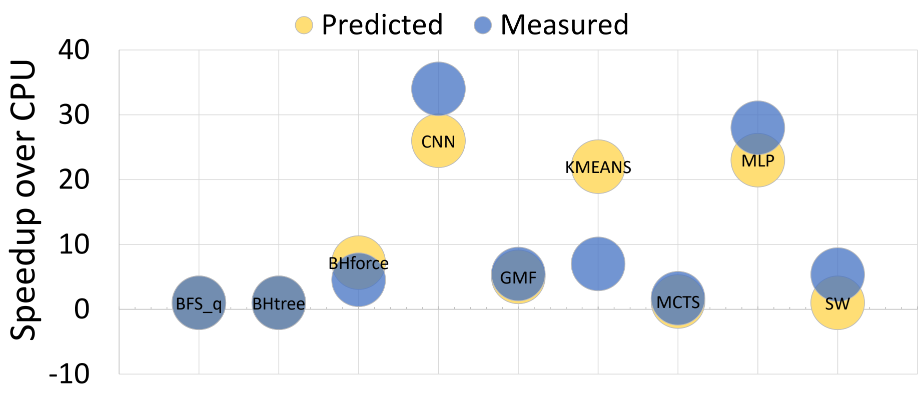


Figure 2.9: Comparison of measured and predicted performance using ChipAdvisor. Comparison is based on the best target platform per application.

space exploration.

2.4.4.3 Approximate performance prediction

From our analysis, we learned that, although we cannot predict the exact speedup/slowdown over CPU, we can estimate the performance with some error margin. Figure 2.9 shows a comparison of the best measured and predicted performance; the circle's area represents error margin. The smaller the circle's area, the smaller the error margin. In the figure, we see that for most of the applications, the measured and the predicted performance overlap, which suggests that our model can estimate expected approximate performance. This result is promising because in early design space exploration, when trying to find the best platform for an application, we are eager to estimate if the performance of a platform lies within a window of speedup over CPU. For example, can we get 2x, 5x speed up and so on? The plot shows that all except *KMEANS* overlap; *KMEANS* follows a dense linear algebra pattern, but it also has control flow irregularity. Nevertheless, because decision-tree decides which edge to traverse from a node based on threshold and because the control flow irregularity is not high enough to change the decision, the accuracy is estimated with applications like *CNN*. Note that the best platform is still predicted accurately because the small irregularity is not high enough to change the prediction.

2.4.4.4 Energy analysis

To get the energy-efficient platform, we trained our model using intrinsic property measurements and energy analysis of the kernels listed in Table 2.4 in CPU, GPU, and FPGA. For energy, we achieve 94% accuracy in predicting the best platform. This accuracy is measured by randomly selecting training and testing datasets using the training data and then by

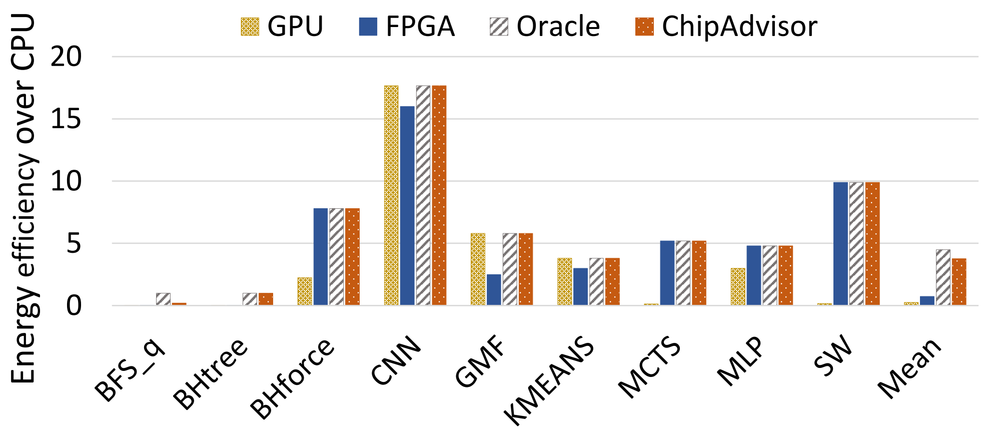


Figure 2.10: Testing kernels' energy efficiency over CPU for GPU, FPGA, oracle (hand optimized best selection) and ChipAdvisor.

performing multiple training (50 runs) and selecting the median accuracy, similar to Section 2.4.3. Prediction for applications that are not part of training is shown in Table 2.7. From the table, we can see that ChipAdvisor predicts the most energy-efficient platform even when the energy-efficient target is different from the best target for execution time. Figure 2.10 plots the energy efficiency of GPU and FPGA over CPU. The figure shows that ChipAdvisor selects the most energy-efficient platform for a specific application with accuracy close to the oracle analysis. This shows that ChipAdvisor provides higher energy efficiency than running all applications on a single target (CPU, GPU, or FPGA). The approximation energy number follows the same trend as the approximate performance prediction.

2.5 Conclusion

In this paper, we analyze the computation, communication, and data access characteristics in diverse applications to identify the important intrinsic properties that make these applications suitable for hardware acceleration. Then, based on these measured properties, we built the framework ChipAdvisor that predicts where the application will perform best among the available compute platforms when targeting best execution time and energy efficiency. Our framework captures the correlation between the intrinsic properties of applications and the microarchitecture of the compute platforms (CPU, GPU, and FPGA). By identifying the most potent application-intrinsic properties, we can predict the best platform and approximate performance benefit achievable from all available platforms. Our framework achieves an accuracy of up to 98% in performance improvements and 94% in energy efficiency compared to an oracle analysis.

CHAPTER 3

Optimized Deployment of Key-Value Stores Using Heterogeneous Memories

High-performance flash-based key-value stores in data-centers utilize large amounts of DRAM to cache hot data. However, motivated by the high cost and power consumption of DRAM, server designs with lower DRAM per compute ratio are becoming popular. These low-cost servers enable scale-out services by reducing server workload densities. This results in improvements to overall service reliability, leading to a decrease in the total cost of ownership (TCO) for scalable workloads. Nevertheless, for key-value stores with large memory footprints, these reduced DRAM servers degrade performance due to an increase in both IO utilization and data access latency. In this scenario, a standard practice to improve performance for sharded databases is to reduce the number of shards per machine, which degrades the TCO benefits of reduced DRAM low-cost servers. In this work, we explore a practical solution to improve performance and reduce the costs and power consumption of key-value stores running on DRAM-constrained servers by using Storage Class Memories (SCM).

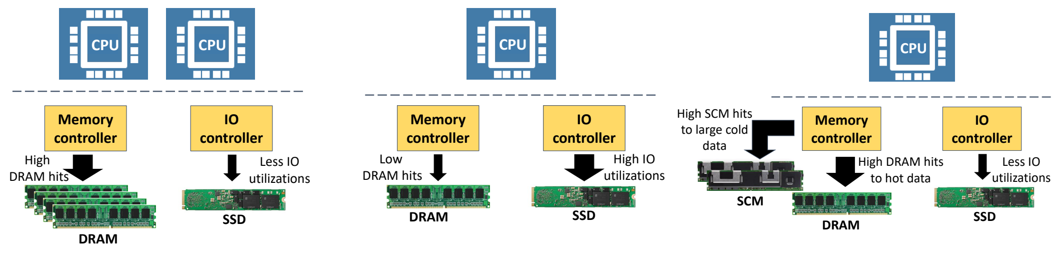
SCMs in a DIMM form factor, although slower than DRAM, are sufficiently faster than flash when serving as a large extension to DRAM. With new technologies like Compute Express Link (CXL) we can expand the memory capacity of servers with high bandwidth and low latency connectivity with SCM. In this paper, we use Intel® Optane™ PMem 100 Series SCMs (DCPMM) in AppDirect mode to extend the available memory of our existing single-socket platform deployment of RocksDB (one of the largest key-value stores at Meta). We first designed a hybrid cache in RocksDB to harness both DRAM and SCM hierarchically. We then characterized the performance of the hybrid cache for 3 of the largest RocksDB use cases at Meta (ChatApp, BLOB Metadata, and Hive Cache). Our results demonstrate that we can achieve up to 80% improvement in throughput and 20% improvement in P95 latency over the existing small DRAM single-socket platform, while maintaining a 43-48% cost improvement over our large DRAM dual-socket platform. To the best of our knowledge, this is the first study of the DCPMM platform in a commercial data center.

3.1 Introduction

High-performance storage servers at Meta come in two flavors. The first, *2P server*, has two sockets of compute and a large DRAM capacity, as shown in Figure 3.1a, and provides excellent performance at the expense of high power and cost. In contrast, *1P server* (Figure 3.1b), has one socket of compute and the DRAM-to-compute ratio is half of the *2P server*. The advantages of *1P server* are reduced cost, power, and increased rack density [28]. For services with a small DRAM footprint, *1P server* is the obvious choice. A large number of services at Meta fit in this category.

However, a class of workloads that may not perform adequately on a reduced DRAM server and may not take advantage of the cost and power benefits of *1P server* at Meta are flash-based key-value stores. Many of these workloads use RocksDB [29] as their underlying storage engine. RocksDB utilizes DRAM for caching frequently referenced data for faster access. A low DRAM to storage capacity ratio for these workloads will lead to high DRAM cache misses, resulting in increased flash IO pressure, longer data access latency, and reduced overall application throughput. Flash-based key-value stores in Meta are organized into shards. An approach to improve the performance of each shard on DRAM constrained servers is to reduce the number of shards per server. However, this approach can lead to an increase in the total number of servers required, lower storage utilization per server, and dilutes the TCO benefits of the *1P server*. This leaves us with the difficult decision between *1P server*, which is cost-effective while sacrificing performance, or *2P server* with outstanding performance at high cost and power. An alternative solution that we explore in this paper is using recent Intel® Optane™ PMem 100 Series SCMs (DCPMM) [30] to efficiently expand the volatile memory capacity for *1P server* platforms. We use SCM to build new variants of the *1P server* platforms as shown in Figure 3.1c. In *1P server variants*, the memory capacity of *1P server* is extended by providing large SCM DIMMs alongside DRAM on the same DDR bus attached to the CPU memory controller.

Storage Class Memory (SCM) is a technology with the properties of both DRAM and storage. SCMs in DIMM form factor have been studied extensively in the past because of their attractive benefits including byte-addressability, data persistence, cheaper cost/GB than DRAM, high density, and their relatively low power consumption. This led to abundant research focusing on the use cases of SCM as memory and persistent storage. The works range from optimizations with varying memory hierarchy configurations [31–35], novel



(a) 2P server. (b) 1P server. (c) 1P server variant.

Figure 3.1: Server configurations with different DRAM sizes: (a) Server with 256GB memory, high hit rate to DRAM and low IO utilization (b) Server with reduced (64GB) memory, lower DRAM hit rate and increased in IO utilization and (c) Server with reduced memory and SCM added to the memory controller, high hit rate to DRAM & SCM due to optimized data placement, which decreases IO utilization.

Table 3.1: Example of memory characteristics of DRAM, SCM and flash per module taken from product specifications.

Characteristics	DRAM	SCM	Flash
Idle read latency (ns)	75	170	85,000
Read bandwidth (GB/s)	15	2.4	1.6
Power (mW / GB)	375	98	5.7
DRAM Relative Cost per GB	1	0.38	0.017
Granularity	byte addressable	byte addressable	blockbased
Device Capacity (GB)	32	128	2048

programming models and libraries [36–38], and file system designs [39–41] to adopt this emerging technology. Past research was focused primarily on theoretical or simulated systems, but the recent release of DCPMM-enabled platforms from Intel motivates studies based on production-ready platforms [42–51]. The memory characteristics of DRAM, DCPMM, and flash are shown in Table 3.1. Even though DCPMM has higher access latency and lower bandwidth than DRAM, it has a much larger density, lower cost, lower power consumption, and its access latency is two orders of magnitude lower than flash. Currently, DCPMM modules come in 128GB, 256GB, and 512GB capacities, much larger than DRAM that typically ranges from 4GB to 32GB in a data-center environment. Hence, we can get a tremendously larger density with DCPMM. In addition to using DDR bus for SCM, with recent high bandwidth and low latency IO interconnect like Compute Express Link (CXL) [52, 53], we can expand the memory capacity of our servers with SCM without the limitations of DDR bus. If we efficiently (cost, power, and performance) use this memory as an extension to DRAM, this would enable us to build dense, flexible, servers with large memory and storage, while using fewer DIMMs and lowering the total cost of ownership (TCO).

Although recent works demonstrated the characteristics of SCM [42,45], the performance gain achieved in large commercial data-centers by utilizing SCM remains unanswered. There are open questions on how to efficiently configure DRAM and SCM to benefit large scale service deployments in terms of cost, power and performance. Discovering the use cases within a large scale deployment that profit from SCM has also been challenging. To address these challenges for RocksDB, we first profiled all flashed-based KV store deployments at Meta to identify where SCM fits in our environment. These studies revealed that we have abundant read-dominated workloads, which focused our design efforts on better read performance. This has also been established in previous work [54–56] where faster reads improved overall performance for workloads serving billions of reads every second. Then, we identified the largest memory consuming component of RocksDB, the block cache used for serving read requests, and redesigned it to implement a hybrid tiered cache that leverages the latency difference of DRAM and SCM. In the hybrid cache, DRAM serves as the first tier cache accommodating frequently accessed data for fastest read access, while SCM serves as a large second tier cache to store less frequently accessed data. Then, we implemented cache admission and memory allocation policies that manage the data transfer between DRAM and SCM. To evaluate the tiered cache implementations we characterize three large production RocksDB use cases at Meta using the methods described in [57] and distilled the data into new benchmark profiles for db_bench [58]. Our results show that we can achieve 80% improvement to throughput, 20% improvement in P95 latency, and 43-48% reduction in cost for these workloads when we add SCM to existing server configurations. In summary, we make the following contributions:

- We characterized real production workloads, identified the most benefiting SCM use case in our environment, and developed new db_bench profiles for accurately benchmarking RocksDB performance improvement.
- We designed and implemented a new hybrid tiered cache module in RocksDB that can manage DRAM and SCM based caches hierarchically, based on the characteristics of these memories. We implemented three admission policies for handling data transfer between DRAM and SCM cache to efficiently utilize both memories. This implementation enables any application that uses RocksDB as its KV Store back-end to be able to easily use DCPMM.
- We evaluated our cache implementations on a newly released DCPMM platform, using commercial data center workloads. We compared different DRAM/SCM size server configurations and determined the cost, power and performance of each configuration compared to existing production platforms.

- We were able to match the performance of large DRAM footprint servers using small DRAM and additional SCM while decreasing the TCO of read dominated services in a production environment.

The rest of the paper proceeds as follows. In Section 3.2 we provide a background of RocksDB, the DCPMM hardware platforms, and brief description of our workloads. Sections 3.3 and 3.4 explain the designs and implementation of the hybrid cache we developed. In Section 4.6 we explain the configurations of our systems and the experimental setup. Our experimental evaluations and results are provided in Section 3.6. We then discuss future directions and related works in Section 4.8 and Section 4.9 respectively, and conclude in Section 4.10.

3.2 Background

3.2.1 RocksDB architecture

A Key-value database is a storage mechanism that uses key-value pairs to store data where the key uniquely identifies values stored in the database. The high performance and scalability of key-value databases promote their widespread use in large data centers [29, 59–61]. RocksDB is a log-structured-merge [62] key-value store engine developed based on the implementation of LevelDB [59]. RocksDB is an industry standard for high performance key-value stores [63]. At Meta RocksDB is used as the storage engine for several data storage services.

3.2.1.1 RocksDB components and memory usage

RocksDB stores key-value pairs in a Sorted String Table (SST) format. Adjacent key-value data in SST files are partitioned into data blocks. Other than the data block, each SST files contains Index and Filter blocks that help to facilitate efficient lookups in the database. SST files are organized in levels, for example, Level0 - LevelN, where each level comprises multiple SST files. Write operations in RocksDB first go to an in-memory write buffer residing in DRAM called the memtable. When the buffered data size in the memtable reaches a preset size limit RocksDB flushes recent writes to SST files in the lowest level (Level0). Similarly, when Level0 exhausts its size limit, its SST files are merged with SST files with overlapping key-values in the next level and so on. This process is called compaction. Data blocks, and optionally, index and filter blocks are cached (typically uncompressed) in an in-memory component called the Block Cache that serves read requests

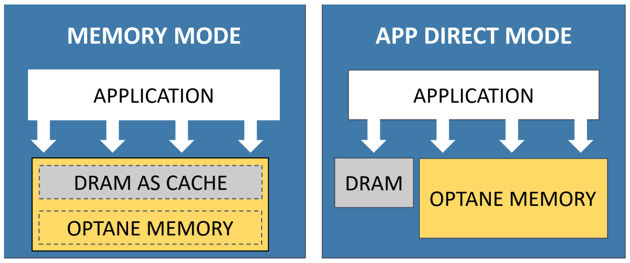


Figure 3.2: Intel® Optane™ memory operation modes overview.

from RocksDB. The size of the Block Cache is managed by global RocksDB parameters. Reads from the database are attempted to be serviced first from the memtable, then next from the Block Cache(s) in DRAM, and finally from the SST files if the key is not found in memory. Further details about RocksDB are found in [29]. The largest use of memory in RocksDB comes from the Block Cache, used for reads. Therefore, in this work we optimize the RocksDB using SCM as volatile memory for the Block Cache.

3.2.1.2 Benchmarking RocksDB

One of the main tools to benchmark RocksDB is db_bench [58]. Db_bench allows us to mock production RocksDB runs by providing features such as multiple databases, multiple readers, and different key-value distributions. Recent work [57] has shown how we can create realistic db_bench workloads from production workloads. To create evaluation benchmarks for SCM we followed the procedures given in [57, 64].

3.2.2 Intel® Optane™ DC Persistent Memory

Intel® Optane™ DC Persistent Memory based on 3D XPoint technology [65, 66], is the first commercially available non-volatile memory in the DIMM form factor and resides on the same DDR bus as DRAM [30]. DCPMM provides byte-addressable access granularity which differentiates it from similar technologies which were limited to larger block-based accesses. This creates new opportunities for low latency SCM usage in data centers as either a volatile memory extension to DRAM or as a low latency persistent storage media.

3.2.2.1 Operation mode overview

DCPMM can be configured to operate in one of two different modes: Memory Mode and App Direct Mode [67]. Illustrations of the modes are shown in Figure 3.2.

In **Memory Mode**, as shown in Figure 3.2, the DRAM capacity is hidden from applications and serves as a cache for the most frequently accessed addresses, while DCPMM capacity is exposed as a single large volatile memory region. Management of the DRAM cache and access to the DCPMM is handled exclusively by the CPU's memory controller. In this mode applications have no control of where their memory allocations are physically placed (DRAM cache or DCPMM).

In **App Direct Mode**, DRAM and DCPMM will be configured as two distinct memories in the system and are exposed separately to the application and operating system. In this case, the application and OS have full control of read and write accesses to each media. In this mode, DCPMM can be configured as block-based storage with legacy file systems or can be directly accessed (via DAX) by applications using memory-mapped files [68].

3.2.3 Meta RocksDB workloads

For our experiments and evaluation we chose the largest RocksDB use cases at Meta, which demonstrate typical uses of key-value storage ranging from messaging services to large storage for processing realtime data and metadata. Note than these are not the only workloads that benefit from our designs. The descriptions of the services are as follows:

3.2.3.1 ChatApp

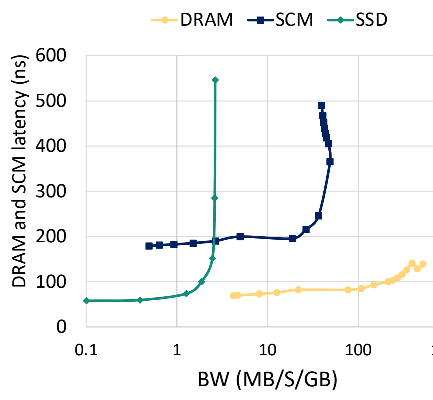
With over a billion active users, ChatApp is one of the most popular messaging applications in the world [69]. ChatApp utilizes ZippyDB as its remote data store. ZippyDB [70] is a distributed KV-store that implements Paxos on top of RocksDB to achieve data reliability and persistence.

3.2.3.2 BLOB Metadata

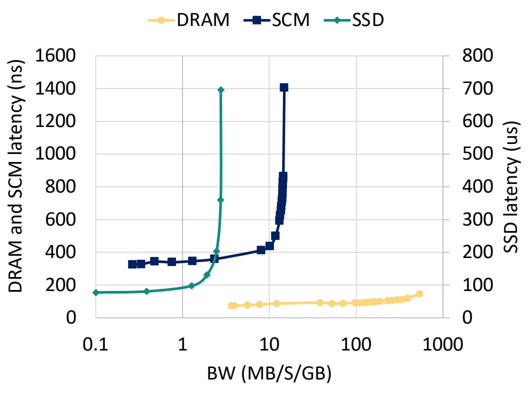
The BLOB Metadata databases are also stored in ZippyDB and are an integral part of the large blob storage service of Meta that serves billions of photos, videos, documents, traces, heap dumps, and source code [71–73]. BLOB Metadata maintains the mappings between file names, data blocks and parity blocks, and the storage nodes that hold the actual blocks. These databases are distributed and fault-tolerant.

3.2.3.3 Hive Cache

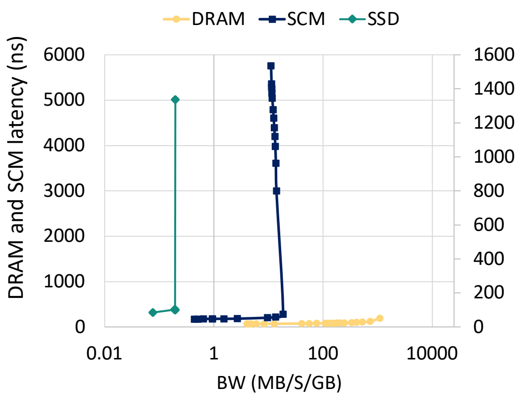
Hive Cache is a high query throughput, low (millisecond) latency, peta-byte scale key-value storage service built on top of RocksDB [74]. Hive Cache reads from any category of Meta's real-time data aggregation services [75] in real-time or from a Hadoop Distributed File System [76] table daily.



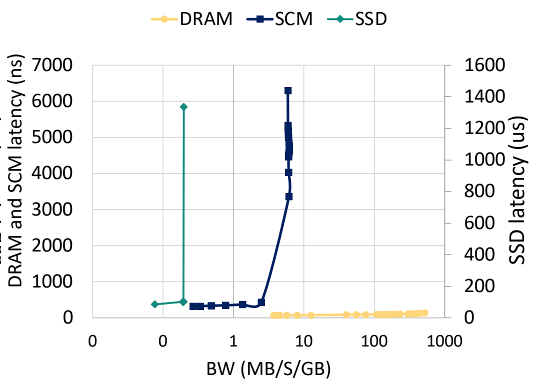
(a) Sequential read



(b) Random read



(c) Sequential read-write (1:1)



(d) Random read-write (1:1)

Figure 3.3: DRAM, SCM, and SSD latency and BW characteristics. x-axis in log scale.

3.2.3.4 Inventory Cache

Inventory Cache contains a diverse collection of objects that might be displayed on a content feed [77]. It is updated rapidly and exhibits more writes than reads.

3.3 Hybrid cache design choices

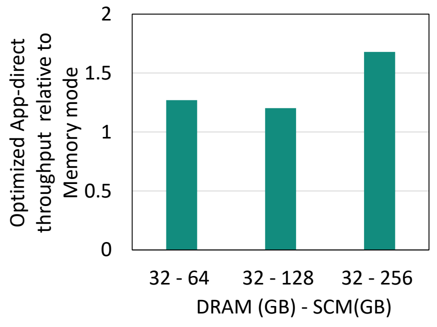
3.3.1 Storage and memory characteristics

Our *1p* and *2p* servers are composed of DRAM memory with SSDs for storage. Our goal is to include SCM in our server designs and evaluate its performance benefits. Before introducing SCM into our systems, we studied the latency and bandwidth characteristics of the three components (DRAM, SCM, and SSD). We used Intel Memory Latency Checker

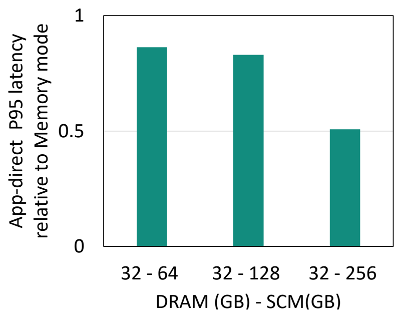
(MLC [78]) to study DRAM and SCM characteristics and Flexible IO tester (FIO [79]) to study SSD characteristics. For all units, we measured the latency vs BW for a single core machine with 32 GB DRAM, 128 GB SCM, and 2TB SSD. We evaluated sequential read, random read, sequential read-write, and random read-write with 1:1 ratio access patterns. To study DRAM and SCM latency we used a 500MB workload and characterized latency and bandwidth using different memory access delays to increase the BW and for SSD we used a 500MB workload with 4KB access granularity using different queue depths to increase the bandwidth. We observe that DRAM latency scales approximately linearly with BW and has similar absolute latencies (75-120ns) across both sequential (Figure 3.3a) and random (Figure 3.3b) read access patterns. Likewise, SSDs, although the latency scales are different (100-750 μs), the latency vs BW characteristics follow the same trend for the sequential and random read. However, for SSDs as the BW utilization increases latency increases exponentially. Hence, optimizing the BW usage of SSDs to decrease latency will improve overall performance for latency-sensitive applications such as RocksDB. For SCMs, we have 190-500 ns latency for sequential access and 350-1400 ns for random access. SCMs BW vs latency curve is also exponential, with the latency increasing exponentially, as we utilize higher BW. SCM shows unique characteristics in that the granularity of access is a critical factor to bandwidth. So 256B sequential access is significantly more performant than 64B random accesses. For mixed read and write (Figure 3.3c and 3.3d), SSDs and SCMs have much lower BW and higher latency than all read workloads. In general, each media has a "knee of the curve" bandwidth target where it can be accessed with reasonable latency. We want to target our usage to this ideal bandwidth for each media to maximize system performance. It will also be beneficial to use SCMs for read-dominated workloads because of the asymmetric read and write latencies/BW.

3.3.2 The challenges of SCM deployment

The first challenge of introducing SCM in RocksDB is identifying which of its components to map to SCM. We chose the uncompressed block cache because it has the largest memory usage in our RocksDB workloads and because our studies reveal that a number of our production workloads, which are read-dominated, benefit from optimizing read operations by block cache extension. We also focused on the uncompressed block cache instead of the compressed ones, so that we can minimize CPU utilization increase when performing compression/decompression. This allowed us to increase the size of SCM (block cache) without requiring additional CPU resources. We also chose block cache over memtable because SCM provides better read bandwidth than writes, hence helping our read-demanding

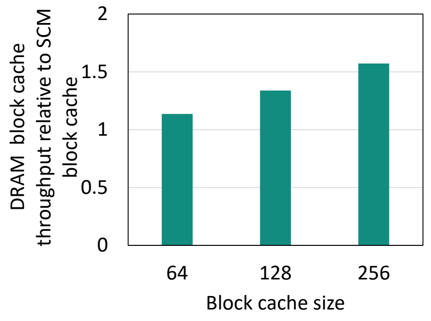


(a)

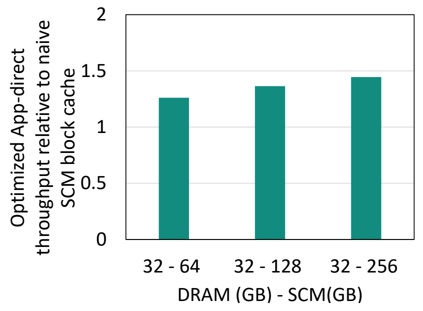


(b)

Figure 3.4: Throughput and latency comparison for memory mode and our optimized hybrid-cache in app-direct mode for ChatApp.



(a)



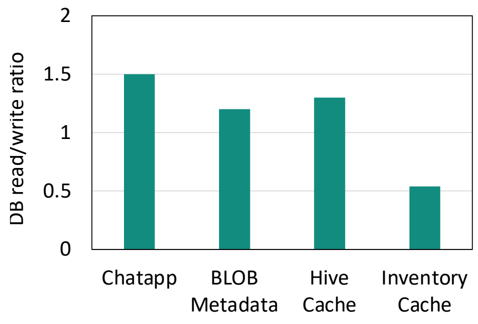
(b)

Figure 3.5: Throughput of ChatApp for DRAM vs SCM block cache (a) and for Naive SCM vs optimized hybrid-cache (b).

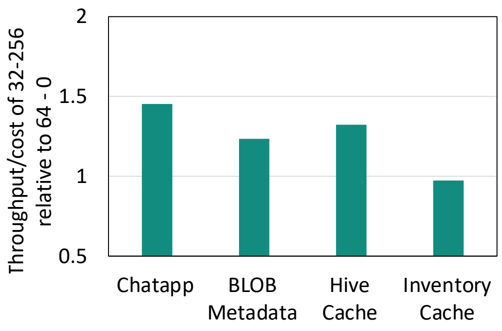
workloads. We then expanded the block cache size by utilizing SCM as volatile memory. We chose this approach because extending the memory capacity while reducing the size of DRAM and the cost of our servers is the primary goal. Although we can benefit from persisting block cache and memtable in SCM for fast cache warmup and fast write access, we left this for future work.

The next challenge is, how we should configure SCM to get the best performance. We have the options of using memory mode, that does not require software architecture changes or app-direct mode that necessitates modification in RocksDB but provides control of DRAM and SCM usage. Figure 3.4 demonstrates how memory-mode compares to our optimized app-direct mode. Optimized app-direct mode with various DRAM and SCM sizes, renders 20-60% throughput improvement and 14-49% lower latency compared with memory mode. This insight supports that our optimized implementation has a better caching mechanism than memory mode, hence we focused our analysis on app-direct mode.

With app-direct we can manage the allocation of RocksDB's components (memtable, data, filter, and index blocks) to DRAM or SCM. But since we know the data access latency



(a) Read to write ratio to DB.



(b) Key-value throughput/cost compare for large block cache friendly workloads and Inventory Cache with higher writes than reads.

Figure 3.6: DB read and writes and throughput/cost comparison of read dominated and write dominated workloads.

of SCM is slower than DRAM (see Table 4.1), we have to consider its effect. We compared the throughput of allocating the block cache to DRAM or SCM in app-direct mode in vanilla RocksDB to understand the impact of the higher SCM access latency. As seen in Figure 3.5a, the slower SCM latency creates 13%-57% difference in throughput when we compare DRAM based block cache to a naive SCM block cache using app-direct mode. This result guided us to carefully utilize DRAM and SCM in our designs. In single-socket machines such as *1P servers*, we have one CPU and 32GB - 64GB DRAM capacity. Out of this DRAM, memtable, index, and filter blocks consume 10-15 GBs. The rest of DRAM and the additional SCM can be allocated for block cache. We compared the naive SCM block cache implementation (all block cache allocated to SCM using app-direct) to a smarter and optimized hybrid cache, where highly accessed data is allocated in DRAM and the least frequently access in SCM. The results in Figure 3.5b show with optimized app-direct we achieve up to 45% better throughput compared to a naive SCM block cache. From this, we can determine that implementing a hybrid cache compensates for the performance loss due to the higher SCM access latency. These results together with the high temporal locality of our workloads (as discussed below) motivated us to investigate a hybrid cache.

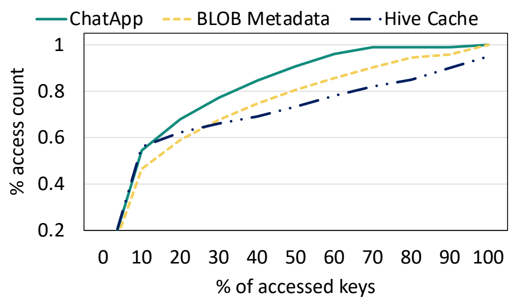
3.3.3 Metrics for identifying workloads

Below we scrutinize the characteristics of our largest RocksDB workloads that guided our hybrid cache design.

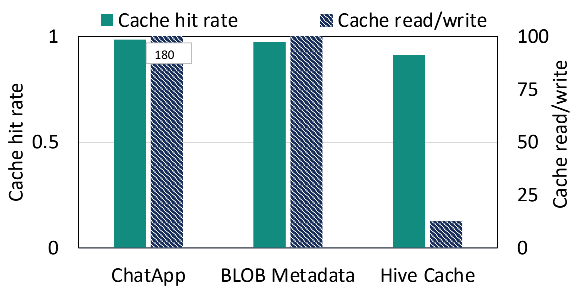
Reads and writes to DB: As we discussed earlier, prior work showed that optimizing reads provides large impact in commercial data center workloads [54–56]. Our studies also show that we have a large number of read-dominated workloads, therefore optimizing the block cache, used for storing data for fast read access, will benefit a number of our workloads. In RocksDB, when a key is updated in memtable it will be invalid in the block cache. Hence, if the workload has more write queries than reads, then the data in the cache will become stale. Note that write-dominated workloads will not be affected by our hybrid cache designs because we did not reduce any DRAM buffer (memtable) in the write path. In our studies, we profiled deployed RocksDB workloads for 24 hours using an internal metrics collection tool to comprehend the read and write characteristics. Figure 3.6a shows the workloads described in Section 3.3.1 reads more bytes from the DB than it writes. To contrast, we evaluated one of our write-dominated workloads, Inventory Cache, also seen in Figure 3.6a. In Figure 3.6b, we calculated the throughput per cost of *1P server variants* with 32 GB DRAM and 256 GB SCM capacity normalized to throughput/cost of *1P server* with 64 GB DRAM capacity. The throughput/cost improvement of Inventory Cache for our largest DRAM-SCM system cannot offset the additional cost due of SCM. Hence, we focus on exporting read-dominated workloads to our hybrid systems.

Key-value temporal locality: Locality determines the cacheability of block data given a limited cache size. A hybrid cache with a small DRAM size will only benefit us if we have a high temporal locality in the workloads. In this case, significant access to the block cache will come from the DRAM cache, and SCM will hold the bulk of less frequently accessed blocks. We used RocksDB trace analyzer [64] to investigate up to 24 hours query statistics of workloads running on production *2P server* and evaluate locality as the distribution of the total database access counts to the total keys accessed per database. Figure 3.7a shows that our workloads possess a power-law relationship [80] between the number of key-value pair access counts and the number of keys accessed. We can observe in the figure that 10% of the key-value pairs carry ~50% of the key-value accesses. This makes a hybrid cache design with small DRAM practical for deployment.

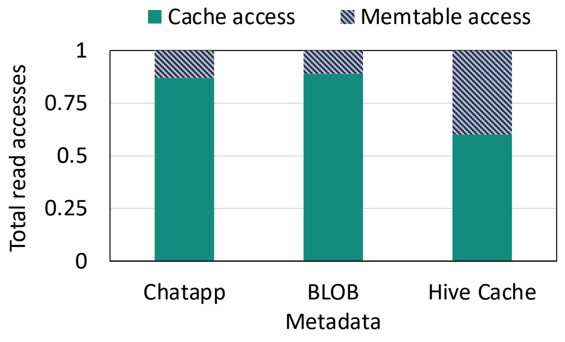
Workload cache utilization: For workloads even with high key-value access locality per database, factors such as reuse distance (number of data access between accessing similar keys) and cache pollution from sharing block cache among multiple shards within a workload can hinder usage of block cache. While improving workloads for better cache utilization is outside of the scope of our project, we studied the current cache utilization of our workloads to understand if a large SCM block cache will give us performance benefits. High block cache hit rate and high read to write ratio in the block cache show the workload is effectively using the caching mechanism. Another circumstance to consider is, despite high key-value locality, if frequently accessed blocks are written to repeatedly then the data will live in the memtable. Here, the workload will not benefit from optimizing the block cache. To study this factor, we looked at the percentage of database accesses that are served from block



(a) Key-value locality.



(b) Cache hit and read/write ratio.

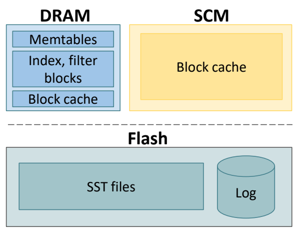


(c) Cache and memtable accesses.

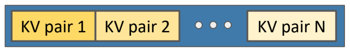
Figure 3.7: Characteristics of ChatApp, BLOB Metadata, and Hive Cache. (a) Key-value access locality shows power-law distribution. (b) Workloads have high cache hit rate and are performing more read that write to the cache. (c) Workloads have more cache access than memtable.

cache and memtable and chose workloads with dominating block cache accesses. Figure 3.7b shows the cache hit rate and read/write ratio of the cache. We can observe from the figure that all workloads have a cache hit rate >90% and have more reads to the cache than writes. Figure 3.7c shows that all workloads have more access to cache than memtable.

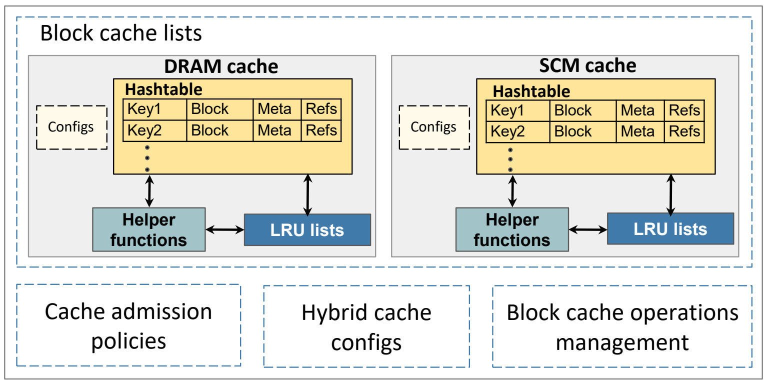
DB and cache sizes: The desirable DRAM and SCM cache sizes required to capture workload's locality is proportional to the size of the DB. Workloads with high key-value locality and large DB sizes can achieve a high cache hit rate with limited cache sizes. But as the locality decreases for large DB sizes, the required cache sizes will grow. In the extreme case of random key-value accesses, all blocks will have similar heat, diluting the value of the DRAM cache and reducing overall hybrid cache performance asymptotically toward that of the SCM-only block cache. For small DBs, locality might not play a significant role because the majority of the DB accesses fit in a small cache. Such types of workloads will not be severely affected by DRAM size reduction, and choosing the *1P server variants* with large SCM capacity will be a waste of resources. In our studies, after looking at various workloads in production, we choose our hybrid DRAM-SCM cache configuration to accommodate



(a) Memory allocation of RocksDB components.



(b) Data block structure.



(c) Hybrid tier cache component and architecture.

Figure 3.8: RocksDB components and memory allocation.

several workloads with larger DB sizes (∼2TB total KVstorage per server).

3.4 DRAM-SCM hybrid cache module

In our RocksDB deployment, we placed the memtables, index blocks, and filter blocks in DRAM. We then designed a new hybrid cache module that allocates the block cache in DRAM and SCM. The database SST files and logs are located in Flash. The overview of RocksDB components allocation in the memory system is shown in Figure 3.8a. Our goal in designing the new hybrid cache module is to utilize DRAM and SCM hierarchically based on their read access latency and bandwidth characteristics. In our design, we aim to place hot blocks in DRAM for the lowest latency data access, and colder blocks in SCM as a second tier. The dense SCM based hybrid block cache provides a larger effective capacity than practical with DRAM alone leading to higher cache hit rates. This dramatically decreases IO bandwidth requirements to the SST files on slower underlying flash media.

The block cache is an integral data structure that is completely managed by RocksDB. Similarly, in our implementations, the new hybrid cache module is fully managed by RocksDB. This module then is an interface between RocksDB and the DRAM and SCM block caches, and fully manages the caches' operations. The overall architecture of the hybrid cache is shown in Figure 3.8c. The details of its internal components are as follows:

3.4.1 Block cache lists

The hybrid cache is a new top-level module in RocksDB that maintains a list of underlying block caches in different tiers. The list of caches are extended from the existing RocksDB block cache with LRU replacement policy. Note that in our implementations we have DRAM and SCM cache, but the module can manage more than these two caches such as multiple DRAM and SCM caches in a complex hierarchy.

3.4.1.1 Block cache architecture and components

The internal structures of DRAM and SCM caches, which are both derived from the block cache, are shown in Figure 3.8c. The block cache storage is divided into cache entries and tracked in a **hashtable**. Each cache entry holds a key, data block, metadata such as key size, hash, current cache usage, and a reference count of the cache entry outside of the block cache. The data block is composed of multiple key-value pairs as shown in Figure 3.8b. Binary searches are performed to find a key-value pair in a data block. The data block size is configurable in RocksDB. In our case, the optimal size was 16KB. As the number of index blocks decreases we can increase the data block size. As a result, with 16KB we were able to reduce the number of index blocks making room for data blocks within our limited DRAM capacity. Every block cache has **configs** that are configured externally. This includes size, a threshold for moving data, a pointer to all other caches for data movement, and the memory allocator for the cache. The cache maintains an **LRU list** that tracks cache entries in order from most to least recently used. The **helper functions** are used for incrementing references, checking against the reference threshold, transferring blocks from one cache to another, checking size limits, and so on. For the components listed above we extended and modified RocksDB to support tiered structure, different kinds of admission policies and we designed new methodologies to enable data movement between different caches and to support memory allocation to different memory types.

3.4.1.2 Data access in the block cache

A block is accessed by a number of external components to the block cache, such as multiple reader clients of the RocksDB database. The number of external referencers is tracked by the reference count. Mapping to a block is created when it is referenced externally, this will increment the reference count. Whereas when the referencer no longer needs a block, mapping is released, and the reference count is decremented. If a block has zero external references, it will be in the hashtable and tracked by the LRU list. If a block gets referenced again, then it will be removed from the LRU list. Note that in the LRU list, newly released blocks with no external references are on the top of the LRU list as the most recently used blocks, and when blocks are evicted, the bottom least recently used blocks are evicted first. The block cache is used for read-only data, hence it doesn't deal with any dirty data management. Therefore, when transferring data between DRAM and SCM we do not have to deal with dirty data.

3.4.2 Cache admission policies

Identifying and retaining blocks in DRAM/SCM based on their access frequencies requires proactive management of data transfer between DRAM, SCM, and flash. Hence, we developed the following block cache admission policies.

3.4.2.1 DRAM first admission policy

In this admission policy, new blocks read from flash are first inserted into the hashtable of the DRAM cache. The block cache data structures are size limited. Hence when the size of the blocks allocated in the DRAM cache exceeds the size limits, the oldest entries tracked by the DRAM LRU list are moved to the next tier cache (SCM cache) by the data mover function of the DRAM cache, using the SCM cache's memory allocator. On lookups, both the DRAM and SCM caches are searched until the cache block is found. If it is not found, it will initiate a flash read. Similar to the DRAM cache when the capacity of the SCM cache exceeds the limit, the oldest entries in the LRU list of the SCM cache are freed to accommodate new cache blocks evicted from the DRAM cache.

3.4.2.2 SCM first admission policy

In this admission policy, new blocks read from flash are first inserted in the hashtable of the SCM cache. Unlike the DRAM first admission policy, this policy has a configurable threshold for moving data from the SCM cache to the DRAM cache. When the external references of cache entries in the SCM cache surpasses the reference threshold, blocks are considered to be hot and will be migrated to the DRAM cache for faster access. The data movement, in this case, is handled by the data mover function of the SCM cache. When the capacity of both the DRAM and SCM caches are full, the oldest LRU blocks are evicted from both caches. In the DRAM cache, LRU entries are moved back to the SCM cache, whereas in the SCM cache, the LRU entries are freed to accommodate new block insertions. On lookup, both the DRAM and SCM caches are searched until the cache block is found.

3.4.2.3 Bidirectional admission policy

In Bidirectional admission policy, similar to the DRAM first admission policy, new data blocks are inserted into the DRAM cache. As the capacity of DRAM and SCM cache reach the limit, the oldest LRU entries are evicted to SCM cache from the DRAM cache and are freed for the case of SCM cache. The difference between DRAM-first and Bidirectional cache is, after the oldest LRU entries are evicted from DRAM to SCM cache, if the external reference to an entry surpasses a preset threshold it is transferred back to the DRAM cache. This property allows us to re-capture fast access performance for blocks with inconsistent temporal access patterns.

In the hybrid cache, we can set the three of the admission policies or we can easily extend a new policy by configuring how to insert, lookup, and move data in the list of block caches. These configs are global parameters in the top-level hybrid cache and are used by the block cache operations manager and list of block caches. Optionally the thresholds for moving data in SCM first and Bidirectional policies can be set to change values based on the current usage of the caches. But in our experiments, we didn't see benefit with changing values. We also performed an analysis with different sizes of cache thresholds and we show the optimal threshold for SCM first and Bidirectional in our evaluations.

3.4.3 Hybrid cache configs

The hybrid cache configurations are set outside of the module by RocksDB, and include pointers to configs of all block caches, the number of block caches, ids, tier numbers of the caches, and admission policy to use. Configs are used during instantiation and at run time to manage database operations.

3.4.4 Block cache operation management

This unit redirects external RocksDB operations such as insert, lookup, update, and so on to the target block cache based on the admission policy. For example, it decides if an incoming insert request should go to the DRAM or SCM cache.

3.5 Systems setup and implementation

3.5.1 DRAM-SCM cache implementation

We configured Intel DCPMMs in App Direct mode using the IPMCTL [81] tool in our experiments. We used Linux Kernel 5.2 that brings support for a volatile use of DCPMM by configuring a hot-pluggable memory region called KMEM DAX. We then used NDCTL 6.7 [82], a utility for managing SCM in the Linux Kernel, to create namespaces in DCPMM in devdax mode. This mode provides direct access to DCPMM and is faster than filesystem-based access. We then used DAXCTL [82] utility for configuring DCPMM in system-ram mode so that DCPMM will be available in its own volatile memory NUMA node. To implement a hybrid DRAM-SCM cache we used memkind library [83], which enables partitioning of the heap between multiple kinds of memories such as DRAM and SCM in the application space. After the system is configured with DRAM and DCPMM memory types, we modified RocksDB block cache to take two types of memory allocators using memkind. We use MEM_KIND_DAX_KMEM kind for SCM cache and a regular memory allocator for DRAM. The overview of our implementation is shown in Figure 3.9.

3.5.2 Evaluation hardware description

In our evaluations we used a Intel Wolf Pass [84] based system, utilizing 2 CPU sockets populated with Intel Cascade Lake processors [85]. Each CPU has 2 memory controllers with 3 channels each, and 2 DIMM slots per channel, for a total of 24 DIMM slots. DIMM slots were populated by default in a 2-2-2 configuration, with 12 total *16GB PC4-23400 ECC Registered DDR4 DRAM DIMMs* and 12 total *128GB Intel® Optane™ PMem 100 DIMMs*. This makes up a total of 192GB DRAM and 1.5TB of SCM per system. Backing storage for the RocksDB database was a *Samsung 983 DCT M.2 SSD*. The detailed specification is listed in Table 5.1.

3.5.3 Meta server designs

The Meta platforms used for TCO analysis are the *2P server* platform based on the OCP Tioga Pass specification [86], and the *1P server* platform based on the OCP Twin Lakes specification [87]. In addition, we propose several *1P server* variants utilizing differing capacities of DRAM and SCM. The detailed specifications and relative costs of these platforms compared to the baseline *2P server* platform are listed in Table 3.3 and 3.4.

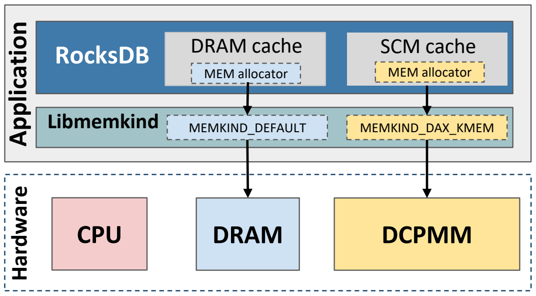


Figure 3.9: DRAM-SCM cache configuration with libmemkind.

Table 3.2: System setup: hardware specs and software configs.

Specification	System config
Application version	RocksDB 6.10
OS	CentOS-8
Linux kernel version	5.2.9
CPU model	Intel(R) Xeon(R) Gold 6252 @ 2.10GHz
Socket/Cores per socket	2/24
Threads total	96
L1I/L1D cache	32 KB
L2/L3 cache	1 MB/35.75 MB
Memory controllers total	4
Channels per controller	3
Slots per channel	2
DRAM size	DDR4 192 GB (16 GB X 12 DIMM slots)
SCM size	DDR-T 1.5 TB (128 GB X 12 DIMM slots)
SSD model	Samsung 983 DCT M.2 NVMe SSD
SSD size/filesystem	2 TB / xfs

Table 3.3: OCP Tioga Pass (2P server) and OCP Twin Lakes Platforms (1P server) details.

Specification	OCP Tioga Pass Config	OCP Twin Lakes Config
CPU model	Xeon(R) Gold 6138	Xeon(R) D-2191
Sockets/cores per socket	2/20	1/18
Threads total	80	36
Memory controllers total	4	2
Channels per controller	3	4
DRAM size	DDR4 256 GB	DDR4 64 GB
SSD size	2 TB	2 TB

Platform costs are calculated based on current OCP solution provider data [88, 89] and DCPMM cost relative to DRAM provided in [90, 91]. The relative cost of DRAM and DCPMM are predicted to maintain similar trends over time [90]. We calculated the cost by adding the cost of individual modules. For DRAM and SCM, we used 16GB and 128GB modules, respectively. In the TCO calculations although introducing SCM adds additional power cost, when we consider the overall power of the system including CPU, NIC, and SSD, the increase of power even for our largest 1P server becomes minimal. We have a power budget for a rack of servers with some power slack and the slight rise in power for SCM is sufficiently below our rack power budget.

Table 3.4: DRAM and SCM cache in server configuration and memory sizes used block cache in our experiments.

Configuration	2P serv.	1P serv.	64-128	32-128	32-256
DRAM size	256	64	64	32	32
SCM size	0	0	128	128	256
DRAM Block cache size	150	40	40	12	12
SCM Block cache size	0	0	100	100	200
Other Rocksdb components	10-15	10 -15	10 - 15	10 -15	10 -15
Codebase	5	5	5	5	5
2P rel. cost	**1**	**0.43**	**0.5**	**0.46**	**0.53**

3.5.4 Cache and memory configurations

To experiment with SCM benefits when added to existing configurations we examined server configurations with different sizes of DRAM and SCM. The configurations we used are shown in Table 3.4. Our experiments verified that *1P server* has significant performance loss compared to *2P server*. The prominent questions here are how much benefit can we achieve by adding SCM to *1P servers*, and can we still maintain the TCO benefits of *1P server* platform. Hence we used the *1P server* with 64 DRAM and no SCM as the baseline for our evaluations. We then evaluated how much performance we can gain by adding 128 GB SCM to the baseline. Since we are interested in DRAM constrained server configurations, we also evaluate the performance gain when we further reduce DRAM to 32 GB while adding 128 or 256 GB SCM. This gives us the four server configurations provided in Table 3.4. Although we experimented with 64GB of SCM, as seen in Section 3.3, to understand the performance of different SCM sizes, DCPMM is not available as a 64GB module, so we didn't consider it in the evaluation below. To run all of the server configurations, we limited the memory usage of the DRAM and SCM for the workloads to the sizes given in Table 3.4. We also use 1 CPU in our experiments because *1P servers* are single-socket machines. We aimed to maximize block cache allocation to evaluate our DRAM-SCM hybrid cache. To do this we studied the memory usage of other components in RocksDB when it runs in our production servers. We then subtracted these usages from total memory and assigned the rest of the memory for the Block cache. The block cache sizes we used in our evaluations are illustrated in Table 3.4.

3.5.5 Workload generation

To generate workloads, we first selected random sample hosts running ChatApp, BLOB Metadata, and Hive Cache in production deployments to collect query statistics traces, being careful to select leaders for use cases relying on Paxos. Note that hosts running the same services manifest similar statistics. Then we followed the methodology in [57, 64] to model the workloads. We also extended these methodologies [57, 64] to scale workloads to match the production deployments. The db_bench workloads we developed mirror the following characteristics of production RocksDB workloads.

3.5.5.1 KV-pair locality

This is characterized by fitting real workload trace profiles to a probability cumulative function using power distributions in MATLAB [92] based on the power-law characteristics of the workloads (see Figure 3.7a).

3.5.5.2 Value distribution

The granularity of value accessed is modeled using Pareto distribution from workload statistics in MATLAB [93].

3.5.5.3 Query composition

The percentage of get, put, and seek quires are incorporated in the db_bench profiles.

3.5.5.4 DB, key, and value sizes

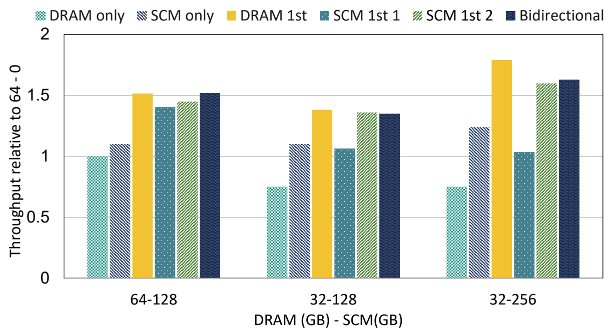
We added the average values of the number of keys per database, key, and value size from collecting data from our production servers to create a realistic DB sizes in the db_bench profile.

3.5.5.5 QPS

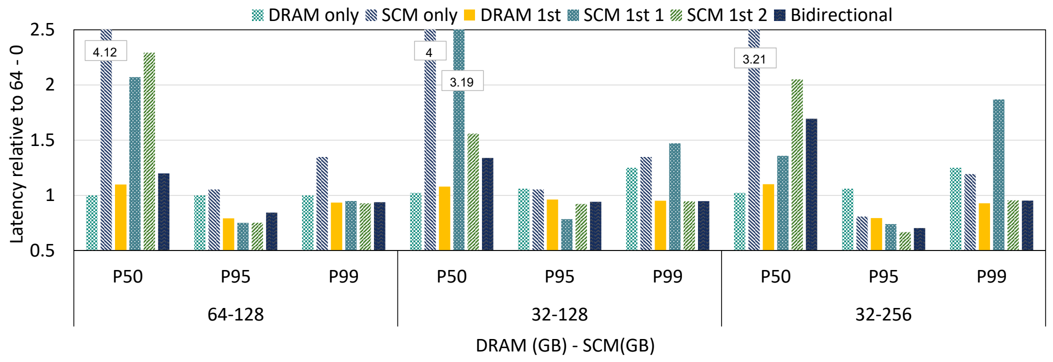
The QPS in db_bench is modeled using sine distributions [94] based on trace collected in the production host.

3.5.5.6 Scaled db_bench profiles

After generating the workloads db_bench profile for a singe database we scaled the workloads by running multiple RocksDB instances, simulating the number of production shards per workload on a single host. These shards share the same block cache. In RocksDB there exists multiple readers and writers to the database. To imitate this property we run multiple threads reading and writing to the set of shards in the db_bench process.



(a) Throughput comparison of admission policies.



(b) Latency comparison of admission policies.

Figure 3.10: Throughput and application read latency comparison using only DRAM for block cache, using only SCM for block cache, and hybrid DRAM-SCM admission policies (DRAM first, SCM first, and Bidirectional) for ChatApp using all server configurations in Table 3.4. (a) Throughput comparisons (db operations/sec) for admission policies. Here the baseline is 64 - 0 configs. (b) P50, P95, and P99 latencies for all admission policies and server configs compared to 64 - 0 server.

3.6 Evaluation

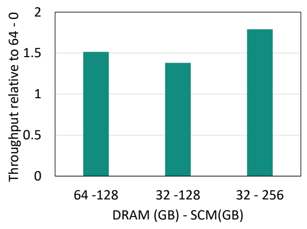
3.6.1 Throughput and latency comparison for admission policies

In Figure 3.10 we compare the throughput achieved for 5 different categories: **DRAM Only**: The Block Cache is only allocated in DRAM. **SCM Only**: The Block Cache is only allocated in SCM using app-direct mode. **DRAM First**: The DRAM first policy discussed in Section 3.4. **SCM First 1 and 2**: The SCM first policy discussed in Section 3.4 with two threshold values. SCM 1st 1, with threshold value of 2 and SCM 1st 2 with the optimal threshold which is 6. **Bidirectional**: The Bidirectional policy discussed in Section 3.4 with the optimal threshold value which is 4.

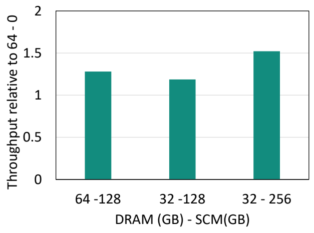
In Figure 3.10a, we demonstrate the throughput differences of all admission policies for ChatApp. The results show that using only SCM for block cache provides 15% throughput improvement for the 128 GB SCM configurations and 25% improvement for the 256GB SCM compared to the baseline (a server configuration with 64GB DRAM and no SCM). If we look at Figure 3.10b, because of latency differences between SCM and DRAM, getting data only from SCM worsens the P50 application read latencies. Therefore we conclude that while the default RocksDB SCM implementation may decrease flash IO utilization, it will have a net negative impact on Meta application performance due to the $2\times$ - $4\times$ worse P50 latency observed. We can also see in the figure for all the server configurations DRAM first policy achieves the best performance. For 64 - 128 and 32 - 128 configurations, SCM first and Bidirectional get close in throughput benefit to DRAM first, but when there is a large block cache, like in the 32 - 256 configuration DRAM first attains the best result. This is because, in DRAM-first, data transfer between DRAM and SCM cache only occurs once, when DRAM cache evicts data to SCM. But in the case of SCM and Bidirectional policies, data transfer occurs when DRAM evicts data to SCM and when hot blocks are transferred from SCM to DRAM. This creates more bandwidth consumption across the DDR bus resulting in performance degradation, especially for configurations with large block cache sizes. For larger DRAM capacities, SCM first 1, SCM first 2, and Bidirectional policies have comparable throughput because the large DRAM size reduces data evictions to SCM. But as DRAM is reduced (in 32 -128), SCM 1st throughput falls quickly because it will move data from SCM to DRAM with a low activation threshold. When we increase DCPMM capacity in the 32 - 256 case, data transfer increases even for SCM 2 and Bidirectional policies, hence the DRAM first policy is the overall performance winner. If we look at read latencies shown in Figure 3.10b, the P50 latency remains similar for DRAM first compared to 64 - 0 configurations. The reason for this is that P50 latencies are primarily governed by DRAM accesses. The effect of data transfer from flash instead of SCM can be observed in the P95 and P99 latencies, where the DRAM First policy does significantly better than other policies and the default configuration with no SCM. BLOB Metadata and Hive Cache (not shown here) also attain the best performance with DRAM first policy.

3.6.2 Performance comparison of DRAM first policy for all workloads

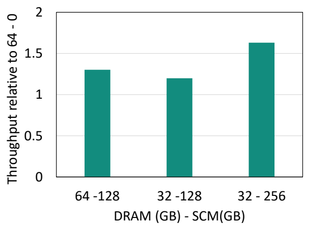
Figure 3.11 shows the throughput and latency comparison of ChatApp, BLOB Metadata, and Hive Cache for DRAM first admission policy (**the best admission policy for all workloads**) for all server configurations. As seen in the figure, our hybrid block cache implementation provides throughput improvement for all the workloads. As seen in Figure 3.11a, 3.11b, and 3.11c throughput is increased up to 50 - 80% compared to the baseline *1P server*'s 64 - 0 due to the addition of SCM. The throughput increase is correlated to



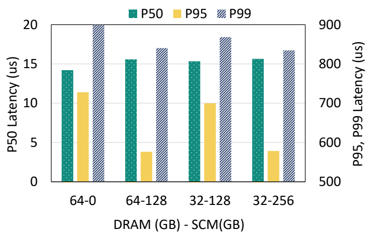
(a) **ChatApp:** throughput comparison.



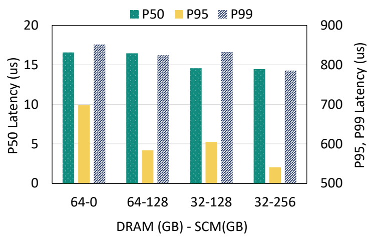
(b) **BLOB Metadata:** throughput comparison.



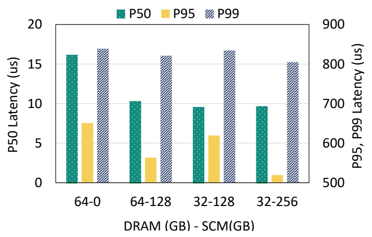
(c) **Hive Cache:** throughput comparison.



(d) **ChatApp:** latency comparison.



(e) **BLOB Metadata:** latency comparison.



(f) **Hive Cache:** latency comparison.

Figure 3.11: Throughput and application read latency comparison for ChatApp, BLOB Metadata and Hive Cache for DRAM first admission policy using all server configurations shown in Table 3.4. (a,b, and c) Throughput comparisons (db operations/sec). Here the baseline is 64 - 0 configs. (d, e and f) absolute P50, P95, and P99 latencies for all workloads and server configs.

the total size of the block cache. Note that increasing the SCM or DRAM capacity further than 256GB will require either more DIMM slots or higher density DIMMs, with different price/performance/reliability considerations. The size of the database also impacts locality and the maximum throughput benefit, as discussed in Section 3.3. Because BLOB Metadata has a larger DB size than ChatApp or Hive Cache the throughput benefit is expected to be smaller for each configuration. Looking at application level read latency in Figure 3.11d, 3.11e, and 3.11f, we observe that P50 latency is relatively stable for ChatApp and BLOB Metadata. While P50 latency does improve for Hive Cache, the absolute magnitude of the improvement is less significant than the improvements to tail latency. P95 and P99 show an overall improvement of 20% and 10% respectively for all services. The P50 latencies primarily reflect situations where the data is obtained from DRAM. The benefit of SCM is reflected in P95 and P99 scenarios where in one case the data is in SCM, while the default case the data is in flash storage.

P50 Write latencies for all workloads is shown in Figure 3.12. In all cases, P50 write latency stays similar since we are only optimizing the block cache used for reads, while writes always go to the memtables, residing in DRAM. We see a slight increase in 64-128 configuration because, in this size of DRAM, we have an extra copy of blocks between DRAM and SCM that increases bandwidth utilization and hence slightly increases the

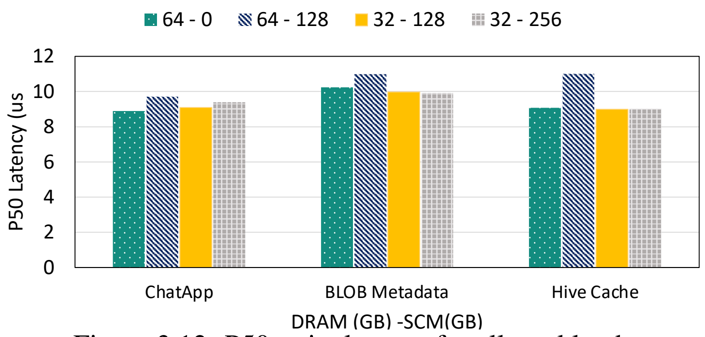


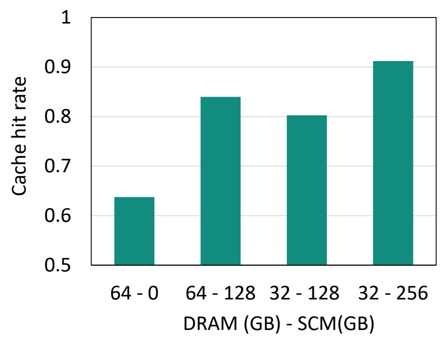
Figure 3.12: P50 write latency for all workloads.

latency. P95 and P99 latency also stay similar for all configurations.

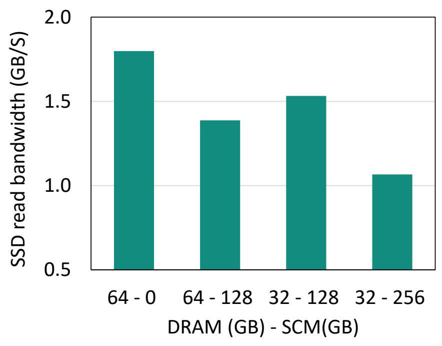
3.6.3 IO bandwidth, cache and CPU utilization

Figure 3.13 illustrates the cache hit rate, IO bandwidth, and latency improvement of DRAM first policy for ChatApp. As seen in Figure 3.13a, the higher capacity of the block cache (sum of DRAM and SCM cache) leads to a higher cache hit rate. We show in Figure 3.13a, that for ChatApp the hit rate increases up to 30% and the increase is correlated to the cache size. BLOB Metadata and Hive Cache (not shown here) also follow a similar pattern of increasing hit rates.

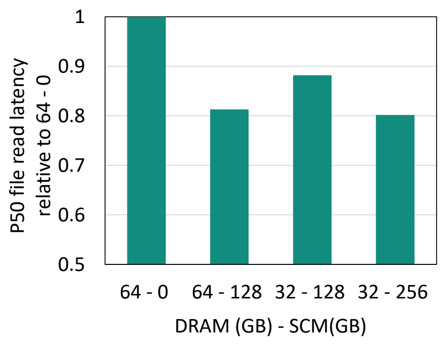
Another important indicator explaining throughput gain is SSD bandwidth utilization. As the cache hit rate increases for the server configurations with SCM it translates to less demand for read access from the SSD, and therefore decreased IO read bandwidth. Figure 3.13b shows for ChatApp adding SCM reduces SSD read bandwidth by up to 0.8 GB/s, or roughly 25% of the SSD's datasheet max read bandwidth. Figure 3.13c shows the file read latency improvement for all server configurations relative to 64 - 0. Decreased demand for read IO bandwidth improves the P50 latency by up to 20%. Latencies at higher percentiles stay the same because there are still scenarios where the IO queue will be saturated with reads, which drives worst-case latency. But for the majority of the requests, latencies are improved because of the decrease in IO bandwidth. The other workloads also show similar patterns. Figure 3.13d shows the CPU utilization for all server configurations. We observe that the CPU utilization increases as the block cache size increases. This is due to the increase in CPU activity as we increase amount of data accessed with low IO wait latency from the cache. One thing to note is, even though CPU utilization increase for our new *1P server variants*, we can still safely service the workloads with 1 CPU even in the largest 256 SCM configuration.



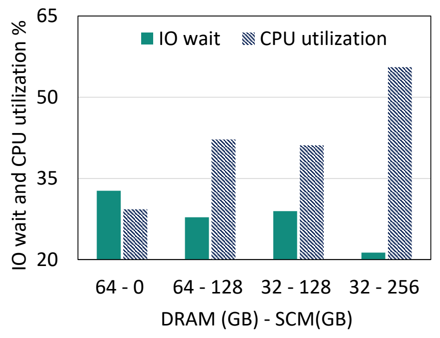
(a) Cache hit rate.



(b) SSD read bandwidth.



(c) File read latency.

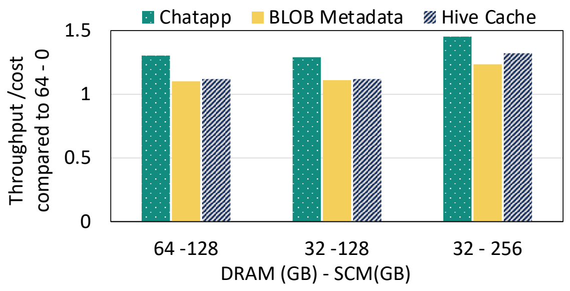


(d) IO wait and CPU utilization.

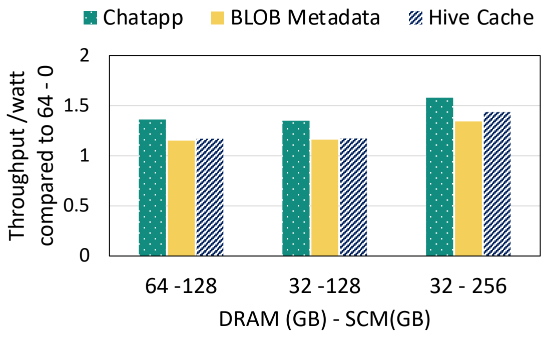
Figure 3.13: Cache utilization, IO read bandwidth, IO read latency, IO wait and CPU utilization of ChatApp for DRAM 1st admission policy.

3.6.4 Cost, performance and power

In the above sections, we aim to understand the performance achieved for different DRAM and SCM variations of the *1P server* configuration. The large capacity of SCM per DIMM slot enables us to dramatically increase the memory capacity of the platform without impacting the server motherboard design. As seen in Table 3.4 adding SCM increases the cost of a server. Figure 3.14a estimates the performance per cost compared to baseline *1P server* (64 - 0) configuration. We observe in the figure that the 32 - 256 configuration gives the best cost relative to performance across all workloads. In this configuration the 23% cost increase over the 64 - 0 baseline produces a 50% - 80% performance improvement. To a smaller degree the 64-128 and 32-128 configurations also provide performance-relative cost benefits over the standard *1P server*. Notable is the fact that the benefit across these two configurations is nearly identical due to the proportional difference in DRAM cost vs. performance increase. If future DRAM/SCM hardware designs provide additional flexibility across capacity & pricing then we may discover new configurations which achieve even larger TCO benefits. We show performance per watt for *1P server variants* relative to *1P server* (64 - 0) in Figure 3.14b. Similarly to the performance per cost, the performance



(a)



(b)

Figure 3.14: Throughput/cost and throughput/watt of ChatApp, BLOB Metadata and Hive Cache normalized to 64 - 0 1P servers throughput/cost.

benefit of adding SCM still offsets the increase in power per platform compared to *1P server* (64 - 0). We achieve 30% to 55% more performance/watt with *1P server variants*, making SCM a power optimized solutions.

In Figure 3.15, we present a throughput comparison between the *2P server* and the various *1P server* configurations. The figure shows that increasing the amount of SCM brings throughput closer to parity with the *2P server* for a minimal increase in relative cost. While the baseline *1P server* (64 - 0) configuration only achieves 50 - 60% of the performance of a *2P server*, the 32 - 256 *1P* variant raises relative performance to 93 - 102%. By dividing the relative cost of the platforms (Table 3.4) by their relative performance (Figure 3.15) for each workload we derive the performance-equivalent TCOs in Table 3.5. In the case of the 32 - 256 configuration, improving *1P* performance with SCM improves the relative TCO to 0.52 - 0.57 of the *2P server*. Therefore, we demonstrate that deploying SCM configurations of *1P servers* instead of *2P servers* results in an **overall cost savings of 43% - 48%** across some of the largest RocksDB workloads at Meta.

The relative power of *1P* and variants relative to *2P* is shown in Figure 3.15. Adding SCM increases power by up to 13% for the 32 - 256 *1P* variant compared to *1P*. But because we can improve performance by 50-80% the overall number of servers required per service will be much less than *1P* hence reducing total power consumption. To examine the power benefits of SCM, we compared the maximum power for *2P*, *1P*, and 32 - 256 *1P* variant. Table 3.5 shows relative power reduction of *1P* and variants relative to *2P*. From the table, we can see that with SCM we can reduce the overall power requirement of services by 54%-60%. Even when we compare it with the 64-0 server, the performance gain allows us to deploy fewer racks, hence it's an overall power benefit. Note that the additional SCM increases power per server, this increase in power enforces us to reduce the number of servers per rack since we have fixed power budges per rack.

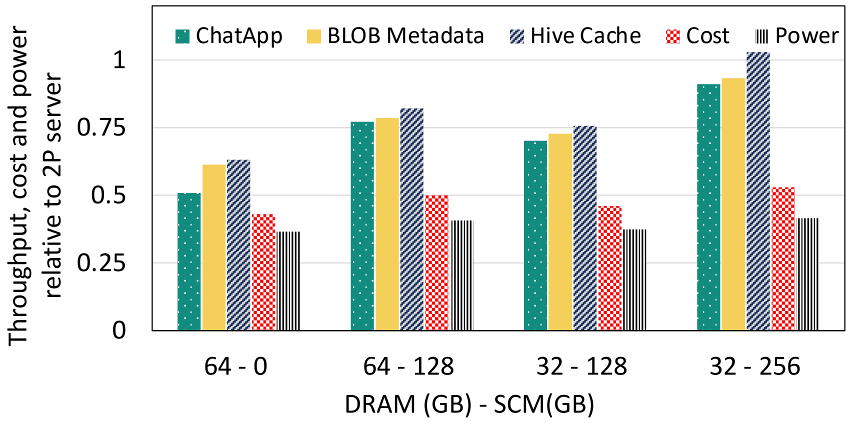


Figure 3.15: Throughput of 1P and its variants compared to 2P servers.

Table 3.5: Performance equivalent TCO relative to 2P.

Server types	2P server	1P server	1P (32 - 256)
Relative TCO	1.0	0.72 - 0.86	0.52 - 0.57
Relative power	1.0	0.6 - 0.82	0.4 - 0.46

Even though the performance and cost of using SCMs are impressive for the chosen workloads, and the performance gain can be translated to other read-dominated workloads in our environment, a discussion on whether we should have a deployment of SCMs in Meta at scale is outside the scope of this paper. We briefly discuss some additional challenges of mass-scale deployment:

Workloads: Section 3.3 talks about the class of applications that would benefit from SCM. We have also identified a number of write-heavy workloads at Meta that would not benefit from SCMs.

Reliability: Since SCMs are not widely available in the market, the reliability of SCMs is a concern until they have been proven in mass deployments.

SKU diversification: Adding a new hardware configuration into the fleet requires consideration of other costs like maintaining a separate code path and creating a new validation and sustainability workflow. This complexity and cost will be added to the practical TCO of any new platform deployment.

3.6.5 General takeaway

From this project, we learned three key design factors to consider for SCM deployment in data centers.

1. **Performance:** SCMs have significantly higher accesses latency and lower bandwidth than DRAM. We can not replace DRAM with SCM without understanding the workloads that are going to run on SCM. It is beneficial to understand the locality of the workload to map highly accessed data in DRAM and low accessed data in SCM to efficiently hide the performance differences between DRAM and SCM.

2. **Cost:** The cost of SCM drives the deployment of SCM. The cost of SCM should scale with the cost of DRAM to be an alternative solution to DRAM. We should always evaluate the performance of our workload for DRAM and SCM and compare how the cost difference gives us TCO benefits. For example, as shown in Figure 3.6b, for inventory Cache workload, though we get added performance with SCM when we compare the performance per cost, the advantage of adding SCM becomes minimum.

3. **SKU:** In general, it's favorable for deployment at scale to limit the number of different hardware platforms. While it would be possible to design a new platform with SCM specifically tailored to one workload and fine-tune performance per workload to get higher performance, it would not likely be practical due to constraints on placement, disaster recovery, and breaking fungibility with previous hardware generations. Hence, it is important to consider SCM benefits across the sum of all workloads and impact to global hardware deployment.

This is a work of a particular Systems research group within Meta. Even though it shows benefits in performance and cost for SCMs, the hardware roadmap of Meta is determined by a large number of complex factors. Therefore the results and use cases illustrated in this paper may not necessarily lead to vast deployment of SCM within the Meta infrastructure.

3.7 Discussion and future work

In the previous section, we demonstrated KV Stores based on RocksDB are examples of the potential advantages of SCM in a large data center. In the future, we want to experiment with extending the memory capacity of other large memory footprint workloads using SCM. Some candidate large-scale workloads that will profit from large memory capacity are Memcached [95, 96] and graph cache [56]. Memcached is a distributed in-memory key-value store deployed in large data centers. Optimizing Memcached to utilize SCM will enable an extension of memory beyond the capacity of DRAM. Graph cache is a distributed read optimized storage for social graphs that exploits memory for graph structure-aware caching. These workloads are read-dominated and have random memory accesses that can benefit from the high density and byte-addressable features of SCM. Although in this paper we did not leverage the persistent capability of SCM for RocksDB uncompressed block cache, in the future we want to study the benefits of fast persistent SCM for the memtable and SST files. We also want to explore with SCM is its performance via emerging connectivity technology such as Compute Express Link (CXL) [52]. The workloads we analyzed in this paper are more latency-bound than memory bandwidth-bound, but for high memory bandwidth-demanding services, sharing DRAM and SCM on the same bus will create interference. In such cases having dedicated SCM access via CXL will avoid contention, but at the same time will potentially increase data access latency, requiring careful design consideration.

3.8 Related work

Performance analysis and characterization in DCPMM: Recent studies proved the potential of commercially available Intel® Optane™ memory for various application domains. [46,97] has determined the performance improvement of DCPMM in memory and app-Direct modes for graph applications. [47,48] evaluated the performance of DCPMM platform as volatile memory compared to DRAM for HPC applications. DCPMM performance for database systems were shown in [43,98–101] both as a memory extension and for persisting data. Works such as [42,45,51,102] also shows the characteristics and evaluations of DCPMM when working alone or alongside of DRAM. Specifically, [45] has identified the deviation of DCPMM characteristics from earlier simulation assumptions. While these works shed light on the usage of DCPMM for data-intensive applications, in our work, based on the memory characteristic findings of these work, we analyzed the performance of DCPMM for large data data-center production workloads. Our work focus on utilizing the DCPMM platform to the best of its capability and study its possible usage as a cost-effective memory extension for future data-center system designs.

Hybrid DRAM-SCM systems: Previous works studied hybrid DRAM-SCM systems to understand how we can utilize these memories with different characteristics together, and how they influence the existing system and software designs. [103–106] have shown the need for a redesign of existing key-value stores and database systems to take into account the access latency differences between DRAM and SCM. Similarly, by noting the latency differences in these memories, we carefully place hot blocks in DRAM and colder blocks in SCM in our implementations. When deploying hybrid memory, another question that arises is, how to manage data placement between DRAM and SCM. In these aspects, [107] demonstrated efficient page migration between DRAM and SCM based on the memory

access pattern observed in the memory controller. In addition, [108–116] perform data/page transfer by profiling and tracking information such as memory access patterns, read/write intensity to a page/data, resource utilization by workloads, and memory features of DRAM and SCM, in hardware, OS, and application level. These works aim to generalize usage of DRAM and SCM to various workloads without involving the application developer, hence requiring hardware and software monitoring that is transparent to the application developers. But in our case, the RocksDB application-level structure exposes separate reads and writes paths and frequency of access of data block. These motivated us to implement our designs in software without requiring any additional overhead in the OS and hardware.
RocksDB performance improvements: [117] demonstrated how to decrease the memory footprint of MyRocks, which is built on top of RocksDB, using block access based non volatile memory (NVM) by implementing secondary block cache. While their methods also decrease DRAM size required in the system, the block-based nature of NVM increases read amplification. This is because, the key-value size in RocksDB is significantly less than the size of the block, whereas in our methods, using byte addressable SCM avoids such issues.

3.9 Conclusion

The increasing cost of DRAM has influenced data centers to design servers with lower DRAM per compute ratio. These servers have shown to decrease the TCO for scalable workloads. Nevertheless, this type of system design diminishes the performance of large memory footprint workloads that relies on DRAM to cache hot data. Key Value stores based on RocksDB is one such class of workloads that is affected by the reduction of DRAM size. In this paper, we propose using Intel® Optane™ PMem 100 Series SCMs (DCPMM) in AppDirect mode to extend the available memory for RocksDB to mitigate performance loss in smaller DRAM designs while still maintaining the desired lower TCO of smaller DRAM systems. We carefully studied and redesigned the block cache to utilize DRAM for the placement of hot blocks and SCM for colder blocks. Our evaluations show up to 80% improvement to throughput and 20% improvement in P95 latency over the existing small DRAM platform when we utilize SCM alongside DRAM, while still reducing the cost by 43-48% compared to large DRAM designs. To our knowledge, this is the first paper that demonstrates practical cost-performance trade-offs for potential deployment of DCPMM in commercial datacenters.

CHAPTER 4

Improving DLRM Training Efficiency Using Heterogeneous Compute and Memory Systems

Recommendation models are very large and require Terabytes (TB) of memory during training. In pursuit of better quality, the model size and complexity grow over time, which requires additional training data to avoid overfitting. This model growth demands a large number of resources in data centers. Hence, the efficiency of training is becoming considerably more important to keep the data center power demand manageable. In Deep Learning Recommendation Models (DLRM), sparse features capturing categorical inputs through embedding tables are the major contributors to model size and require high memory bandwidth. In this paper, we study the bandwidth requirement and locality of embedding tables in real-world deployed models at Meta, and observe that the bandwidth requirement is not uniform across different tables, and that embedding tables show high temporal locality. We then design MTrainS, which leverages hierarchical memory, including byte and block addressable Storage Class Memory for DLRM. This allows for higher memory capacity per node and increases training efficiency by lowering the need to scale out to multiple hosts in memory capacity-bound models. By optimizing the platform memory hierarchy, we are able to reduce the number of nodes for training by up to 8×, saving power and cost of training while meeting our target training performance.

4.1 Introduction

Recommendation models are broadly deployed in big technology companies to personalize the experience of their audience. For example, Google uses such models for personalized advertisements [118], Amazon and Alibaba for recommending items in their catalogs [119, 120], Microsoft for recommending news to users [121], and Meta for ranking and click-through prediction [122].

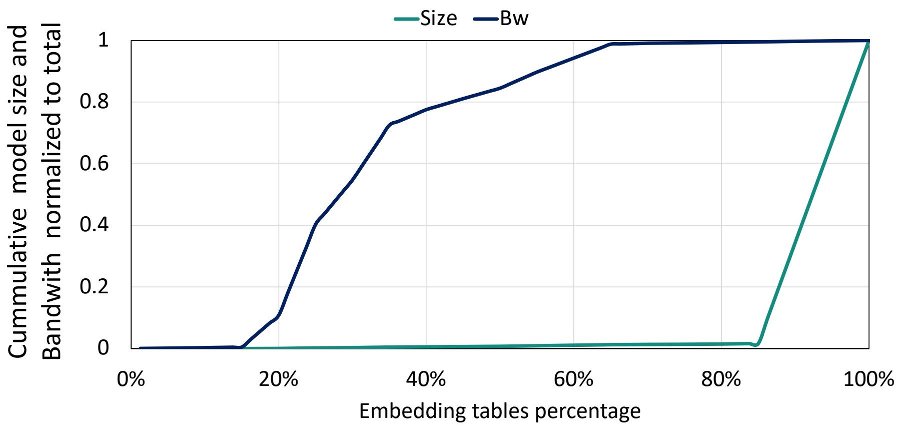


Figure 4.1: Cumulative bandwidth and Memory size for one of the real world models we evaluate.

Recommendation models are very large, requiring Terabytes (TB) of memory during training [123], and 100s of Gigabytes (GB) during inference [124]. Accelerator-enabled platforms such as GPU-enabled systems [125], with 10s to 100s of accelerators, are commonly used to train such models [123, 126]. These models take a significant amount of resources in data centers. For example, recommendation model training consumes over 50% of AI training resources at Meta [127]. In pursuit of better recommendation quality, both model size and complexity are increasing by more than 1.5$\times$ year over year [128], which requires additional training data to avoid overfitting. Training models with n times more parameters, requiring m times more data, at the same speed, increase the resource and power demand at $O(n \times m)$. Hence, improving the efficiency of recommendation model training to manage the resource and power demand is becoming increasingly important in data centers.

Deep Learning Recommendation Models (DLRM) are neural network-based personalization and recommendation models [122]. In DLRM, sparse features capturing categorical inputs through embedding tables are the major contributor to the TB scale model size, while dense features composed of Multi-Layer Perceptron (MLP) contribute to the model compute complexity. In addition to significant memory capacity requirements, sparse features may require high memory bandwidth. Due to DLRM's considerable resource requirements and growth rate, the typical solution to accommodate these models in data centers is to scale out the model to multiple hosts. We can categorize reasons for scaling out a DLRM deployment beyond a single host at Meta's data center as follows:

- **Memory capacity-bound**: The training model size (model parameters, optimizer states, and activations) dictates the minimum number of hosts (HBM and DRAM) allocated to serve the memory size demand.

- **Compute-bound/bandwidth bound**: Given the compute intensity (e.g., in terms of

Petaflops/s-day) and/or memory bandwidth requirements of the model, a number of GPUs (hosts) are used to scale the training speed (e.g., Query Per Second) to the desired target.

Diverse configurations of DLRM workloads falling into the above two categories are developed and deployed in production. Models in each category utilize the underlying hardware differently, exhibiting unique challenges to improve performance and efficiency. One such class of configurations is those that are memory capacity bound. In this case, the degree of scale-out is guided primarily by the maximum number of model parameters each host can contain. Figure 4.1 shows the cumulative memory size and bandwidth requirement normalized to the total capacity for one of the most significant capacity-bound model use cases at Meta's data center. In Figure 4.1, each embedding table has a size calculated from the number of rows and embedding dimension. In addition, by multiplying the table's pooling factor (per-table average number of rows read per sample) by the embedding dimension size and target query per second (QPS), we can calculate the bandwidth requirement for each table. Interestingly, in the figure, the majority of the larger capacity embedding tables have a relatively low bandwidth requirement, and the tables that contribute to the most bandwidth have small sizes. By taking advantage of a hierarchy of heterogeneous memories such as high bandwidth memory (HBM), DRAM, and Storage Class Memory (SCM), it is possible to address both capacity and bandwidth requirements using fewer hosts. This increases training efficiency by limiting the need to scale out to multiple hosts only when the computation/bandwidth requirements warrant it.

Storage Class Memories (SCMs) are technologies with the properties of both memory and storage. SCM complements HBM and DRAM by providing memory at a unique capacity, bandwidth, power, and cost target. SCM can be utilized as byte-addressable (using DIMM, or Compute Express Link (CXL) [52] in the future), which provides $\sim 5\times$ increase in capacity at latency and bandwidth close to that of DRAM [30], or as block addressable (using NVMe) with $\sim 20\times$ higher capacity but at lower latency and bandwidth [129]. This flexibility allows the various memory technologies to cover a wide range of training system solutions by balancing bandwidth and capacity per host. However, such a design is not without challenges. The higher latency and lower bandwidth of the denser memory types must be accounted for when developing any scheme that distributes model parameters across different memory types.

Previous works have shown the benefit of SCMs to complement DRAM [42,44,51,130]. But little is known about the challenges and benefits of these technologies in commercial data centers for recommendation systems. Recent works [131,132] studied how we can use block-addressable storage for embedding tables but focused on inference, which has a different size and memory bandwidth demand than training, and as a result, imposes distinct challenges on the hardware. Previous studies also focus on using block-addressable storage with CPU. GPU-based accelerators are becoming commonplace for training recommendation systems. Hence, it is crucial to study how SCMs with lower bandwidth can be used with GPUs that have high memory bandwidth demand.

In this paper, we characterize diverse DLRM deployments at Meta and find that we have mainly capacity-bound and bandwidth-bound models. We then study these models' bandwidth, size, and locality and determine nonuniform size and bandwidth requirements across the embedding tables within a model. Our studies also show that embedding tables have low spatial locality, but high temporal locality with power-law [80] distribution. These characteristics make our DLRM models suitable for hierarchical memory with HBM, DRAM, and byte and block addressable SCM. We then design MTrainS, an end-to-end trainer that efficiently leverages a heterogeneous memory hierarchy along with GPUs. MTrainShas a key-value [29] storage system residing in SSDs for large (TB scale) embedding tables management. Then, to hide the low bandwidth and high latency of SSDs, MTrainSimplements a GPU-managed, software-based configurable hierarchical cache that uses DRAM and SCM for hot and cold embedding rows, respectively. We use two of our extensive resource-consuming models representing both capacity- and bandwidth-bound workloads for our experiments. Our results demonstrate that for capacity-bound models, by using MTrainS, we can reduce the number of hosts used for training by 4× for current models and by 8× for future scaled models while meeting our service level agreement (SLA) QPS target. To the best of our knowledge, this is the first work that studies SCM usage in GPU-enabled systems in a commercial data center for recommendation system training. In summary, we make the following contributions:

- Extend the memory hierarchy of DLRM training beyond HBM and DRAM to SCM in byte and block addressable forms.
- Characterize and experiment on large scale production-based real workloads, and discuss the real world scenarios where such hierarchical memory can win efficiency.
- Discuss the system-level performance trade-offs of byte and block addressable SCMs for the DLRM training usecase.
- Extend the open source DLRM with different memory support to facilitate hardware/-software research on larger AI models.

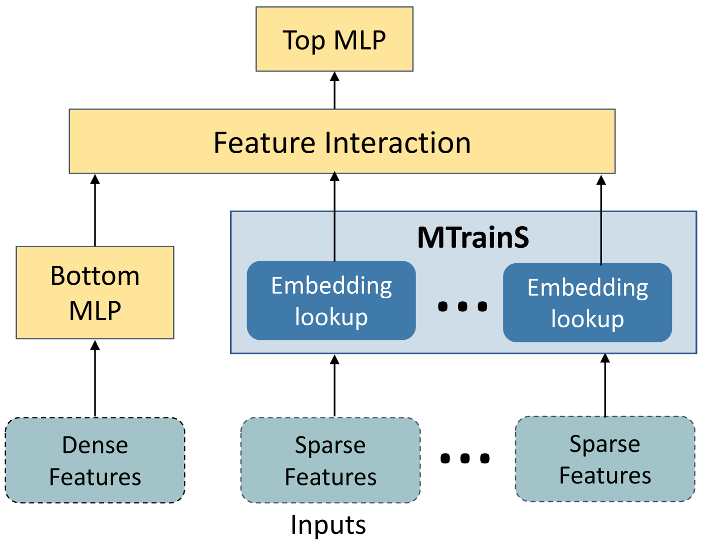


Figure 4.2: DLRM architecture and where MTrainS fits in DLRM.

4.2 Background

This section discusses the architecture of deep learning recommendation models (DLRM), FBGEMM_GPU (an optimized open-source GPU kernel library we used in our designs), and the various memory and storage components in our systems.

4.2.1 Deep learning recommendation model

Recommendation systems are widely deployed to rank content such as news feeds, videos, and products based on user preferences and interactions. To accurately rank user preferences, recommendation systems such as DLRM [122] use deep neural networks. However, DLRM is unique from DNNs because it has dense and sparse features to capture user and item attributes, leading to different characterizations as far as the underlying system is concerned.

4.2.1.1 DLRM architecture and components

Figure 4.2 shows the internal components of DLRM, the details are described below:
Input features: Users' data and products are represented by dense and sparse input features. **Sparse** features represent categorical inputs, such as a page a user likes in a list of pages. As the name suggests, this data is sparse, i.e., a user has likely interacted with a small subset of billions of pages available. **Dense** features include continuous inputs, such as user age.
Embedding tables: A naïve representation of categorical inputs would be to use binary vectors. For example, if we have 4 pages with IDs from 0 to 3 and if a user likes ID 0 and 2, the embedding vector for the user will be (1,0,1,0). But we have billions of pages,

Table 4.1: Characteristics of NAND SSD, Optane SSD, DRAM, Optane memory and HBM per module taken from product specifications. For SSDs, the numbers are for Gen3 PCIe.

Characteristics	**Nand Flash SSD**	**Optane SSD**	**DRAM**	**Optane memory**	**HBM**
Power (mW / GB) (mW / GB/ s for HBM)	5.7	35	375	98	5000
Cost per GB relative to Nand Flash SSD	1	10.4	68.8	26.5	-
Granularity of access	block	block	byte	byte	byte
Total capacity per host (GB)	8192	2048	384	2048	320
Total BW per host (GB/s)	6	6	200	84	12800
Endurance (DWPD)	0.8	100	-	-	-

hence such representation will be very large and sparse. Furthermore, binary vectors do not represent the relationship between similar pages. To avoid these problems, DLRM uses embedding tables that map categorical features into a dense representation. In this case, categorical inputs such as a page will be represented by a short vector, and similar pages will be located closer in Euclidian space. In the embedding tables, the column represents the embedding vector, and the rows are items in a category. Within a model, there are multiple categorical features, and the number of rows varies across tables. Embedding table operators look up a subset of rows in an embedding table, and pool the result using *sum*, *mean* or *max* [133].

Bottom MLP layer: The continuous inputs are transformed and projected into a dense space by a bottom multi-layer preceptron (MLP) that is composed of a series of fully connected (FC) layers and activation functions.

Feature interaction and top MLP: The dense projections of categorical and continuous features are aggregated (e.g. through concatenation), and then a set of MLP layers capture the interaction between different features.

4.2.1.2 Embedding tables and operators

Embedding tables impose unique challenges in systems designed for recommendation systems. Real-world use cases of embedding tables require a large memory capacity, up to 10s of TB. Equation 4.1 formulates the memory capacity requirements for a model with T embedding tables, H average number of rows per table (also known as embedding table hash size), D elements per row (embedding vector dimension), and element precision of p bytes.

$$Memory(SparseParameters) = T \times H \times D \times p \tag{4.1}$$

Given the massive size of the embedding tables, typical optimizers with a small number of states per row, such as Adagrad [134], is commonly used for sparse features. We can rewrite Equation 4.1 to include both model parameters and optimizer states (o).

$$MemoryCapacity = T \times H \times (D + o) \times p \tag{4.2}$$

Embedding tables are also memory BW intensive, as each training sample accesses multiple rows per embedding lookup. Assuming L rows are accessed per table for each training sample, Equation 4.3 formulates the BW requirement for embedding tables training to achieve a given QPS. Since both the forward and backward passes consume all the rows accessed, the equation is multiplied by 2.

$$MemoryBW = QPS \times T \times D \times p \times L \times 2 \tag{4.3}$$

4.2.2 FBGEMM_GPU kernel library

All software development contributions of this paper are on top of the FBGEMM_GPU (FBGEMM GPU kernel library) [135, 136], which is a high-performance GPU CUDA operator library for deep neural network training and inference. FBGEMM_GPU provides an efficient embedding table operator, data layout transformations, and other optimizations. It supports efficient embedding table access by providing HBM-based caching and DRAM utilization using unified virtual memory. We extended FBGEMM_GPU to add operators for SSD/SCM-based training and for various caching mechanisms to hide the high access latency of SSD.

4.2.3 Storage and memory types

We examined different storage and memory technologies to accommodate larger models per node. Each unit has its benefits and drawbacks. Table 4.1 shows the characteristics of each technology. The detailed descriptions are as follows:
Nand Flash SSD: It is the densest and cheapest memory technology we leverage, as seen in Table 4.1. However, these SSDs have low input/output Operations Per Second (IOPS) and significant latency compared with other memory types we evaluate. Moreover, they have limited write endurance, defined as Drive Writes Per Day (DWPD). NAND flash-based solution's limited IOPs make it best suited for a limited range of sparse features with high memory capacity and low bandwidth.
Optane SSD: It is based on Intel's 3D XPoint technology. It has higher IOPS, especially for lower access granularity requests and lower latency compared to NAND flash SSD. These SSDs have balanced read and write latency as well as 100$\times$ better write endurance (DWPD)

(see Table 4.1). However, these SSDs come at a 10× higher price per GB than NAND SSDs. We refer to this memory as *BLA-SCM* for BLock-Addressable SCM.

Optane Memory (PMEM): Intel Optane memory sits between DRAM and SSDs in the memory hierarchy. It is a cheaper alternative memory to DRAM with a 4-8× higher density but has lower bandwidth and higher latency than DRAM. We refer to this memory as *BYA-SCM* for BYte-Addressable SCM. *BYA-SCM* operates in Memory Mode and App Direct Mode. In **Memory Mode** DRAM serves as a direct map cache, while *BYA-SCM* is exposed as a single volatile memory region. The DRAM cache and *BYA-SCM* accesses are handled exclusively by the CPU's memory controller, and applications have no control of where their memory allocations are placed (DRAM cache or *BYA-SCM*). In **App Direct Mode**, DRAM and *BYA-SCM* are configured as two distinct memories. Here, the applications fully control read and write access to each memory. Note that the persistence characteristics of these memories are not relevant for our use case of training DLRM.

DRAM: DRAM has high bandwidth, and low read/write latencies. However, as seen in Table 4.1. it is more expensive per GB, has a lower density, and has higher power consumption per GB compared to *BYA-SCM* and SSDs. Additionally, the maximum DRAM capacity is limited by the number of DIMMs available on a host.

HBM: Many modern GPUs designed for HPC/AI Training utilize a memory technology even faster than DRAM. High Bandwidth Memory(HBM) has higher memory bandwidth than conventional DRAM. These memory modules are soldered onto the GPU, so the capacity is fixed. Therefore, the size available for embedding table storage is limited per GPU.

Table 4.2: Workload specifications.

	model 1	*model 1+*	*model 2*
Features	$\sim 10s$	$\sim 10s$	$\sim 100s$
Total bw (GB/s)	1300	2600	7136
Embedding dimension	128	256	128
Data type	4 byte	4 byte	4 byte
Average pooling factor	33	33	18
Num MLP layers	7	7	20

4.3 Workload characterization

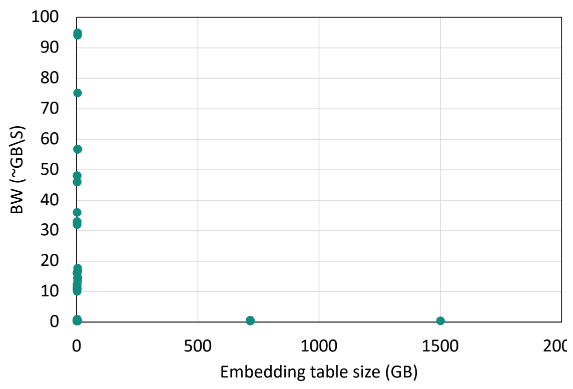
The large size and bandwidth requirements of DLRM workloads impose challenges in systems design. We studied the size, bandwidth, and locality of embedding tables in various deployed DLRM models in production to understand how we can improve their performance and power efficiency with heterogeneous memories. We select two of the most prominent representative models at Meta's data center with distinct characteristics and show the details here. The models are *model 1*, which is used for ranking content in various services, and *model 2*, which is used for click-through rate (CTR) prediction for user content and item recommendations. *model 1*+is a future scaling of *model 1* with similar BW and locality characteristics as *model 1*. These models have TB scale sizes. The characteristics are shown in Table 5.2.

4.3.1 Bandwidth and size distribution

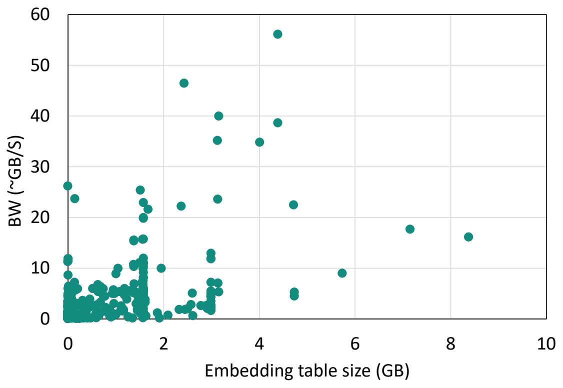
Figure 4.3a and 4.3b show a sample of the bandwidth and size distributions of embedding tables found in *model 1* and *model 2*. We use Equation 4.1 and 4.3 to study size and bandwidth. We use the tables' row numbers, dimensions, and precision from production configurations to calculate the size of embedding tables. We then use the acceptable QPS (SLA) for training each model in our data center and the pooling factor to calculate bandwidth. Pooling factor is determined from a large historical data of how many times each embedding table is accessed per lookup. As seen in the figures, the bandwidth vs. size distributions for the two models are distinctive. In *model 1*, we have smaller size embedding tables with high bandwidth requirements and large size embedding tables with lower bandwidth requirements. These types of tables fit intrinsically to a hybrid memory system with both large size/low bandwidth and small size/large bandwidth components. The cumulative bandwidth for our target QPS for *model 1* can be satisfied by a single HBM+DRAM system. Here, we scale out the model to multiple hosts to fit the model parameters with more memory capacity. On the contrary, in *model 2*, there are considerably more embedding tables that vary significantly in size and bandwidth. In this case, we scale models to multiple hosts to increase both the memory capacity and bandwidth. Due to this wide variance in size/bandwidth requirements, bandwidth and size distribution of models are key factors in DLRM memory hierarchy designs.

4.3.2 Locality in embedding table lookups

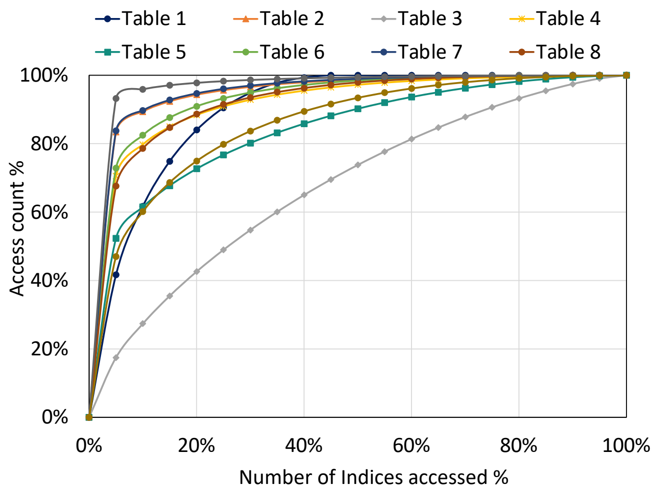
As discussed in [124], the spatial locality in embedding tables is very low because embedding tables represent sparse categorical features, and the embedding row access is very irregular. However, we have a considerable temporal locality that makes caching effective in our trainer design. To investigate temporal locality, we examine the frequency of access of embedding table Indices for several embedding tables of *model 1* and *model 2* running in production for 24 hours. We show the results for the representative embedding tables in



(a) Model 1



(b) Model 2



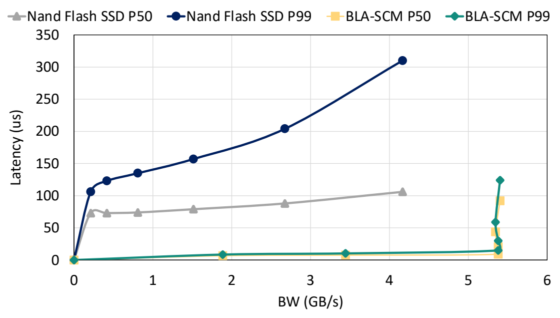
(c) Temporal locality analysis of embedding tables.

Figure 4.3: BW vs Size distribution of Model 1 and Model 2 and locality analysis of various embedding tables.

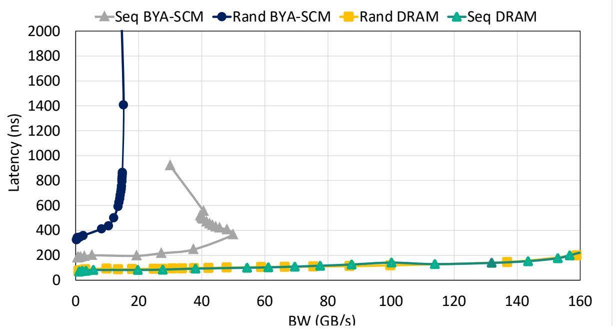
Figure 4.3c. The figure shows that access to most tables follows a power-law distribution. As also shown in [124], we observe 80% of the indices accessed come from 10%-40% of the total Indices for most tables. Hence, we can take advantage of heterogenous memories and storage by placing colder embedding tables and embedding rows in large but slower memories like *BYA-SCM*, *BLA-SCM*, or Nand Flash SSD, while hot embedding tables and embedding rows can still enjoy faster but smaller size memories like HBM and DRAM through caching because of the high temporal locality in DLRM workloads. In our design in Section 4.5, we emphasize how we can maximize caching to hide latency and provide higher BW by using hierarchical caching.

4.4 System design challenges and considerations

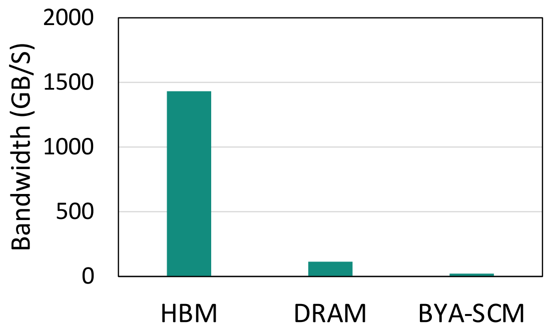
This section discusses the challenges of adopting heterogeneous memories and storage for DLRM training and our considerations in the workloads and hardware characteristics for our designs.



(a) P50, P99 latency and BW of *BLA-SCM*and NAND SSD.



(b) Latency and BW of DRAM and *BYA-SCM*.



(c) BW comparison of HBM, DRAM and *BYA-SCM*.

Figure 4.4: System memory and storage characterization.

4.4.1 Memory Performance evaluation

We compare the latency and BW of the memories and storage to understand how they fit with our workload characteristics. We measure the *BLA-SCM* and Nand Flash SSD latency and BW with FIO [137] for random read workload with different queue depths to increase BW utilization. In Figure 4.4a, *BLA-SCM* has a latency in ∼10µs range for both P50 and P99 at similar BW, whereas Nand Flash SSD has a latency of 100µs, and P99 latency is significantly higher than P50. Also, note that increasing BW utilization in Nand Flash SSD increases the access latency. This shows that for Nand Flash SSDwe have to be careful with the BW utilization to prevent significant latency. Given the high temporal locality in our models, it is advantageous to implement caching to reduce SSD traffic, especially with the performance limitations of Nand Flash SSD.

We use Intel's Memory Latency Checker (MLC) [78] to measure DRAM and *BYA-SCM* latency and BW. We use sequential and random read workloads with different memory traffic rates. In Figure 4.4b, *BYA-SCM* achieves $\sim$15 GB/s and DRAM 170 GB/s BW. Further, *BYA-SCM*'s latency increases with increased memory traffic ($\sim$200ns - 800ns for a sequential and $\sim$350ns - 1500ns for a random read), and the BW saturates at high traffic. Then the latency increases with no BW change. However, DRAM has a much lower latency and higher BW than *BYA-SCM*, and it maintains the same latency for sequential and random access. While spatial locality in our workloads is low in the 4KB block access granularity of SSDs, for *BYA-SCM*, the sequential access granularity is 256 bytes. The access granularity of an embedding lookup in our workloads is 512-1024B. Hence, we can still achieve *BYA-SCM* sequential access performance. However, because of the BW and latency differences between DRAM and *BYA-SCM*, the most practical design is a hierarchical configuration, where DRAM is used for hot embedding rows and *BYA-SCM* for colder ones. We should also consider the traffic to *BYA-SCM* to avoid large latency and BW saturation.

We use gpumembench [138] for HBM and MLC for DRAM and *BYA-SCM*to compare BW differences. In Figure 4.4c, we see that HBM has significantly higher BW than DRAM and *BYA-SCM*. These BW and size differences (as seen in Table 4.1) show that we need to optimize the placement of embedding tables and rows to these memories to maximize BW utilization.

4.4.2 IOPS vs BW

Although we have temporal locality in the embedding tables, adjacent rows are accessed in a non-sequential manner and lack spatial locality. Additionally, the embedding dimensions typically range from 64 to 256 (i.e., 256-1024 bytes with single precision). When considering block addressable technologies such as *BLA-SCM* and Nand Flash SSD, each access to an embedding row could consume less than the block size (e.g., 4KB), resulting in a waste of BW (referred to as read amplification). To account for the impact of read amplification, we track IOPS instead of BW for the block addressable units and study how we should use these memories to satisfy IOPS demand. Equation 4.4 formulates the required IOPS, assuming T_B tables are placed on SSD, with an average pooling factor of L_B. α is used to factor in the locality of accessing embedding tables, which reduces access to the lower level block addressable memory.

$$IOPS = QPS \times T_B \times L_B \times \alpha \tag{4.4}$$

The IOPS requirement for *model 1* and *model 2* to accommodate the entire model in one node while placing T_B tables on SSD for our target QPS is 6.25M and 75M, respectively, without considering the locality. Typically, SSDs have IOPS in a range of 500K-1M IOPS limit. For example, if we have a cache hit rate of 70% for *model 1*, the IOPS will be 1.875M. Whereas for *model 2* with 70% locality, we still require 22.5M IOPS. Hence, the locality is not only important for optimizing the high latency of SCM, but it will help us to operate within the IOPS limit of the hardware. Therefore, we focus on maximizing the locality of the models in our designs.

4.4.3 Endurance

Nand Flash SSD and *BLA-SCM* [1] have a limited number of program/erase cycles that can be performed before a memory cell wears out. This is measured as how many times the entire drive can be written to each day of its lifetime (typically 3-5 years) which is called Drive Writes Per Day (DWPD). Equation 4.5 formulates the amount of data written per day during training for a given QPS, where T_B tables are placed on SSD, with L_B average pooling factor, D elements per row, α locality, and element precision of p bytes.

$$write/day = 24 \times 3600 \times QPS \times T_B \times L_B \times D \times p \times \alpha \tag{4.5}$$

For example we write ∼10TB per day to SSDs for *model 1* and ∼100TB for *model 2* while placing T_B tables in SSDs. DLRM workloads are write-intensive. When using SSDs for embedding table storage, we want to limit our writes per day below the stated DWPD limit to avoid premature drive failure.

4.4.4 Workload scaling

Deep learning models, including DLRM, are scaling rapidly in complexity and size. A number of sparse features are used in the model, and hence the number of embedding tables, along with the embedding dimensions per table, are among the main contributors to the increase in the memory capacity of DLRM workloads. Any system solution needs to be able to consider such scaling during the lifetime of the system. In this paper, we design systems for current models and test whether our design holds in future scaling.

[1] Optane in DIMM form factor, referred to as *BYA-SCM* in this paper, is claimed to not be bounded by endurance

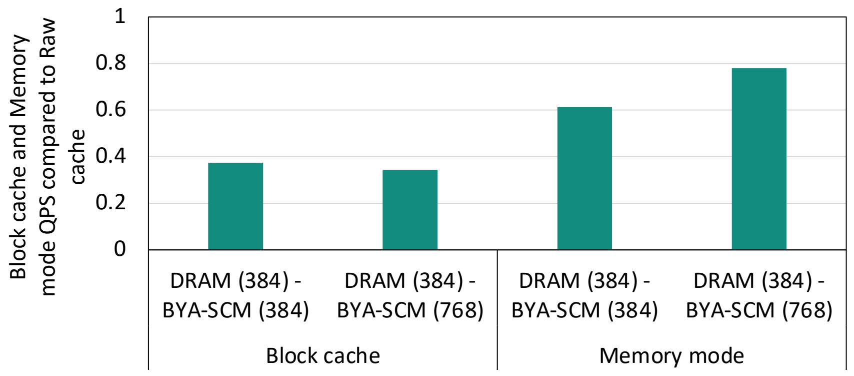


Figure 4.5: Caching efficiently comparison for *model 1*.

4.4.5 Software design choices

The first consideration of our software design is the efficient utilization of block-based storage. Block devices must always write a minimum of one block, even if only a single byte has changed. In order to efficiently utilize the drive, we use RocksDB [29]. RocksDB is a key-value storage engine that provides low latency database operations. RocksDB's structure, such as efficient database sharding, allows for fast storage access. In addition, RocksDB uses an in-memory data structure for faster write operations. Since new writes go to memory (DRAM) first, RocksDB can compact many writes into a single large contiguous drive write. This significantly reduces SSD writes and increases SSD lifetime.

Second, due to the high temporal locality in embedding tables, caching effectively hides the latency of SSDs. However, to adequately utilize DRAM and *BYA-SCM*, we must consider how we organize the cache. We first examine re-using the existing RocksDB block cache, which uses memory to cache data for fast read access, and store it in DRAM/pmem as shown in [3]. Another alternative is using the hardware-managed DRAM cache that comes with Intel Optane memory (memory-mode). In this mode, the hardware transparently uses DRAM as a direct-map cache of *BYA-SCM*. We compared these two methods to a raw cache with access granularity equal to the embedding table row dimension. The raw cache hierarchically uses DRAM and *BYA-SCM*in app-direct mode, where DRAM is the first level of cache, and *BYA-SCM*is the second-level cache. Figure 4.5 compares the QPS of the raw cache versus both block cache and memory mode in two configurations for *model 1*. Since the block cache is designed for best performance in read-only cases, we did not find it performant in both hardware configurations (0.35-0.38× of the performance of the raw cache) for training DLRM models where the read/write mix is 50/50. This is because once an embedding row is updated, its location changes on disk due to write compaction and thus isn't accessible from the original block cache line. In addition, the block cache results in double-caching values, wasting capacity. Similarly, from Figure 4.5, we can also see that memory mode is not helpful because of double caching. Hence, it is essential to design caching that exposes the unified capacity of DRAM and *BYA-SCM*. For these reasons, we design an exclusive hierarchical cache using the app-direct mode to fully control access granularity and embedding table and row placement.

4.5 MTrainS design

4.5.1 Overview

Adding heterogenous memories in DLRM system design requires considering the memory technologies' latency, size, and bandwidth differences and a closer look at our models, as discussed above. We design MTrainS, an end-to-end training pipeline to leverage HBM, DRAM, *BYA-SCM*, and *BLA-SCM* or Nand Flash SSD in DLRM. MTrainSextends the embedding table storage to *BLA-SCM* and/or Nand Flash SSDusing **RocksDB-based embedding table storage**, giving us the flexibility to accommodate large TB scale models with varying sizes and BW embedding. Based on the temporal locality of our models (see Section 4.3.2), MTrainSimplements **a hierarchical cache module** using the least recently used (LRU) policy for embedding tables stored in SSDs to hide the large SSD latencies. It uses DRAM and *BYA-SCM*for caching and places hot embedding rows in DRAM for fast accesses and colder ones in *BYA-SCM*, providing slower but ample cache space. Given the wide ranges of size and bandwidth requirements (see Section 4.3.1) in embedding tables, to maximize the size and BW availability in the memories and storage, MTrainSuses **embedding table placement** based on a mixed-integer programming solver, with table size and data volume per access (pooling factor) as inputs, and memory size and bandwidth as constraints, with the goal of increasing the bandwidth of embedding tables and minimizing tables' access time. Figure 4.2 shows the overview of where MTrainS fits in DLRM, and Figure 4.6 shows the architecture of MTrainS.

4.5.2 Embedding table storage

MTrainSuses RocksDB-based storage to place embedding tables in block-addressable storage (*BLA-SCM* and Nand Flash SSD). Storing tables in SSDs allows for extensive storage per node for the big-size embedding tables. RocksDB [29] is a key-value store optimized for high-speed storage. In the context of embedding tables, the key is the embedding table row index, and the value stored is the embedding row containing the weights. RocksDB organizes the key-value database into blocks (4KB in our implementation) in a

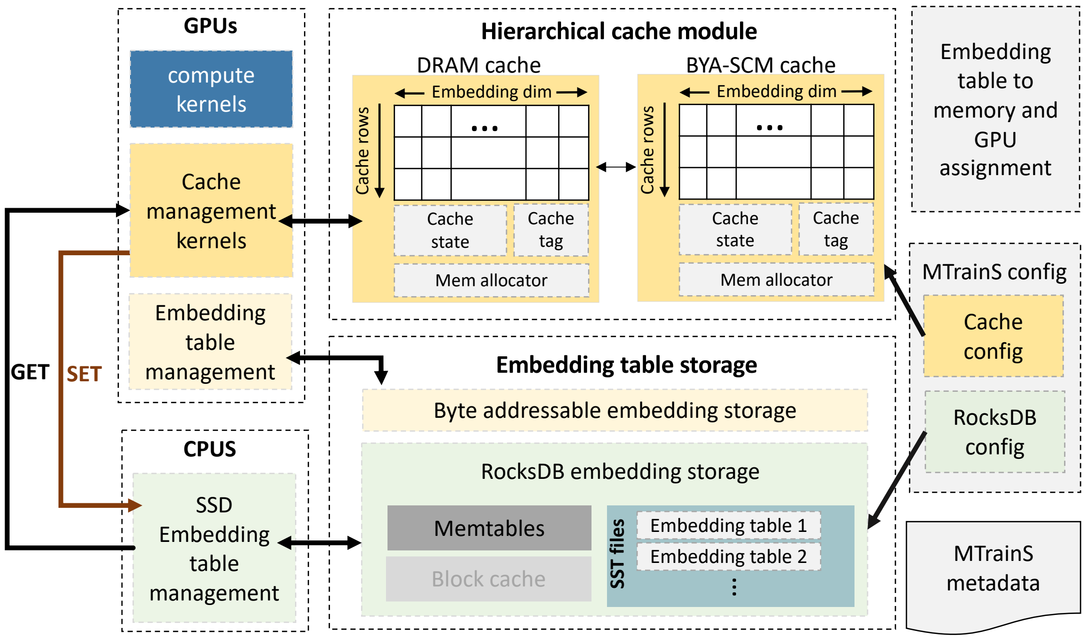


Figure 4.6: MTrainS component and architecture.

Sorted String Table (SST) format. We sharded these databases (SST files) of embedding tables to load balance and for fast key lookup. Our RocksDB implementation uses memtable, located in DRAM, to optimize writes. However, as discussed above, we turn off the block cache because it does not benefit our DLRM workloads. For faster read, we use the MultiGet() [139] API in RocksDB, optimized for batched lookups.

MTrainSalso allows embedding table placement in the byte-addressable memories (HBM, DRAM, *BYA-SCM*) using a two dimensional tensor. These tensors have the same structure in all memory types with different memory allocator parameters.

4.5.3 Hierarchical cache module

We leverage DRAM and *BYA-SCM*as software-managed caches on top of the RocksDB embedding storage to hide the SSD's low bandwidth and high latency. The hierarchical cache module has a list of caches derived from a cache class and a configurable cache hierarchy that organizes the list of caches to multiple levels. Figure 4.6 shows the cache overview.

4.5.3.1 Cache class

This class stores the hot rows of the embedding tables placed in SSDs. It has a **cache memory** that keeps raw embedding rows (as opposed to a block of multiple rows) as shown in Figure 4.6 because of the lack of spatial locality in the embedding tables. To allocate cache memory, the class has a **memory allocator** parameter that can be set to different memory types. The class also has tags and states. The **cache tags** track which Indices of the embedding rows reside in the cache and which cache entries are occupied/free. The **cache states** track each cache entry based on timestamp. DRAM and *BYA-SCM*caches are an instance of the cache class with different cache memory allocators.

4.5.3.2 Cache hierarchy

We use DRAM and *BYA-SCM*as a configurable multi-level cache. The cache is configured as a one-level in the presence of only DRAM in the system. When we have DRAM and *BYA-SCM*in the system, we organize the two memories as a two-level exclusive cache, with DRAM cache as the first level and *BYA-SCM*cache as the second level. We use exclusive cache settings for more efficient use of the memory space. Note that we can use *BYA-SCM*only as a one-level cache, but we didn't find this configuration performant because of its high latency. We identify cache location-specific operations, such as data movement between caches and which cache to access first, based on the cache hierarchy structure. Note that, we only have DRAM and *BYA-SCM*cache here, but the cache structure can handle more than these two caches, such as multiple DRAM and SCM caches in a complex hierarchy.

4.5.4 Embedding table management

This module accepts and responds to embedding table requests stored in all memory types. Embedding table lookups and updates are managed by the GPU for tables placed in the byte-addressable memories and by the CPU for tables placed in SSDs. While embedding lookups are initiated by the GPUs, they will be handed off to the CPU to get data from RocksDB embedding storage when Indices miss from the cache modules. Note that we can directly access the SSDs from the GPUs using GPU Direct Storage (GDS) [140], but we do not use it in our design to leverage the host side memory (DRAM and *BYA-SCM*) as SSD caches. Using GDS limits the cache to HBM. This module exposes two APIs, GET and SET, from the SSD embedding storage to the rest of the trainer, as shown in Figure 4.6. GET and SET APIs must synchronize the CPU and GPU to maintain data consistency. For efficiency, we designed a multithreaded management unit for RocksDB embedding storage in the CPU that looks up the RocksDB shards in parallel. The module is also responsible for initializing the embedding table weights before training starts. We provide two options for initialization:

4.5.4.1 Pre-initialization

Initialize all the weights of the embedding tables stored in all memory types with random values following the desired distribution before training starts.

4.5.4.2 Deferred initialization on read

Embedding tables stored in SSDs have very large sizes. Pre-initializing all these weights takes a long time. We designed a deferred initialization technique to prevent this long initialization process at the start of training and to preserve SSD write endurance. In this technique, we initialize embedding values on-demand upon the first read if a key is not found inside the database Indices. During deferred initialization, to reduce initialization latency, we have a separate background thread that generates a queue of randomly initialized values following the desired distribution. This separate thread optimization is especially helpful in reducing latency when a single request tries to read many uninitialized rows. When we attempt to read an embedding row that has never been accessed, we consume values from the queue to randomly initialize the desired embedding row. This technique reduces writes by $\sim$15% for *model 1*.

4.5.5 Cache management

In our design, the cache is managed by the GPU, similar to the caching proposed in [141]. We extend their work and design GPU kernels in FBGEMM_GPU that support caches in multiple memories, such as DRAM and *BYA-SCM*, in a hierarchy. GPU-managed cache gives us the advantage of using the higher GPU compute and BW capability to accelerate cache management operations. We discussed the cache management kernels below.

4.5.5.1 Tag/state lookup

This GPU kernel looks up the cache tags and states to check if the embedding rows are in the caches or the SSD storage for incoming embedding row requests. In a two-level cache, the kernel looks up the tags and states for both caches in parallel for efficiency. After determining hits and misses in all the caches, the kernel groups the Indices of incoming lookup requests based on the memory destination, i.e., DRAM, *BYA-SCM*, or SSD.

4.5.5.2 Cache algorithm

The caching algorithm kernel uses the LRU policy. It looks up the grouped indices of each memory from the tag and state lookup kernel. Then based on this, 1) it updates the time of access (LRU status) of the cache hit indices in each cache state (DRAM and *BYA-SCM*), 2) it determines the cache slots to insert the missed indices for the indices that are a miss, 3) it resolves the cache slots to evict if there are no free cache slots available. We explored the least recently (LRU) and least frequently (LFU) used caching algorithm. Our experiments on show that LRU provides $\sim$8-10% better performance than LFU. This is because, while both LRU and LFU capture the temporal locality of embedding tables, with LRU after inserting embedding rows to the cache during the forward pass, the rows are still going to be the recently used row in the backward pass. This increases the chances of the rows being in the cache in the backward pass even when it is not the frequently accessed row, increasing cache hit rates in the backward pass compared to LFU.

4.5.5.3 Data lookup/update

It takes the grouped cache Indices from the tag lookup kernel and returns the rows. It also accepts new data or updates to Indices in the caches and returns evicted rows.

4.5.5.4 Data movement between DRAM, *BYA-SCM*, and SSD

In the one-level cache, if incoming embedding lookup requests are hits, the requested embeddings rows will be fetched by the requesting GPU. In case of misses, the control is passed to the CPU to access SSDs. The CPU then fetches the requested rows from the SSD storage. Newly fetched rows will be inserted into the first-level cache from SSD for fast access. If the cache is full, LRU rows will be evicted back to SSD to make room for the new rows.

For a two-level cache, we access both DRAM and *BYA-SCM*caches in parallel during lookup. Hits from both caches are returned to GPU. DRAM cache misses that are hits in *BYA-SCM*cache and *BYA-SCM*cache misses fetched from SSD are promoted to DRAM cache for fast access. When DRAM capacity is full, DRAM LRU rows are evicted to *BYA-SCM*cache. Similarly, when *BYA-SCM*cache is full, LRU rows are evicted back to SSD storage.

4.5.6 Memory and GPU assignment

4.5.6.1 Embedding table assignment

A single DLRM model has various embedding tables having different sizes and BW. We have multiple memory types in our systems with varying sizes and bandwidths. While we use DRAM and *BYA-SCM* for caching lower BW embedding tables stored in SSD, we can use HBM and some part of DRAM to place the high and medium BW embedding tables and still satisfy the latency and BW requirement of embedding tables stored in SSDs through

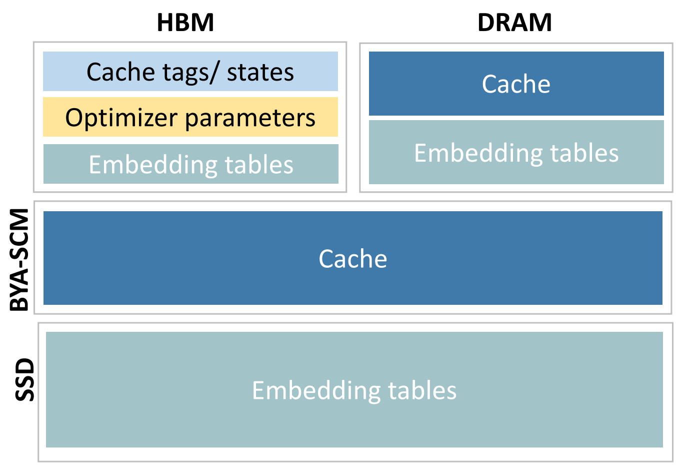


Figure 4.7: Memory allocation of MTrainS.

caching. Figure 4.7 shows memory allocation of MTrainS components. HBMs are used to store high bandwidth embedding tables, optimizer parameters, and the tags and states of the caches. DRAM is used to store medium BW embedding tables for caching hot embedding rows, and *BYA-SCM*, for storing colder embedding rows of tables stored in SSDs. Our experiments show that using *BYA-SCM* only for caching instead of an explicit assignment is better. The search space for embedding tables assignment is vast because we can assign any tables in any of the memories. We use a simple linear solver to optimize the assignment. Table assignment is a complex problem, and it is possible that with a more sophisticated heuristic, we can achieve better table assignments than our current solution by considering other factors, such as locality. We leave such complex designs to future work.

Input variables: The input variables for assignment are the sizes and BW of each embedding table in a model and the size and BW of the memory types.

Constraints: The constraints for table assignment are:

1. Each table can only be assigned to one memory type, but each memory type can hold multiple tables.

2. The cumulative size of tables assigned to each memory type can not be larger than the memory size.

Objective function: Minimize the total embedding lookup time, approximated according to Equation 4.6.

$$\begin{aligned} lookup\_time &= Max(time(g)), g \ \varepsilon \ GPUCount \\ time(g) &= \sum_{M} \sum_{T_{gm}} (D \times L \times p) / BW_{gm} \end{aligned} \tag{4.6}$$

M in Equation 4.6 stands for memory type (e.g. HBM or DRAM). T_{gm} represents embedding tables assigned to a specific memory type for a given GPU, and BW_{gm} represents BW for memory type m for shard g. For example, for HBM, BW_{gm} represents HBM BW. For the shared DRAM, it would represent $DRAM\_BW/num_{gpus}$. We show the effect of placements in section 4.7.6.

4.5.6.2 GPU assignment

Based on the table assignment, for N GPUs, embedding tables assigned to the HBM of the GPU_i will be handled by GPU_i. In addition, the embedding tables assigned to DRAM and SSD will be distributed to be managed by the N GPUs by the table placement algorithm by minimizing lookup time in Equation 4.6.

4.5.7 Pipelining

To hide some of the latency of accessing SSD for cache misses, we can pipeline access to the caches several batches in advance. Instead of sequentially 1) Fetch, 2) Preprocess, 3) Load on GPU, 4) Train, we split each step into its own stage and execute them simultaneously for different batches. In our case, we added a step: 4a) Prefetch Sparse Indices into cache before training. As long as we can maintain an invariant that embedding rows prefetched into the cache are not evicted until that batch has been trained, we can have an arbitrary number of batches in the pipeline. By adding additional stages between 4a) Prefetch and 4) Train, we can increase the latency hiding capability of the pipeline until it exceeds the typical SSD latency for a GET call. If the demanded bandwidth required to meet the QPS goals exceeds the capabilities of the SSD, no amount of extra stages will help.

4.5.8 MTrainS configurations and metadata

4.5.8.1 MTrainS metadata

The metadata keeps the memory and GPU assignment of all embedding tables and is used in every embedding table lookup to direct requests to the responsible memory.

4.5.8.2 Cache config

This includes cache configurations to expose DRAM and *BYA-SCM*memories hierarchically or to be used alone as the first layer of cache, cache row, and column sizes to fit the embedding dimensions of the target DLRM model.

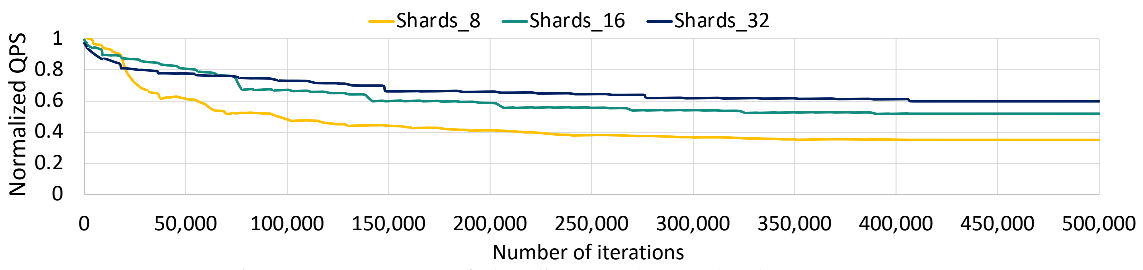


Figure 4.8: Impact of sharding RocksDB database on QPS.

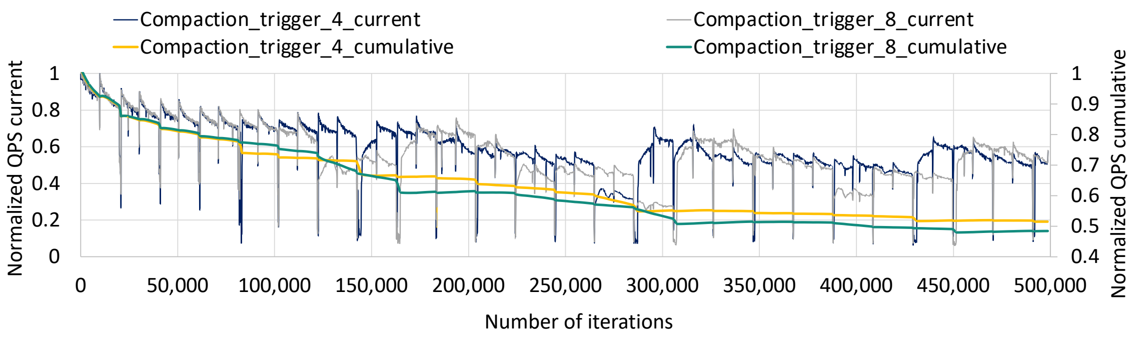


Figure 4.9: Impact of Database compaction on QPS.

4.5.8.3 RocksDB configs

The RocksDB embedding storage exposes knobs to tune performance. The knobs include the number of CPU threads used for embedding table lookup, DB shards, compaction time, memtable sizes, and turning the block cache on/off. Sharding is one of the most important knobs, which increases key lookup efficiency and decreases compaction time. As seen in Figure 4.8, sharding DB increases QPS by up to 40%. Another knob is database compaction, which is necessary for RocksDB to maintain a manageable database size during training. Synchronized database compaction from all RocksDB shards and trainers causes a major thundering herd problem that results in large memory and IO spikes. We observe considerable drops in QPS (over 50% in some cases) during database compaction (seen in Figure 4.9). Tuning compaction knobs, such as compaction trigger time as shown in Figure 4.9, improves the cumulative QPS by 5-8%. In our experiments, we show the results for the best RocksDB configuration we found for each model.

4.5.9 End-to-end trainer

Figure 4.10 shows the end-to-end trainer. We first run the embedding table assignment and distribute embedding tables to HBM, DRAM, and SSD according to the optimal placement. Note that we don’t need to run the placement for every training unless the model changes significantly. For the training data, the dense features are distributed across the batch

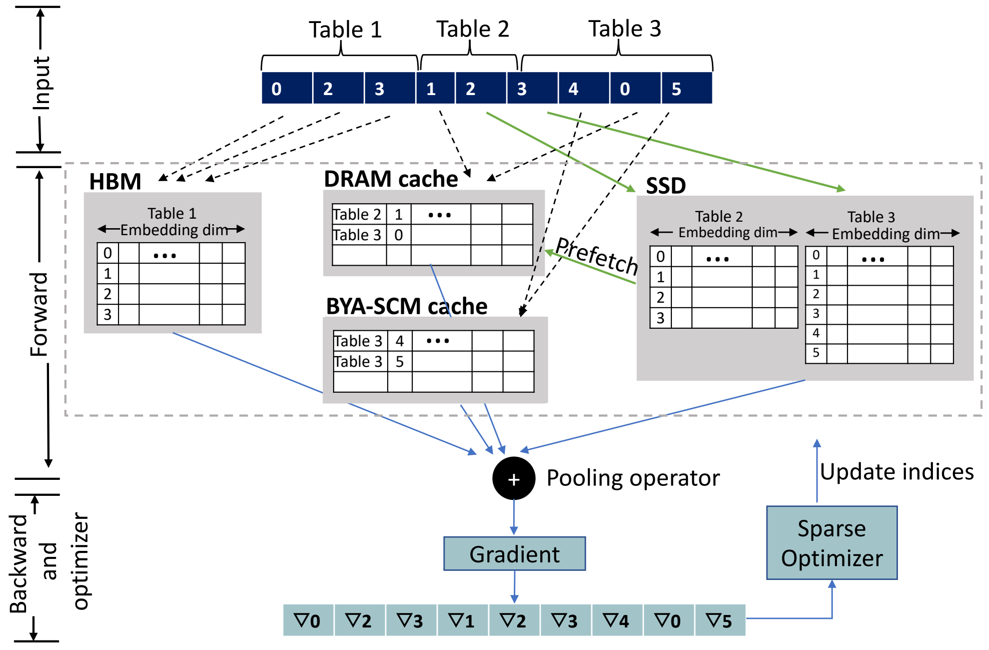


Figure 4.10: {MTrainS trainer overview.

dimension among multiple GPUs. The model embedding tables are distributed among the GPUs table-wise, and every GPU will handle the lookup for the embedding table assigned to it. For our current workloads, table-wise partitioning provides sufficient model parallelism across GPUs. Then, during embedding lookups, MTrainS will distribute the incoming input indices to the GPUs and memories. In Figure 4.10, for example, Table 1 is placed in HBM and Table 2 and 3 in SSD. The GPU initiates embedding table lookup for the indices using MTrainSin the forward pass. MTrainS for example, gets index 1 of Table 2 from DRAM in the figure. Once the GPU gets the embedding rows from the memories, it performs aggregations and optimizers, then updates the weights in the respective memories in the backward pass.

4.6 Systems setup and implementations

4.6.1 Software design and implementation

We use the PyTorch version of DLRM and implement a new EmbeddingBag that uses heterogeneous memories and storage for embedding table operations instead of using the default EmbeddingBag [142] implemented with PyTorch in DLRM. Our new EmbeddingBag is integrated with the FBGEMM_GPU kernels we developed to manage cache and memory. Figure 4.11 shows the software overview.

In our experiments, we set up Intel DCPMMs in App Direct mode using IPMCTL [81].

We use Linux Kernel 5.4, which enables a volatile use of DCPMM. We then use NDCTL 6.7 [82] utilities to configure SCM in the devdax mode. This mode gives direct access to DCPMM, which is faster than filesystem-based access. We use DAXCTL [82] to set up DCPMM in the system-ram mode so that DCPMM will be available in its own volatile memory NUMA node. To implement SCM in DLRM, we use the Memkind library [83]. Memkind partitions the heap into multiple kinds of memories, such as DRAM and SCM, in the application space. We use MEM_KIND_DAX_KMEM for SCM accesses.

We use CUDA 11 for our GPU kernel implementations. As shown in the Figure, we utilize cudaMallocManaged [143] that uses a unified memory system to access HBM and DRAM from the GPU transparently. For *BYA-SCM*, since the GPU can't access these types of memories directly with unified memory, we use cudaHostRegister [143], which registers an existing host memory range already allocated by the Memkind library to CUDA. We have multiple cache management and computation kernels. Using PyTorch's torch.cuda.Stream [144], we launch kernels that can run in parallel, such as looking up tags or getting data from DRAM and *BYA-SCM*caches, in different streams. We then use torch.cuda.synchronize [145] to synchronize the kernels getting data from the cache before computation kernels start. While performing calculations for a batch, in parallel, we update tags/states and manage insertion, update, and eviction in all caches and SSD storage. Then these kernels are synchronized with the subsequent batch data lookups to have updated cache/data for the next batch.

4.6.2 Workloads description and setup

In our experiments, we use two of the most significant DLRM models derived from real use cases (*model 1* and *model 2*). These models show distinct features (capacity- and memory-bound) present in most of our models. We also use a model with 2x the size of *model 1* (*model 1+*), representing how the model will grow in the next two years. We show the characteristics of the models in Table 5.2.

4.6.3 Evaluation hardware description

In our evaluations, we used an Intel Barlow Pass-based system, with 2 CPU sockets populated with Intel Ice Lake processors and 8 Nvidia A100 GPUs. This platform is designed for AI/ML, Deep Learning, and HPC applications, and has 8 NVidia A100 GPUs behind a fully connected NVLink / NVSwitch [146] fabric. The hardware specification is shown in Table 5.1.

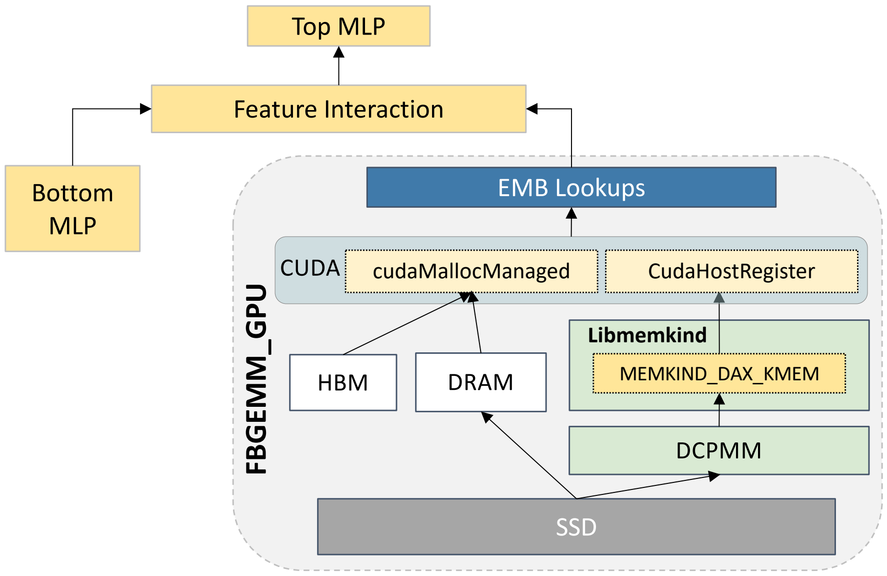


Figure 4.11: MTrainS software design.

4.6.4 Server design

We consider diverse server designs by varying the memory and storage types and sizes. Table 4.4 summarizes the configurations we used in our experiments. In all of our system configurations, we limited the DRAM size to 384GB. We use two sizes of *BYA-SCM*, 384GB and 768GB. We chose these sizes because we want to determine the ratio of DRAM and *BYA-SCM* required in the system. We also experiment with other configurations, such as increasing the *BYA-SCM* size. However, increasing *BYA-SCM* beyond 768GB does not show additional benefit for our existing workloads because we will be compute-bound at this config. Similarly, with *BLA-SCM* with 768GB configuration.

We use half of the DRAM in our system (192 GB) for caching and the rest to store smaller size embedding tables and for other system requirements of DLRM. We use all of BYA-SCM for caching, i.e., 360GB in 384GB configurations and 720GB in 768GB configurations. The remaining BYA-SCM is used for optane metadata.

4.7 Evaluation

4.7.1 Baseline

We compare MTrainS to the baseline system configuration shown in Table 4.4. Our implementation in the baseline also uses an integration of DLRM and FBGEMM_GPU. We use HBM and DRAM in the baseline for embedding storage and caching. This implementation

Table 4.3: System setup: hardware specs.

Specification	**System config**
OS	CentOS-8
Linux kernel version	5.4.135
CPU model	Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz
Sockets	28
Core per socket	28
Threads total	112
L1I/L1D cache	32 KB/48 KB
L2/L3 cache	1.25 MB/42 MB
GPU model/GPUs	A100-SXM4-40GB/8
GPU Driver	450.142.00
CUDA Version	450.142.00
HBM size	HBM2e 320 GB (40 X 8)
DRAM size	DDR4 384 GB (32 GB X 12 DIMM slots)
SCM size	DDR-T 2 TB (128 GB X 16 DIMM slots)

Table 4.4: System Configurations (all sizes in GB).

Config.	HBM	DRAM	*BYA-SCM*	*BLA-SCM*	Nand SSD
Baseline	320	384			
configNand	320	384			8192
configBLA	320	384		2048	
configBYA-1	320	384	384		8192
configBYA-2	320	384	768		8192
configSCM	320	384	384	2048	

uses the same techniques as CDLRM [147]. While CDLRM uses the CPU to manage the cache, for a fair comparison with MTrainS, which uses GPU for fast cache management, we move the cache management in CDLRM to GPU. We call this CDLRM+. We also added the efficient embedding table placement in CDLRM+. Our experiments compare the number of nodes required to train a model running CDLRM+ with the performance we get with a single node running MTrainS. To compare performance, we use target QPS, an acceptable QPS in our data center, to train a specific model based on how often we need to train models and the rate at which new training data becomes available. QPS in our experiment represents the number of input data we can use to train a model per second, including the forward and backward pass. It is inversely proportional to the training time.

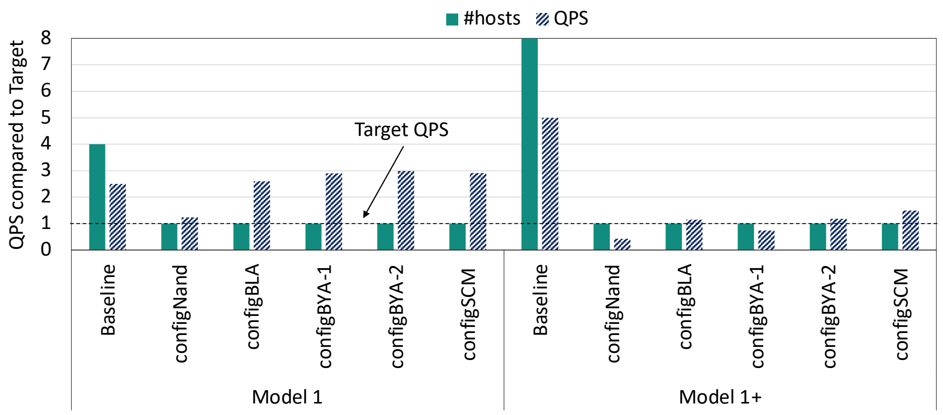


Figure 4.12: Training QPS and number of host Comparison of CDLRM+ and MTrainS with SLA QPS target as the baseline for *model 1* and *model 1+*

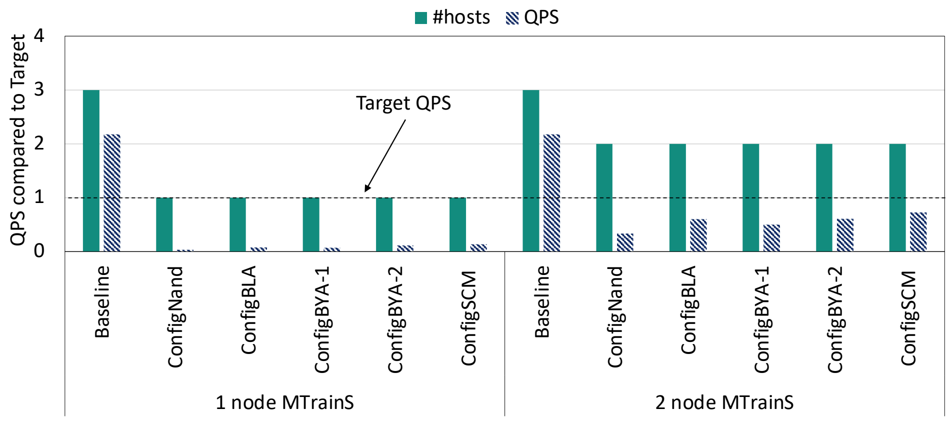
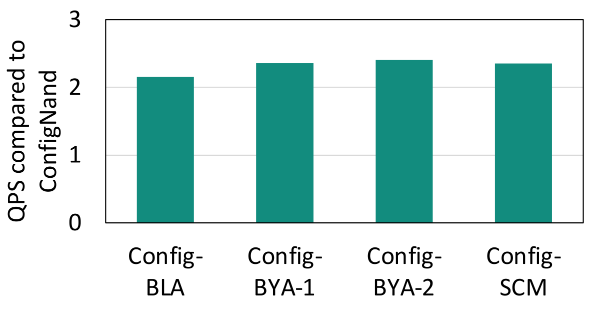


Figure 4.13: Training QPS and number of host Comparison of CDLRM+ and MTrainS with SLA QPS target as the baseline for *model 2*.

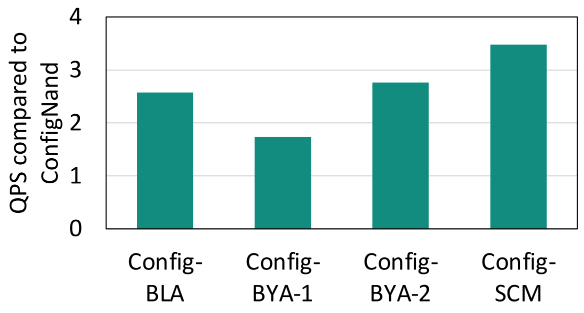
4.7.2 Training efficiency

One of the principal arguments for adopting denser memory technologies is to use fewer nodes for specific model training and improve power and cost-efficiency. Hence, we first compare the overall deployment efficiency of MTrainS and CDLRM+ for *model 1* and *model 2* based on the target QPS.

Figure 4.12 shows the efficiency comparison of *model 1* and *model 1+*, which are memory capacity bound. Note that *model 1+* is the future scaling of *model 1* with a larger size ($2\times$). These models require 4-8 hosts to load and train with the baseline CDLRM+. With MTrainS, one host can provide sufficient extended memory capacity to load and train the model. We can also reach the target QPS (SLA) in such a setting, as seen in Figure 4.12. While the baseline without MTrainS achieves higher QPS, it requires 4-8$\times$ more hosts, so this performance is stranded and does not contribute additional efficiency to the deployment

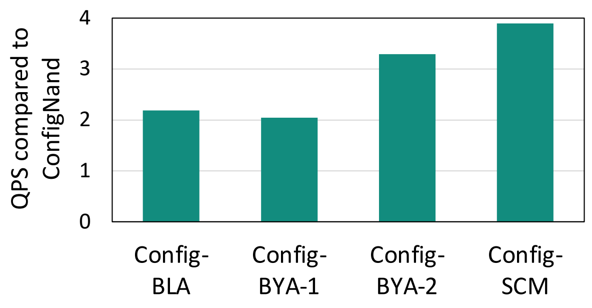


(a) *model 1*

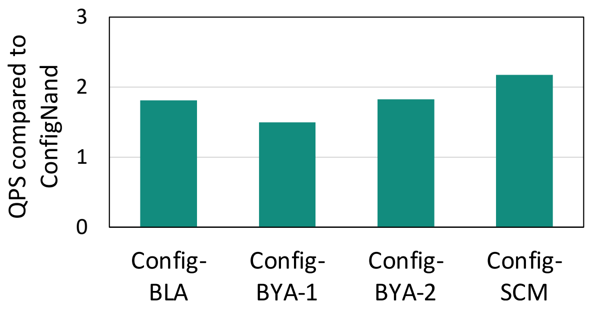


(b) *model 1+*

Figure 4.14: QPS comparison of different configuration of MTrainS with configNandas the baseline for *model 1*and *model 1+*.



(a) 1 Node MTrainS



(b) 2 Nodes MTrainS

Figure 4.15: QPS comparison of different configuration of MTrainS with configNandas the baseline for *model 2*.

because we already met our training QPS requirement. As a result, MTrainS can improve power and cost efficiency for *model 1* and *model 1+* while providing SLA performance. Our experiments show that models scaled beyond *model 1+*require multi-node training, each node leveraging MTrainS. Nevertheless, MTrainS still provides a lower host count because it can store a larger model per host compared to the baseline.

Figure 4.13 shows a comparison of MTrainS vs CDLRM+ for *model 2* (BW bound workload). This model requires 3 hosts to load and train with the baseline configuration using CDLRM+. While MTrainS allows the model to be loaded and trained on one host, it does not meet the QPS requirements because of the higher memory bandwidth required by the model. Also shown in the figure, using 2 nodes, each running MTrainS, significantly improves the performance of *model 2* compared to the single node. Regardless, the 2 nodes still fail to meet the QPS target because the additional BW with SCMs in the system, even with 2 nodes, is not enough to accommodate the high BW demand of *model 2*. However, such capability extends the efficiency of research and development without high QPS production requirements.

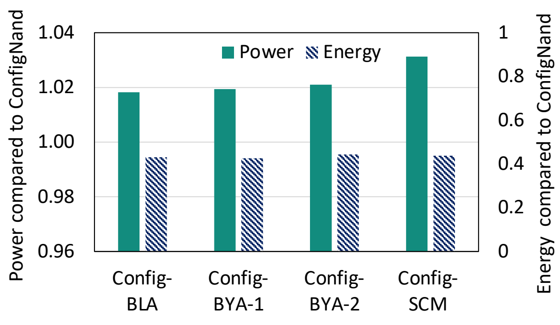
Next, we evaluate the performance of the various system configurations compared to configNand to understand the performance implication of *BYA-SCM* and *BLA-SCM* compared to Nand Flash SSD. In Figure 4.14a and 4.14b, we compare QPS achieved for *model 1* and *model 1+* for different configurations of MTrainS. As shown in Figure 4.14a, for *model 1*, using *BLA-SCM* (configBLA) instead of Nand Flash SSD (configNand) increases QPS by 2×. Using *BYA-SCM* with Nand Flash SSD (configBYA-1 and configBYA-2) increases cache size. Hence lowers traffic to Nand Flash SSD and therefore provides 2.4× QPS. Our results indicate no further improvement in QPS for the setup with both *BYA-SCM* and *BLA-SCM* for configSCMbecause once we increase the cache size per host using *BYA-SCM*, the access to SSDs is out of the critical path.

For *model 1+*, shown in Figure 4.14b, Nand Flash SSD with 384GB of *BYA-SCM* (configBYA-1) increases QPS by 1.73× whereas, 768GB of *BYA-SCM* (configBYA-2) increases by 2.76× due to the increased cache hit rate. When we use both *BYA-SCM* with *BLA-SCM* (configSCM), we achieve 3× QPS. *model 1+* has higher BW requirement due to increased embedding dimension. With Nand Flash SSD we already read 4KB blocks per IO, which is larger than the embedding dimension. Nevertheless, increased embedding dimension results in fewer embedding rows maintained in the cache (DRAM or *BYA-SCM*), given fixed cache size. This increases the miss rate and hence results in increased IO to the SSDs. *BLA-SCM* with higher IOPS can support such an increase in IO, hence showing higher QPS.

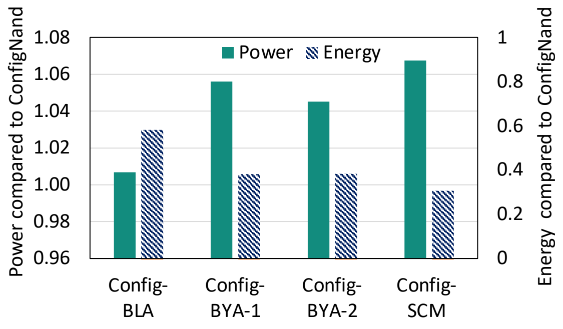
In Figure 4.15a, for *model 2*with 1 node MTrainS, because of *BLA-SCM*'s higher bandwidth and lower latency compared to Nand Flash SSDin configNand, performance increases by 2.2×. By adding *BYA-SCM*we get 2× and 3.2× more performance because the cache hit increases. Similar to *model 1*, the combination of *BYA-SCM*and *BLA-SCM*achieves the best performance. In figure 4.15b, the model is sharded between 2 nodes, and the model size per node is half of 1 node MTrainS. The performance improvement for 2 nodes follows the same pattern as 1 node MTrainS, but the speedup drops compared to 1 node because SSD traffic decreases when the model size per node is reduced, hence caching has less impact compared to 1 node. Although compared to configNand, the various configurations show performance improvement for *model 2*, using SCMs does not provide adequate performance (QPS target) for memory bandwidth-bound workloads, as seen in Figure 4.13.

4.7.3 Power and energy analysis

In Figure 4.16a and Figure 4.16b we show the power and energy of the various MTrainS configurations for *model 1* and *model 1+* to study the increase in power and the total energy usage when introducing *BYA-SCM* and *BLA-SCM*to our system. The power consumption of adding *BYA-SCM* and *BLA-SCM* only increases the overall platform power consumption by

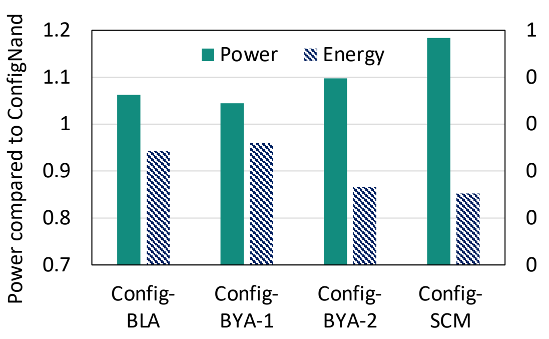


(a) *model 1*

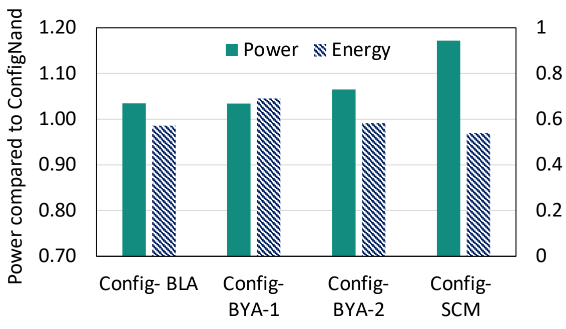


(b) *model 1+*

Figure 4.16: Average power and total energy consumption.



(a) 1 node MTrainS



(b) 2 nodes MTrainS

Figure 4.17: Average power and total energy consumption.

1-3.2%. This is because of the low power consumption of these individual units and because major power consumption contributors are the GPU, CPU, and DRAM. This extra power consumption per node is justified when considering reduced execution time and overall Energy consumption ($Energy = Power \times Time$). We observe a 60% to 70% reduction in the energy consumption across the two models compared to configNand. Figure 4.17a and 4.17b show the power and energy of MTrainS for *model 2*. In this case, adding SCMs increases the power by 3-18%. The higher power in *model 2* is because there are more caching and embedding storage operations in *model 2*due to the larger data access volume compared to *model 1*that increases the power of *BYA-SCM* and *BLA-SCM*. Figure 4.18 shows the power and energy of *model 1* and *model 1+* using MTrainScompared to the baseline configuration with only HBM and DRAM. Compared to training using the baseline, which requires 4-8 nodes to accommodate the models, we see $\sim$ 1/4 - 1/8 power reduction with MTrainS. This reduction is mainly driven by the decrease in the number of nodes required for training with MTrainS. This leads to up to 50% energy reduction. However, as seen in the figure, configNandhas higher energy than the baseline system for *model 1+*because of the low QPS (longer training time). Note that this high reduction is because we have a QPS target for

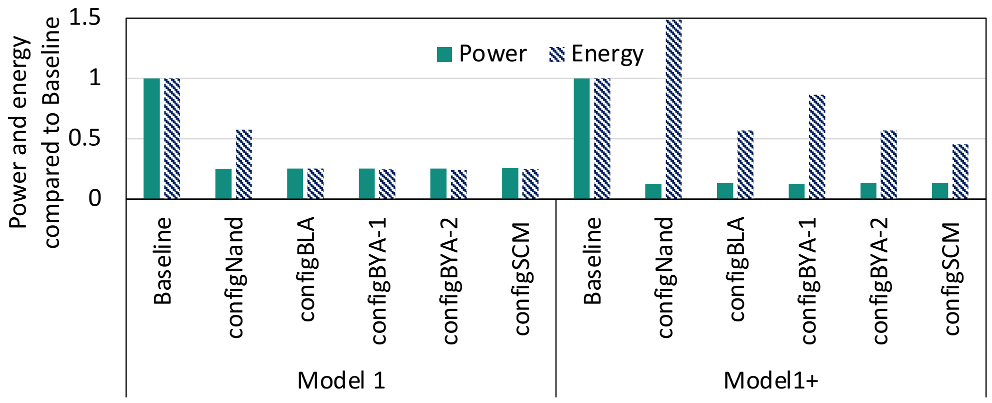


Figure 4.18: Power and energy comparison of MTrainS to baseline system running CLDRM+ for *model 1* and *model 1+*.

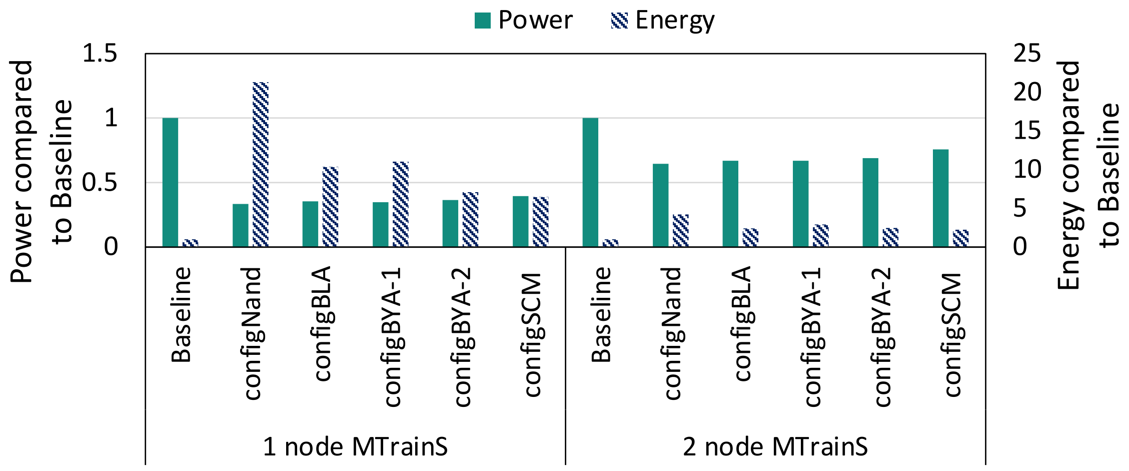


Figure 4.19: Power and energy comparison of MTrainS to baseline system running CLDRM+ for *model 2* using 1 and 2 nodes eith MTrainS.

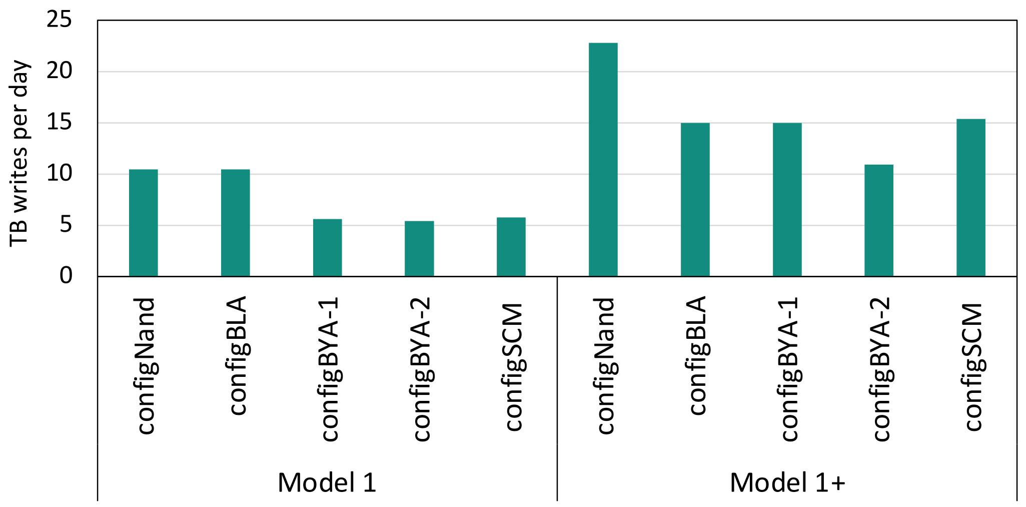
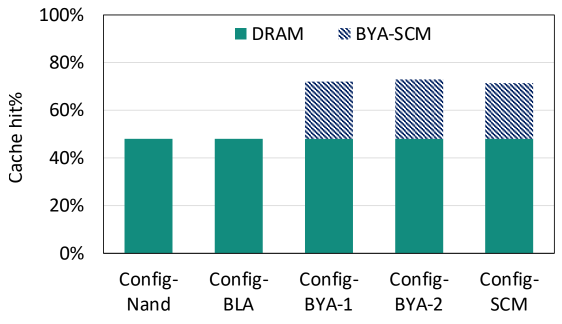


Figure 4.20: Wear out comparison for *model 1* and *model 1+*.

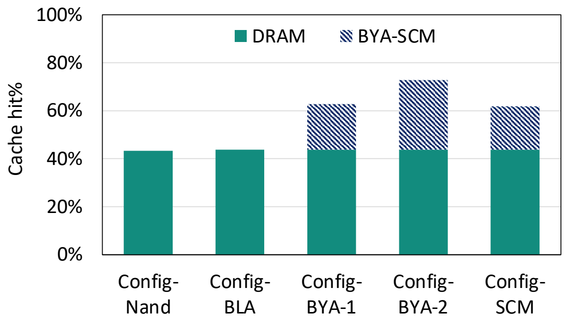
training. If we were to compare absolute performance achieved by the baseline system and MTrainS, the number of nodes running MTrainSrequired would be higher, lowering power and energy reduction. Figure 4.19 shows *model 2*'s average power consumption is reduced by up to 60% in 1 node and 30% in 2 node MTrainSconfigurations. Nonetheless, because MTrainS's QPS for *model 2*is considerably lower than the baseline config (longer training time), even when using 2 nodes running MTrainS, energy consumption is higher compared to the baseline in all configurations, as seen in Figure 4.19.

4.7.4 Storage endurance and wear out

In this and the following sections, we show detailed hardware characteristic analysis for the models that benefit from MTrainS, i.e., *model 1* and *model 1+*. Figure 4.20 measures the TB written per day for *model 1* and *model 1+* to study if we meet our endurance requirements with the diverse configurations of MTrainS. Based on the sizes of *BLA-SCM* and Nand Flash SSD, the endurances to avoid storage wear out are 200TB and 8TB data writes per day (DWPD), respectively. As seen in the figure, while we satisfy our QPS target with configNand for *model 1*, the write per day exceeds the endurance of Nand Flash SSD (8TB). configBYA-1 and configBYA-2satisfy the endurance because of the increase in cache size that decreases writes to storage. All configurations with *BLA-SCM*(configBLAand configSCM) meet the endurance because of its higher DWPD (200TB). For *model 1+*, all configurations backed by Nand Flash SSDdo not meet the endurance. Therefore, for future models *BLA-SCM* is the best option to prevent storage wear out.

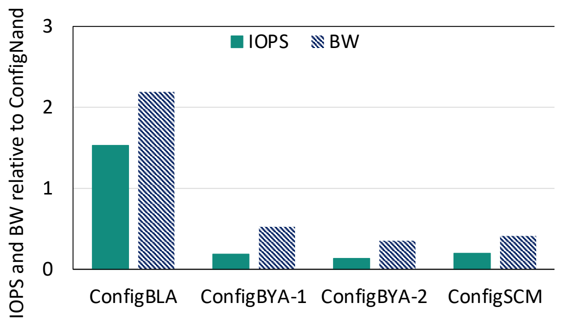


(a) *model 1*

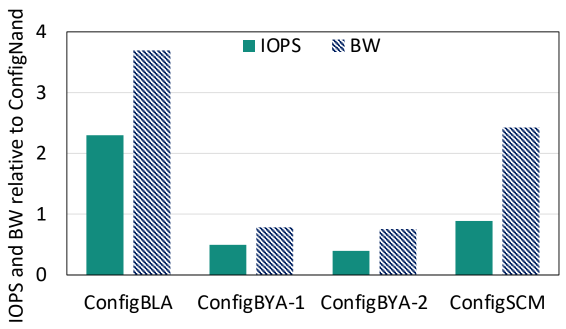


(b) *model 1+*

Figure 4.21: Cache hit rates for *model 1*(a) and *model 1+*(b) for different MTrainS configurations.



(a) *model 1*



(b) *model 1+*

Figure 4.22: IOPS and effective SSD BW for *model 1* and *model 1+*.

4.7.5 Cache hit and IO utilization

Figure 4.21 shows the cache hit rates for *model 1* and *model 1+*. For *model 1* in Figure 4.21a, we observe 50% cache hit rate with configNandand configBLAthat use only DRAM for cache, and addition of *BYA-SCM* (configBYA-1) increases the cache hit to 75%. The 768GB *BYA-SCM* configuration (configBYA-2) does not increase the cache hit further because the 384GB *BYA-SCM*is enough to capture the temporal locality. As a result, both configurations of *BYA-SCM* reach similar QPS, as shown in Figure 4.14a. For *model 1+*, as shown in Figure 4.21b, we measure 40% cache hit rate with configNand. The addition of *BYA-SCM* increases the cache hit to 60% and 70% for configBYA-1 and configBYA-2, respectively. In this case, as shown in Figure 4.14b, higher *BYA-SCM* translates to higher QPS. The reason for the different behavior by *model 1* and *model 1+* is that the bigger embedding dimension in the latter model manifests higher pressure on the caches. As a result, a larger cache size can capture more locality, resulting in less IO traffic and improved QPS.

As shown in Figure 4.22, the trainer achieves higher IOPS and bandwidth with *BLA-SCM*. In Figure 4.22a for *model 1* with *BLA-SCM* 1.5× more IOPS and 2.3× more effective BW

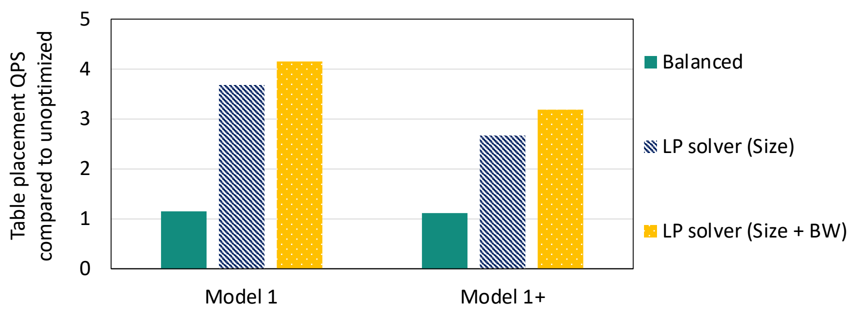


Figure 4.23: QPS increase for different types of embedding table placement for *model 1* and *model 1+*

($IOPS \times embedding\_dim$) is measured, and consecutively more than 2× QPS compared to Nand Flash SSD, as shown in Figure 4.14a. Adding *BYA-SCM*decreases IOPS by up to 80% and BW by up to 60%. Here, the IO traffic is substantially reduced to the point that SSD traffic is not on the critical performance path. Figure 4.22b shows similar trends for *model 1+* with the main difference that the higher *BYA-SCM* continues to improve QPS, and hence IOPS and effective BW. In summary, the MTrainSconfigurations in Table 4.4 improve QPS 1) by reducing IO traffic using *BYA-SCM*and/or 2) by providing more IOPS using *BLA-SCM*.

4.7.6 Embedding table assignment efficiency

In this section, we compare the benefit of four embedding table placements for *model 1*and *model 1+*. The first is placing all embedding tables in RocksDB storage (Nand Flash SSDand *BLA-SCM*) and using the available size of DRAM and *BYA-SCM* memories for caching. We use this as the baseline in Figure 4.23 and call it unoptimized because table placement is only optimized for maximizing the utilization of the sizes of SSDs instead of BW; hence, the data access volume is not balanced among the GPUs. When we compare the unoptimized with using BW to balance the data access of the GPUs, we increase QPS by 15%. Note that using just SSD with DRAM and *BYA-SCM* caching has lower HBM utilization. We then compare this to applying a linear programming solver with only the size of embedding tables and memory types as input. We get 2.5-3.5× more QPS with this placement. Using both size and bandwidth aware placement further increases the QPS to 3.2-4.2×. Hence, table placement is critical when training DLRM with heterogeneous memories, and considering both the size and BW provides the best performance.

4.8 Discussion and future work

In this paper, we demonstrate that using different storage class memories helps us reduce the number of nodes used for training by 4-8× for capacity-bound DLRM models. We learned that for models that are both capacity and bandwidth bound, the additional memory bandwidth with storage class memories is not enough to accommodate the models. Our results show that while we can use *BYA-SCM* with Nand Flash SSD to meet our QPS target, server configuration with just *BLA-SCM* can meet our target QPS for current models. For future scaling using *BYA-SCM* and *BLA-SCM* can help us meet our target QPS. Although these memories are slower than DRAM, they provide enough performance for our memory capacity-bound workloads. Hence we can harness the cost and power benefit of SCMs. In the future, we plan to extend our systems with Compute Express Link (CXL) to access *BYA-SCM*, which will provide us with better memory bandwidth because we don't have to share the memory bus with DRAM and we wont be limited by the system's available DIMM slots. This will help our design to suit bandwidth-bound workloads too.

4.9 Related work

Memory capacity extension for recommendation systems: The high memory capacity and BW demand of embedding operation in recommendation systems impose a challenge on the memory system. Training and Inference use cases present their own unique challenges. For example, in Inference, the latency of each query is important. Ardestani *et al.* [124] present an end-to-end system to leverage SSDs while keeping the latency manageable, and Wilkening *et al.* [132] use the controller in SSD to offload some of the compute closer to the data. Eisenman *et al.* [131] uses SCM to increase memory capacity per host. However, it requires offline preprocessing of embedding tables, which is not applicable for training.

Training is less sensitive to latency but requires higher BW and frequent parameter updates, and hence read and write traffic to the SSDs. Zhao *et al.* [126] present a training system that leverages HBM, DRAM, and SSD. They leverage pipelining to hide SSD latency and use caching to hide lower SSD BW. They follow a parameter server scheme for training large models, as opposed to our distributed, synchronized scheme. Also, in contrast, to sustain high throughput, we leverage the GPU for cache management. Additionally, we show different SCM technologies' performance and power impact on training. Balasubramanian *et al.* [147] implements a CPU-managed cache to leverage HBM and DRAM to expand the memory capacity of embedding tables but only extends to DRAM. Yang *et al.* [141] propose and implement a software caching scheme in GPU backed by CPU memory for embeddings. We build on a similar scheme as [141] and enable multi-level caching with DRAM and *BYA-SCM*, backed by SSDs. However, while they also use caching with DLRM, they focus on using different precisions to reduce embedding storage sizes. In contrast, we extend the memory hierarchy to *BYA-SCM*and flash to expand memory per host and remain neutral to the accuracy of training.

SCM usage for AI workloads: Hildebrand *et al.* [148] developed an integer linear programming-based system that moves tensors between DRAM and SCM. They show that optimized data placement achieves higher performance than a naive approach. Similarly, in our designs, based on the characteristics of applications, we carefully place data on different memories.

4.10 Conclusion

In the pursuit of better model quality, recommendation model complexity, size and amount of training data is increasing over time. This imposes considerable pressure on the compute, bandwidth, IO, and memory capacity provided by the underlying platforms. In this paper we tackle the pressure on memory capacity, and present a hierarchical memory based training to increase memory capacity per host. We quantitatively compare the performance and system level implications of Byte and Block addressable SCMs. Given the temporal locality observed in embedding table access across large scale production workloads, we apply a multi-level cache, utilizing Storage Class Memory (SCM) to reduce the number of nodes required to load and train the memory bound models by 1/4-1/8th. Adding SCM to a platform results in minimal increase in platform cost and power, given that other components such as GPU dominate the power. As a result, we observe overall efficiency benefits due to lower number of nodes required to train a given model. Considering $nodes/Performance$, we observe 2.3× improvement while meeting usecase performance requirement. There are limits to this approach for bandwidth bound models. In these cases the multi-level cache fails to deliver enough bandwidth to satisfy the model performance requirements under the hardware configurations evaluated. By contributing the MTrainS multi-level cache to the open source DLRM software community, we plan to enable these efficiency optimizations across many diverse applications, and enable hardware designers to further optimize future training platforms for SCM.

CHAPTER 5

Resource Management in Integrated CPU-GPU for Collaborative Workloads

Computing systems are evolving towards CPUs, GPUs, and other specialized compute units tightly integrated with shared last-level cache and memory. These integrations enable systems to run applications in fine-grain collaborations between different compute units, which were previously based on offloading the entire application to a specialized unit such as GPU. These collaborations increase performance and resource utilization by partitioning applications among different compute units. While collaboration provides attractive benefits, if we do not manage and monitor the shared resources such as cache, interconnect, and memory rigorously, interference and unfair resource usage inevitably degrade performance. The degradations result from different compute units generating varying demands and pressures on the shared resources. In this paper, for an integrated CPU-GPU architecture, we design shared resource management strategies based on the collaboration patterns of applications. We design cache partitioning and memory transaction prioritization in the interconnect and the memory controller to facilitate collaboration and resource usage to increase the system's overall performance. We achieve, on average, 25% performance improvement and 30% memory read reduction by designing collaboration pattern-based resource management.

5.1 Introduction

The increasing demand for high computing capability to process today's computation and data-intensive workloads drove emerging heterogeneous systems incorporating compute units such as GPUs, FPGAs, and other specialized devices along with CPUs. As a result, heterogeneous systems are abundant in modern systems ranging from mobile to data center architectures [149–153]. The effort to constantly increase performance and reduce energy usage and cost of these specialized architectures led to tightly integrated architectures, such as CPU and GPU, on the same die sharing the same memory and address space.

Sharing memory and address space in heterogeneous systems manifests flexible programmability and efficient collaboration among computing devices via fast and fine-grain data sharing and synchronization. These create new opportunities to design various sophisticated collaborative workloads by utilizing heterogeneous systems together, leading to performance and resource utilization increase. Especially in embedded and desktop systems, where we are limited by lower compute resources availability, energy, and cost, collaborating workloads between compute units improves the efficiency of workloads. Partitioning workloads between different specialized compute units increases the performance of running applications by dividing different tasks of an application into the compute unit that can better perform a specific task or by partitioning the data points of an application among compute units. Previous studies [154–157] have confirmed the substantial performance gain when we partition workloads between different units in heterogeneous systems for diverse applications.

The benefits of collaborative workloads motivate the study of the impact of shared cache, interconnect, and memory controller in integrated architecture to facilitate data sharing between different compute units [158, 159]. Although shared resources management in multicore architecture has been studied for a long time, the distinct architectural properties of computing devices in heterogeneous systems require unique resource usage. For example, the CPU would be more sensitive to cache sharing in integrated CPU-GPU systems because of its latency sensitivity. At the same time, the GPU will be less affected by cache misses. We also have disproportionate memory request because of the architectural differences between CPU and GPU, resulting in interference. Recognizing these differences, previous works have shown the benefits of shared resource management in the cache and memory controller based on both the demand of the compute devices and the applications running on it [160, 161]. But these works focus on disparate applications running on different compute units. However, in collaborative workloads, the collaboration pattern of applications aids in better managing shared resources to increase performance. Recent works have shown the trend towards collaborative workloads and the benefits of shared cache in fine-grain collaborative workloads running in integrated architectures. But this work only considered a shared cache, and a shared cache is not always helpful even in collaborative workloads, depending on the data sharing patterns in the applications. Furthermore, managing other shared resources, such as the interconnect and the memory controller, also improves performance. Hence, exploring efficient resource management in the memory controller and interconnect is essential to increase efficiency and serve the continuous growth of the compute requirements in applications.

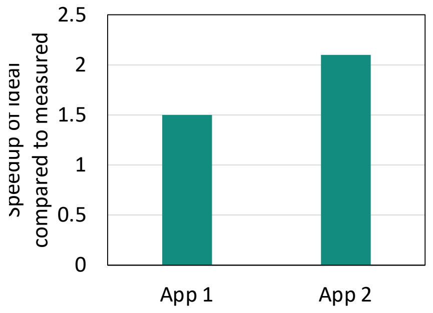
In this paper, considering the interference created by shared resources in integrated systems, we designed CoACT (Collaborative Applications Cache and memory Transaction management) to manage shared resources in integrated CPU-GPU architecture efficiently. We first study the interference created in the cache, interconnect, and memory controller when shared by CPU and GPU. We find that, even though collaboration improves performance, we can potentially lose 50% performance by this interference. We then characterize compute and memory properties of collaborative workloads and learn that the collaborative patterns in applications can help us better manage shared resources. In our solution, we design cache partitioning based on the collaboration pattern of the application to decrease interference and increase data sharing between CPU and GPU. Then again, based on the collaboration pattern of application and characteristics of GPU and CPU, we design memory transaction prioritization in the interconnect and memory controller. Our design provides multiple cache partition and memory transaction prioritization configurations to select from based on what the application requires. Finally, to select the best configuration for an application, we design a simple analytical model that aids in selecting the best configuration for an application. Our experimentation shows that we achieve in average 25% improvement in performance and 30% reduction in memory read with minimal hardware modification. In summary, we contribute the following:

- We designed an efficient resource management unit for heterogeneous integrated CPU-GPU architecture with minimal hardware modifications. Our designs expose multiple policies to orchestrate the shared cache, interconnect, and memory controller.
- We characterize collaborative workloads and analyze parameters that help better manage shared resources in heterogeneous systems. We build a simple analytical model that selects efficient shared resource policies from this.
- We implemented our design in a cycle-accurate CPU and GPU simulator and achieved an average of 25% performance improvement for diverse collaborative applications.

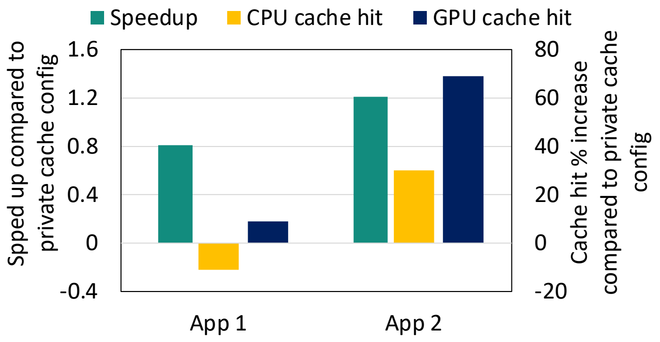
5.2 Background and Motivation

5.2.1 Integrated architecture

Integrated architecture compromises CPU and other accelerators, such as GPUs, together on the same die. This integration enables fast and fine-grain computation and data sharing between CPU and GPU at relatively lower power and cost than discrete systems. These



(a) Ideal vs real system.



(b) Interference in the cache.

Figure 5.1: Ideal speedup and cache interference.

benefits foster multiple commercially available integrated architecture units [149, 150]. With heterogeneous system architecture (HSA) features [162], such as coherent and unified shared memory, it has become easier to utilize integrated architectures in the hardware and software layers. These also led to considerable research focusing on even a much tighter integration sharing cache [158, 159, 161] and software tools to facilitate application development such as OpenCL [163, 164].

5.2.2 Collaborative workloads

In discrete heterogeneous systems, accelerators and compute units are usually attached to the host (CPU) via IO ports, and data and computation are offloaded to accelerators before applications run. In contrast, in integrated systems, we can run the application by dividing it among the host and other accelerators, such as GPUs, in parallel because of the fast communication and synchronization between the compute units. These types of workloads are called collaborative workloads [165]. Collaborative applications help to improve workload performance, energy, and power efficiency by leveraging multiple compute units in parallel. This is helpful when there are tightly communicating tasks within a workload with different computation patterns that can benefit from using compute units/accelerators specialized for different types of computation patterns. We also benefit from collaborative workloads by dividing applications between different units when a workload performs the same task on a large dataset giving us an immense computing resource. We focus on the following two main collaborative patterns discussed in [165].

Task partitioning: In these collaborative workloads, multiple sub-tasks run in parallel within an application, and these different tasks run on different compute units.

Data partitioning: In data partitioning, the application performs the same task for different data points. In this case, the different compute units run the task divided into data points.

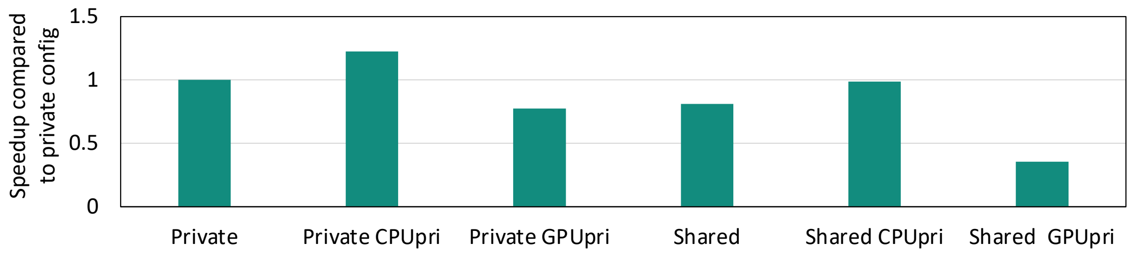


Figure 5.2: Importance of selecting policies.

5.2.3 Shared resources in integrated systems

While guaranteeing fair resources is not a problem in homogeneous computing such as multicore CPU, in a system with CPU and GPU, it can create interference. This interference is because of the architectural structure differences in how computations and memory accesses are organized and performed in the different units. When CPU and GPU work independently, we want to provide fair resource assignments. However, with collaborative workloads, we want to increase resource utilization based on how CPU and GPU perform tasks in the application. In collaborative workloads, the collaborative pattern also guides us on how to share resources—leading to a better resource management.

5.2.3.1 Shared resource in collaborative CPU-GPU workloads

In discrete heterogeneous systems, CPUs and GPUs have their own dedicated resources. However, for integrated heterogeneous systems, one of the primary shared resources is memory. When memory is shared between CPU and GPU, because CPUs are optimized to lower memory latency and GPU for high memory bandwidth, the GPU will have much more requests in the memory controller than the CPU. In tightly integrated architecture, the cache is also another resource that is beneficial to share, as shown in [159]. When a cache is shared between CPU and GPU, the GPU will have much more requests, and at the same time, the performance is not sensitive to cache misses. Another resource that will be shared when we have a shared cache is the interconnect. Similar to the memory controller, we must be careful about sharing and issuing requests in the interconnect.

5.2.3.2 Performance degradation in shared resource

As discussed above, shared resources between various compute units create unfair resource sharing between CPU and GPU, leading to performance degradation. Even though dividing applications between CPU and GPU increases performance, because of the interferences, we might not achieve the maximum performance we can get. In collaborative workloads, the interactions between CPU and GPU are also tailored to the collaborative pattern of the

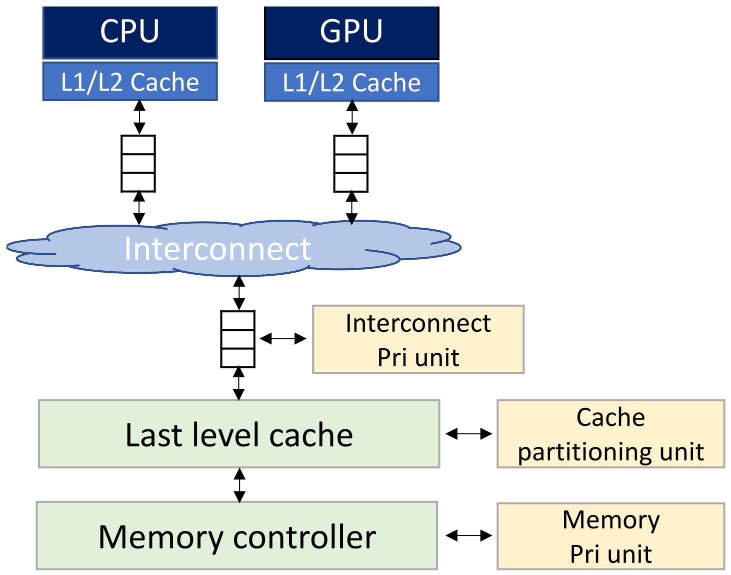


Figure 5.3: CoACT architecture overview.

application. Hence, data access and sharing are specific to the applications and require a unique solution for each application. We performed the following studies to understand the impact of sharing resources in integrated systems and application-specific shared resource management.

To understand the performance benefits we could get from collaborative workloads, we compared the measured performance of collaborative workloads divided between CPU and GPU, to an ideal scenario, where CPU and GPU are not affected by running parts of the workloads in parallel while sharing cache, interconnect and memory. To estimate an ideal scenario, we calculated the rate at which CPU and GPU perform computation if we dedicated the entire resource to them. While this is an ideal scenario, it shows us how far we are from the maximum performance we can achieve. Figure 5.1a shows these results for two applications. Here, the performance gap between the ideal and the measured speedup indicates that if we reduce interference by managing the shared resources, we will have the potential to increase performance. While the ideal scenario is unattainable, this paper explores how we can drive performance improvement towards the ideal speedup by efficiently developing resource management.

In Figure 5.1b, we show cache interference created by CPU and GPU sharing cache and how we can use collaborative patterns to combat these problems. For App 1 and App 2, we compare a private cache configuration where equal cache space is dedicated to CPU and GPU to a configuration where the cache is shared. In the figure, we see that, for App 1, which has private data sharing between CPU and GPU, when we compare private and shared configs, because of the interference between requests of CPU and GPU in the cache in the shared config, we lose performance. When we see the cache hit, in shared configuration, we lose 10% cache hit compared to private for CPU, whereas, for GPU, we gain 9% cache hit in shared config. But we can see that even when we get more GPU cache hits, we still have lower performance than private because cache misses do not hurt GPU's performance. However, a private config is not always the best configuration, as we see the opposite behavior for App 2 because it shares data between CPU and GPU. Hence, depending on the collaborative patterns, the cache should be configured differently.

Figure 5.2 we show the performance of App 1 for multiple management policies. We examined private and shared policies and CPU and GPU memory transaction prioritization in the interconnect and memory controller. In the figure, we see that prioritizing memory transactions affects the performance of applications because managing interconnect and memory controller potentially decreases data access latency. We can also observe that different policies influence the performance we achieve. Hence, it is crucial to select the best policy that boosts the performance of an application. Note that the best policy is different for each application and is tailored to the collaborative pattern of the application.

5.3 CoACT Design

In our designs, we focus on managing three critical shared resources in integrated CPU-GPU architecture that contribute to interference and, as a result, performance degradation. These resources are the last level cache, the interconnect, and the memory controller. We carefully designed configurable strategies to partition and use these resources among cores in CPU and GPU based on the characteristic of workloads to improve overall performance. We then implemented a simple analytical model that considers the collaboration strategies of a particular application and its computation and memory characteristics to determine resource management configurations for the shared resources before the application runs. Our design focuses on two parts. The first part is **shared resource management design in the hardware** to enable configurable resource utilization strategies. The second part is understanding applications' computation and memory access and deciding the configuration for **selecting the best configuration per application based on its characteristics.**

5.3.1 Hardware design

Figure 5.3 shows the overview of where our design fits in integrated CPU-GPU architecture. In our designs, to enable configurable resource sharing, we added a configurable cache partitioning unit to assign cache resources for CPU and GPU based on the application's

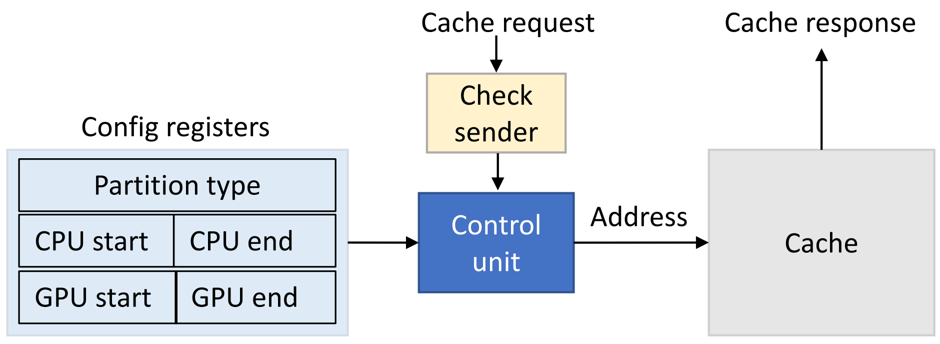


Figure 5.4: Cache partitioning overview.

characteristics. This helps to minimize interference between CPU and GPU and eases fast communication, depending on the application. We then added prioritization units in the queues of the interconnect and the memory controller to optimize the data access latency. The details of the designs are shown below.

5.3.1.1 Memory request packets

Our design depends on identifying packets coming from the CPU and GPU. This identification enables us to treat CPU and GPU memory access differently, helping us perform cache sharing/partitioning and packet prioritization. We do this by reusing the core id bits in the packets. As seen in Figure 5.4 and 5.5, both cache partition and prioritization units have a sender checker that compares the id and resolves if the packet is coming from the CPU or GPU.

5.3.1.2 Cache partitioning

The last level cache (LLC) partitioning provides dedicated cache resources for CPU and GPU in an integrated architecture. In our design, for all the cache lines in LLC, we configure what percentage of the lines we should dedicate to CPU and what percentage to GPU. As shown in Figure 5.4, designated config registers are used to hold the cache line address range used for CPU and GPU. The control unit in the cache is then used to translate the incoming address to the target space (CPU or GPU) based on the register values and the packet's origin. This will give us the address to access the correct partition space. We implemented three Cache partitioning policies. The cache partitioning options are as follows:
Private partitioning: In this policy, both CPU and GPU will have cache address range dedicated to their memory access. In this scenario, Write/read to the cache are private to CPU and GPU cache space. Depending on the application, we can configure the percentage

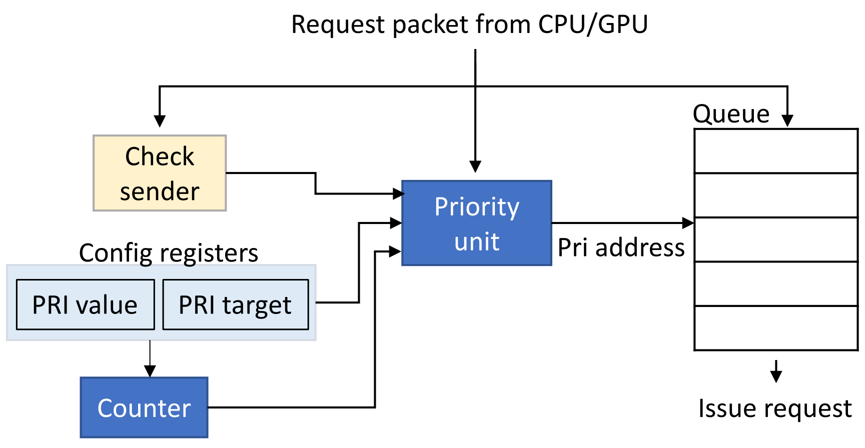


Figure 5.5: Interconnect and memory prioritization architecture overview.

of the cache we want to provide to the CPU and GPU. This policy will be beneficial when the CPU and GPU collaborate by partitioning the input or/and output data, and each unit operates on private data. If the rate at which CPU and GPU access data is different, then there will be cache interference. This type of policy will help in this case.

Shared partition: In this policy, the entire cache space is used by both CPU and GPU. The partition type configuration register is used to indicate shared policy. When both CPU and GPU try to read/write to the cache, it can access the entire LLC space. This will be an optimal policy when data is shared between CPU and GPU in the collaboration pattern of the applications. Depending on the percentage of data shared, we can reduce compulsory misses because the CPU can use data brought by GPU and vice versa.

Private write, shared read partitioning In this case, both CPU and GPU will have dedicated cache space for writing, but the devices can read from the whole cache. This will help when we have some portion of the input/output data shared between CPU and GPU. By carefully partitioning the private writing space, we can achieve better performance with this policy depending on the application's collaboration pattern.

Supporting cache partitioning requires modifying the design of the coherence protocol and caching algorithm. We changed the coherence protocol in the LLC to handle CPU and GPU transactions separately based on the partition type. Similarly, in the caching algorithm, such as the replacement policy, we must consider the partitioning types. For example, in private partitioning, during replacement, if a transaction is from the CPU, only CPU address ranges should be considered. We configure cache partitioning before the application starts running. The configurations include partition type and percentage of CPU and GPU space.

5.3.1.3 Interconnect and memory prioritization

The CPU and GPU have different memory access characteristics. Since the GPU runs much larger threads than the CPU, it will require much more memory bandwidth than the CPU. But this will lead to the CPU's memory request taking longer. However, we know the CPU is more sensitive to memory access latency. Hence, we designed strategies to prioritize transactions based on the criticality of incoming memory requests. The overall design is shown in Figure 5.5. A packet is critical when the memory request coming from the particular compute unit is small or if we surpass a preset threshold of prioritizing a particular unit. In our implementations, we designed prioritization in the interconnect buffer queues and the read queue of the memory controller. Based on the criticality of each transaction, we decide which transaction to prioritize. In the memory controller, we use registers to set the target device to prioritize and the threshold of prioritization. The prioritization target tells which incoming transactions to the memory controller have priority (CPU or GPU). On the other hand, the threshold prevents starving the non-prioritization compute unit. For example, if the target is CPU and the threshold is 5, we will prioritize 5 CPU transactions before prioritizing the GPU. The prioritization in the interconnect buffers also follows the same strategy. The policies in the memory controller and interconnect are as shown below.
Prioritize CPU or GPU: In this case, we first specify which compute unit's memory request to prioritize. Then, based on the specified target, we perform prioritization in the interconnect buffers and the memory controller until we hit a threshold that is specified before the execution of the application starts. If no threshold is specified, we will prioritize memory requests coming from the target in the complete execution of an application.
Adaptive prioritization: In this case, we prioritize based on the current state of the queues in the interconnect and the memory controller. If we have fewer memory requests from the CPU, we prioritize the CPU, whereas if we have fewer GPU requests, we prioritize that. In adaptive prioritization, besides the target and threshold registers, we will also monitor the total number of CPU and GPU requests in the queues.

5.3.2 Policy selection

The cache partition and memory prioritization policies above give us numerous policies to choose from for an application. To determine the best policy for an application, we designed a simple analytical model based on the collaboration patterns of applications. The analytical model takes input parameters from offline profiling and the static characteristics of the applications. Note that the above config register values can be a vast space to choose from, and we can use a sophisticated heuristic to select a better policy. We left such intricate solutions for future work.

5.3.2.1 Policy selection parameters

We use parameters from offline profiling and the static characteristics of the applications to select the best policy. From the static properties of applications, we get the type of partitioning (task or data), input or output data partitioning, and the proportion of data shared between the CPU and GPU. From offline profiling applications, we get the best partition proportion for an application and locality. The details of the parameters are shown below.
Type of collaborations: These tell us if the application partitions data or tasks between CPU and GPU. This is static information that is decided at the design time of the application. These help us understand the data access characteristics of the applications. While in data partitioning, there are possibilities where the applications might not share data, in task partitioning, the most common type of collaboration is a producer-consumer pattern that usually requires data sharing between CPU and GPU.
Data partition proportion: For applications that are data partitioned, the proportion of data processing performed by GPU and CPU will tell us about the cache space we should provide for CPU or GPU. If we were dealing with just homogeneous cores, then all cores would perform an equal amount of work, but in heterogeneous systems, we will have a disproportion of work done between CPU and GPU, depending on the tasks in the application. Hence, this property will tell us about the desired cache partition. This parameter also tells us the criticality of the memory transactions. Data partitioned proportion is collected from offline processing because we can only find the best partition by profiling the application.
Shared and private data percentage: This property is decided on the design time of the application. The shared and private data parentage is a critical parameter because it will inform us whether we want to share the cache or dedicate private space to the CPU and GPU.
Locality: Locality helps us understand the partitioning strategy. It helps us understand if we should provide more cache space to the CPU than GPU or vice versa despite the partition percentage. Sometimes, depending on the locality, we can adjust the cache space for each compute unit to maximize the cache hits. We measured locality based on offline profiling and quantified it as the cache hit rates for both CPU and GPU.

5.3.2.2 Analytical model for policy selection

We use the metrics above to decide the best configuration for a collaborative application. Based on the characterization of a handful of applications, we use the following rules to

Algorithm 1 Best policy selection algorithm

Input: Coll_type, partition, shared_p, cache_hit.

Output: partition_type, pri_type, cpu_cache_space.

```
1: if $coll\_type$ is task then
2:     $partition\_type \leftarrow shared$
3:     $pri\_type \leftarrow adaptive$
4: else if $coll\_type$ is data then
5:     if $shared\_p \geq 50\%$ then
6:         $partition\_type \leftarrow shared$
7:         if $partition < 0.5$ then
8:             $pri\_type \leftarrow cpu$
9:         else if $partition > 0.5$ then
10:            $pri\_type \leftarrow gpu$
11:        else
12:            $pri\_type \leftarrow adaptive$
13:    else
14:        $partition\_type \leftarrow private$
15:        if $cache\_hit < 50\%$ then
16:            $cpu\_cache\_space \leftarrow 100\%$
17:            $pri\_type \leftarrow gpu$
18:        else
19:            $cpu\_cache\_space \leftarrow 50\%$
20:            if $partition < 0.5$ then
21:                $pri\_type \leftarrow cpu$
22:            else if $partition > 0.5$ then
23:                $pri\_type \leftarrow gpu$
24:            else
25:                $pri\_type \leftarrow adaptive$
```

build our simple analytical model. The algorithm for the analytical model is shown in Algorithm 1.

1. The type of collaboration, task, or data partition helps us determine if we need a private or shared cache. In task partitioning, the most common type of collaboration is a producer-consumer scheme. All of the tasks partitioned workloads we evaluate show these characteristics. Here, one of the compute units (CPU or GPU) produces data to be used by the other. Hence, most of the data is shared. As a result, for task partition, we use a shared cache configuration. For data partitioning, we need to further understand how data is partitioned between CPU and GPU.

2. For data partitioned workloads, the first factor to investigate is the percentage of data shared between GPU and CPU. From our characterizations, we saw that most

applications are partitioned in the input or output data, also discussed in [165]. If the compute units are sharing either input or output data and the percentage of data shared is greater than 50% then the cache should be shared. Otherwise, a private configuration provides better performance because it minimizes interference between CPU and GPU.

3. The proportion of input or output data partitioned between CPU and GPU in a private setting can help us understand how to partition the cache. Here, the locality is also a valuable parameter. In our experiments, we saw that utilizing the CPU cache hit rate in the LLC provides more information for selecting the best policy. If we have locality (high CPU cache hit) in the application when we run it on the baseline, we keep the cache partition in the baseline to maintain the cache hit. When the application's locality is low (low CPU cache hit), we can dedicate the whole cache space to the CPU to extract more cache hits. Here, we focus more on increasing cache hits for the CPU because of its sensitivity to a cache miss.

4. We considered prioritizing in both the interconnect and memory controller together. For example, if we prioritize the CPU in the interconnect, we also prioritize it in the memory controller. Our strategy in prioritizing is that, when we consider task partitioning since it follows a producer-consumer pattern, an adaptive configuration fits the best because both CPU and GPU contribute a comparable number of memory accesses.

5. For data partitioning, if we have a shared cache or a private configuration with 50-50 cache space to GPU and CPU, then depending on the CPU-GPU partition proportion, we prioritize the one that has a smaller partition as seen in Algorithm 1. In a private configuration, if we dedicate the entire cache space to the CPU, we prioritize the GPU, assuming the cache will improve the performance of the CPU.

5.4 Experimental Setup

5.4.1 Simulation setup

We design CoACT using the gem5-gpgpu simulator [166], a cycle-accurate environment that integrates the gem5 [167] and gpgpu-sim [168] simulators. We run all of our simulations using the full system mode of gem5 to model the interaction between CPU and GPU accurately. We modeled our cache and coherence using gem5 ruby, which is a detailed memory subsystem simulator. We use MOESI_hammer cache coherence protocol that uses

Table 5.1: Simulation setup.

Specification	**Simulation config**
CPU cores	8
CPU clock	3GHz
GPU cores	16 (Fermi cores)
GPU clock	700MHz
CPU L1I/L1D cache	32KB assoc = 8
GPU L1I/L1D cache	32KB assoc = 8
CPU L2 cache	512KB assoc= 16 (private to the cores)
GPU L2 cache	4MB assoc= 16 (shared by the cores)
L3 cache	8MB assoc= 16
Coherence protocol	MOESI_hammer
Memory	DDR4_2400
Interconnect	crossbar with 128 bits bus width

MOESI protocol for CPU caches and LLC and hammer for GPU. We use a crossbar for the interconnect. We use cacti [169] to model the latencies of the private and shared caches of both CPU and GPU and the LLC. Table 5.1 shows the simulation hardware configurations.

5.4.2 Workloads

We use collaborative workloads from CHAI benchmarks [165] for our detail analysis and Rodinia benchmarks for a much broader analysis. Chai benchmarks follow data and task parallel collaborative patterns. Chai provides workloads from diverse application domains with very distinct collaboration patterns and fine-grain data sharing. For Rodinia benchmarks [27], we modified the benchmarks to divide the work between CPU and GPU. We use openmp for the CPU and cuda for GPU in our Rodinia implementations. In the Rodinia benchmarks, our modification uses data partitioning. In our evaluation, for all of the applications, we used data set sizes larger than the total cache sizes of the CPU, GPU, and LLC to evaluate memory access of collaborative workloads. We show the details of the workloads in Table 5.2.

5.4.3 Baselines

For our baseline, we use a private L3 cache configuration with the cache size divided equally among CPU and GPU and without transaction management in the interconnect and the memory controller. We compared this baseline with CoACT and the following related works. (1) Private_cache [161], this works evaluates different cache partitioning proportions for CPU and GPU. To compare CoACT to this work, we took the different cache partitioning

Table 5.2: Collaborative workloads characterization.

Apps	Coll type	Partition	Shared %	Cache hit %
HSTO	Data	0.3	98	9
HSTI	Data	0.2	3	11.8
SC	Data	0.4	∼0	17
PAD	Data	0.6	∼0	33
RSCD	Data	0.2	96	32
RSCT	Task	-	∼100	45
BS	Data	0.2	∼0	35
SSSP	Task	-	∼100	67
BFS	Task	-	∼100	74
KMEANS	Data	0.1	∼0	71
TQ	Task	-	∼100	30
TQH	Task	-	∼100	2.5
CEDT	Task	-	∼100	11
NN	Data	0.5	∼0	23.8
PF	Data	0.6	∼0	74

they use and chose the best performing partition. (2) Shared_cache [159], uses a shared cache configuration for all workloads. (3) SMS [160], that manages CPU and GPU memory transactions in the memory controller. Since SMS does not deal with cache partitioning, we use basic cache partitioning (50-50% private and shared) on top of SMS's implementation and choose the best one. We also compare CoACT with different cache partition baselines, namely private cache with 25%, 75% dedicated for CPU to evaluate how the baseline cache configuration affects our performance improvement. We implemented all of our baselines and previous work comparison in gem5-gpgpu with the configuration in Table 5.1.

5.5 Evaluations

5.5.1 Workload characteristic measurement

For all the workloads we use in our experimentation, we first measure the policy selection parameters described in Section 5.3. We show the result of the measurements in Table 5.2. We can see in the table that the applications have diverse collaboration types (both task and data). This property is decided during algorithm design time. Next, we have the partition proportion between CPU and GPU. To get the best partition, we run the applications by partitioning CPU and GPU data points from 0 to 100% by increasing the partition by 10% and chose the best performing partition in our baseline configuration, which has 50-50% cache space dedicated to both CPU and GPU. Here we use offline profiling because we

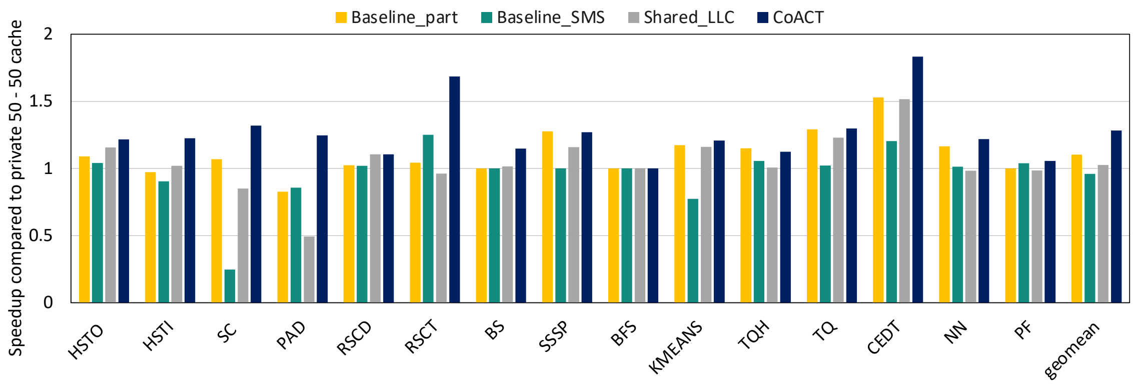


Figure 5.6: Performance analysis and improvement for all workloads.

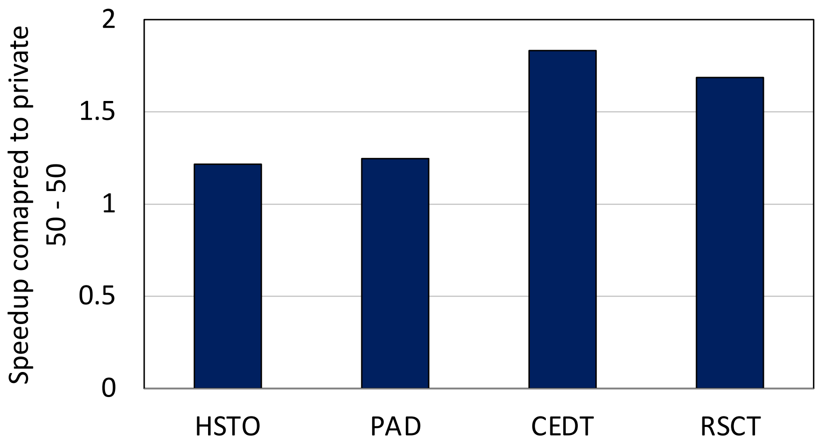


Figure 5.7: Speedup of representative applications.

perform profiling and choose the best policy once, and we can use that policy afterward when we require running the application multiple times. We see that the best partition for most of the applications has a smaller CPU partition, as expected, because of the more significant computing capability of the GPU. We measured the data shared percentage by counting the size of input and output data shared between CPU and GPU divided by the total data. To quantify CPU locality, we measure the LLC CPU cache hit rate in the baseline system shown in the table.

5.5.2 Performance analysis for all workloads

Figure 5.6 show the speedup of CoACT, *Private_cache* that uses just cache partitioning, *Shared_cache* that always uses shared cache and *SMS* that manages CPU and GPU memory request in the memory controller and compared to the baseline system with 50-50% cache partition. From the figure, we can see that CoACT, because it uses both cache management and interconnects and memory controller prioritization, it provides up to 90% performance

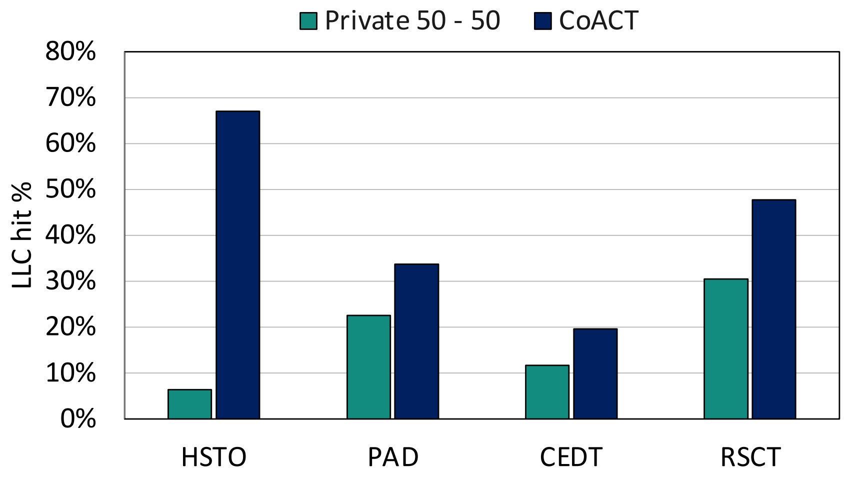


Figure 5.8: LLC cache hit for representative applications.

increase and 25% improvement on average. When we compare these to the other configurations, we see that in most of the applications, using a collaborative pattern in conjunction with both cache and memory management achieves better performance. When we compare private_shared cache with CoACT, we see that in workloads with minimal shared data like HSTI and SC, it has better performance than the baseline. Still, because we include an additional management step (prioritization) in our design, even in private configured workloads, CoACT performs better. The shared cache configuration has a closer performance to CoACT in applications with a large percentage of shared data, such as HSTO, TQ, and RSCD. However, because it is only optimized for applications with shared cache, it loses significant performance in workloads such as SC and PAD with less shared data. In SMS, which mainly optimizes the memory controller for fair CPU and GPU memory access management, CoACT outperforms it because of the additional cache partitioning in our design and because we use collaborative patterns for resource management.

5.5.3 Application and hardware characteristic analysis

5.5.3.1 Speedup and LLC cache hit

In Figure 5.7 and 5.8 we show the speedup and LLC cache hit rate for four applications that are representative of the unique characteristics of collaborative workloads. We use these applications for deeper analysis and compare CoACT to the baseline configuration. For HSTO, which shares a large percentage of the data and with CPU-GPU partitioning less than 0.5, we use a shared cache with CPU prioritization. From the figure, we see that with this config, we attain 21% improvements. When we see the cache hit in Figure 5.8,

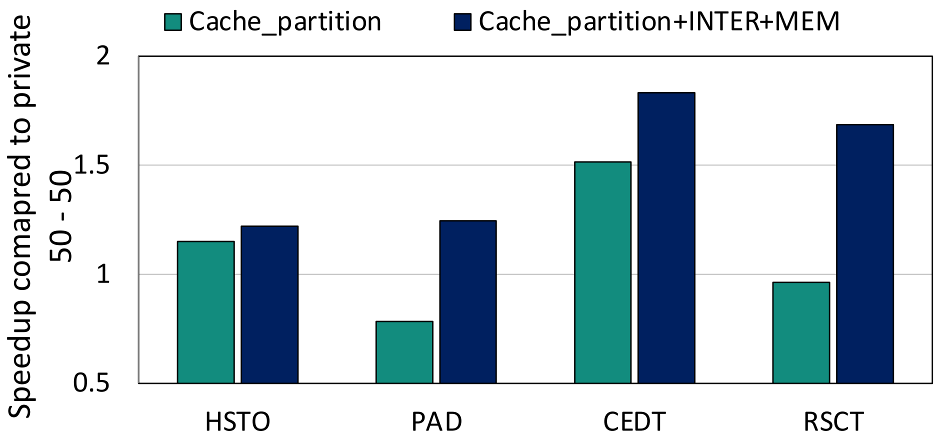


Figure 5.9: Benefit of individual features.

because of the large shared data, the LLC cache hit rate increases. PAD, as seen in Table 5.2 has no shared data and low locality. Based on Algorithm 1, the best policy is all of the LLC reserved for the CPU with GPU prioritization. As seen in Figure 5.7, we achieve 25% speedup and LLC cache hit increase (Figure 5.8). For RSCT and CEDT, which are both task partitioned, the best policy is shared cache with adaptive prioritization config. These configs provide performance improvement, and LLC cache hit increases, as shown in the figures.

5.5.3.2 Benefits of each feature

In Figure 5.9, we show the benefit of each feature (cache partitioning and interconnect + memory prioritization) for the four representative applications. The figure shows that in all cases, cache partition alone does not provide the best performance. In HSTO, we see that we get an additional 5% performance increase with prioritization. For CEDT and RSCT, the shared configuration combined with the prioritization provides the best performance. In general, we can conclude that shared cache alone provides 10% improvement, and the addition of the prioritization feature delivers an additional 50% benefit on average across all workloads.

5.5.3.3 Memory read reduction

Figure 5.10 shows the memory read reduction when we use CoACT compared to the baseline with 50-50% cache partition. From the figure we can see that the memory read is reduced proportionally to the LLC cache hit rate. HSTO and RSCT, which gain the largest LLC hit rate with our design, achieve 55% memory read reduction. In general, we attain on

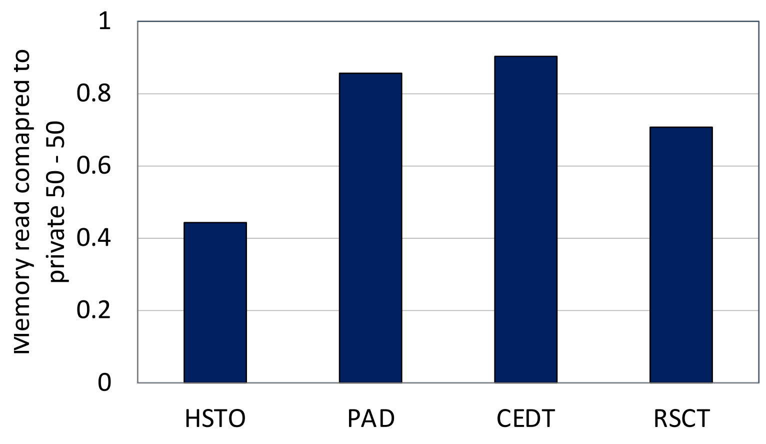


Figure 5.10: Memory read reduction compared to baseline 50-50%.

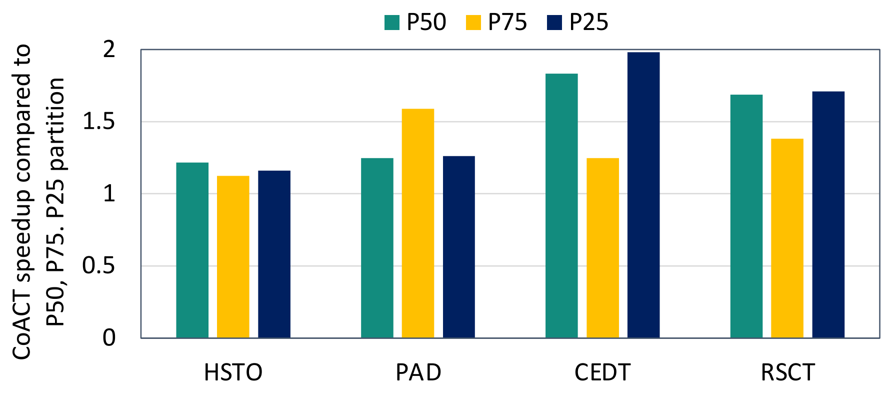


Figure 5.11: Effect of changing baseline partition.

average 30% memory read deduction with CoACT. This reduction improves memory access latency.

5.5.3.4 Changing baseline partition

In Figure 5.11 we show the speed of CoACT compared to different partitions, P50 (baseline in all above studies), P75 (cache partition with 75% assigned to CPU), and P25 (25% of the cache assigned to CPU). From the figure, we see that in all cases CoACT achieves performance improvement compared to all of the baselines. In the figure, we see that when the cache partition percentage for the CPU is larger (P75), the speedup is reduced because having a large cache space for the CPU improves the performance of the P75 baseline. In contrast, we have a higher speedup for the P25 baseline because it has lower performance due to the small CPU cache space. In general, we can observe that CoACT achieves better performance for various baselines.

5.6 Related Works

Collaborative workloads are shown to improve performance and energy efficiency. [154–157] show the performance improvement we achieve by strategically partitioning workloads between CPU and GPU. [154] shows the partitioning of workloads to CPU and GPU on the fly, and [155] shows the partitioning strategy for irregular workloads. While these works show the benefits of performing meticulous partitioning of workloads to different compute units, in contrast, we focus on the hardware characteristics of collaboration workloads in our work.

Previous works have shown the importance of managing the shared resource in heterogeneous systems. [159] shows the benefits of shared last level cache in CPU-GPU system. They show that a shared cache is advantageous for providing fast data sharing. In our work, we extend their studies and show that while the shared cache is helpful in some types of collaborative workloads that are sharing data between CPU and GPU, in some workload configurations, it increases interference between CPU and GPU and degrades performance. [161], shows in integrated architecture, when we have independent applications running in CPU and GPU, private cache configuration can aid in performance. In this work, we show that private cache settings can also be used when CPU and GPU are working together, and we can use collaborative characteristics of applications to share the cache between CPU and GPU efficiently. [160] uses a staged memory controller design for integrated CPU-GPU that shares memory. They focus on when the CPU and GPU are running a separate application. In contrast, we use the collaboration pattern of applications to manage the memory controller better.

5.7 Conclusion

Heterogeneous systems are abundant in modern computing systems because they support the performance demand of today's high compute and data-demanding applications. The continuous growth and workload diversification lead computing systems towards CPUs, GPUs, and other specialized compute units tightly integrated into the same die sharing last-level cache and memory. This integrated architecture creates the opportunity to run applications by dividing them among various compute units using fine-grain data sharing, boosting performance and energy efficiency. However, if we do not carefully manage and monitor the shared resources in the compute units, we will not reap the performance improvement benefits of integrated architecture. In this paper, we design shared resource management CoACT for collaborative workloads. In our design, we manage shared cache in integrated architecture by carefully partitioning available resources between CPU and GPU. We also manage memory transactions by implementing prioritization units in the shared interconnect and the memory controller using collaborative patterns that are unique for each application. We achieve, on average, 25% performance improvement across a diverse set of applications from different domains.

CHAPTER 6

Conclusion

This dissertation studies software, systems, and hardware mechanisms to efficiently utilize heterogeneous compute and memory units in different system configurations for diverse workloads. We analyze the challenges of heterogeneous systems designs and utilization for server-grade and integrated systems and provide workload characteristics-aware solutions. We first study how to map applications in integrated and discrete systems with various compute units using intrinsic properties of applications and design a framework that achieves 98% accuracy in selecting the best performant and energy efficient platform. We then study how to design heterogeneous memory-based servers for large resource-consuming workloads in data centers based on applications' locality and bandwidth characteristics and how these new memory technologies fit with CPUs and GPUs. Our studies show that by introducing heterogeneous memories, we achieve 80% performance and 43-48% cost reduction for key-value stores and up to $8\times$ training host and power reduction for recommendation systems. For integrated CPU-GPU, we build lightweight software and hardware mechanisms to manage the shared cache, interconnect, and memory controller using collaborative patterns of applications and achieve 25% performance improvements on average.

BIBLIOGRAPHY

[1] Yasin, A., “A Top-Down method for performance analysis and counters architecture,” *Proc. ISPASS*, 2014.

[2] Kassa, H. T., Verma, T., Austin, T., and Bertacco, V., “ChipAdvisor: A Machine Learning Approach for Mapping Applications to Heterogeneous Systems,” *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, 2021, pp. 292–299.

[3] Kassa, H. T., Akers, J., Ghosh, M., Cao, Z., Gogte, V., and Dreslinski, R., “Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension,” *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, USENIX Association, July 2021, pp. 821–837.

[4] Kassa, H. T., Akers, J., Ghosh, M., Cao, Z., Gogte, V., and Dreslinski, R., “Power-Optimized Deployment of Key-Value Stores Using Storage Class Memory,” *ACM Trans. Storage*, Vol. 18, No. 2, mar 2022.

[5] Asanovic, K., Bodik, R., et al., “The Landscape of Parallel Computing Research: A View from Berkeley,” Tech. rep., EECS Department, University of California, Berkeley, Dec 2006.

[6] Weinberg, J., McCracken, M., Strohmaier, E., and Snavely, A., “Quantifying Locality In The Memory Access Patterns of HPC Applications,” *Proc. SC*, 2005.

[7] Reagen, B., Adolf, R., Shao, Y., Wei, G., and Brooks, D., “MachSuite: Benchmarks for accelerator design and customized architectures,” *Proc. IISWC*, 2014.

[8] *OpenCL Best Practices Guide*, https://hpc.oit.uci.edu/nvidia-doc/sdk-cuda-doc/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf.

[9] *Intel FPGA SDK for OpenCL*, https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf.

[10] Jiang, J. et al., “Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs,” *FPGA*, 2020.

[11] Farooqui, N. et al., “Leo: A Profile-Driven Dynamic Optimization Framework for GPU Applications,” *Proc. TRIOS*, 2014.

[12] Cong, J., Fang, Z., et al., "Understanding Performance Differences of FPGAs and GPUs," *Proc. FPGA*, 2018.

[13] Che, S., Li, J., et al., "Accelerating Compute-Intensive Applications with GPUs and FPGAs," *Proc. SASP*, 2008.

[14] Prabhakar, R., Koeplinger, D., Brown, K., et al., "Generating Configurable Hardware from Parallel Patterns," *Proc. ASPLOS*, 2016.

[15] Wang, S., Liang, Y., and Zhang, W., "Poly: Efficient Heterogeneous System and Application Management for Interactive Applications," *Proc. HPCA*, 2019.

[16] Hill, M. and Janapa Reddi, V., "Gables: A Roofline Model for Mobile SoCs," *Proc. HPCA*, 2019.

[17] Shao, Y. and Brooks, D., "ISA-independent workload characterization and its implications for specialized architectures," *Proc. ISPASS*, 2013.

[18] Ould-Ahmed-Vall, E., Woodlee, J., et al., "Using Model Trees for Computer Architecture Performance Analysis of Software Applications," *Proc. ISPASS*, 2007.

[19] Wang, Z., Grewe, D., and O'boyle, M., "Automatic and Portable Mapping of Data Parallel Programs to OpenCL for GPU-based Heterogeneous Systems," *ACM Trans. Archit. Code Optim. 2014*.

[20] *Intel VTune Amplifiers*, https://software.intel.com/en-us/vtune.

[21] Williams, S., Waterman, A., and Patterson, D., "Roofline: An Insightful Visual Performance Model for Multicore Architectures," *Commun. ACM*, 2009.

[22] Pedregosa, F., Varoquaux, G., and others., "Scikit-learn: Machine Learning in Python," *J. Mach. Learn. Res.*, Nov. 2011.

[23] He, X., Liao, L., et al., "Neural Collaborative Filtering," *Proc. WWW*, 2017.

[24] Silver, D., Huang, A., Maddison, C., et al., "Mastering the game of Go with deep neural networks and tree search," *Nature*, 2016.

[25] Rucci, E. et al., "SWIFOLD: Smith-Waterman implementation on FPGA with OpenCL for long DNA sequences." *BMC Syst Biol*, 2018.

[26] Barnes, J. and Hut, P., "A hierarchical O(N log N) force-calculation algorithm," *Nature*, 1986.

[27] Che, S., Boyer, M., Meng, J., et al., "Rodinia: A benchmark suite for heterogeneous computing," *Proc. IISWC*, 2009.

[28] Facebook, "Introducing "Yosemite": the first open source modular chassis for high-powered microserver," 2015, https://engineering.fb.com/core-data/introducing-yosemite-the-first-open-source-modular-chassis-for-high-powered-microservers/.

[29] Facebook, “Rocksdb,” 2020, https://rocksdb.org/.

[30] Intel, “Intel® Optane™ Persistent Memory,” 2019, https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[31] Qureshi, M. K., Srinivasan, V., and Rivers, J. A., “Scalable High Performance Main Memory System Using Phase-Change Memory Technology,” *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA ’09, Association for Computing Machinery, New York, NY, USA, 2009, p. 24–33.

[32] Zhao, J., Li, S., Yoon, D. H., Xie, Y., and Jouppi, N. P., “Kiln: Closing the performance gap between systems with and without persistence support,” *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013, pp. 421–432.

[33] Jung, J.-Y. and Cho, S., “Memorage: Emerging Persistent RAM Based Malleable Main Memory and Storage Architecture,” *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS ’13, Association for Computing Machinery, New York, NY, USA, 2013, p. 115–126.

[34] Gottesman, Y., Nider, J., Kat, R., Weinsberg, Y., and Factor, M., “Using Storage Class Memory Efficiently for an In-Memory Database,” *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR ’16, Association for Computing Machinery, New York, NY, USA, 2016.

[35] Jeong, J., Hong, J., Maeng, S., Jung, C., and Kwon, Y., “Unbounded Hardware Transactional Memory for a Hybrid DRAM/NVM Memory System,” *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 525–538.

[36] Volos, H., Tack, A. J., and Swift, M. M., “Mnemosyne: Lightweight Persistent Memory,” *SIGARCH Comput. Archit. News*, Vol. 39, No. 1, March 2011, pp. 91–104.

[37] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S., “NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories,” *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, Association for Computing Machinery, New York, NY, USA, 2011, p. 105–118.

[38] Zhang, L. and Swanson, S., “Pangolin: A Fault-Tolerant Persistent Memory Programming Library,” *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, USENIX Association, Renton, WA, July 2019, pp. 897–912.

[39] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D., “Better I/O through Byte-Addressable, Persistent Memory,” *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP ’09, Association for Computing Machinery, New York, NY, USA, 2009, p. 133–146.

[40] Ou, J., Shu, J., and Lu, Y., “A High Performance File System for Non-Volatile Main Memory,” *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys ’16, Association for Computing Machinery, New York, NY, USA, 2016.

[41] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J., “System Software for Persistent Memory,” *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys ’14, Association for Computing Machinery, New York, NY, USA, 2014.

[42] Izraelevitz, J., Yang, J., Zhang, L., Kim, J., Liu, X., Memaripour, A., Soh, Y. J., Wang, Z., Xu, Y., Dulloor, S. R., Zhao, J., and Swanson, S., “Basic Performance Measurements of the Intel Optane DC Persistent Memory Module,” 2019.

[43] Zarubin, M., Damme, P., Habich, D., and Lehner, W., “Polymorphic Compressed Replication of Columnar Data in Scale-Up Hybrid Memory Systems,” *Proceedings of the 13th ACM International Systems and Storage Conference*, SYSTOR ’20, Association for Computing Machinery, New York, NY, USA, 2020, p. 98–110.

[44] Wu, K., Ober, F., Hamlin, S., and Li, D., “Early Evaluation of Intel Optane Non-Volatile Memory with HPC I/O Workloads,” 2017.

[45] Yang, J., Kim, J., Hoseinzadeh, M., Izraelevitz, J., and Swanson, S., “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory,” *18th USENIX Conference on File and Storage Technologies (FAST 20)*, USENIX Association, Santa Clara, CA, Feb. 2020, pp. 169–182.

[46] Gill, G., Dathathri, R., Hoang, L., Peri, R., and Pingali, K., “Single machine graph analytics on massive datasets using Intel optane DC persistent memory,” *Proceedings of the VLDB Endowment*, Vol. 13, No. 10, Jun 2020, pp. 1304–1318.

[47] Patil, O., Ionkov, L., Lee, J., Mueller, F., and Lang, M., “Performance Characterization of a DRAM-NVM Hybrid Memory Architecture for HPC Applications Using Intel Optane DC Persistent Memory Modules,” *Proceedings of the International Symposium on Memory Systems*, MEMSYS ’19, Association for Computing Machinery, New York, NY, USA, 2019, p. 288–303.

[48] Peng, I., Wu, K., Ren, J., Li, D., and Gokhale, M., “Demystifying the Performance of HPC Scientific Applications on NVM-based Memory Systems,” *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2020.

[49] Ma, T., Zhang, M., Chen, K., Song, Z., Wu, Y., and Qian, X., “AsymNVM: An Efficient Framework for Implementing Persistent Data Structures on Asymmetric NVM Architecture,” *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’20, Association for Computing Machinery, New York, NY, USA, 2020, p. 757–773.

[50] Anderson, T. E., Canini, M., Kim, J., Kostić, D., Kwon, Y., Peter, S., Reda, W., Schuh, H. N., and Witchel, E., “Assise: Performance and Availability via Client-local NVM

in a Distributed File System," *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 1011–1027.

[51] Wang, Z., Liu, X., Yang, J., Michailidis, T., Swanson, S., and Zhao, J., "Characterizing and Modeling Non-Volatile Memory Systems," *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 496–508.

[52] "Compute Express Link™: The Breakthrough CPU-to-Device Interconnect," 2020, https://www.computeexpresslink.org/.

[53] Knowlton, S., "Introduction to Compute Express Link (CXL): The CPU-To-Device Interconnect Breakthrough." 2019, https://www.computeexpresslink.org/post/introduction-to-compute-express-link-cxl-the-cpu-to-device-interconnect-breakthrough.

[54] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M., "Workload Analysis of a Large-Scale Key-Value Store," *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 53–64.

[55] Shi, X., Pruett, S., Doherty, K., Han, J., Petrov, D., Carrig, J., Hugg, J., and Bronson, N., "FlightTracker: Consistency across Read-Optimized Online Stores at Facebook," *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, Nov. 2020, pp. 407–423.

[56] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H., Marchukov, M., Petrov, D., Puzar, L., Song, Y. J., and Venkataramani, V., "TAO: Facebook's Distributed Data Store for the Social Graph," *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, USENIX Association, San Jose, CA, June 2013, pp. 49–60.

[57] Cao, Z., Dong, S., Vemuri, S., and Du, D. H., "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook," *18th USENIX Conference on File and Storage Technologies (FAST 20)*, USENIX Association, Santa Clara, CA, Feb. 2020, pp. 209–223.

[58] Facebook, "db_bench," 2020, https://github.com/facebook/rocksdb/wiki/Benchmarking-tools#db_bench.

[59] Google, "LevelDB," 2011, https://dbdb.io/db/leveldb.

[60] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: Amazon's Highly Available Key-Value Store," *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, Association for Computing Machinery, New York, NY, USA, 2007, p. 205–220.

[61] Redis, "Redis," 2020, https://redis.io/.

[62] O'Neil, P., Cheng, E., Gawlick, D., and O'Neil, E., "The Log-Structured Merge-Tree (LSM-Tree)," *Acta Inf.*, Vol. 33, No. 4, June 1996, pp. 351–385.

[63] Facebook, "RocksDB Users and Use Cases," 2020, https://github.com/facebook/rocksdb/wiki/RocksDB-Users-and-Use-Cases.

[64] Facebook, "RocksDB Trace, Replay, Analyzer, and Workload Generation," 2020, https://github.com/facebook/rocksdb/wiki/RocksDB-Trace%2C-Replay%2C-Analyzer%2C-and-Workload-Generation.

[65] Micron, "3D XPoint Technology," 2020, https://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[66] Intel, "3D XPoint™: A Breakthrough in Non-Volatile Memory Technology," 2020, https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[67] Intel, "Intel Optane DC Persistent Memory," 2020, https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf.

[68] "NDCTL user guide: Managing Namespaces," 2020, https://docs.pmem.io/ndctl-user-guide/managing-namespaces.

[69] IgorsIstocniks, "How WhatsApp Moved 1.5B Users Across Datacenters," 2019, https://docplayer.net/161220289-How-whatsapp-moved-1-5b-users-across-data-senters-igors-istocniks-code-beam-sf-2019.html.

[70] Annamalai, M., "ZippyDB: a modern, distributed keyvalue data store." 2015, https://www.youtube.com/watch?v=DfiN7pG0D0k.

[71] Pan, S., Stavrinos, T., Zhang, Y., Sikaria, A., Zakharov, P., Sharma, A., P, S. S., Shuey, M., Wareing, R., Gangapuram, M., Cao, G., Preseau, C., Singh, P., Patiejunas, K., Tipton, J., Katz-Bassett, E., and Lloyd, W., "Facebook's Tectonic Filesystem: Efficiency from Exascale," *19th USENIX Conference on File and Storage Technologies (FAST 21)*, USENIX Association, Feb. 2021, pp. 217–231.

[72] Muralidhar, S., Lloyd, W., Roy, S., Hill, C., Lin, E., Liu, W., Pan, S., Shankar, S., Sivakumar, V., Tang, L., and Kumar, S., "f4: Facebook's Warm BLOB Storage System," *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, USENIX Association, Broomfield, CO, Oct. 2014, pp. 383–398.

[73] Muralidhar, Lloyd, R. H. L. L. P. S. S. T. K., "f4: Facebook's Warm BLOB Storage System," USENIX, 2014.

[74] Chen, G. J., Wiener, J. L., Iyer, S., Jaiswal, A., Lei, R., Simha, N., Wang, W., Wilfong, K., Williamson, T., and Yilmaz, S., "Realtime data processing at Facebook," *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1087–1098.

[75] Thusoo, A., Shao, Z., Anthony, S., Borthakur, D., Jain, N., Sen Sarma, J., Murthy, R., and Liu, H., "Data warehousing and analytics infrastructure at facebook," *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010, pp. 1013–1020.

[76] Facebook, "Hive - A Petabyte Scale Data Warehouse using Hadoop," 2009, https://www.facebook.com/notes/facebook-engineering/hive-a-petabyte-scale-data-warehouse-using-hadoop/89508453919/.

[77] Kumar, S., "Social Networking at Scale." 2012, pp. 40–49, https://www.ece.lsu.edu/hpca-18/files/HPCA2012_Facebook_Keynote.pdf.

[78] Intel, "Intel Memory Latency Checker v3.9a," 2021, https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

[79] "fio," 2021, https://github.com/axboe/fio.

[80] Chow, C., "On optimization of storage hierarchies," *IBM Journal of Research and Development*, Vol. 18, No. 3, 1974, pp. 194–203.

[81] Intel, "IPMCTL," 2020, https://github.com/intel/ipmctl.

[82] "NDCTL and DAXCTL," 2020, https://github.com/pmem/ndctl.

[83] "memkind library," 2020, https://github.com/memkind/memkind.

[84] Intel, "Intel Server Board S2600WFTR Specification," 2019, https://ark.intel.com/content/www/us/en/ark/products/192581/intel-server-board-s2600wftr.html.

[85] Intel, "Intel Xeon Gold 6252 Processor Specification," 2019, https://ark.intel.com/content/www/us/en/ark/products/192447/intel-xeon-gold-6252-processor-35-75m-cache-2-10-ghz.html.

[86] OCP, "OCP Tioga Pass 2S Server Design Specification V1.1," 2018, https://www.opencompute.org/documents/open-compute-project-fb-2s-server-tioga-pass-v1p1-1-pdf.

[87] OCP, "OCP Twin Lakes 1S Server Design Specification V1," 2018, https://www.opencompute.org/documents/facebook-twin-lakes-1s-server-design-specification.

[88] Hyperscalers, "RACKGO X YOSEMITE VALLEY," 2019, https://www.hyperscalers.com/Rackgo-X-Yosemite-Valley.

[89] Hyperscalers, "RACKGO X LEOPARD CAVE," 2019, https://www.hyperscalers.com/OCP-Hyperscale-Systems?product_id=194.

[90] Handy, J., "Intel's Optane DIMM Price Model," 2019, https://thememoryguy.com/intels-optane-dimm-price-model/#more-2291.

[91] Alcorn, J., "Intel Optane DIMM Pricing," 2019, https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[92] MATLAB, "Fit Power Series Models Using the fit Function," 2020, https://www.mathworks.com/help/curvefit/power.html.

[93] MATLAB, "gpfit: Generalized Pareto parameter estimates," 2020, https://www.mathworks.com/help/stats/gpfit.htmll.

[94] MATLAB, "Fit sine Models Using the fit Function," 2020, https://www.mathworks.com/help/curvefit/sum-of-sine.html.

[95] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V., "Scaling Memcache at Facebook," *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, USENIX Association, Lombard, IL, April 2013, pp. 385–398.

[96] Fitzpatrick, B., "Distributed Caching with Memcached," *Linux Journal*, Vol. 30, No. 10, 2004, pp. 2223–2236.

[97] Peng, I. B., Gokhale, M. B., and Green, E. W., "System evaluation of the Intel optane byte-addressable NVM," *Proceedings of the International Symposium on Memory Systems*, Sep 2019.

[98] van Renen, A., Vogel, L., Leis, V., Neumann, T., and Kemper, A., "Persistent Memory I/O Primitives," *Proceedings of the 15th International Workshop on Data Management on New Hardware - DaMoN'19*, 2019.

[99] Psaropoulos, G., Oukid, I., Legler, T., May, N., and Ailamaki, A., "Bridging the Latency Gap between NVM and DRAM for Latency-Bound Operations," *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, Association for Computing Machinery, New York, NY, USA, 2019.

[100] Shanbhag, A., Tatbul, N., Cohen, D., and Madden, S., "Large-Scale in-Memory Analytics on Intel® Optane™ DC Persistent Memory," *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, Association for Computing Machinery, New York, NY, USA, 2020.

[101] Wu, Y., Park, K., Sen, R., Kroth, B., and Do, J., "Lessons learned from the early performance evaluation of Intel optane DC persistent memory in DBMS," *Proceedings of the 16th International Workshop on Data Management on New Hardware*, Jun 2020.

[102] Imamura, S. and Yoshida, E., "FairHym: Improving Inter-Process Fairness on Hybrid Memory Systems," *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, 2020, pp. 1–6.

[103] Xia, F., Jiang, D., Xiong, J., and Sun, N., "HiKV: A Hybrid Index Key-Value Store for DRAM-NVM Memory Systems," *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, USENIX Association, Santa Clara, CA, July 2017, pp. 349–362.

[104] Huang, Y., Pavlovic, M., Marathe, V., Seltzer, M., Harris, T., and Byan, S., "Closing the Performance Gap Between Volatile and Persistent Key-Value Stores Using Cross-Referencing Logs," *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, USENIX Association, Boston, MA, July 2018, pp. 967–979.

[105] Bailey, K. A., Hornyack, P., Ceze, L., Gribble, S. D., and Levy, H. M., "Exploring Storage Class Memory with Key Value Stores," *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, Association for Computing Machinery, New York, NY, USA, 2013.

[106] Oukid, I., Lasperas, J., Nica, A., Willhalm, T., and Lehner, W., "FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory," *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 371–386.

[107] Ramos, L. E., Gorbatov, E., and Bianchini, R., "Page Placement in Hybrid Memory Systems," *Proceedings of the International Conference on Supercomputing*, ICS '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 85–95.

[108] Zhang, Z., Fu, Y., and Hu, G., "DualStack: A High Efficient Dynamic Page Scheduling Scheme in Hybrid Main Memory," *2017 International Conference on Networking, Architecture, and Storage (NAS)*, 2017, pp. 1–6.

[109] Bock, S., Childers, B. R., Melhem, R., and Mossé, D., "Concurrent Migration of Multiple Pages in software-managed hybrid main memory," *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 420–423.

[110] Liu, H., Chen, Y., Liao, X., Jin, H., He, B., Zheng, L., and Guo, R., "Hardware/-Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures," *Proceedings of the International Conference on Supercomputing*, ICS '17, Association for Computing Machinery, New York, NY, USA, 2017.

[111] Liu, L., Yang, S., Peng, L., and Li, X., "Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 30, No. 10, 2019, pp. 2223–2236.

[112] Wu, K., Ren, J., and Li, D., "Runtime Data Management on Non-Volatile Memory-based Heterogeneous Memory for Task-Parallel Programs," *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 401–413.

[113] Hassan, A., Vandierendonck, H., and Nikolopoulos, D. S., "Software-Managed Energy-Efficient Hybrid DRAM/NVM Main Memory," *Proceedings of the 12th ACM International Conference on Computing Frontiers*, CF '15, Association for Computing Machinery, New York, NY, USA, 2015.

[114] Chang, H., Chang, Y., Kuo, T., and Li, H., "A light-weighted software-controlled cache for PCM-based main memory systems," *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 22–29.

[115] Li, Y., Ghose, S., Choi, J., Sun, J., Wang, H., and Mutlu, O., "Utility-Based Hybrid Memory Management," *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, 2017, pp. 152–165.

[116] Dhiman, G., Ayoub, R., and Rosing, T., "PDRAM: A hybrid PRAM and DRAM main memory system," *2009 46th ACM/IEEE Design Automation Conference*, 2009, pp. 664–669.

[117] Eisenman, A., Gardner, D., AbdelRahman, I., Axboe, J., Dong, S., Hazelwood, K., Petersen, C., Cidon, A., and Katti, S., "Reducing DRAM Footprint with NVM in Facebook," *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, Association for Computing Machinery, New York, NY, USA, 2018.

[118] Cheng, H.-T., Koc, L., Harmsen, J., Shaked, T., Chandra, T., Aradhye, H., Anderson, G., Corrado, G., Chai, W., Ispir, M., Anil, R., Haque, Z., Hong, L., Jain, V., Liu, X., and Shah, H., "Wide and Deep Learning for Recommender Systems," *arXiv:1606.07792*, 2016.

[119] Smith, B. and Linden, G., "Two Decades of Recommender Systems at Amazon.Com," *IEEE Internet Computing*, Vol. 21, No. 3, May 2017, pp. 12–18.

[120] Zhou, G., Mou, N., Fan, Y., Pi, Q., Bian, W., Zhou, C., Zhu, X., and Gai, K., "Deep Interest Evolution Network for Click-Through Rate Prediction," 2018.

[121] Elkahky, A. M., Song, Y., and He, X., "A Multi-View Deep Learning Approach for Cross Domain User Modeling in Recommendation Systems," *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 2015, p. 278–288.

[122] Naumov, M., Mudigere, D., Shi, H.-J. M., Huang, J., Sundaraman, N., Park, J., Wang, X., Gupta, U., Wu, C.-J., Azzolini, A. G., Dzhulgakov, D., Mallevich, A., Cherniavskii, I., Lu, Y., Krishnamoorthi, R., Yu, A., Kondratenko, V., Pereira, S., Chen, X., Chen, W., Rao, V., Jia, B., Xiong, L., and Smelyanskiy, M., "Deep Learning Recommendation Model for Personalization and Recommendation Systems," 2019.

[123] Mudigere, D., Hao, Y., Huang, J., Jia, Z., Tulloch, A., Sridharan, S., Liu, X., Ozdal, M., Nie, J., Park, J., Luo, L., Yang, J. A., Gao, L., Ivchenko, D., Basant, A., Hu, Y., Yang, J., Ardestani, E. K., Wang, X., Komuravelli, R., Chu, C.-H., Yilmaz, S., Li, H., Qian, J., Feng, Z., Ma, Y., Yang, J., Wen, E., Li, H., Yang, L., Sun, C., Zhao, W., Melts, D., Dhulipala, K., Kishore, K., Graf, T., Eisenman, A., Matam, K. K., Gangidi, A., Chen, G. J., Krishnan, M., Nayak, A., Nair, K., Muthiah, B., khorashadi, M., Bhattacharya, P., Lapukhov, P., Naumov, M., Mathews, A., Qiao, L., Smelyanskiy, M., Jia, B., and Rao, V., "Software-Hardware Co-design for Fast and Scalable Training of Deep Learning Recommendation Models," 2021.

[124] Ardestani, E. K., Kim, C., Lee, S. J., Pan, L., Rampersad, V., Axboe, J., Agrawal, B., Yu, F., Yu, A., Le, T., Yuen, H., Juluri, S., Nanda, A., Wodekar, M., Mudigere, D., Nair, K., Naumov, M., Peterson, C., Smelyanskiy, M., and Rao, V., "Supporting Massive DLRM Inference Through Software Defined Memory," 2021.

[125] NVIDIA, "NVIDIA DGX SYSTEMS Purpose-Built for the Unique Demands of AI," 2021, https://www.nvidia.com/en-us/data-center/dgx-systems/.

[126] Zhao, W., Xie, D., Jia, R., Qian, Y., Ding, R., Sun, M., and Li, P., "Distributed Hierarchical GPU Parameter Server for Massive Scale Deep Learning Ads Systems," 2020.

[127] Naumov, M., Kim, J., Mudigere, D., Sridharan, S., Wang, X., Zhao, W., Yilmaz, S., Kim, C., Yuen, H., Ozdal, M., et al., "Deep learning training in facebook data centers: Design of scale-up and scale-out systems," *arXiv preprint arXiv:2003.09518*, 2020.

[128] Jouppi, N. P., Yoon, D. H., Ashcraft, M., Gottscho, M., Jablin, T. B., Kurian, G., Laudon, J., Li, S., Ma, P., Ma, X., et al., "Ten lessons from three generations shaped google's tpuv4i: Industrial product," *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2021, pp. 1–14.

[129] Intel, "Intel® Optane™ SSD 905P Series for Demanding Storage Workloads," 2022, https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/enthusiast-ssds/optane-ssd-905p-brief.html.

[130] Wu, K., Arpaci-Dusseau, A., and Arpaci-Dusseau, R., "Towards an Unwritten Contract of Intel Optane SSD," *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, USENIX Association, Renton, WA, July 2019.

[131] Eisenman, A., Naumov, M., Gardner, D., Smelyanskiy, M., Pupyrev, S., Hazelwood, K., Cidon, A., and Katti, S., "Bandana: Using non-volatile memory for storing deep learning models," *Proceedings of Machine Learning and Systems*, Vol. 1, 2019, pp. 40–52.

[132] Wilkening, M., Gupta, U., Hsia, S., Trippel, C., Wu, C.-J., Brooks, D., and Wei, G.-Y., "RecSSD: Near Data Processing for Solid State Drive Based Recommendation

Inference," *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 717–729.

[133] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S., "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in Neural Information Processing Systems 32*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Curran Associates, Inc., 2019, pp. 8024–8035.

[134] Lydia, A. and Francis, S., "Adagrad—an optimizer for stochastic gradient descent," *Int. J. Inf. Comput. Sci*, Vol. 6, No. 5, 2019.

[135] Khudia, D., Huang, J., Basu, P., Deng, S., Liu, H., Park, J., and Smelyanskiy, M., "FBGEMM: Enabling High-Performance Low-Precision Deep Learning Inference," *arXiv preprint arXiv:2101.05615*, 2021.

[136] "FBGEMM_GPU," 2021, ://github.com/pytorch/FBGEMM/tree/main/fbgemm_gpu.

[137] Axboe, J., "FIO benchmark," 2013.

[138] Konstantinidis, E. and Cotronis, Y., "A Quantitative Performance Evaluation of Fast on-Chip Memories of GPUs," *2016 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, 2016, pp. 448–455.

[139] Facebook, "MultiGet Performance," 2021, https://github.com/facebook/rocksdb/wiki/MultiGet-Performance.

[140] NVIDIA, "NVIDIA Magnum IO GPUDirect Storage Overview Guide," 2021, https://docs.nvidia.com/gpudirect-storage/overview-guide/index.html.

[141] Yang, J. A., Huang, J., Park, J., Tang, P. T. P., and Tulloch, A., "Mixed-Precision Embedding Using a Cache," 2020.

[142] "EmbeddingBag," 2022, https://pytorch.org/docs/stable/generated/torch.nn.EmbeddingBag.html.

[143] "CUDA C++ Programming Guide," 2022, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[144] PyTorch, "torch.cuda.Stream," 2022, https://pytorch.org/docs/stable/generated/torch.cuda.Stream.html.

[145] PyTorch, "torch.cuda.synchronize," 2022, https://pytorch.org/docs/stable/generated/torch.cuda.synchronize.html?highlight=torch%20cuda%20synchronize#torch.cuda.synchronize.

[146] Ishii, A. and Foley, D., "Switching Fabrics and FPGA Architectures," *Hot Chips: A Symposium on High Performance Chips (HC30, 2018)*, 2018, https://old.hotchips.org/hc30/2conf/2.01_Nvidia_NVswitch_HotChips2018_DGX2NVS_Final.pdf.

[147] Balasubramanian, K., Alshabanah, A., Choe, J. D., and Annavaram, M., "cDLRM: Look Ahead Caching for Scalable Training of Recommendation Models," *Fifteenth ACM Conference on Recommender Systems*, 2021, pp. 263–272.

[148] Hildebrand, M., Khan, J., Trika, S., Lowe-Power, J., and Akella, V., "AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems Using Integer Linear Programming," *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 875–890.

[149] Nvidia, "Tegra X1: The Powerful Processor Behind SHIELD," 2017, https://blogs.nvidia.com/blog/2017/07/20/shield-tegra-x1-processor/.

[150] AMD, "AMD accelerator proccessing unit (APU)," 2022, https://www.amd.com/en/products/embedded-r-series-2nd-gen-apu.

[151] Intel, "Iris® Xe Graphics," 2022, https://www.intel.com/content/www/us/en/architecture-and-technology/visual-technology/graphics-overview.html.

[152] Intel, "Intel® Agilex™ FPGA and SoC," 2022, https://www.intel.com/content/www/us/en/products/details/fpga/agilex.html.

[153] Chung, E., Fowers, J., Ovtcharov, K., Papamichael, M., Caulfield, A., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Abeydeera, M., Adams, L., Angepat, H., Boehn, C., Chiou, D., Firestein, O., Forin, A., Gatlin, K. S., Ghandi, M., Heil, S., Holohan, K., El Husseini, A., Juhasz, T., Kagi, K., Kovvuri, R. K., Lanka, S., van Megen, F., Mukhortov, D., Patel, P., Perez, B., Rapsang, A., Reinhardt, S., Rouhani, B., Sapek, A., Seera, R., Shekar, S., Sridharan, B., Weisz, G., Woods, L., Yi Xiao, P., Zhang, D., Zhao, R., and Burger, D., "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave," *IEEE Micro*, Vol. 38, No. 2, 2018, pp. 8–20.

[154] Kaleem, R., Barik, R., Shpeisman, T., Lewis, B. T., Hu, C., and Pingali, K., "Adaptive Heterogeneous Scheduling for Integrated GPUs," *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, Association for Computing Machinery, New York, NY, USA, 2014, p. 151–162.

[155] Zhang, F., Wu, B., Zhai, J., He, B., and Chen, W., "FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures," *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2017, pp. 27–38.

[156] Zhang, F., Zhai, J., He, B., Zhang, S., and Chen, W., "Understanding Co-Running Behaviors on Integrated CPU/GPU Architectures," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 28, No. 3, 2017, pp. 905–918.

[157] Pérez, B., Bosque, J. L., and Beivide, R., "Simplifying Programming and Load Balancing of Data Parallel Applications on Heterogeneous Systems," *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, GPGPU '16, Association for Computing Machinery, New York, NY, USA, 2016, p. 42–51.

[158] Alsop, J., Sinclair, M. D., and Adve, S. V., "Spandex: A Flexible Interface for Efficient Heterogeneous Coherence," *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, IEEE Press, 2018, p. 261–274.

[159] Garcıa, V., Gomez-Luna, J., Grass, T., Rico, A., Ayguade, E., and Pena, A. J., "Evaluating the effect of last-level cache sharing on integrated GPU-CPU systems with heterogeneous applications," *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[160] Ausavarungnirun, R., Chang, K. K.-W., Subramanian, L., Loh, G. H., and Mutlu, O., "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012, pp. 416–427.

[161] Fang, J., Liu, S., and Zhang, X., "Research on Cache Partitioning and Adaptive Replacement Policy for CPU-GPU Heterogeneous Processors," *2017 16th International Symposium on Distributed Computing and Applications to Business, Engineering and Science (DCABES)*, 2017, pp. 19–22.

[162] "Heterogeneous System Architecture Foundation," 2022, `http://hsafoundation.com/`.

[163] "AMD Fusion: How It Started, Where It's Going, And What It Means," 2022, `https://www.tomshardware.com/reviews/fusion-hsa-opencl-history,3262-8.html`.

[164] "The OpenCL Specification," 2022, `https://www.khronos.org/registry/OpenCL/`.

[165] Gómez-Luna, J., Hajj, I. E., Chang, L.-W., García-Floreszx, V., de Gonzalo, S. G., Jablin, T. B., Peña, A. J., and Hwu, W.-m., "Chai: Collaborative heterogeneous applications for integrated-architectures," *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017, pp. 43–54.

[166] Power, J., Hestness, J., Orr, M. S., Hill, M. D., and Wood, D. A., "gem5-gpu: A Heterogeneous CPU-GPU Simulator," *IEEE Computer Architecture Letters*, Vol. 14, No. 1, 2015, pp. 34–36.

[167] Power, J., Hestness, J., Orr, M. S., Hill, M. D., and Wood, D. A., "gem5-gpu: A Heterogeneous CPU-GPU Simulator," *IEEE Computer Architecture Letters*, Vol. 14, No. 1, 2015, pp. 34–36.

[168] gpgpu sim, 2022, `https://github.com/gpgpu-sim/gpgpu-sim_distribution`.

[169] Shivakumar, P. and Jouppi, N. P., "Cacti 3.0: An integrated cache timing, power, and area model," 2001.