Doctoral Thesis

Heterogeneity-Aware Resource Management Techniques for Data-Intensive Applications

Myeonggyun Han

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

2024

Heterogeneity-Aware Resource Management Techniques for Data-Intensive Applications

Myeonggyun Han

Department of Computer Science and Engineering

Ulsan National Institute of Science and Technology

Heterogeneity-Aware Resource Management Techniques for Data-Intensive Applications

A dissertation submitted to
Ulsan National Institute of Science and Technology
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Myeonggyun Han

11.28.2023 of submission
Approved by

Advisor
Woongki Baek

Heterogeneity-Aware Resource Management Techniques for Data-Intensive Applications

Myeonggyun Han

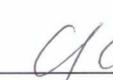
This certifies that the dissertation of Myeonggyun Han is approved.

11.28.2023 of submission

Signature

Advisor: Woongki Baek

Signature

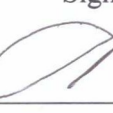


Young-ri Choi

Signature

Myeongjae Jeon

Signature



Seulki Lee

Signature

Jongeun Lee

Abstract

A wide range of applications have become data-intensive as they operate on the massive amounts of data generated by social network services, multimedia devices, and Internet of Things sensors. These data-intensive applications typically require enormous computational and memory resources to extract useful information from the massive amounts of data they encounter. To accommodate the enormous computing and memory demands of data-intensive applications, hardware resources in computing systems are becoming highly heterogeneous. Specifically, numerous hardware accelerators, such as tensor processing units (TPUs) and neural processing units (NPUs), have been developed to address the ever-increasing computing demands of deep-learning applications. In addition, new memory devices, such as high-bandwidth memory (HBM) and non-volatile memory (NVM), have been developed to tackle the growing demand for increased memory performance, capacity, and cost-efficiency.

Heterogeneous computing and memory have great potential to significantly improve the performance and efficiency of data-intensive applications. However, taking full advantage of the capabilities of heterogeneous computing and memory poses significant challenges to system software in that it is the responsibility of the underlying system software to manage the heterogeneous computing and memory resources effectively so as to maximize the metric of interest, such as the performance or energy efficiency. This dissertation presents heterogeneity-aware resource management techniques that significantly improve the performance and efficiency of data-intensive applications by effectively exploiting heterogeneous computing and memory resources.

First, we investigate system software techniques that effectively schedule computations on heterogeneous computing devices for efficient deep-learning inference. To this end, we propose MOSAIC, a software-based system for heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference on heterogeneous embedded systems. MOSAIC employs accurate models for estimating the execution and communication costs of the target inference workload. MOSAIC generates an efficient model slicing and execution plan for the target workload using an algorithm based on dynamic programming.

Second, we propose HERTI, a reinforcement learning-augmented system for efficient real-time inference on heterogeneous embedded systems. HERTI efficiently explores the state space and robustly finds an efficient state that significantly improves the efficiency of the target inference workload while satisfying the corresponding deadline constraint through reinforcement learning. In addition, HERTI significantly accelerates the training process based on the accurate and lightweight cost estimators.

Third, we investigate a system software technique that effectively manages heterogeneous memory for high-performance deep-learning. We analyze the characteristics of representative deep-learning workloads on a real heterogeneous memory system. Guided by the characterization results, we propose HALO, hotness- and lifetime-aware data placement and migration for high-performance deep-learning on heterogeneous memory systems. HALO extracts the hotness and lifetime information on the tensors

of the target deep-learning application based on its dataflow graph. HALO then dynamically places and migrates the tensors on heterogeneous memory nodes based on their hotness and lifetime characteristics.

Finally, we investigate a system software technique for QoS-aware and efficient workload consolidation on heterogeneous memory systems based on software-defined far memory. We conduct an in-depth characterization of the impact of cores, memory, and compressed memory swap (CMS) on the QoS and throughput of consolidated latency-critical (LC) and batch applications. Guided by the characterization results, we propose COSMOS, a software-based runtime system for the coordinated management of cores, memory, and CMS for QoS-aware and efficient workload consolidation for memory-intensive applications. COSMOS dynamically collects runtime data from consolidated applications and the underlying system and allocates the resources to the consolidated applications in a way that achieves high throughput with strong QoS guarantees.

Contents

List of Figures

List of Tables

I Introduction

An enormous volume of data is being generated through the widespread use of social network services, multimedia devices, and Internet of Things sensors [101]. Various applications, including machine learning, graph analytics, and databases, are becoming increasingly data-intensive as they operate on extensive datasets to enable innovative services. Data-intensive applications typically require large computational and memory resources to derive valuable insights and knowledge from extensive datasets and to make predictions and informed decisions. Given that data-intensive applications are widely used across various computing domains ranging from embedded systems to high-performance computing, it is crucial to optimize the performance and efficiency of these applications.

Hardware resources in computing systems are becoming highly heterogeneous due to the emergence of new hardware accelerators and memory technologies. On the computing device side, numerous hardware accelerators, such as Google's tensor processing units (TPUs) [76, 77], Meta's MTIA [53], Amazon's Inferentia [1], and various neural processing units (NPUs) [5, 13, 14, 20], have been developed to address the ever-growing computing demands in deep-learning applications. Along with these hardware accelerators, heterogeneous computing systems that employ multiple types of computing devices (e.g., CPU, GPU, and NPU) have emerged as promising solutions for efficient machine learning. For example, the Huawei Kirin 9000 [14] comprises eight CPU cores of two different types (i.e., four Cortex-A77 and four Cortex-A55 cores), a 24-core GPU, and a 3-core NPU of two different types (i.e., two Ascend Lite and one Ascend Tiny cores). In addition, the Apple A17 Pro [13] contains six CPU cores of two different types (i.e., two performance and four efficiency cores), a 6-core GPU, and a 16-core NPU.

On the memory device side, new memory devices, such as high-bandwidth memory (HBM) [6] and non-volatile memory (NVM) [9], have been developed to address the growing demand for improved memory speeds and capacities. With these new memory devices, heterogeneous memory systems, which consist of various memory nodes with a wide range of architectural characteristics, are rapidly emerging as a promising solution to increase memory performance and capacity in a cost-effective manner. The key concept behind heterogeneous memory systems is to offer useful properties, such as performance, energy efficiency, durability, and cost efficiency, that cannot be obtained with a single memory type. For instance, the recently-released Intel Xeon CPUs (i.e., Intel Xeon Max Series) [11], which comprise HBM and DRAM nodes, can effectively provide high memory bandwidths (e.g., 1638.4 GB/s with HBM) as well as large memory capacities (e.g., 4 TB with DRAM).

Heterogeneous computing and memory have great potential to significantly improve the performance and efficiency of data-intensive applications. However, taking full advantage of the capabilities of heterogeneous computing and memory poses several challenges in terms of resource management. First, it is necessary to schedule computations on heterogeneous computing devices according to the characteristics of the workload and the computing devices. Second, it is necessary to place and migrate data between various memory nodes according to the memory access patterns. Third, it is necessary to allocate an appropriate amount of heterogeneous resources to each application when sharing resources between multiple applications.

In this dissertation, we investigate system software techniques that effectively manage heterogeneous computing and memory resources to address the aforementioned challenges. To address the first challenge, we investigate model slicing and execution techniques [58, 61]. Model slicing and execution techniques divide the deep-learning model into multiple slices and schedule them on heterogeneous computing devices to improve the performance and energy efficiency of deep-learning inference [61] or to improve the energy efficiency while satisfying a deadline constraint [58]. To address the second challenge, we investigate hotness- and lifetime-aware data placement and migration techniques to improve the performance of deep-learning application on heterogeneous memory systems [60]. Finally, to address the third challenge, we investigate a resource allocation technique for QoS-aware and efficient workload consolidation on heterogeneous memory systems based on software-defined far memory [62].

1.1 Contributions

The focus of this dissertation is to improve the performance and efficiency of data-intensive applications by fully utilizing heterogeneous computing and memory resources. To this end, we designed and implemented system software techniques that effectively manage hardware resources in a heterogeneity-aware manner and evaluated them.

Specifically, this dissertation makes the following contributions:

- **Model Slicing and Execution for Efficient Inference on Heterogeneous Embedded Systems**
  We propose MOSAIC, a software-based system for heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference on heterogeneous embedded systems [61]. MOSAIC utilizes accurate models to estimate the execution and communication costs of the target inference workload. MOSAIC generates the efficient model slicing and execution plan for the target inference workload through dynamic programming. We quantify the effectiveness of MOSAIC with widely-used inference workloads on a real heterogeneous embedded system, which consists of big core cluster, little core cluster, GPU, and NPU. Our experimental results demonstrate the effectiveness of MOSAIC as it significantly reduces inference latency and energy, exhibits high estimation accuracy, and incurs small overheads.

- **RL-Based System for Efficient Real-Time Inference on Heterogeneous Embedded Systems**
  We propose HERTI, a reinforcement learning (RL)-augmented system for efficient real-time inference on heterogeneous embedded systems [58]. HERTI efficiently explores the state space and robustly finds an efficient state that significantly improves the efficiency of the target real-time inference workload while satisfying the corresponding deadline constraint through reinforcement learning. Our quantitative evaluation conducted on a full heterogeneous embedded system demonstrates the effectiveness of HERTI in that HERTI achieves high inference efficiency in multiple metrics (i.e., energy and energy-delay product) with a strong deadline guarantee, delivers larger gains as the inference deadline and the system heterogeneity increase, provides strong generality for hyper-parameter tuning, and significantly reduces the training time through its estimation-based approach across all the evaluated inference workloads and scenarios.

- **Memory Management for High-Performance DL on Heterogeneous Memory Systems**
  We propose HALO, hotness- and lifetime-aware data placement and migration for high-performance deep-learning (DL) on heterogeneous memory systems [60]. We conduct an in-depth characterization of widely-used deep-learning workloads on a real heterogeneous memory system and design and implement HALO guided by the characterization results. HALO analyzes the hotness and lifetime characteristics of tensors based on a dataflow graph of the target deep-learning application. HALO dynamically places and migrates the tensors across the heterogeneous memory nodes in a hotness- and lifetime-aware manner based on the proposed algorithm. Through quantitative evaluation, we demonstrate the effectiveness of HALO in that it significantly outperforms various memory management policies supported by the underlying system software and hardware, achieves performance comparable to the ideal case with infinite HBM, incurs small performance overheads, and delivers high performance across a wide range of application working-set sizes.

- **Resource Allocation for QoS-Aware and Efficient Workload Consolidation on Heterogeneous Memory Systems Based on Software-Defined Far Memory**
  We propose COSMOS, a software-based system for coordinated management of cores, memory, and compressed memory swap (CMS) for QoS-aware and efficient workload consolidation for memory-intensive applications [62]. CMS is used as a software-defined far memory in this work. We conduct an in-depth characterization of the impact of cores, memory, and CMS on the QoS and throughput of the consolidated latency-critical (LC) and batch applications and design and implement COSMOS guided by the characterization results. COSMOS dynamically collects the runtime data from the consolidated applications and the underlying system and allocates cores, memory, and CMS in a way that significantly improves the throughput of the consolidated applications while satisfying the LC application's QoS. Our quantitative evaluation based on a real system and widely-used memory-intensive benchmarks demonstrates that COSMOS provides strong QoS guarantees and achieves high throughput across all the workload mixes with various loads for the LC application and memory overcommit ratios.

1.2 Organization

The rest of this dissertation is organized as follows. Chapter II provides background information related to the dissertation. Chapter III summarizes previous work. Chapter IV presents MOSAIC, a software-based system for heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference on heterogeneous embedded systems. Chapter V describes HERTI, an RL-augmented system for efficient real-time inference on heterogeneous embedded systems. Chapter VI presents HALO, hotness- and lifetime-aware data placement and migration for high-performance deep-learning on heterogeneous memory systems. Chapter VII describes COSMOS, a software-based system for coordinated management of cores, memory, and CMS for QoS-aware and efficient workload consolidation for memory-intensive applications. Finally, Chapter VIII concludes this dissertation.

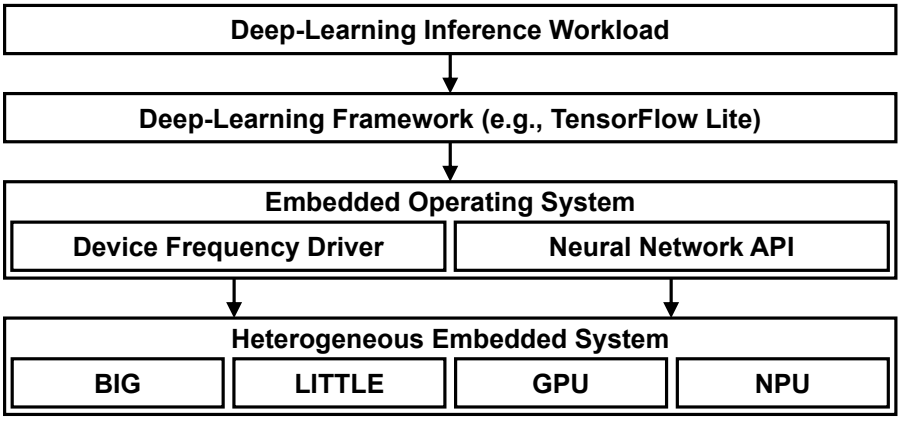


Figure 1: Hardware and software stacks for deep-learning inference on heterogeneous embedded systems.

II Background

2.1 Heterogeneous Embedded Systems and Inference

Heterogeneous embedded systems consist of various computing devices (e.g., big core cluster, little core cluster, GPU, and NPU). Each heterogeneous computing device exhibits widely-different characteristics in terms of performance, power efficiency, communication overheads, functionality, and memory capacity [4, 5, 13, 15, 20].

Widely used deep-learning (DL) frameworks (e.g., TensorFlow Lite [23], PyTorch [122]) simplify inference workload programming on heterogeneous embedded systems. With such frameworks, parts of inference workloads are offloaded to computing devices through the invocations of API functions provided by the frameworks, eliminating the need to implement device-specific code. Figure 1 illustrates the hardware and software stacks for DL inference on heterogeneous embedded systems.

Inference workloads consist of *layers*. A layer is defined as a set of associated mathematical operations such as convolution and rectifier. Each layer takes a set of input tensors (i.e., multidimensional arrays), performs computations specified by its mathematical operations, and generates a set of output tensors.

In this dissertation, we define a *model slice* as a set of consecutive layers, to be executed on the same computing device of the underlying heterogeneous embedded system. There exist communication overheads between adjacent slices as they need to communicate through input and output tensors.

Due to memory or functionality constraints, it may be infeasible to execute a model slice on certain heterogeneous computing devices. For example, if a model slice includes mathematical operations unsupported by a computing device or has a memory footprint larger than the memory capacity of a computing device, the model slice cannot be executed on the device.

While the importance of efficient inference is ever increasing, it is highly challenging to maximize the efficiency of inference workloads on heterogeneous embedded systems. As we will show in Section 4.3, each layer exhibits greatly differing performance and power consumption characteristics on

each heterogeneous computing device and different model slicing plans incur different execution and communication costs. In addition, each heterogeneous computing device imposes different constraints. Therefore, the design complexity and the state space that an optimizing system needs to cover increase drastically.

2.2 Heterogeneous Memory Systems

Heterogeneous memory systems comprise two or more types of memory nodes (e.g., non-volatile memory (NVM) and DRAM [9, 52] high-bandwidth memory (HBM) and DRAM [11, 144]) that exhibit a broad range of different characteristics, including bandwidth, latency, persistence, and durability.

Heterogeneous memory systems can be largely classified into two categories. The heterogeneous memory systems that belong to the first category organize heterogeneous memory nodes in a *flat* manner in that the heterogeneous memory nodes are mapped to a single physical address space [28, 82, 136, 157, 161, 162, 166]. The heterogeneous memory systems in the second category organize heterogeneous memory nodes in a *tiered* manner in that faster and smaller memory nodes are used as hardware caches for slower and larger memory nodes [46, 56, 165, 167]. In Chapter VI, we investigate the system software technique (i.e., HALO) that significantly improves the performance of the target deep-learning (DL) application on heterogeneous memory systems with the flat memory organization by effectively utilizing the application-level information (e.g., hotness, lifetime).

One of the most representative heterogeneous memory systems is the Intel Xeon CPU-based heterogeneous memory systems, such as the Intel Xeon CPU Max Series [11], the Intel Xeon CPU with Optane Memory [9], and the Intel Xeon Phi KNL [143]. A prior work has characterized the performance of the HBM and LBM nodes[1] using a Xeon Phi KNL CPU-based system [129]. Because HBM employs a high-performance multi-channel interface between the HBM nodes and the CPU, it achieves significantly higher bandwidth (i.e., 480 GB/s vs. 80 GB/s) than LBM. In contrast, HBM nodes incur slightly longer latency (i.e., 160-175 nanoseconds vs. 130-140 nanoseconds) than LBM nodes [129].

Further, the capacity of HBM is significantly smaller (i.e., 16 GB vs. 192 GB) compared to that of LBM as the area budget of HBM is tightly limited to employ the high-bandwidth package-on-package technology. Given that data-intensive applications typically require a large amount of physical memory, their working-set size often significantly exceeds the capacity of HBM.

The Xeon Phi KNL architecture provides the `Flat` mode (i.e., flat memory organization) in which the HBM and LBM nodes are mapped to a single physical address space. In addition, the Xeon Phi KNL architecture provides the `Cache` mode (i.e., tiered memory organization). In the `Cache` mode, HBM is fully managed by hardware as a last-level hardware cache and is transparent to the OS. Note that other Xeon CPU-based heterogeneous memory systems, such as the Intel Xeon CPU Max Series [11] and Intel Xeon CPU with Optane Memory [9], also provide the `Flat` mode and `Cache` mode.

Another representative type of heterogeneous memory system is a system that employs software-defined far memory. Production cloud computing systems and datacenters (e.g., Google [93] and

[1] We denote multi-channel DRAM (MCDRAM) and DRAM as high-bandwidth memory (HBM) and low-bandwidth memory (LBM), respectively.

Meta [160]) employ the compressed memory swap (CMS) as a software-defined far memory to effectively increase the memory capacity of the underlying server system in a cost-effective manner. CMS compresses cold pages and stores them in the in-memory swap area to effectively create a second-tier slow memory in software.

CMS is a promising technique for hosting memory-intensive applications without increasing the physical memory of the underlying server system [93, 160]. With CMS, pages selected as victim pages by the memory reclaim algorithm in the OS are compressed and evicted to CMS instead of the disk swap. CMS mitigates the page swapping overhead by storing the victim pages in the in-memory swap area instead of the disk. While CMS incurs overheads when compressing and decompressing pages, it is still significantly faster than a disk swap as it eliminates expensive disk I/O operations. Widely used OSes (e.g., zswap in Linux [25], memory compression in Windows [18], and compressed memory in macOS [19]) support CMS.

III Related Work

3.1 Model Slicing and Execution for Efficient Deep Learning Inference

Prior works have extensively investigated deep-learning model analysis and optimization techniques to improve the inference latency and/or address the constraints of inference workloads [29, 35, 43, 68, 74, 78, 84, 114, 128, 132, 146, 158]. While insightful, none of the prior works considers the efficiency heterogeneity, communication overheads, and constraints of inference workloads and emerging computing devices in an integrated manner.

The prior works proposed in [29, 35, 68, 74, 78, 114, 128, 146] lack the consideration of the efficiency heterogeneity and memory and functionality constraints of inference workloads and emerging computing devices (e.g., NPU), which are crucial factors to achieve the best possible efficiency on heterogeneous embedded systems. The works proposed in [43, 132, 158] lack the consideration of the efficiency heterogeneity, communication overheads, and functionality constraints of computing devices and deep-learning workloads. Our works (i.e., MOSAIC and HERTI) significantly differ in the sense that we analyze the characteristics of inference workloads on a real heterogeneous embedded system that includes a highly-optimized NPU, propose efficient algorithms to solve the model slicing and execution problem in a heterogeneity-, communication-, and constraint-aware manner, and design, implement, and evaluate the proposed systems using full hardware and software stacks.

The systems (e.g., AutoScale) proposed in [35, 84] consider the real-time property of inference workloads and employ DVFS to enhance energy efficiency. However, they lack the consideration of the heterogeneity- and constraint-aware model slicing and execution for real-time inference workloads and emerging computing devices (e.g., NPU), which are crucial for achieving the best possible efficiency on heterogeneous embedded systems. For instance, as we will show in Chapter V, the heterogeneity-oblivious inference workload execution employed by the aforementioned systems significantly degrades efficiency. Our work (i.e., HERTI) significantly differs as it proposes a reinforcement learning (RL)-augmented system that robustly addresses the model slicing and execution planning problem for real-time inference workloads in a heterogeneity- and constraint-aware manner and quantifies the effectiveness of HERTI on a real heterogeneous embedded system with a highly-optimized NPU.

Prior works have proposed techniques to apply machine learning to optimize the efficiency of computer systems in various contexts such as cloud resource management [39, 49, 83, 85, 112, 113], memory system design and management [51, 140], and compiler analysis and optimizations [106, 107]. Their common theme is that machine learning can effectively be used to address problems whose optimal solutions are unknown or exact solutions are computationally too expensive due to their high complexity. In line with the prior works, our work (i.e., HERTI) demonstrates that RL can effectively be used to solve the model slicing and execution planning problem for real-time inference workloads in order to maximize their efficiency while meeting their deadlines.

Prior works have explored the design and implementation of hardware accelerators for deep learning [30, 31, 42, 44, 50, 53, 53, 57, 66, 70, 76, 116, 145, 148, 159]. As hardware accelerators for deep learning

become more widely-used and diverse, we believe that systems like MOSAIC and HERTI will become even more crucial to effectively utilize all the heterogeneous computing devices in the underlying system with heterogeneity, communication, and constraint awareness for efficient inference.

Prior works have investigated the architectural and system software techniques to improve the efficiency of heterogeneous computing systems [63, 64, 91, 110, 115, 117, 118, 124, 125, 141, 155, 168]. Our works (i.e., MOSAIC and HERTI) differ as we investigate the characteristics of various inference workloads on heterogeneous computing devices including an NPU and present systems that efficiently execute inference workloads on heterogeneous embedded systems.

3.2 Data Placement and Migration for High-Performance Deep Learning

Prior works have extensively investigated system software techniques to improve the performance and scalability of the parallel and distributed deep-learning (DL) systems [26,43,45,72,75,92,108,132,158, 171]. While impactful, most prior works propose optimization techniques on homogeneous memory systems [26, 45, 72, 75, 108, 171]. Our work (i.e., HALO) significantly differs in that it investigates the hotness- and lifetime-aware tensor placement and migration techniques on heterogeneous memory systems, where data can be directly placed, accessed, and migrated across the heterogeneous memory nodes in a fully parallel manner.

Some of the prior works (e.g., vDNN) are closely related to ours [43, 92, 132, 158]. Based on the memory usage pattern of each layer in the target application, the techniques presented in [43,92,132,158] dynamically transfer the data between the CPU and GPU memory [43,132,158] or the memory modules attached to the device-side interconnection network (e.g., NVIDIA NVLink) and GPU memory [92] to ensure that the GPU memory usage of the target application does not exceed the capacity of the GPU memory during the entire execution time of the target application. While insightful, those works lack the consideration of the hotness of tensors because the data in the CPU host memory [43, 132, 158] or the memory modules within the device-side interconnect [92] cannot be directly accessed by the computation units in the GPU. Our work significantly differs in the sense that HALO dynamically places and migrates the tensors based on their hotness and lifetime characteristics to significantly improve the performance of the target DL applications on heterogeneous memory systems on which the computation units can directly and simultaneously access all the heterogeneous memory nodes.

Prior works have investigated the design and implementation of hardware accelerators for high-performance and low-power DL [44, 53, 66, 145]. While effective, the prior works primarily focus on the design and implementation of computation units that are customized for efficiently executing the mathematical operations (e.g., convolution) that are frequently employed by DL applications and lack the consideration of the performance issues on heterogeneous memory systems. We believe that heterogeneity-aware memory management techniques such as HALO will become more crucial with the widespread use of the hardware accelerators as the data transfer time is expected to account for a larger portion of the total execution time of DL applications as the hardware accelerators effectively reduce the time spent for computations.

Prior works have extensively explored architectural [28, 46, 56, 82, 157, 165, 167] and system software [103, 130, 136, 161, 162, 166] techniques for heterogeneous memory systems. Some of the prior works [28, 82, 136, 157, 161, 162, 166] investigate the placement [28, 82, 161, 166] and/or migration [103, 130, 136, 157, 162, 166] policies for heterogeneous memory systems that employ the flat memory organization. The other prior works [46, 56, 165, 167] explore the cache management (e.g., placement, replacement, bypassing) policies [56, 165] and the techniques to reduce the traffic between memory nodes [46, 167] for heterogeneous memory systems with the tiered memory organization.

While insightful, the prior works lack the consideration of application-level information, which is readily available from the dataflow graph of the target DL application. In contrast, HALO robustly extracts the application-level information (e.g., hotness, lifetime) from the target DL application and effectively places and migrates tensors across the heterogeneous memory nodes based on the application-level information.

3.3 Resource Management for QoS-Aware and Efficient Workload Consolidation

Prior works have extensively investigated resource management techniques for workload consolidation [40, 59, 65, 99, 111, 113, 119, 121, 163, 172]. Most of the prior works focus on partitioning resources such as cores and memory bandwidth and lack the consideration of dynamically partitioning the memory capacity between the consolidated applications [59, 65, 99, 111, 113, 119, 121, 163, 172]. Our work (i.e., COSMOS) differs in that it investigates system software support for coordinated management of cores, memory, and compressed memory swap (CMS) for QoS-aware and efficient workload consolidation for memory-intensive applications.

The technique proposed in [40] considers memory capacity partitioning between the consolidated applications. While insightful, it lacks the consideration of dynamic allocation of the CMS, which is crucial to improve the throughput of the consolidated applications with strong QoS guarantees for the latency-critical application. Our work (i.e., COSMOS) is significantly different in the sense that it presents the in-depth characterization of the impact of cores, memory, and CMS on the QoS and throughput of the consolidated applications and design, implement, and evaluate a software-based runtime system for QoS-aware and efficient workload consolidation for memory-intensive applications based on a real system.

Prior works have explored memory offloading techniques based on the CMS [93, 160]. While the prior works have a similarity to our work in that they show that the CMS can effectively be used to improve the resource efficiency of cloud computing systems and datacenters, they lack the consideration of dynamic management of the CMS in the context of workload consolidation. Our work (i.e., COSMOS) significantly differs in the sense that it investigates coordinated resource management of cores, memory, and CMS for QoS-aware and efficient workload consolidation.

Prior works have studied the feasibility of the CMS for embedded applications [69, 81, 95, 147]. The prior works show that the CMS can be used to reduce the response time and improve the interactivity of applications on embedded systems with limited memory capacity. However, they lack the capability of

holistically allocating the resources to the consolidated applications. Our work is considerably different in that COSMOS dynamically allocates cores, memory, and CMS to the consolidated applications in a coordinated manner for QoS-aware and efficient workload consolidation.

IV Heterogeneity-, Communication-, and Constraint-Aware Model Slicing and Execution for Accurate and Efficient Inference

4.1 Introduction

The need for accurate and efficient deep-learning inference on mobile systems continues to increase to enable intelligent and interactive services such as augmented reality and personal mobility. For a wide range of mobile applications such as security- and privacy-sensitive applications, it is crucial to execute inference workloads within mobile systems without relying on cloud services, which may leak sensitive information through various security attacks.

Heterogeneous embedded systems are rapidly emerging as a promising solution to enable accurate and efficient inference on mobile systems [4, 5, 13, 15, 20]. Heterogeneous embedded systems consist of various computing devices (e.g., big core cluster, little core cluster, GPU, and neural processing unit (NPU)), which exhibit widely-different characteristics in terms of performance, energy consumption, functionality (e.g., supported operations), memory capacity, and communication overheads.

Inference workloads also exhibit widely-different characteristics in terms of heterogeneity in performance, energy efficiency, communication overheads, and constraints across computing devices on heterogeneous embedded systems. Despite extensive prior works, it still remains unexplored to investigate the system-software support that efficiently executes inference workloads on heterogeneous embedded systems by judiciously considering their characteristics.

To bridge this gap, this chapter[2] proposes MOSAIC, heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference on heterogeneous embedded systems. MOSAIC builds on the accurate models for estimating the execution and communication costs, generates the efficient model slicing and execution plan with low time complexity, and executes the target inference workload to significantly improve its efficiency based on the user-specified metric such as latency and energy.

Specifically, this work makes the following contributions:

- We propose MOSAIC, a software-based system for heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference on heterogeneous embedded systems. MOSAIC employs the accurate models for estimating the execution and communication costs of the target inference workload. MOSAIC generates the efficient model slicing and execution plan for the target workload using an algorithm based on dynamic programming.

- We design and implement the prototype of MOSAIC as a user-level runtime system using the TensorFlow Lite programming framework [23] for deep-learning inference on the Android OS. MOSAIC achieves high efficiency by executing the slices of the target inference workload across computing devices on the underlying heterogeneous embedded system in a heterogeneity-, communication-, and constraint-aware manner.

[2]The work presented in this chapter was also published in [61]

- We quantify the effectiveness of MOSAIC with widely used inference workloads on a full heterogeneous embedded system that consists of a big core cluster, a little core cluster, a GPU, and an NPU. Our experimental results demonstrate the effectiveness of MOSAIC as it significantly improves the efficiency of inference (e.g., 29.2% lower inference latency than an NPU-preferred version (i.e., `TF-NPU-P`) with the performance governor and large models and 36.6% lower energy consumption than an NPU-preferred version (i.e., `TF-NPU-O`) with an on-demand governor and large models), achieves high estimation accuracy, and incurs small overheads.

The rest of this chapter is organized as follows. Section 4.2 describes the experimental methodology. Section 4.3 presents the need for heterogeneity-, communication-, and constraint-aware inference. Section 4.4 discusses the design and implementation of MOSAIC. Section 4.5 quantifies the effectiveness of MOSAIC. Section 4.6 concludes the chapter with a summary.

4.2 Experimental Methodology

To investigate the characteristics of deep-learning inference workloads and the effectiveness of MOSAIC, we use a heterogeneous embedded system, the HiKey 970 embedded development board [8]. The evaluated system is equipped with the Kirin 970 mobile processor [15] that comprises a CPU including four Cortex-A73 (big) cores, four Cortex-A53 (little) cores, a Mali-G72 GPU, and an NPU. The big core cluster, little core cluster, and GPU support DVFS. The available frequency ranges of the big core cluster, little core cluster, and GPU are 682–2362MHz, 509–1844MHz, and 104–767MHz, respectively. The NPU lacks DVFS support.

The NPU has a memory constraint in that it cannot execute a model slice whose size exceeds 100MB [8]. While the exact memory constraints of the big core cluster, the little core cluster and the GPU for executing inference workloads on the evaluated system are undocumented, they are sufficiently large for the evaluated inference workloads.

With regard to the system software stack, the evaluated heterogeneous embedded system is installed with Android 8.1. In addition, all of the evaluated inference workloads and MOSAIC are implemented using TensorFlow Lite 1.11.0 [23].

Table 1 shows the inference workloads (i.e., Inception V4 (IN) [152], MnasNet with model width parameters (p_W) of 1.0 (MN-1.0) and 1.3 (MN-1.3) [153], MobileNet V2 with $p_W = 1.3$ (MO-1.3) and $p_W = 1.4$ (MO-1.4) [138], ResNet V2 (RN) [67], and VGG (VGG) [142]) with large models. They exhibit high accuracy (i.e., Top-1 accuracy with the ImageNet [135] dataset) and widely-different characteristics such as the model size (i.e., the memory used by the model, which is reported by TensorFlow Lite), the layer count, and the inference latency (on the evaluated GPU). The evaluated workload set includes the state-of-the-art inference workloads (e.g., MobileNet V2 [138], MnasNet [153]), which are highly optimized for mobile systems. MobileNet V2 and MnasNet provide a mechanism to exploit the tradeoff between inference accuracy and latency through hyperparameters. Specifically, the model width parameter (i.e., p_W) determines the number of channels, which generally results in higher accuracy and longer latency when it is set to a larger value.

Table 1: Evaluated deep-learning inference workloads

Workload	Accuracy	Model Size	Layers	Latency
Inception V4 (IN) [152]	80.1%	183.7MB	20	430.0ms
MnasNet with $p_W = 1.0$ (MN-1.0) [153]	74.1%	321.9MB	20	66.8ms
MnasNet with $p_W = 1.3$ (MN-1.3) [153]	75.2%	511.6MB	20	81.3ms
MobileNet V2 with $p_W = 1.3$ (MO-1.3) [138]	74.4%	187.8MB	18	49.1ms
MobileNet V2 with $p_W = 1.4$ (MO-1.4) [138]	75.0%	214.7MB	18	55.4ms
ResNet V2 (RN) [67]	77.8%	260.4MB	53	603.8ms
VGG (VGG) [142]	71.5%	407.4MB	16	218.7ms
MnasNet with $p_W = 0.5$ (MN-0.5) [153]	68.0%	98.3MB	20	42.7ms
MobileNet V2 with $p_W = 1.0$ (MO-1.0) [138]	71.8%	94.5MB	18	36.6ms
SqueezeNet (SN) [71]	49.0%	34.8MB	10	32.5ms

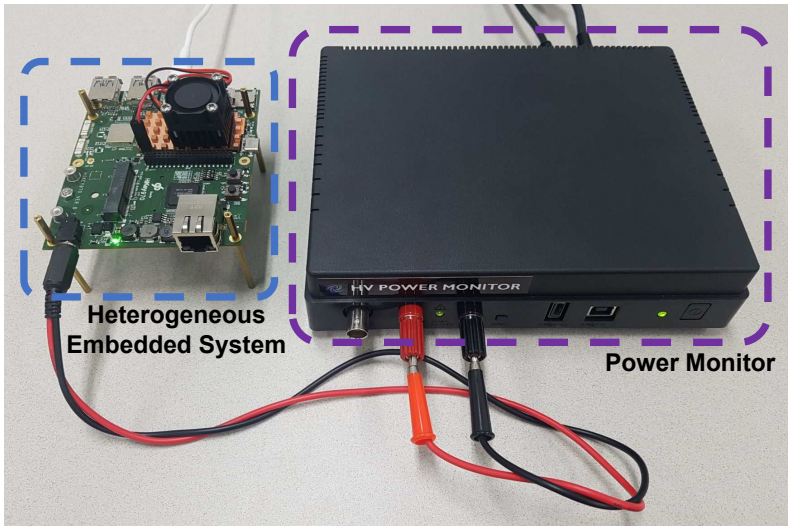


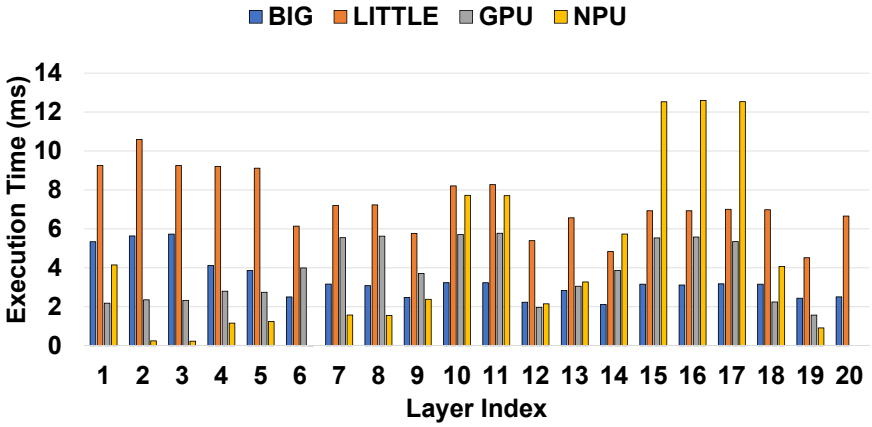
Figure 2: Evaluated heterogeneous embedded system and power monitor

As shown in Table 1, we use MnasNet with $p_W = 0.5$ (MN-0.5), MobileNet V2 with $p_W = 1.0$ (MO-1.0), and SqueezeNet (SN) [71] to investigate the impact of MOSAIC with smaller models. Due to the use of smaller models, they tend to exhibit lower accuracy.

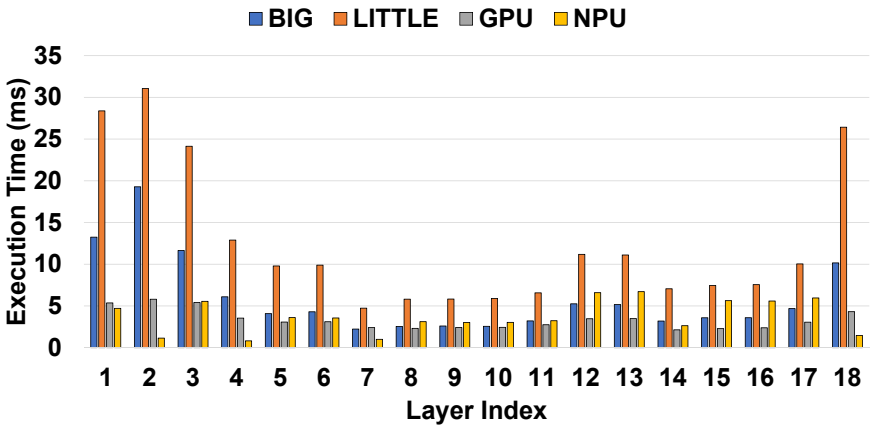
To measure the latency of the inference workloads, we use the `high_resolution_clock` function in the C++ standard library. To measure the energy consumption of the inference workloads, we use an external power monitor [7], which collects the voltage and current applied to the evaluated heterogeneous embedded system at a data sampling rate of 5000 samples per second. Figure 2 shows the power monitor connected to the evaluated heterogeneous embedded system.

4.3 Need for Heterogeneity-, Communication-, and Constraint-Aware Inference

We investigate the characteristics of widely-used deep-learning inference workloads in terms of the model size, performance and energy heterogeneity, and communication overheads on the evaluated heterogeneous embedded system. For conciseness, we mainly report the data with MN-1.0 and MO-1.4, which represent accurate and highly-optimized inference workloads on mobile systems.



(a) MnasNet 1.0



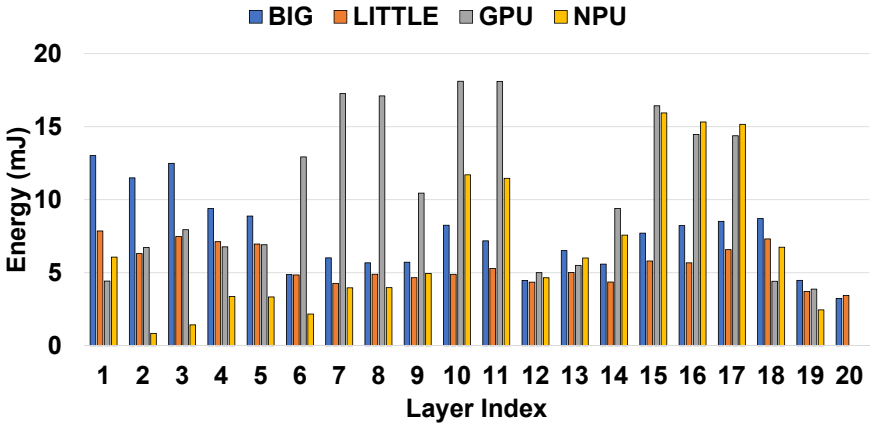
(b) MobileNet V2 1.4

Figure 3: Performance heterogeneity of inference workloads

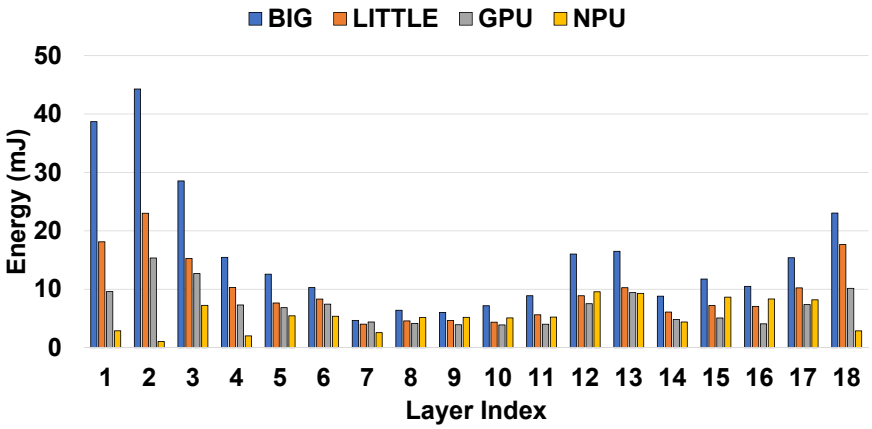
As shown in Table 1, accurate inference workloads including those (e.g., MN-1.3, MO-1.4) highly optimized for mobile systems employ large models, that exceed the memory constraint of the NPU (i.e., 100MB) on the evaluated heterogeneous embedded systems. Further, the evaluated inference workloads exhibit widely-different model sizes, indicating that different numbers of slices are required to execute them using the NPU.

Figure 3 shows the execution time of each layer of MN-1.0 and MO-1.4 when executed on the big core cluster, the little core cluster, the GPU, and the NPU. We observe that the layers of MN-1.0 and MO-1.4 exhibit widely different performance characteristics across the devices. For instance, the GPU achieves significantly higher performance than the NPU when executing layers 12–17 of MO-1.4. In contrast, the NPU significantly outperforms the GPU when executing layers 2–9 of MN-1.0.

Figure 4 shows the energy consumption of each layer of MN-1.0 and MO-1.4 when executed on the big core cluster, the little core cluster, the GPU, and the NPU. Similarly to the performance heterogeneity, we also observe that the layers of MN-1.0 and MO-1.4 exhibit disparate energy consumption characteristics across the devices. For example, the little core cluster consumes significantly lower energy than the NPU when executing layers 9–17 of MN-1.0. The little core cluster tends to achieve



(a) MnasNet 1.0



(b) MobileNet V2 1.4

Figure 4: Energy heterogeneity of inference workloads

higher efficiency when executing less computational intense layers. In contrast, the NPU significantly outperforms the little core cluster in terms of energy efficiency when executing layers 1–7 of MO-1.4.

Figure 5 shows the inference latency of MO-1.4 when decomposing MO-1.4 into three slices with various slicing plans and with the preferred computing device of each slice is set to the NPU. We observe that their inference latency is highly sensitive to the slicing plan. For instance, the performance difference between the best and worst slicing plans is 34.1%, which is significant. This data trend indicates that communication overhead has a significant impact on the efficiency of the target inference workload.

Overall, our experimental results show that the evaluated inference workloads exhibit broadly different characteristics in terms of the model size, performance and energy heterogeneity, and communication overheads. Therefore, it is crucial to investigate the system-software support for heterogeneity-, communication-, and constraint-aware model slicing and execution to achieve the best possible efficiency of inference workloads on heterogeneous embedded systems.

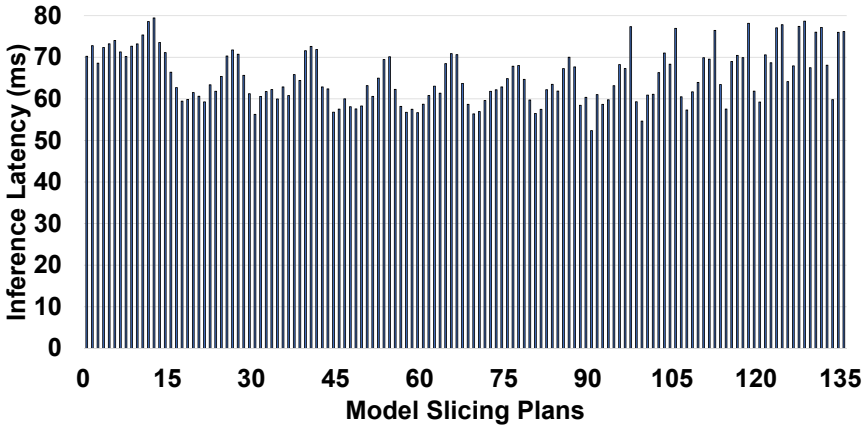


Figure 5: Communication overheads

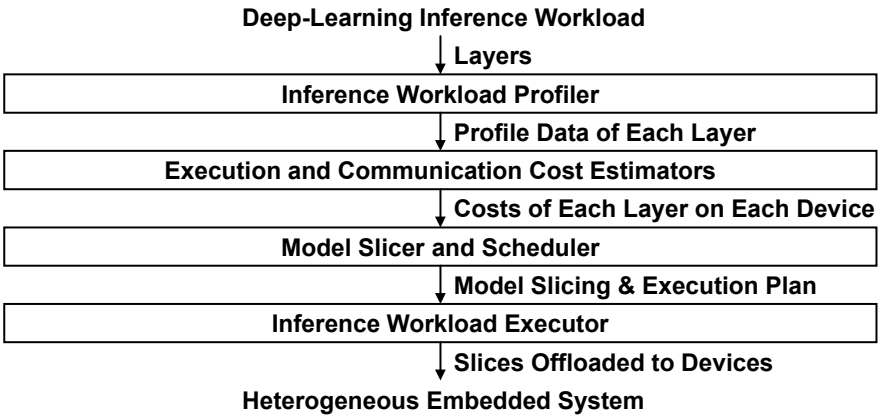


Figure 6: Overall architecture of MOSAIC

4.4 Design and Implementation

MOSAIC is a software-based system that determines the efficient model slicing and execution plan for the target deep-learning inference workload based on the user-defined metrics such as latency and energy consumption. Figure 45 shows the overall architecture of MOSAIC, which mainly consists of the inference workload profiler, the execution and communication cost estimators, the model slicer and scheduler, and the inference workload executor.

4.4.1 Inference Workload Profiler

The inference workload profiler of MOSAIC executes each of the layers in the target inference workload and profiles the total costs (e.g., latency, energy consumption) for executing each layer on computing devices in the underlying heterogeneous embedded system. If a computing device supports DVFS, the total costs for executing the layer are collected at two frequencies (i.e., the maximum and minimum frequencies available on the computing device).

If the layer cannot be executed on a certain computing device due to a constraint (e.g., memory constraint), the total cost for executing the layer on the device is set to an infinite value. The inference

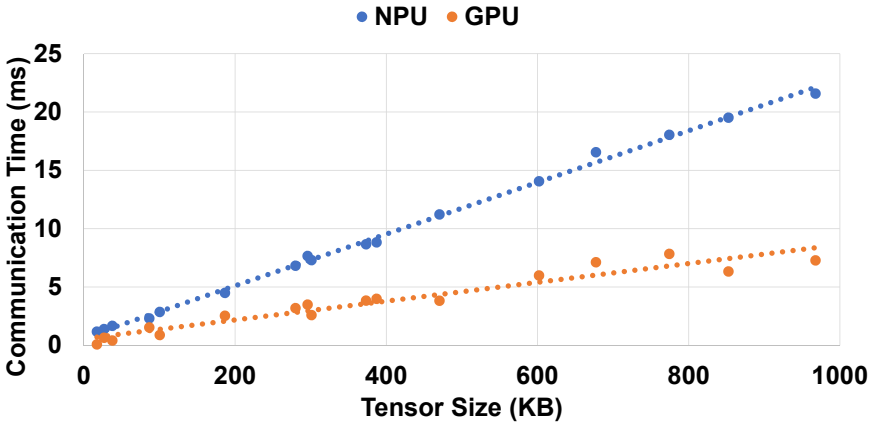


Figure 7: Communication time with various tensor sizes

workload profiler also collects the sizes of the input and output tensors of the layer, which are used to estimate the communication costs of the layer.

4.4.2 Execution and Communication Cost Estimators

The total cost of each layer measured by the inference workload profiler includes both the execution and communication costs associated with the layer. Based on the total cost and the input and output tensor sizes of each layer, the execution and communication cost estimators of MOSAIC estimate the execution and communication costs of the layer on each computing device in the heterogeneous embedded system.

Communication Cost Estimator: To investigate the relationship between the communication cost and the tensor size, we developed a micro-benchmark that consists of layers with various tensor sizes. Figure 7 shows the communication time with various tensor sizes on the GPU (at its maximum frequency) and the NPU. The communication time data with the CPU are omitted as our experimental results show that they are insignificant due to no or small data copy and format transformation overheads. We observe the following data trends.

First, on both the GPU and NPU, the communication time is linearly proportional to the size of the tensors associated with the layer. Second, the NPU incurs significantly larger communication overheads than the GPU. While the implementation details of the evaluated NPU are undisclosed [15], we conjecture that the amounts of data copy and/or format transformation overheads for the NPU are significantly larger than those for the GPU.

Guided by the aforementioned observations, we design and implement the communication cost estimator based on the linear regression technique. Specifically, the communication cost estimator employs Equations 1 and 2 to estimate the communication costs associated with the input and output tensors of the layer l, where $T_{in,l}$, $T_{out,l}$, α_{d,f_d}, and β_{d,f_d} correspondingly denote the total sizes of the input and output tensors of the layer l and the regression coefficients of computing device d (i.e., the GPU or NPU)

at frequency f_d.

$$c_{in,l,d,f_d} = \alpha_{d,f_d} \cdot T_{in,l} + \beta_{d,f_d} \tag{1}$$

$$c_{out,l,d,f_d} = \alpha_{d,f_d} \cdot T_{out,l} + \beta_{d,f_d} \tag{2}$$

Execution Cost Estimator: The execution cost estimator estimates the cost incurred when executing each layer without including the communication costs. Specifically, the execution cost estimator first estimates the total execution cost of each layer on a computing device d running at f_d based on the profile data collected at the maximum and minimum frequencies of d. The execution cost estimator then simply subtracts the communication costs of the layer estimated by the communication cost estimator from the estimated total cost of the layer. The execution cost estimator consists of the performance and power estimators.

Performance Estimator: The performance estimator of the execution cost estimator estimates the latency that arises when executing layer l on computing device d, which runs at frequency f_d. The performance estimator builds on a linear model, as shown in Equation 3, where t_{l,d,f_d}, $\gamma_{l,d}$, and $\varepsilon_{l,d}$ denote the estimated latency and coefficients. The coefficients (i.e., $\gamma_{l,d}$, $\varepsilon_{l,d}$) are computed based on the profile data collected by the inference workload profiler at the maximum and minimum frequencies of d.

$$t_{l,d,f_d} = \frac{\gamma_{l,d}}{f_d} + \varepsilon_{l,d} \tag{3}$$

Power Estimator: The power estimator of the execution cost estimator estimates the power consumption for executing a layer l on a computing device d running at frequency f_d. As shown in Equation 4, the power estimator decomposes the total power consumption into the dynamic (i.e., $P_{\text{dynamic},l,d,f_d}$) and static (i.e., P_{static,d,f_d}) power consumption. Because static power consumption is the inherent property of a computing device and independent of the characteristics of the target inference workload, we only profile it once for each computing device (i.e., no need for per-workload profiling).

$$P_{l,d,f_d} = P_{\text{dynamic},l,d,f_d} + P_{\text{static},d,f_d} \tag{4}$$

Since dynamic power consumption is dependent on not only the characteristics of the computing device (and its frequency) but also on the characteristics of the layer, the power estimator estimates dynamic power consumption to eliminate the need for extensive offline profiling. Specifically, the power estimator employs Equation 5 to estimate the dynamic power consumption (i.e., $P_{\text{dynamic},l,d,f_d}$) of layer l running on computing device d at frequency f_d (and the corresponding voltage level V_{f_d}), where $f_{d,\max}$, $V_{f_{d,\max}}$, and $P_{\text{dynamic},l,d,f_{d,\max}}$ denote the maximum frequency of d, the corresponding voltage level, and the

Table 2: Voltage and frequency levels of the evaluated computing devices

Device	Voltage and frequency levels
Big cores	(0.7V, 682MHz), (0.8V, 1018MHz), (0.8V, 1210MHz), (0.8V, 1364MHz), (0.9V, 1498MHz), (0.9V, 1652MHz), (0.9V, 1863MHz), (1.0V, 2093MHz), (1.1V, 2362MHz)
Little cores	(0.7V, 509MHz), (0.8V, 1018MHz), (0.9V, 1210MHz), (0.9V, 1402MHz), (1.0V, 1556MHz), (1.0V, 1690MHz), (1.1V, 1844MHz)
GPU	(0.6V, 104MHz), (0.7V, 151MHz), (0.7V, 237MHz), (0.7V, 332MHz), (0.8V, 415MHz), (0.8V, 550MHz), (0.9V, 667MHz), (1.0V, 767MHz)

dynamic power consumption at the maximum frequency collected by the inference workload profiler.

$$P_{\text{dynamic},l,d,f_d} = \frac{V_{f_d}^2 \cdot f_d}{V_{f_{d,\max}}^2 \cdot f_{d,\max}} \cdot P_{\text{dynamic},l,d,f_{d,\max}} \tag{5}$$

The voltage and frequency levels of computing devices are readily available through their specifications or direct measurements. Table 2 shows the voltage and frequency levels of each computing device on the evaluated heterogeneous embedded system, taken from publicly available device tree source (DTS) files. The dynamic power consumption can be estimated by plugging in specific values of f_d and V_{f_d} in Equation 5.

4.4.3 Model Slicer and Scheduler

The main goal of the model slicer and scheduler (MSS) of MOSAIC is to generate an efficient model slicing and execution plan for the target deep-learning inference workload on the heterogeneous embedded system. Specifically, MSS determines the number of slices, the layers that belong to each slice, and the computing device that executes each slice in order to maximize the efficiency of the target inference workload on the heterogeneous embedded system based on the user-defined metric (e.g., latency, energy consumption).

We formulate the model slicing and execution problem as a dynamic-programming problem [37]. Without loss of generality, we assume that the target inference workload employs a deep-learning model that consists of Λ layers. $C_{m,n,\delta}$ denotes the total cost for executing the $n-m+1$ consecutive layers from the m-th layer to the n-th layer of the model on the computing device δ, where $1 \leq m \leq n \leq \Lambda$, and $\delta \in \mathbb{D}$. Note that we consider identical physical computing devices running at different frequencies as different computing devices to simplify the energy optimization problem formulation.

For performance optimization, $\mathbb{D}$ is defined by Equation 6, where $B_{f_{B,\max}}$, $L_{f_{L,\max}}$, $G_{f_{G,\max}}$, and N denote the big core cluster at its maximum frequency, the little core cluster at its maximum frequency, the GPU at its maximum frequency, and the NPU, respectively. Note that the NPU on the evaluated heterogeneous embedded system lacks support for DVFS. On the evaluated system, $|\mathbb{D}|$ is 4 for performance optimization.

$$\mathbb{D} = \{B_{f_{B,\max}}, L_{f_{L,\max}}, G_{f_{G,\max}}, N\} \tag{6}$$

With regard to energy optimization, $\mathbb{D}$ is defined as Equation 7, where $B_{f_{B,\min}}$, $B_{f_{B,\min+1}}$, $\cdots$, and $B_{f_{B,\max}}$ denote the big core cluster at its minimum, second minimum, $\cdots$, and maximum frequencies, $L_{f_{L,\min}}$, $L_{f_{L,\min+1}}$, $\cdots$, and $L_{f_{L,\max}}$ indicate the little core cluster at its minimum, second minimum, $\cdots$, and maximum frequencies. Additionally, $G_{f_{G,\min}}$, $G_{f_{G,\min+1}}$, $\cdots$, and $G_{f_{G,\max}}$ correspondingly denote the GPU at its minimum, second minimum, $\cdots$, and maximum frequencies, and N indicates the NPU. On the evaluated system, $|\mathbb{D}|$ is 25 for energy optimization (see Table 2).

$$\begin{aligned}\mathbb{D} = \{&B_{f_{B,\min}}, B_{f_{B,\min+1}}, \cdots, B_{f_{B,\max}},\\ &L_{f_{L,\min}}, L_{f_{L,\min+1}}, \cdots, L_{f_{L,\max}},\\ &G_{f_{G,\min}}, G_{f_{G,\min+1}}, \cdots, G_{f_{G,\max}},\\ &N\}\end{aligned} \tag{7}$$

$C_{m,n,\delta}$ is computed using Equation 8, where $e_{k,\delta}$, $c_{in,m,\delta}$, and $c_{out,n,\delta}$ represent the execution cost of the k-th layer ($m \leq k \leq n$) and the communication cost associated with the input tensors of the m-th layer, and the communication cost associated with the output tensors of the n-th layer, respectively. MSS employs the execution and communication cost estimators to estimate $e_{k,\delta}$, $c_{in,m,\delta}$, and $c_{out,n,\delta}$. If there is any layer that cannot be executed on δ due to a constraint, $C_{m,n,\delta}$ is set to an infinite value to avoid selecting the corresponding model slicing and execution plan.

$$C_{m,n,\delta} = \begin{cases} \sum_{k=m}^{n} e_{k,\delta} + c_{in,m,\delta} + c_{out,n,\delta} & \text{if } \delta \text{ can execute} \\ \infty & \text{otherwise} \end{cases} \tag{8}$$

$C_{tot,l}$ represents the total cost to execute the consecutive layers from the first layer to the l-th layer. To formulate the model slicing and execution problem as a dynamic-programming problem, $C_{tot,l}$ must exhibit the optimal substructure and overlapping subproblem properties [37] in that $C_{tot,l}$ can be efficiently computed if $C_{tot,0}$, $C_{tot,1}$, $\cdots$, and $C_{tot,l-1}$ are known. We consider all the possible cases, in each of which the l-th layer belongs to a unique slice. Because each layer belongs to only one slice and each slice consists of consecutive layers, there are l cases in total, where the l-th layer belongs to unique slices. Specifically, the slice that contains the l-th layer may comprise only a single layer (i.e., the l-th layer), two layers (i.e., the l-th layer and the $(l-1)$-th layer), $\cdots$, or l layers (i.e., the l-th layer, the $(l-1)$-th layer, $\cdots$, and the first layer).

Without a loss of generality, we assume that the slice that contains the l-th layer comprises $l-k$ layers (i.e., the l-th, $(l-1)$-th, $\cdots$, and $(k+1)$-th layers). In this case, the lowest cost for executing the consecutive layers from the first layer to the l-th layer is computed by summing $C_{tot,k}$ and the total cost for executing the last $l-k$ layers on the device that incurs the minimum total cost among all the computing devices on the heterogeneous embedded system.

As shown in Equation 9, $C_{tot,l}$ can be then computed by finding the minimum total cost among all l aforementioned cases. Because the model slicing and execution problem has the optimal substructure and overlapping subproblem properties, its optimal solution can be determined based on dynamic

Algorithm 1 The findEfficientSlicingAndExecutionPlan function

```
1:  procedure FINDEFFICIENTSLICINGANDEXECUTIONPLAN(layers, devices)
2:      sliceSets[0] ← ∅
3:      sliceSets[0].cost ← 0
4:      for l ← 1 to layers.length do                          ▷ layers.length = Λ
5:          sliceSet ← ∅
6:          sliceSet.cost ← ∞
7:          slice ← createNewSlice()
8:          for k ← 0 to (l − 1) do
9:              slice.layers ← getConsecutiveLayers(layers, k + 1, l)
10:             for δ in devices do                            ▷ devices = 𝔻
11:                 if δ.canExecute(slice) = true then
12:                     slice.device ← δ
13:                     cost ← sliceSets[k].cost + estimateTotalCost(slice)
14:                     if cost < sliceSet.cost then
15:                         sliceSet ← sliceSets[k]
16:                         sliceSet.insert(slice)
17:                         sliceSet.cost ← cost
18:                     end if
19:                 end if
20:             end for
21:         end for
22:         sliceSets[l] ← sliceSet
23:         sliceSets[l].cost ← sliceSet.cost
24:     end for
25:     return sliceSets[layers.length]
26: end procedure
```

programming.

$$C_{tot,l} = \begin{cases} 0 & \text{if } l = 0 \\ \min\limits_{0 \le k < l, \forall \delta \in \mathbb{D}} \left(C_{tot,k} + C_{k+1,l,\delta} \right) & \text{otherwise} \end{cases} \tag{9}$$

Algorithm 2 shows the pseudocode for the `findEfficientSlicingAndExecutionPlan` function that determines the efficient model slicing and execution plan based on dynamic programming. In the outer loop (Lines 4–24), MSS iterates the layer count from 1 to Λ. MSS uses the solutions found in previous iterations to find the solution for the current iteration in the outer loop.

In the inner loop (Lines 8–21), MSS iterates all the cases, in each of which the last layer belongs to a unique slice. MSS determines the set of slices that minimizes the total cost when executing the consecutive layers and memoizes its cost and slicing and execution plan to reuse them in the next iteration in the outer loop.

The proposed algorithm has low time complexity (i.e., $O(\Lambda^2 \cdot |\mathbb{D}|)$, where Λ and $|\mathbb{D}|$ denote the number of layers in the inference workload and the number of computing devices on the heterogeneous embedded system, respectively). As quantified in Section 4.5, MOSAIC achieves high inference efficiency with small overheads due to the use of an efficient algorithm.

4.4.4 Inference Workload Executor

The inference workload executor of MOSAIC is a user-level runtime system that executes the model slices of the target inference workload across the computing devices on the heterogeneous embedded system based on the efficient model slicing and execution plan generated by MSS. The current version of the inference workload executor is implemented in the C++ programming language based on the TensorFlow Lite framework [23] on the Android OS. However, we believe that MOSAIC is readily applicable to other widely used deep-learning frameworks as it builds on a framework-agnostic approach for slicing and executing the target inference workload.

4.5 Evaluation

4.5.1 Overview

This section quantifies the effectiveness of MOSAIC. Specifically, we aim to investigate (1) inference latency, (2) inference energy, (3) impact of the MOSAIC components, (4) efficiency with smaller models, (5) estimation accuracy, and (6) overheads for generating the model slicing and execution plan.

For each inference workload, we evaluate ten versions – the big core cluster-preferred (`TF-BIG-P`), little core cluster-preferred (`TF-LITTLE-P`), GPU-preferred (`TF-GPU-P`), NPU-preferred (`TF-NPU-P`) with the performance governor of the Android OS, the big core cluster-preferred (`TF-BIG-O`), little core cluster-preferred (`TF-LITTLE-O`), GPU-preferred (`TF-GPU-O`), NPU-preferred (`TF-NPU-O`) with the on-demand governor of the Android OS, exhaustive, and MOSAIC versions.

The big core cluster-, little core cluster-, GPU-, and NPU-preferred versions execute the slices of each inference workload on the corresponding preferred computing device. For the big core cluster-, little core cluster-, GPU-, and NPU-preferred versions, we include as many consecutive layers as possible in each slice if they satisfy all the constraints such as memory and functionality constraints. If a layer cannot be included in the slice that contains its previous layer due to a memory constraint, the layer is included in the next slice that is executed on the preferred device. If a layer cannot be executed on the preferred device due to a memory or functionality constraint, the layer is executed in a separate slice on the NPU (if feasible), GPU (if feasible), or big core cluster (as the final fallback execution path).

As quantified by our experimental results, the performance governor tends to achieve higher performance than the on-demand governor as the performance governor always executes the target inference workload at the maximum frequency of the underlying computing device. In contrast, the on-demand governor, which is the default governor of the Android OS, tends to exhibit lower energy consumption than the performance governor as the on-demand governor performs DVFS based on the dynamic load of the target inference workload.

The exhaustive version uses the model slicing and execution plan with the highest inference efficiency (e.g., the lowest latency), which is empirically determined through an exhaustive search. Note that it takes excessive computing time and resources to determine the model slicing and execution plan for the exhaustive version, which is impractical. For instance, it is estimated to take 1335 days to empir-

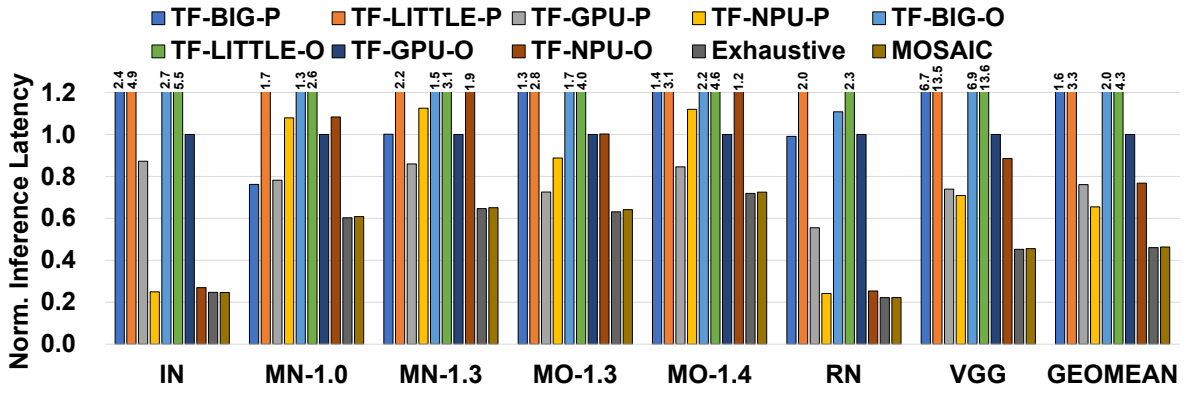


Figure 8: Inference latency

Table 3: Model slicing and execution plans for performance optimization

Workload	Model slicing and execution plan
IN	N_1, N_{11}, B_{20}
MN-1.0	$N_1, B_{10}, G_{18}, B_{20}$
MN-1.3	N_1, B_9, G_{12}, B_{20}
MO-1.3	N_1, G_9, N_{18}
MO-1.4	N_1, G_8, N_{18}
RN	$B_1, N_2, B_4, N_5, B_{12}, N_{13}, N_{31}, B_{48}, N_{49}, B_{52}$
VGG	N_1, G_{14}, B_{16}

ically determine the best model slicing and execution plan for MN-1.3 through exhaustive search using the evaluated system.

Due to limited computing time and resources, we determine the model slicing and execution plan for the exhaustive version of each inference workload by executing the workload with 1000 unique model slicing and execution plans that are randomly selected and choosing the best plan among the randomly selected plans and the plans used by the other versions including the MOSAIC version. We report the results with the exhaustive version as the efficiency that can be potentially achieved by any competitive model slicing and execution technique.

Finally, the MOSAIC version uses MOSAIC to generate the efficient model slicing and execution plan.

4.5.2 Inference Latency

We investigate the effectiveness of MOSAIC in terms of inference latency. Figure 8 shows the inference latency of each version of the inference workloads with large models, normalized to the `TF-GPU-O` version, which is the default setting of the Android OS. The rightmost bars show the average (i.e., geometric mean) inference latency of each version across the workloads. In addition, Table 3 shows the model slicing and execution plans generated by MOSAIC to minimize the latency of each workload. Each letter (i.e., big core cluster (B), little core cluster (L), GPU (G), and NPU (N)) indicates a slice and the device used to execute the slice. The subscript to each letter denotes the ID of the first layer of the

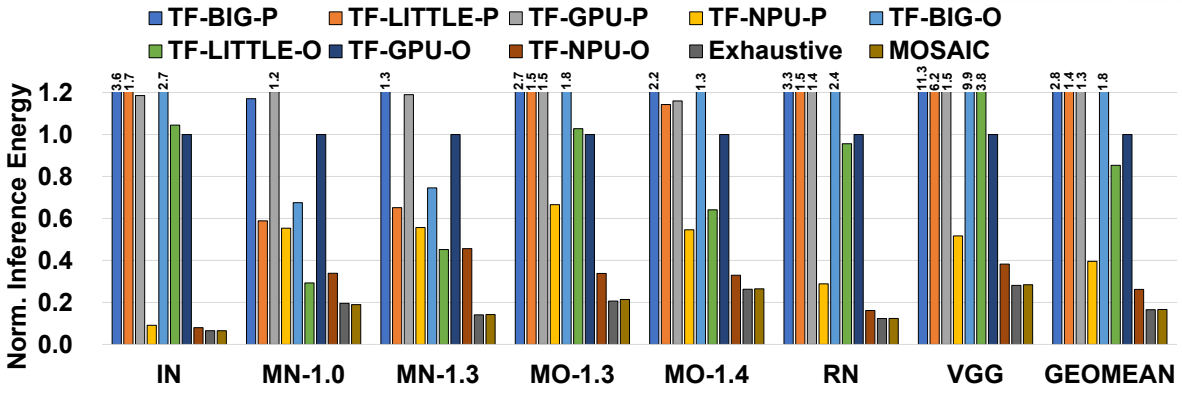


Figure 9: Inference energy

slice.

First, MOSAIC significantly outperforms the big core cluster-, little core cluster-, GPU-, and NPU-preferred versions. Specifically, MOSAIC exhibits 70.3%, 86.1%, 39.1%, and 29.2% lower inference latency than the TF-BIG-P, TF-LITTLE-P, TF-GPU-P, and TF-NPU-P versions, respectively. MOSAIC significantly reduces inference latency by slicing and executing the model of the target inference workload in a heterogeneity-, communication-, and constraint-aware manner. For instance, as shown in Table 3, MOSAIC effectively utilizes various computing devices (i.e., big core cluster, GPU, and NPU) when executing MN-1.3 and significantly reduces its inference latency.

MOSAIC exhibits the inference latency similar to that of the TF-NPU-P version with IN and RN. This is mainly because the NPU exhibits the highest performance among the computing devices and the model slicing plan used for the TF-NPU-P version happens to incur small communication overheads when executing IN and RN. Nevertheless, the NPU-preferred versions provide no guarantee for maximizing the efficiency across a wide range of inference workloads as they execute the target inference workload in a heterogeneity- and communication-oblivious manner.

Second, MOSAIC achieves the performance similar to that of the exhaustive version, which empirically determines the efficient model slicing and execution plan for the target inference workload through exhaustive search. Specifically, the average performance difference between the exhaustive and MOSAIC versions is 0.67% across the workloads, which is small. Note that the exhaustive version is guaranteed to exhibit (at least) the same efficiency as MOSAIC because we always include the model slicing and execution plan determined by MOSAIC in the plans that are explored by the exhaustive version. Our experimental results clearly demonstrate the effectiveness of MOSAIC in that it achieves high inference efficiency without the need for the exhaustive search process, which requires excessive computing time and resources.

4.5.3 Inference Energy

We investigate the effectiveness of MOSAIC in terms of inference energy. Figure 9 shows the energy consumption of each version of the inference workloads with large models, normalized to the TF-GPU-O

Table 4: Model slicing and execution plans for energy optimization

Workload	Model slicing and execution plan
IN	$N_{1}^{960}, N_{11}^{960}, L_{20}^{509}$
MN-1.0	$G_{1}^{767}, L_{7}^{509}, G_{12}^{667}, L_{15}^{509}, G_{18}^{667}, L_{20}^{509}$
MN-1.3	$G_{1}^{767}, L_{7}^{509}, G_{12}^{667}, L_{14}^{509}, G_{18}^{667}, L_{20}^{509}$
MO-1.3	$N_{1}^{960}, L_{5}^{509}, B_{7}^{682}, L_{10}^{1018}, L_{11}^{509}, L_{14}^{1018}, L_{15}^{509}, N_{18}^{960}$
MO-1.4	$N_{1}^{960}, L_{5}^{1018}, L_{6}^{509}, L_{10}^{1018}, L_{11}^{509}, L_{13}^{1402}, G_{14}^{767}, N_{18}^{960}$
RN	$B_{1}^{682}, N_{2}^{960}, B_{4}^{682}, N_{5}^{960}, L_{12}^{1018}, N_{13}^{960}, N_{29}^{960}, L_{48}^{1210}, N_{49}^{960}, L_{52}^{509}, B_{53}^{682}$
VGG	$N_{1}^{960}, G_{14}^{550}, G_{15}^{667}, B_{16}^{682}$

version. The rightmost bars show the average (i.e., geometric mean) energy consumption across the workloads. In addition, Table 4 shows the model slicing and execution plans generated by MOSAIC for energy optimization. The superscript attached on each letter denotes the frequency of the computing device in MHz.

First, MOSAIC significantly outperforms the big core cluster-, little core cluster-, GPU-, and NPU-preferred versions across the workloads in terms of energy efficiency. For instance, MOSAIC consumes 91.0%, 80.5%, 83.4%, and 36.6% lower energy than the `TF-BIG-O`, `TF-LITTLE-O`, `TF-GPU-O`, and `TF-NPU-O` versions, respectively. As shown in Table 4, MOSAIC effectively utilizes various computing devices at various frequencies, significantly reducing the energy consumption of the inference workloads.

Second, MOSAIC exhibits the energy consumption similar (i.e., the average difference of 0.53%) to that of the exhaustive version, which requires excessive computing time and resources. Our experimental results demonstrate that MOSAIC can be effectively used for both inference latency and energy optimizations on heterogeneous embedded systems.

4.5.4 Impact of the MOSAIC Components

We investigate the impact of the MOSAIC components in terms of inference latency and energy. To this end, we synthesize the heterogeneity- and constraint-aware (HCA) version, which is an intermediate version that generates the model slicing and execution plan by only considering the efficiency heterogeneity and constraints of each layer (i.e., in a communication-oblivious manner). We report the results with the HCA version to investigate the efficiency impact of heterogeneity- and constraint-aware model slicing and execution. Note that the efficiency difference between the HCA and MOSAIC versions shows the efficiency impact of communication-aware model slicing and execution.

Figure 10 shows the latency impact of the MOSAIC components. We observe that the HCA version considerably outperforms the `TF-GPU-O` version, which demonstrates the effectiveness of heterogeneity- and constraint-aware model slicing and execution. Further, MOSAIC considerably outperforms the HCA version, which demonstrates the impact of communication-aware model slicing and execution. The HCA version incurs higher inference latency than MOSAIC as it slices and executes the target inference workload in a communication-oblivious manner.

We quantify the energy impact of the MOSAIC components. To this end, we synthesize an additional

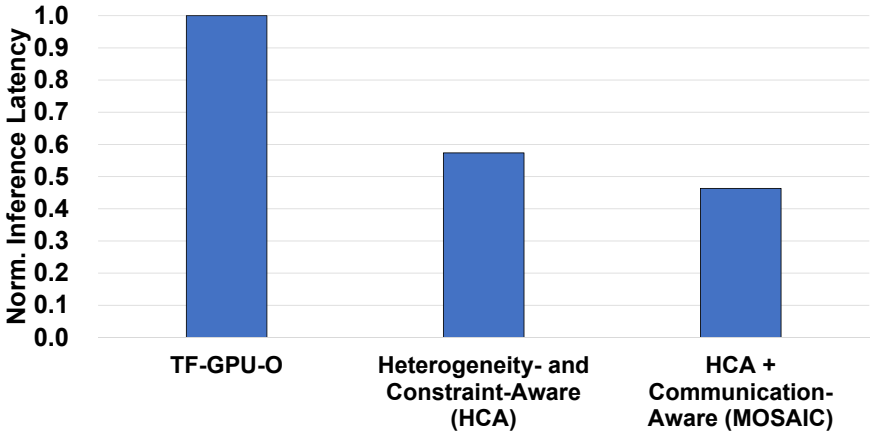


Figure 10: Latency impact of the MOSAIC components

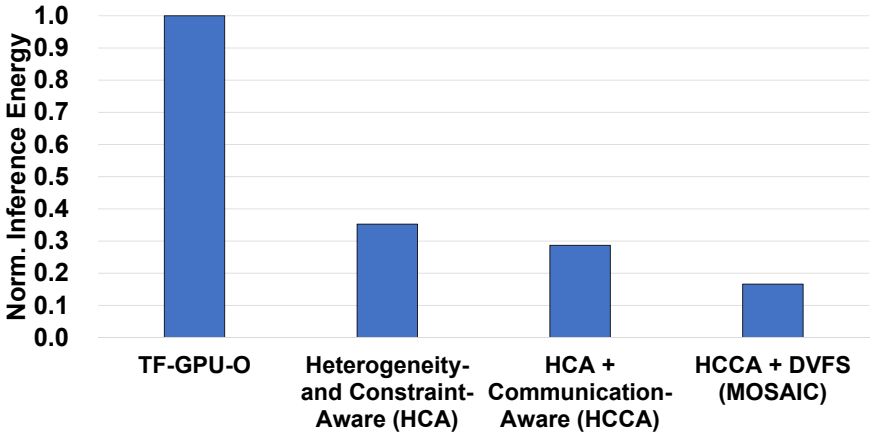


Figure 11: Energy impact of the MOSAIC components

intermediate version called the HCCA version, which generates the model slicing and execution plan while considering the heterogeneity of the physical computing devices, constraints of each layer, and communication overheads between devices in a DVFS-oblivious manner. Figure 11 shows the energy impact of the MOSAIC components.

The HCA version exhibits considerably lower energy consumption than the `TF-GPU-O` version, demonstrating the impact of heterogeneity- and constraint-aware model slicing and execution. In addition, the HCCA version consumes considerably lower energy than the HCA version, showing the effectiveness of communication-aware model slicing and execution. Finally, the MOSAIC version significantly exhibits significantly lower energy consumption than the HCCA version, which demonstrates the effectiveness of DVFS. In summary, our quantitative evaluation demonstrates that the individual components of MOSAIC compose in a constructive manner, providing additional performance and energy-efficiency gains.

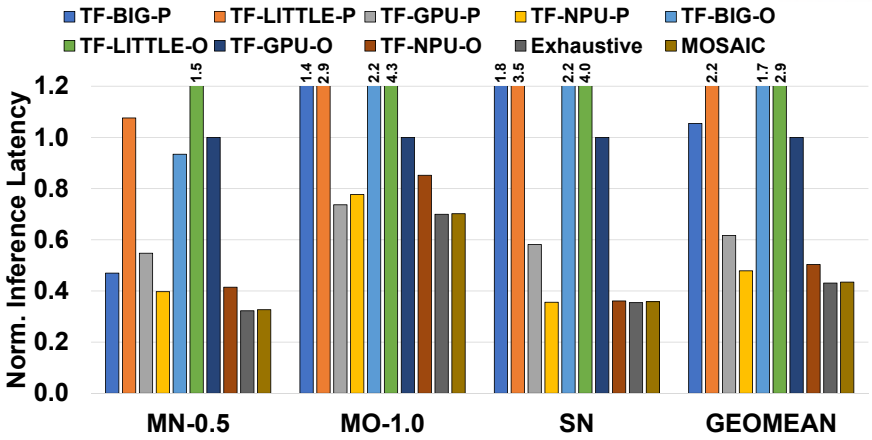


Figure 12: Inference latency with smaller models

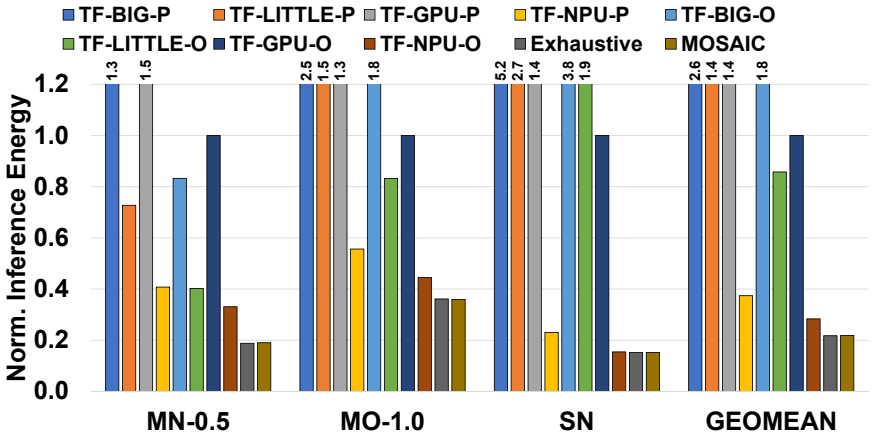


Figure 13: Inference energy with smaller models

4.5.5 Discussion

Smaller models: Figures 12 and 13 show the inference latency and energy consumption with smaller models. Our experimental results show that MOSAIC continues to achieve high efficiency with inference workloads with smaller models (i.e., MN-0.5, MO-1.0, SN). For instance, MOSAIC exhibits 58.8%, 80.4%, 29.5%, and 9.2% lower inference latency than the `TF-BIG-P`, `TF-LITTLE-P`, `TF-GPU-P`, and `TF-NPU-P` versions and performs similarly (i.e., the average difference of 0.89%) to the exhaustive version. Further, MOSAIC consumes 87.6%, 74.5%, 78.2%, and 22.9% lower energy than the `TF-BIG-O`, `TF-LITTLE-O`, `TF-GPU-O`, and `TF-NPU-O` versions and achieves similar energy efficiency (i.e., the average difference of 0.32%) to the exhaustive version. The efficiency gains of MOSAIC with smaller models decrease as fewer slices are generated, which reduces optimization opportunities.

Estimation Accuracy: Figures 14 and 15 correspondingly show the latency and energy consumption estimation accuracy of MOSAIC. Our results show that the estimation accuracy of MOSAIC is high. Specifically, the average latency and energy estimation errors are 3.0% and 4.3%, which are small.

Overheads: Figures 16 and 17 shows the overheads for performance and energy optimization. Our experimental results show that MOSAIC incurs small overheads for generating model slicing and exe-

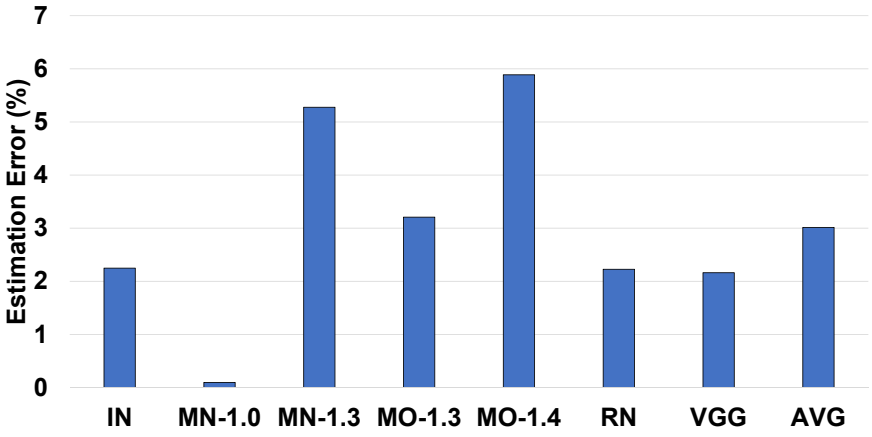


Figure 14: Latency estimation accuracy

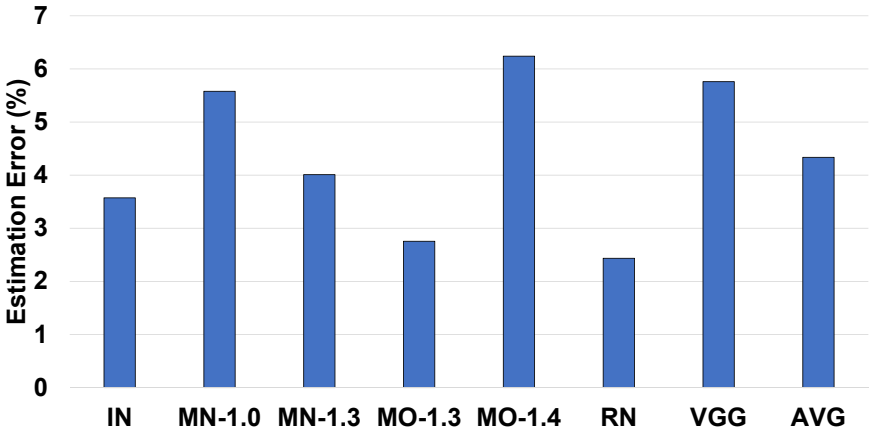


Figure 15: Energy estimation accuracy

cution plans. Specifically, the average times required to generating the model slicing and execution plans across the workloads for performance and energy optimization are 0.15ms and 0.82ms, which are short. Because MOSAIC produces the model slicing and execution plan based on the efficient algorithm with low time complexity (i.e., $O(\Lambda^2 \cdot |\mathbb{D}|)$), it incurs insignificant overheads. Further, note that MOSAIC generates the model slicing and execution plan only once for each inference workload and incurs no overheads during the execution of the workload.

MOSAIC incurs larger overheads for energy optimization than performance optimization. This is mainly because the number of computing devices (i.e., $|\mathbb{D}|$) increases (from 4 (i.e., performance optimization) to 25 (i.e., energy optimization)) as MOSAIC considers each computing device at its maximum frequency for performance optimization but considers each computing device at all its available frequencies for energy optimization.

Overall, our quantitative evaluation shows the effectiveness of MOSAIC in that it significantly improves the efficiency of inference workloads in terms of latency and energy consumption, achieves high estimation accuracy, and incurs small overheads on the evaluated heterogeneous embedded system.

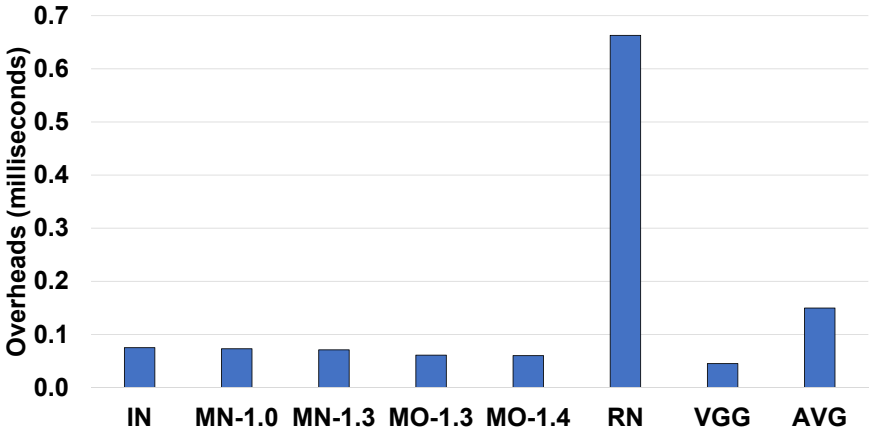


Figure 16: Overheads for performance optimization

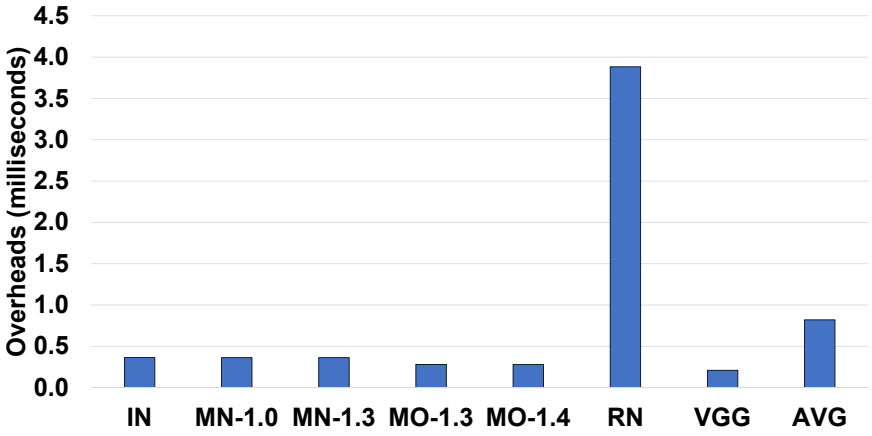


Figure 17: Overheads for energy optimization

4.6 Summary

This chapter presents MOSAIC, heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference on heterogeneous embedded systems. MOSAIC uses the accurate models for estimating the execution and communication costs of the target inference workload and generates the efficient model slicing and execution plan with low time complexity. Our quantitative evaluation with the state-of-the-art inference workloads and heterogeneous embedded system shows that MOSAIC significantly reduces inference latency and energy (e.g., 29.2% lower inference latency than an NPU-preferred version (i.e., `TF-NPU-P`) with the performance governor and large models and 36.6% lower energy than an NPU-preferred version (i.e., `TF-NPU-O`) with the on-demand governor and large models), achieves high estimation accuracy, and incurs small overheads.

V Reinforcement Learning-Augmented System for Efficient Real-Time Inference on Heterogeneous Embedded Systems

5.1 Introduction

The need for real-time deep-learning inference is ever increasing to enable latency-critical intelligent services such as autonomous driving, interactive image editing, and augmented reality. For such services, it is highly crucial to meet the deadline of real-time inference workloads to prevent any dangerous and disastrous situations from happening and deliver the best possible user experiences. Real-time inference workloads are often executed within a single system without relying on cloud services. This approach provides various advantages such as low latency and enhanced security and privacy.

Heterogeneous embedded systems have emerged as a promising solution for efficient real-time inference in a variety of computing domains ranging from mobile to high-performance computing. Heterogeneous embedded systems consist of various computing devices (e.g., big core cluster, little core cluster, GPU, and neural processing unit (NPU)), each of which exhibits widely-different architectural and system-level characteristics in terms of performance, power consumption, functionality (e.g., supported mathematical operations), memory capacity, and communication overheads.

Inference workloads also exhibit widely-different characteristics on heterogeneous computing devices in terms of performance, power efficiency, communication overheads, and constraints. For example, some parts of inference workloads may efficiently be executed on hardware accelerators such as NPUs, whereas other parts of inference workloads need to be executed on CPUs to avoid excessive communication overheads. If inference workloads are scheduled across heterogeneous computing devices in a suboptimal manner, they may fail to meet their deadlines and/or achieve low efficiency.

Despite extensive prior works on enhancing the efficiency of inference workloads on various systems, little work has been done to investigate the design and implementation of a practical system that enables efficient real-time inference on heterogeneous embedded systems. The main challenge when developing such a system is the high design complexity, which is caused by various characteristics of inference workloads and heterogeneous computing devices and critical constraints such as the inference deadline and the functional and capacity constraints of heterogeneous computing devices.

To bridge this gap, this chapter[3] proposes HERTI, a reinforcement learning (RL)-augmented system for efficient real-time inference on heterogeneous embedded systems. HERTI employs the accurate and efficient execution and communication cost estimators for inference workloads, which significantly reduce the training time. HERTI efficiently explores the state space with heterogeneity and constraint awareness and robustly finds the efficient state that executes the target inference workload across the heterogeneous computing devices while satisfying the deadline constraint through RL.

Specifically, this work makes the following contributions:

- We propose HERTI, an RL-augmented system for efficient real-time inference on heterogeneous embedded systems. HERTI builds on the accurate and lightweight execution and communication

[3]The work presented in this chapter was also published in [58]

cost estimators to significantly accelerate the training process. HERTI generates the efficient model slicing and execution plan for the target real-time inference workload with a strong deadline guarantee based on RL.

- Because the model slicing and execution planning problem for efficient real-time inference on heterogeneous embedded systems is a variant of the NP-hard multiple-choice quadratic knapsack problem [80] and has the Markov property [150], we formulate it as an RL problem. To overcome the state space explosion problem, we employ the RL algorithm (i.e., deep Q-network (DQN) [109]) to solve the model slicing and execution planning problem.

- We design and implement the prototype of HERTI as a user-level runtime system based on the TensorFlow Lite programming framework [23] for deep-learning inference on the Android OS. HERTI significantly improves the efficiency of the target inference workload with a strong deadline guarantee by executing its slices across computing devices on the underlying heterogeneous embedded system in a heterogeneity- and constraint-aware manner.

- We quantify the effectiveness of HERTI using widely-used inference workloads on a real heterogeneous embedded system that consists of the big core cluster, little core cluster, GPU, and NPU. Our quantitative evaluation demonstrates the effectiveness of HERTI in that it achieves high efficiency and outperforms the deadline-conscious state-of-the-art technique (i.e., AutoScale [84]) in multiple metrics (i.e., energy (i.e., 28.4%) and energy-delay product (i.e., 29.2%)) while satisfying the deadline constraint, consistently meets inference deadlines in contrast to the deadline-oblivious state-of-the-art technique (i.e., MOSAIC [61]), effectively translates increased inference deadlines and system heterogeneity into larger efficiency gains, exhibits the strong generality in that the RL hyper-parameters tuned for a specific configuration can effectively be used in other configurations, and significantly reduces the training time through its estimation-based approach across all the inference workloads and scenarios.

The rest of this chapter is organized as follows. Section 5.2 provides background information. Section 5.3 presents the design and implementation of HERTI. Section 5.4 describes the experimental methodology and Section 5.5 quantifies the effectiveness of HERTI. Finally, Section 5.6 concludes the chapter with a summary.

5.2 Background: Deep Q-Network

Reinforcement learning (RL) is a machine learning technique that aims to learn a sequence of decisions that maximizes a cumulative reward [150]. In RL, there are two main components – the agent and the environment. The agent interacts with the environment in discrete time steps. The state of the environment at step t is denoted as s_t. The agent chooses an action a_t from the available action set according to a policy and performs it. The state of the environment is then transitioned to s_{t+1} and the agent receives the reward r_t. The goal of the agent is to learn the best policy that maximizes the cumulative reward.

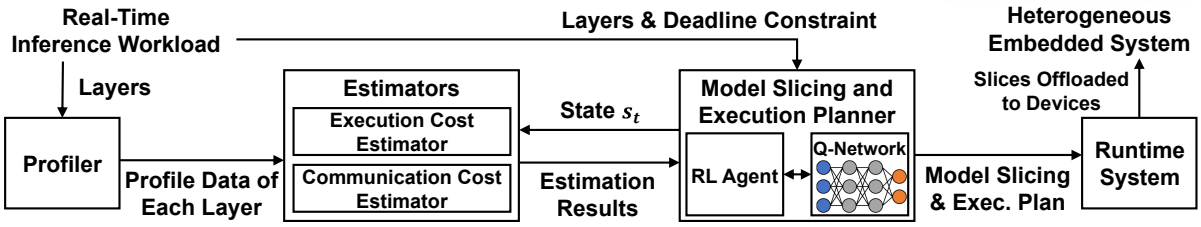


Figure 18: Overall architecture of HERTI

Q-learning is one of the widely used RL algorithms owing to its effectiveness and simplicity [150]. Q-learning maintains a matrix called a Q-table whose row and column counts are equal to the number of feasible states of the environment and the number of actions that can be performed by the agent, respectively. Each element in the Q-table encodes the expected value (i.e., the Q-value) of its associated action when the environment is in its associated state. At each time step t, the agent performs the action that has the maximum value in the current state. The corresponding Q-table entry is then updated using Equation 10, where α and γ denote the learning rate ($0 < \alpha < 1$) and the discount factor ($0 < \gamma < 1$), respectively. The learning rate controls how rapidly Q-values change based on new observations. The discount factor determines the relative importance of future rewards over immediate rewards.

$$Q(s_t,a_t) \leftarrow (1-\alpha)\cdot Q(s_t,a_t) + \alpha\cdot(r_t + \max_a \gamma\cdot Q(s_{t+1},a)) \tag{10}$$

Despite its effectiveness and simplicity, Q-learning has a major drawback in that its memory usage for storing the Q-table becomes infeasible with modern computer systems when it is applied to solve complex problems with large numbers of states and/or actions. Unfortunately, the problem that this work aims to address belongs to such a category. For instance, the numbers of states and actions associated with the deadline-conscious energy efficiency optimization of the RN inference workload evaluated in this work are 1.1×10^{90} and 328, respectively (more details in Section 5.3.3). This indicates that the memory usage for solving the energy optimization problem for RN using Q-learning is 1.4×10^{93} bytes (assuming each Q-value requires 4 bytes), which is infeasible on any kind of modern computer systems.

To address this limitation, recent work has proposed a variant of Q-learning called deep Q-network (DQN) [109]. The main idea of DQN is to replace the aforementioned memory matrix required for storing accurate the Q-values with a deep neural network called the Q-network, which is used to approximate the Q-table. Given that the reasonably-sized Q-network can effectively approximate the Q-table, this method eliminates the excessive memory usage issue of the original Q-learning algorithm. To mitigate the instability issues caused by the correlations in the sequence of observations and the correlations between the Q and target values, Mnih et al. use the experience replay and iterative update techniques [109]. Owing to its effectiveness and practicality, we have made a design decision to employ DQN for HERTI.

5.3 Design and Implementation

HERTI is a software-based system that determines the efficient model slicing and execution plan for the target real-time inference workload on the underlying heterogeneous embedded system. Figure 18 shows its overall architecture, which mainly consists of the profiler, the execution and communication cost estimators, the model slicing and execution planner, and the runtime system.

5.3.1 Profiler

The profiler collects the total cost data (i.e., execution time and energy consumption) for executing each layer in the target inference workload on each computing device in the underlying heterogeneous embedded system. The collected total cost data is then used to construct the execution and communication cost estimators.

If a layer can be executed on a computing device and the computing device supports DVFS, the total cost data for executing the layer is collected at the minimum and maximum available frequencies of the computing device as multiple data points are needed to construct the DVFS-aware performance estimator. If a layer cannot be executed on a computing device due to a constraint (e.g., memory or functional constraint), the total costs for executing the layer on the computing device are set to an infinite value. The profiler also collects the input and output tensor sizes of each layer, which are used to estimate the communication costs associated with the layer.

5.3.2 Execution and Communication Cost Estimators

The total cost data of each layer collected by the profiler includes both the execution and communication costs. Similar to MOSAIC in Chapter IV, HERTI employs the execution and communication cost estimators. Specifically, they estimate the execution and communication costs of the layer on each heterogeneous computing device based on the total cost data and the input and output tensor sizes of the layer.

Communication Cost Estimator: Our experimental results with a micro-benchmark that undertakes communications with various tensor sizes show that the communication cost is linearly proportional to the transferred tensor size. Based on this observation, we design the communication cost estimator such that it estimates the communication costs associated with the input and output tensors of the layer l using Equations 11 and 12. Specifically, $\tau_{i,l}$, $\tau_{o,l}$, β_{d,f_d}, and δ_{d,f_d} are the total sizes of the input and output tensors of layer l and the regression coefficients for computing device d at frequency f_d.

$$C_{i,l,d,f_d} = \beta_{d,f_d} \cdot \tau_{i,l} + \delta_{d,f_d} \tag{11}$$

$$C_{o,l,d,f_d} = \beta_{d,f_d} \cdot \tau_{o,l} + \delta_{d,f_d} \tag{12}$$

Execution Cost Estimator: The execution cost estimator serves to estimate the cost for executing each layer on each computing device without including the communication costs. Specifically, the execution cost estimator first estimates the total execution cost of each layer on a computing device d running at frequency f_d using the profile data collected at the maximum and minimum frequencies of d. The execution cost estimator then subtracts the estimated communication costs of the layer (generated by the communication cost estimator) from the estimated total cost of the layer. The execution cost estimator consists of the performance and power estimators.

The performance estimator estimates the latency when executing a layer l on a computing device d at frequency f_d. The performance estimator employs a linear model shown in Equation 13, where T_{l,d,f_d}, $\theta_{l,d}$, and $\rho_{l,d}$ indicate the estimated latency and coefficients, respectively. $\theta_{l,d}$ and $\rho_{l,d}$ are computed using the data collected by the profiler at the maximum and minimum frequencies of d.

$$T_{l,d,f_d} = \theta_{l,d} \cdot \frac{1}{f_d} + \rho_{l,d} \tag{13}$$

The power estimator estimates the power consumption when executing a layer l on a computing device d at frequency f_d using Equation 14. Specifically, the power estimator decomposes the total power consumption into dynamic (i.e., $P_{\text{dynamic},l,d,f_d}$) and static (i.e., P_{static,d,f_d}) power consumption. Static power consumption is the inherent property of a computing device and independent of the characteristics of the target inference workload. Therefore, it can be profiled only once for each computing device without the need for per-workload profiling.

$$P_{l,d,f_d} = P_{\text{dynamic},l,d,f_d} + P_{\text{static},d,f_d} \tag{14}$$

Because dynamic power consumption is dependent on the characteristics of the computing device (and its frequency) and the characteristics of the layer, the power estimator estimates dynamic power consumption in order to eliminate the need for extensive offline profiling. Specifically, the power estimator uses Equation 15 to estimate the dynamic power consumption (i.e., $P_{\text{dynamic},l,d,f_d}$) of layer l on computing device d at frequency f_d (and the corresponding voltage level V_{f_d}). In Equation 15, $f_{d,\text{max}}$, $V_{f_{d,\text{max}}}$, and $P_{\text{dynamic},l,d,f_{d,\text{max}}}$ denote the maximum frequency of d, the voltage level at $f_{d,\text{max}}$, and the dynamic power consumption at $f_{d,\text{max}}$.

$$P_{\text{dynamic},l,d,f_d} = \frac{V_{f_d}^2 \cdot f_d}{V_{f_{d,\text{max}}}^2 \cdot f_{d,\text{max}}} \cdot P_{\text{dynamic},l,d,f_{d,\text{max}}} \tag{15}$$

The voltage and frequency levels of computing devices can be found in their specification documents or directly measured using instruments. Table 2 summarizes the voltage and frequency levels of the big core cluster, little core cluster, and GPU on the evaluated heterogeneous embedded system, which we found in their device tree source (DTS) files.[4] Dynamic power consumption can be estimated by substituting voltage and frequency levels into Equation 15.

Our experimental results show that the performance and power estimators achieve high accuracy.

[4]The evaluated NPU runs at 960 MHz and lacks a DVFS capability.

Specifically, the average performance and energy consumption estimation errors across all the evaluated inference workloads on the target heterogeneous embedded system are 3.0% and 4.3%, which are small.

The main reason for adopting the estimation-based approach in this work is to significantly accelerate the training process. The deep Q-network model used in HERTI could be trained by repeatedly executing the target inference workload on the underlying heterogeneous embedded system in a large number of different configurations. However, it will significantly increase the training time due to the time required to repeatedly execute the target inference workload on the heterogeneous embedded system.

To significantly reduce the training time, we have made a design decision to estimate the total costs for executing the target inference workload in each configuration. For instance, estimating the execution cost of the RN inference workload (Section 5.4) based on the estimators may be up to 14,313 times faster than executing it on the heterogeneous embedded system. Since the estimators achieve high estimation accuracy, the quality of the trained model based on the estimators is high and effectively guides to efficient states.

5.3.3 Model Slicing and Execution Planner

The main goal of the model slicing and execution planner (MSEP) of HERTI is to generate an efficient model slicing and execution plan for the target inference workload while satisfying its deadline constraint. Specifically, MSEP generates an efficient model slicing and execution plan, which specifies the number of slices, the specific layers that belong to each slice, and the specific computing device and frequency for each slice to efficiently execute the target inference workload with a strong deadline guarantee. MSEP can perform optimizations based on the user-defined metric (e.g., energy consumption, energy-delay product).

The model slicing and execution planning problem to maximize the inference efficiency on heterogeneous embedded systems with the deadline constraint is a variant of the multiple-choice quadratic knapsack problem, which is an NP-hard [80] problem. In addition, it has the Markov property (i.e., the memoryless property) in that the next state is determined solely by the current state and the action without any dependency on the previous states or actions [150]. Therefore, we formulate the model slicing and execution planning problem as a reinforcement learning (RL) problem because RL can effectively solve problems with the Markov property.

In our RL formulation, the environment is the target inference workload and the underlying heterogeneous embedded system and the agent is MSEP. The state of the environment is the current model slicing and execution plan. Equation 16 shows the state at step t. The state is expressed as a vector whose length is equal to the number of layers (i.e., Λ in Equation 16) in the target inference workload.

$$s_t = (\vec{e}_1, \vec{e}_2, \cdots, \vec{e}_\Lambda) \tag{16}$$

Each element of the state vector is also a vector, as shown in Equation 17. l ($1 \leq l \leq \Lambda$) denotes the layer index. σ ($\sigma \in \{0,1\}$) specifies whether the corresponding layer is the first layer of a slice (i.e., $\sigma = 1$) or not (i.e., $\sigma = 0$). d ($d \in D$) denotes the computing device where the layer is scheduled

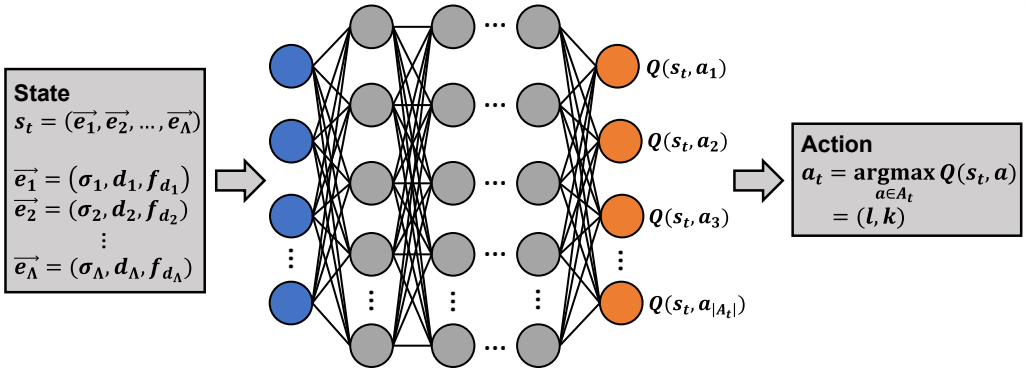


Figure 19: DQN architecture of MSEP

to execute. For heterogeneous embedded system evaluated in this work, the device set D is equal to $\{\text{big}, \text{little}, \text{GPU}, \text{NPU}\}$. f_d ($f_d \in F_d$) denotes the frequency of device d and F_d specifies the available frequencies of d.

$$\vec{e}_l = (\sigma, d, f_d) \tag{17}$$

At each step t, the agent (i.e., MSEP) interacts with the environment (i.e., the target inference workload and the underlying heterogeneous embedded system) by performing an action. We design the agent to choose a single layer (instead of multiple layers) to perform an action. This is done to ensure that the state space is gradually explored. Equation 18 shows the action performed by the agent at step t.

$$a_t = (l, k) \tag{18}$$

Specifically, l ($1 \leq l \leq \Lambda$) indicates the layer index selected to perform the action. k ($k \in K$) denotes the specific change associated with the action. For each layer, the agent can apply one of the changes (i.e., K) defined in Equation 19 – creating a new slice starting from the layer (i.e., k_{split}), merging the slice where the layer belongs with the previous slice (i.e., k_{merge}), changing the computing device by traversing the device list in either direction (i.e., $k_{d,\text{next}}$ or $k_{d,\text{prev}}$), or increasing (i.e., $k_{f_d,\text{up}}$) or decreasing (i.e., $k_{f_d,\text{down}}$) the computing device frequency by one level.

$$K = \{k_{\text{split}}, k_{\text{merge}}, k_{d,\text{next}}, k_{d,\text{prev}}, k_{f_d,\text{up}}, k_{f_d,\text{down}}\} \tag{19}$$

MSEP employs the RL algorithm (i.e., deep Q-network (DQN) [109]) to solve the model slicing and execution planning problem. Figure 19 shows the DQN architecture of MSEP, which mainly comprises a fully connected layer and ReLU functions.

Algorithm 2 shows the pseudocode for MSEP. After initializing key variables, MSEP iterates the main loop to train the DQN model (Lines 7–23). Without loss of generality, we discuss the main logic of MSEP in the context of step t.

At step t, MSEP chooses the most feasible action (i.e., a_t) to perform based on the Q-network with a probability of $1 - \varepsilon(t)$ (Line 12 in Algorithm 2). To prevent the algorithm from converging to a

Algorithm 2 The findEfficientSlicingAndExecPlan function

```
1:  procedure FINDEFFICIENTSLICINGANDEXECPLAN(Λ)
2:      Q ← initializeQNetwork(Λ, depth, width)
3:      M_r ← initializeReplayMemory()
4:      s_1 ← s_init
5:      s_best ← s_init
6:      r_best ← −Φ
7:      for t ← 1 to t_max do
8:          A_t ← getValidActions(s_t)
9:          if generateRandomNumber() < ε(t) then
10:             a_t ← getRandomAction(A_t)
11:         else
12:             a_t ← argmax_{a∈A_t} Q(s_t, a)
13:         end if
14:         r_t ← calculateImmediateReward(s_t, a_t)
15:         s_{t+1} ← applyAction(s_t, a_t)
16:         if r_t > r_best then
17:             r_best ← r_t
18:             s_best ← s_{t+1}
19:         end if
20:         M_r ← M_r ∪ {(s_t, a_t, r_t, s_{t+1})}
21:         B ← sampleMiniBatch(M_r)
22:         trainAndUpdateQNetwork(Q, B, t)
23:     end for
24:     return s_best
25: end procedure
```

local optimum, MSEP randomly (instead of consulting the Q-network) chooses a feasible action with a probability of $\varepsilon(t)$ (Line 10). As t increases, $\varepsilon(t)$ decreases to facilitate choosing the action based on the trained Q-network.

MSEP then computes the immediate reward (i.e., r_t) associated with the state transition caused by performing the action by invoking the `calculateImmediateReward` function shown in Algorithm 3 (Line 14 in Algorithm 2). The `calculateImmediateReward` function uses the execution and communication cost estimators discussed in Section 5.3.2 to compute the immediate reward (Lines 3–11 in Algorithm 3).

The reward computation equation is shown in Equation 20, where Γ_s denotes the cost (e.g., energy consumption, energy-delay product) associated with state s and Φ indicates the penalty, which is set to 100, for states that violate the deadline constraint. If the deadline constraint of the target inference workload is satisfied in state s, the reward is set to $-\sqrt{\Gamma_s}$, which increases as the cost decreases. Otherwise, the reward is set to Φ to exclude states in which the deadline constraint is violated.

$$r_s = \begin{cases} -\sqrt{\Gamma_s} & \text{if } T_s \leq T_{\text{deadline}} \\ -\Phi & \text{otherwise} \end{cases} \tag{20}$$

MSEP determines the next state (i.e., s_{t+1}) and memorizes it if the associated immediate reward is

Algorithm 3 The calculateImmediateReward function

```
1:  procedure CALCULATEIMMEDIATEREWARD(s, a)
2:      s' ← applyAction(s, a)
3:      C_s' ← estimateCommunicationCost(s')
4:      T_s' ← estimatePerformance(s')
5:      P_s' ← estimatePowerConsumption(s')
6:      Γ_s' ← calculateCost(C_s', T_s', P_s')
7:      if T_s' ≤ T_deadline then
8:          r_s' ← −√Γ_s'
9:      else
10:         r_s' ← −Φ
11:     end if
12:     return r_s'
13: end procedure
```

Table 5: Tunable hyper-parameters

Hyper-parameter	Candidate values	Tuned value
Learning rate	10^{-5}, 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}	10^{-3}
Discount factor	0.9, 0.99	0.9
Q-network depth	2, 4, 8, 16	4
Q-network width	32, 64, 128, 256, 512, 1024	512
Batch size	32, 64	32
Epsilon in Adam optimizer	10^{-8}, 10^{-7}, 10^{-6}, 10^{-5}, 10^{-4}	10^{-5}

higher than the highest reward that has been discovered up to this point (Lines 15–19 in Algorithm 2). MSEP then inserts the tuple of (s_t, a_t, r_t, s_{t+1}) into the replay memory (Line 20). The replay memory is the key data structure enabling the experience replay mechanism of DQN [109]. Specifically, the replay memory contains the previous observations (i.e., the immediate rewards received by previous state transitions), which are used to train the Q-network.

MSEP generates a batch of training data by randomly choosing samples from the replay memory (Line 21 in Algorithm 2). MSEP then trains the Q-network using the batch of training data (Line 22). Because the Q-network is continuously trained using the previous observations, it can help MSEP to perform more effective actions in subsequent steps. MSEP repeats this process until a predefined iteration count is reached and returns the best state with the highest reward (Lines 7–24).

MSEP employs six tunable hyper-parameters. We use the Hyperband algorithm [96, 97], which is one of the most widely used and effective algorithms, to automatically tune the hyper-parameters. Table 5 summarizes the six hyper-parameters and their candidate values explored by the tuner. Table 5 also shows the values of the hyper-parameters, which have been tuned using the MN-1.3 inference workload and its medium deadline (Table 6). We have made a design decision to use only a single inference workload and a single deadline for hyper-parameter tuning in order to eliminate the need for per-workload and/or per-deadline hyper-parameter tuning. Section 5.5.4 quantifies the generality of this approach.

Table 6: Evaluated real-time inference workloads

Workload	Accuracy	Size (MB)	Layers	`short`/`medium`/`long` deadlines (milliseconds)
IN [152]	80.1%	183.7	20	190/210/250
MN-1.0 [153]	74.1%	321.9	20	85/100/130
MN-1.3 [153]	75.2%	511.6	20	105/120/150
MO-1.3 [138]	74.4%	187.8	18	75/90/120
MO-1.4 [138]	75.0%	214.7	18	85/100/130
RN [67]	77.8%	260.4	53	155/170/200
VGG [142]	71.5%	407.4	16	135/150/180

5.3.4 Runtime System

The runtime system of HERTI is a user-level process that executes the model slices of the target inference workload across the heterogeneous computing devices based on the model slicing and execution plan generated by the model slicing and execution planner. The runtime system is implemented in the C++ programming language based on the TensorFlow Lite framework [23] on the Android OS because it supports various heterogeneous computing devices and provides a well-established programming environment. However, since HERTI employs a framework-agnostic approach for slicing and executing the target inference workload, we believe that HERTI can be applied widely to other representative deep-learning frameworks as well, such as PyTorch [122].

5.4 Experimental Methodology

Inference Workloads: Table 6 summarizes the evaluated inference workloads – Inception V4 (IN) [152], MnasNet with the width parameters (p_W) of 1.0 (MN-1.0) and 1.3 (MN-1.3) [153], MobileNet V2 with $p_W = 1.3$ (MO-1.3) and $p_W = 1.4$ (MO-1.4) [138], ResNet V2 (RN) [67], and VGG (VGG) [142]. The evaluated inference workloads achieve high accuracy (e.g., the Top-1 accuracy with the ImageNet dataset [135] in Table 6) and exhibit widely-different characteristics such as the model size (i.e., the memory usage reported by TensorFlow Lite), the layer count, and the inference latency.

MobileNet V2 and MnasNet provide a mechanism to tune their inference accuracy and latency using the width parameter (i.e., p_W), which controls the number of channels. When it is set to a larger value, their inference accuracy and latency tend to increase.

The deadline is determined using Equation 21, where w, ζ, $T_{w,\text{best}}$, and $T_{w,\text{energy\_optimized}}$ denote the inference workload, a scale factor, and the latencies that incur the highest performance (i.e., the best-case execution time) and the lowest energy consumption that are determined by the technique proposed in [61] (more details in Section 5.5.1), respectively. As shown in Table 6, we use three deadlines (i.e., `short` ($\zeta = 0.25$), `medium` ($\zeta = 0.5$), and `long` ($\zeta = 1.0$)) for each inference workload to investigate the sensitivity of HERTI to the inference deadline.

$$\text{Deadline}_w = T_{w,\text{best}} + \zeta \cdot (T_{w,\text{energy\_optimized}} - T_{w,\text{best}}) \tag{21}$$

Heterogeneous Embedded System: To investigate the effectiveness of HERTI, we use a heterogeneous embedded system, the HiKey 970 development board [8]. The evaluated system is equipped with the Kirin 970 mobile processor [15] which consists of a CPU with four Cortex-A73 (big) cores, four Cortex-A53 (little) cores, a Mali-G72 GPU, and an NPU. Table 2 summarizes their available voltage and frequency levels.

Due to the memory constraint, the NPU cannot execute a model slice larger than 100 MB [8]. The exact memory constraints of the big core cluster, little core cluster, and GPU are unknown. However, they can successfully execute all the evaluated inference workloads without any memory capacity issues. Therefore, we assume that their memory constraints are sufficiently large for the evaluated inference workloads.

With regard to the system software stack, Android 8.1 is installed on the evaluated heterogeneous embedded system. In addition, HERTI and the evaluated inference workloads are implemented in the TensorFlow Lite 1.11.0 [23].

We measure the inference latency through the `high_resolution_clock` C++ standard library function. We collect the inference energy consumption data using an external power monitor [7]. The power monitor periodically samples the voltage and current applied to the evaluated heterogeneous embedded system at the rate of 5000 samples per second. Figure 2 shows the evaluated heterogeneous embedded system and the power monitor.

Training Server System: To train the DQN model of HERTI, we use a server system equipped with two Intel Xeon E5-2640 16-core CPUs running at 2.6 GHz and 32 GB memory. The server system is installed with Ubuntu 18.04 and TensorFlow 2.3.1 [26].

5.5 Evaluation

5.5.1 Overview

This section quantifies the effectiveness of HERTI. Specifically, we aim to investigate (1) the inference latency, (2) the inference energy, (3) the sensitivity of the efficiency gains to the inference deadline and the system heterogeneity, (4) the generality, (5) the energy-delay product (EDP) efficiency, and (6) the training time reduction through the estimation-based approach.

For each inference workload, we evaluate 13 versions – the big core cluster-preferred (`TF-BIG-P`), little core cluster-preferred (`TF-LITTLE-P`), GPU-preferred (`TF-GPU-P`), NPU-preferred (`TF-NPU-P`) with the performance governor of the Android OS, the big core cluster-preferred (`TF-BIG-O`), little core cluster-preferred (`TF-LITTLE-O`), GPU-preferred (`TF-GPU-O`), NPU-preferred (`TF-NPU-O`) with the on-demand governor of the Android OS, MOSAIC configured to conduct performance optimization (`MOSAIC-P`), MOSAIC configured to perform energy optimization (`MOSAIC-E`), AutoScale, exhaustive, and HERTI versions.

With the big core cluster-, little core cluster-, GPU-, and NPU-preferred versions, the slices of each inference workload are executed on the corresponding preferred computing device. To generate the big core cluster-, little core cluster-, GPU-, and NPU-preferred versions, we include as many consecutive

layers as possible in each slice if they satisfy the memory and functionality constraints. If a layer cannot be included in the current slice due to a memory constraint, a new slice is created and the layer is added to the new slice, which is executed on the preferred device. If a layer cannot be executed on the preferred device due to a memory or functionality constraint, a separate slice is created for the layer and executed on a feasible device among NPU, GPU, and big core cluster (in the preference order).

The performance governor tends to deliver higher performance and lower energy efficiency than the on-demand governor because the performance governor always executes the target inference workload at the maximum frequency of the underlying computing device. In contrast, because the on-demand governor, which is the default governor of the Android OS, performs DVFS based on the dynamic load of the target inference workload, it typically exhibits lower performance and higher energy efficiency than the performance governor.

MOSAIC is a technique in Chapter IV that generates the optimal model slicing and execution plan of the target inference workload on heterogeneous embedded system in terms of the user-specified metric. However, it is deadline-oblivious and provides no deadline guarantee for real-time inference workloads. MOSAIC can perform optimizations based on various user-specified metrics. For the `MOSAIC-P` and `MOSAIC-E` versions, MOSAIC is configured to generate the optimal model slicing and execution plan in terms of inference latency and energy efficiency, respectively. While the `MOSAIC-E` version lacks a deadline guarantee, its energy efficiency is always optimal. As quantified in this work, when a deadline is configured in a way that the `MOSAIC-E` version also happens to meet the deadline, HERTI achieves the same efficiency (i.e., the optimal energy efficiency) as the `MOSAIC-E` version, demonstrating its effectiveness.

AutoScale is a state-of-the-art technique that executes the target inference workload in a deadline-conscious manner and employs DVFS to enhance inference efficiency based on machine learning [84]. However, it lacks the consideration of the heterogeneity- and constraint-aware model slicing and execution for the target inference workload and executes the entire inference workload on a single device. In other words, AutoScale cannot exploit the optimization opportunities for slicing and executing the model at the layer level.

The exhaustive version empirically determines the model slicing and execution plan that exhibits the highest inference efficiency through exhaustive search. Note that exhaustive search is highly time- and resource-consuming and impractical. For example, it is estimated to take 6.1×10^{11} years to empirically find the best model slicing and execution plan for MN-1.3 through exhaustive search even using the lightweight execution and communication cost estimators. The experimental data for the exhaustive version of each inference workload has been collected for ten days, covering up to 3.7×10^{9} states. Finally, the HERTI version uses HERTI to generate the efficient model slicing and execution plan.

5.5.2 Inference Latency and Energy Efficiency

In this section, we evaluate the effectiveness of HERTI when it is configured to optimize the energy efficiency of the inference workload while satisfying its deadline constraint. The deadline for each

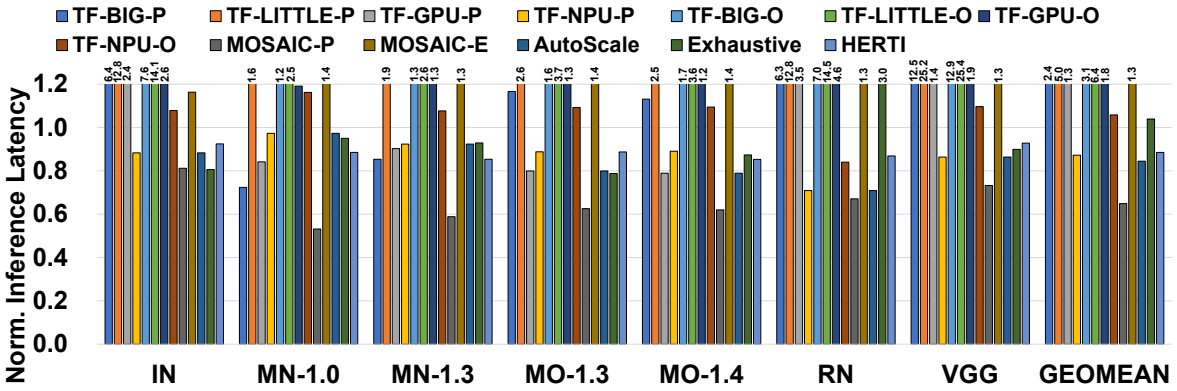


Figure 20: Inference latency

inference workload is set to the medium case shown in Table 6.

We first investigate the effectiveness of HERTI in terms of inference latency. Figure 20 shows the inference latency of each version of the inference workload, normalized to the deadline.[5] If the normalized inference latency of a certain version is 1 or lower, the version meets the deadline. Otherwise, it fails to satisfy the deadline constraint.

First, HERTI robustly satisfies the deadline constraint across all of the evaluated inference workloads, which empirically demonstrates that HERTI provides a strong deadline guarantee for real-time inference workloads. Interestingly, HERTI chooses states that lead to the inference latency lower than the deadline (i.e., the normalized latency is lower than 1). This is mainly because such states result in higher energy efficiency than the other states that incur inference latency close to the deadline because such states facilitate the use of the evaluated NPU that lacks DVFS support (but still more efficient than other devices in many cases) and reduce static energy consumption.

Second, aside from HERTI, only the `TF-NPU-P`, `MOSAIC-P`, and AutoScale versions meet the deadlines across the evaluated inference workloads. Because the NPU is effective for reducing the inference latency and the performance governor of the Android OS is optimized for performance, the `TF-NPU-P` version meets the inference deadline. As the `MOSAIC-P` version conducts performance optimizations without considering the deadline of each inference workload, it incurs the latency significantly lower than the other versions at the cost of reduced inference efficiency. In addition, the AutoScale version meets the inference deadline by finding a single device and its DVFS setting and executing the entire inference workload (i.e., no model slicing) on the device.

Third, all the other versions fail to meet the deadlines of the inference workloads. The big core cluster-, little core cluster-, and GPU-preferred and `TF-NPU-O` versions fail to meet the inference deadline because they lack the capability of exploiting heterogeneity in the underlying system and the Android OS is deadline-oblivious. The `MOSAIC-E` version fails to meet the inference deadline as it performs optimizations solely to reduce the inference energy consumption without considering the deadline constraint. Despite the extensive search conducted for each workload for ten days, the exhaustive version still fails to meet the deadline. Due to the excessively large state space, the optimal state still remains

[5] All the reported experimental data is the average of 100 experiments.

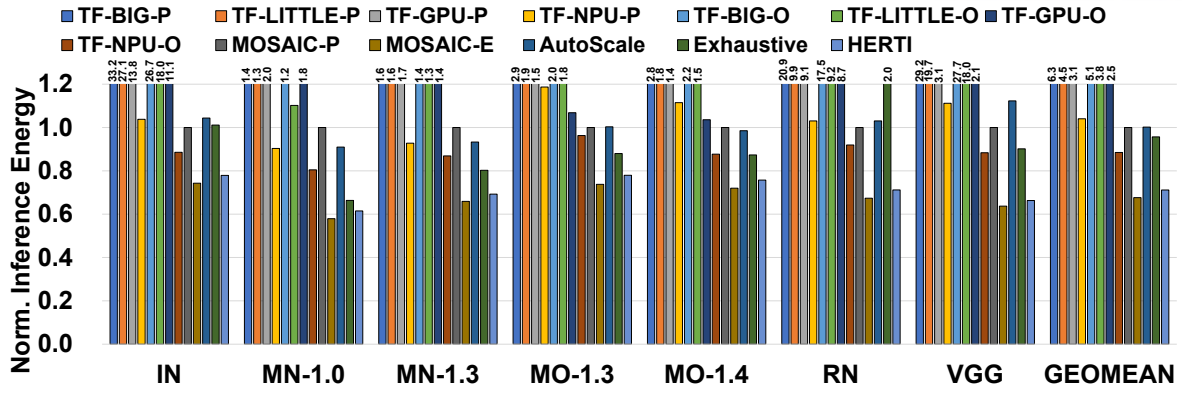


Figure 21: Inference energy

Table 7: Model slicing and execution plans

Workload	Model slicing and execution plan
IN	N_{1}^{960}, N_{11}^{960}, L_{20}^{1690}
MN-1.0	N_{1}^{960}, L_{14}^{1690}, N_{18}^{960}, L_{20}^{1402}
MN-1.3	N_{1}^{960}, B_{9}^{1210}, G_{12}^{332}, B_{14}^{1210}, G_{18}^{332}, B_{20}^{1210}
MO-1.3	N_{1}^{960}, G_{8}^{550}, L_{15}^{1402}, N_{18}^{960}
MO-1.4	N_{1}^{960}, G_{9}^{667}, L_{15}^{1690}, N_{18}^{960}
RN	B_{1}^{1863}, N_{2}^{960}, L_{4}^{1690}, N_{5}^{960}, B_{12}^{1364}, N_{13}^{960}, N_{30}^{960}, B_{48}^{1364}, N_{49}^{960}, B_{53}^{1863}
VGG	N_{1}^{960}, G_{14}^{332}, G_{15}^{237}, B_{16}^{682}

undiscovered even after the extensive search.

We now investigate the effectiveness of HERTI in terms of energy efficiency. Figure 21 shows the energy consumption of each version, normalized to that of the `MOSAIC-P` version. First, we observe that HERTI significantly outperforms other versions in terms of energy efficiency. For instance, HERTI consumes 28.0% and 28.4% lower energy than the `MOSAIC-P` and AutoScale versions, respectively. Our experimental results provide empirical evidence that HERTI robustly finds the optimal state that minimizes the energy consumption of each inference workload with a strong deadline guarantee through reinforcement learning.

Second, the `MOSAIC-E` version exhibits slightly lower energy consumption than the HERTI version when the inference deadline is set to the medium one. This is mainly because the `MOSAIC-E` version aggressively applies DVFS by solely focusing on minimizing the energy consumption without considering the inference deadline. Hence, as discussed above (Figure 20), the `MOSAIC-E` version fails to meet the deadlines of all the inference workloads.

In contrast, since HERTI performs inference energy efficiency optimizations in a deadline-conscious manner, it achieves the energy efficiency comparable to that of the `MOSAIC-E` version while robustly satisfying the deadline constraint for all the evaluated inference workloads. Further, as quantified in Section 5.5.3, our experimental results show that HERTI achieves energy efficiency identical to that of the `MOSAIC-E` version when given a sufficiently long deadline with which the `MOSAIC-E` version happens to meet the inference deadline.

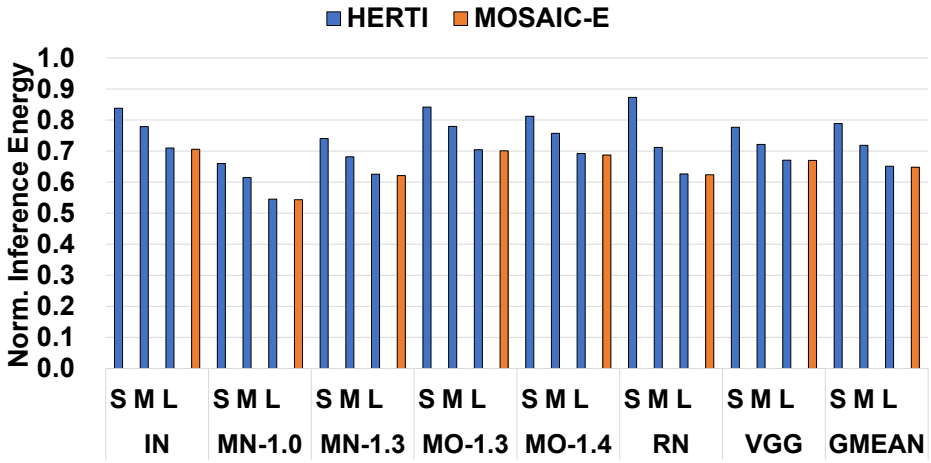


Figure 22: Sensitivity to the inference deadline

Table 7 summarizes the model slicing and execution plans generated by HERTI for deadline-conscious energy optimization. Each letter (i.e., big core cluster (B), little core cluster (L), GPU (G), and NPU (N)) denotes a slice and its associated device. The subscript and the superscript to each letter represent the ID of the first layer of the corresponding slice and the frequency of the computing device in MHz, respectively. HERTI tends to execute slices with relatively-low computation intensity and large communication overheads on the big or little cluster, computation-intensive slices with functional/memory constraints on the GPU, and computation-intensive slices without any constraints on the NPU.

5.5.3 Sensitivity

In this section, we analyze the sensitivity of the efficiency gains of HERTI to the inference deadline and the degree of the heterogeneity of the underlying system. Figure 22 shows the energy consumption of HERTI for each inference workload, normalized to that of the `MOSAIC-P` version. Each bar represents the energy consumption when the inference deadline is configured to the `short` (S), `medium` (M), or `long` (L) deadline in Table 6. Figure 22 also shows the normalized energy efficiency of the `MOSAIC-E` version. Because the `MOSAIC-E` version is deadline-oblivious, we only show one bar for the `MOSAIC-E` version with each inference workload.

First, as the inference deadline increases, the energy efficiency gains of HERTI increase across all the inference workloads. For instance, the average energy efficiency gain increases by 14.1% as the inference deadline increases from the `short` deadline to the `long` deadline. This is mainly because the more relaxed constraint with a longer inference deadline allows HERTI to explore and find states that result in higher energy efficiency but still meet the inference deadline.

Second, with the `long` deadline, HERTI achieves the same energy efficiency as the `MOSAIC-E` version across all the inference workloads. While the `MOSAIC-E` version is deadline oblivious and lacks a deadline guarantee, the `MOSAIC-E` version happens to meet the inference deadline when the inference deadline for each inference workload is set to the `long` deadline. In this case, HERTI robustly generates the same model slicing and execution plan generated by the `MOSAIC-E` version, which is optimal.

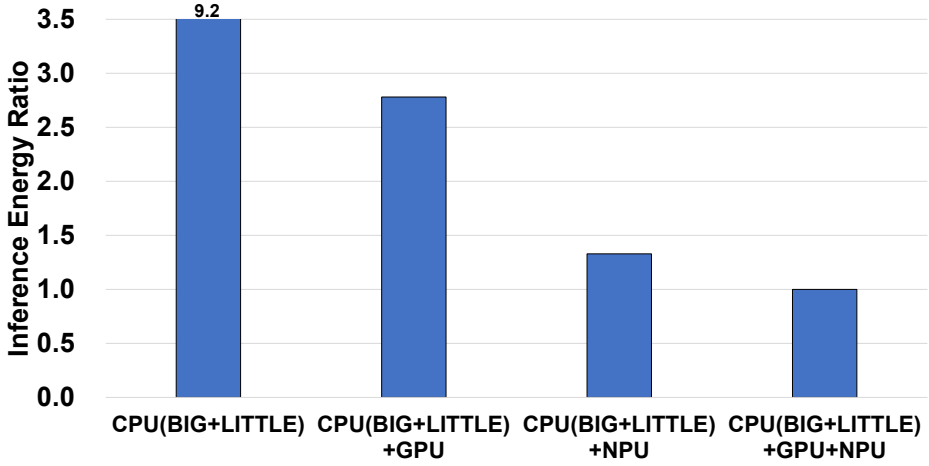


Figure 23: Sensitivity to the system heterogeneity

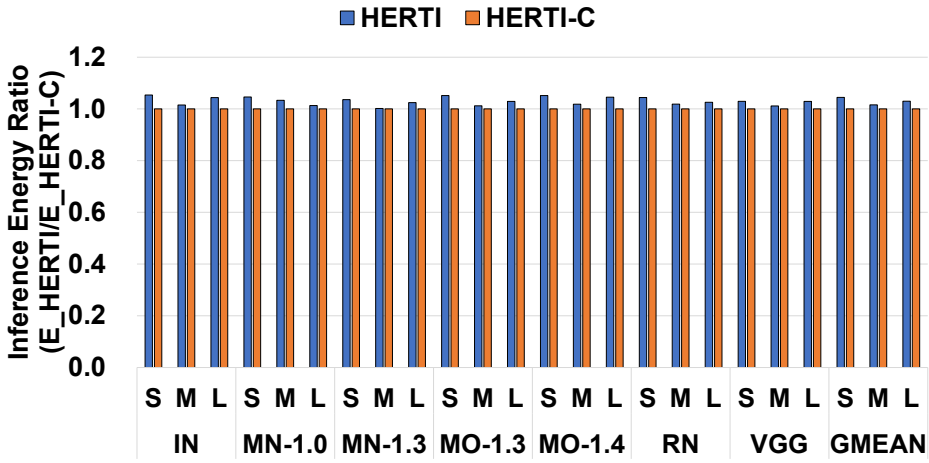


Figure 24: Generality of HERTI

We investigate the sensitivity of the efficiency gains of HERTI to the degree of the heterogeneity of the underlying system. To this end, we have created synthetic versions of HERTI, which only employ the CPU (i.e., the big and little core clusters), the CPU and another device (e.g., GPU or NPU), and all the computing devices (i.e., the full version of HERTI). Figure 23 shows the energy efficiency gain of each version, normalized to that of the full version of HERTI.[6]

We observe that the HERTI continues to achieve higher energy efficiency with more heterogeneous computing devices by effectively utilizing all the available devices with a strong deadline guarantee. Our experimental results also demonstrate the importance of heterogeneity-aware systems for maximizing the efficiency of real-time inference workloads on heterogeneous embedded systems.

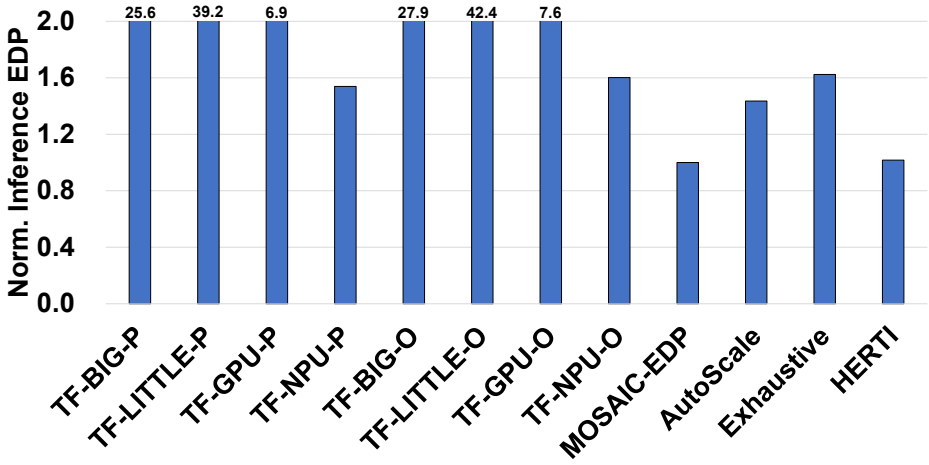


Figure 25: Energy-delay product

5.5.4 Generality

The hyper-parameters of the DQN model used in HERTI have been tuned using the MN-1.3 inference workload and its medium deadline. This design decision has been made to eliminate the need for per-workload and/or per-deadline hyper-parameter tuning. To quantify the generality of this approach, we compare the energy efficiency of the default version of HERTI with a highly customized version (i.e., `HERTI-C`) of HERTI whose hyper-parameters have been tuned for each pair of the inference workload and its target deadline. Figure 24 reports the energy efficiency of the two versions, normalized to the `HERTI-C` version.

We observe that the default version of HERTI achieves the energy efficiency similar to that of the `HERTI-C` version. For instance, the average energy consumption of the `HERTI-C` version is 2.9% lower than the default version of HERTI. Our experimental results demonstrate that HERTI exhibits strong generality in that it achieves high efficiency across all of the evaluated inference workloads and scenarios without requiring per-workload or per-deadline hyper-parameter tuning. HERTI inherits this generality property from RL, which can be widely applied to various tasks with a set of hyper-parameters tuned for a single task.

5.5.5 Energy-Delay Product Efficiency

So far, we have evaluated the effectiveness of HERTI in terms of energy efficiency. HERTI is versatile in that it can perform optimizations in various user-specified metrics. To evaluate the versatility of HERTI, we configure HERTI to perform optimizations based on the energy-delay product (EDP) metric while meeting the deadline constraint. For conciseness, Figure 25 shows the EDP of each version, which is averaged (i.e., geometric mean) across all the inference workloads and normalized to that of the `MOSAIC-EDP` version. While MOSAIC is deadline-oblivious, the EDP of the `MOSAIC-EDP` version is

[6]While omitted for conciseness, our experimental results show that HERTI outperforms MOSAIC-P and AutoScale by 20.6% and 11.8% when only CPU and GPU are used and by 21.7% and 13.3% when only CPU and NPU are used in terms of energy efficiency.

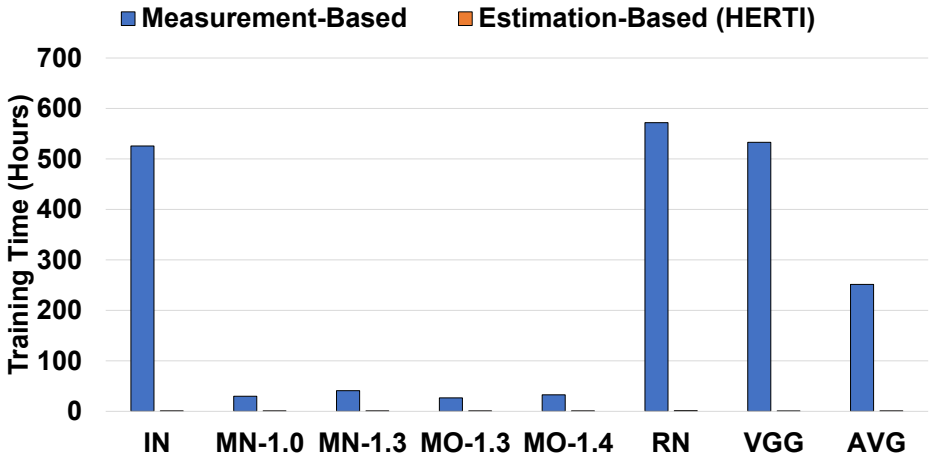


Figure 26: Training time comparison

optimal.

We observe that HERTI achieves the optimal EDP efficiency (i.e., same as the `MOSAIC-EDP` version) and significantly outperforms other versions. While the inference latency data is omitted for brevity, our experimental results demonstrate that HERTI robustly satisfies the deadline constraint across all of the inference workloads when configured to perform EDP optimizations.

5.5.6 Training Time

HERTI employs the estimation-based approach to significantly reduce the time spent for training the DQN model. To quantify the effectiveness of this approach, we compare the training time of the estimation-based approach with that of the measurement-based approach. Since the measurement-based approach incurs excessively long training time for each inference workload, we were unable to measure its training time. To estimate the training time of the measurement-based approach, we first conduct training based on the estimation-based approach. We then compute the estimated training time with the measurement-based approach by summing the estimated execution times of all the explored states, which are estimated by the execution and communication cost estimators (Section 5.3.2).

Figure 26 shows the training time of the estimation-based and measurement-based approaches for deadline-conscious energy optimization. First, the estimation-based approach significantly reduces the training time in comparison with the measurement-based approach. For instance, the estimation-based approach (i.e., 48 minutes) incurs 314 times shorter training time than the measurement-based approach (i.e., 15,088 minutes (251 hours)) on average across the evaluated inference workloads. Because executing the inference workload on the heterogeneous embedded system takes drastically longer time than estimating its execution and communication costs using the lightweight estimators, the estimation-based approach significantly reduces the training time.

Second, the estimation-based approach leads to more significant reduction in training time with some inference workloads such as `IN` and `RN`. This is mainly because such inference workloads are more complex than the others, requiring longer execution time on the underlying heterogeneous embedded

system with the measurement-based approach.

Overall, our experimental results clearly demonstrate the effectiveness of HERTI in that it achieves high inference efficiency in terms of both energy and EDP metrics while satisfying the deadline constraint, effectively translates longer inference deadlines and a higher degree of heterogeneity of the underlying system into higher efficiency, exhibits strong generality with respect to hyper-parameter tuning, and significantly reduces the training time through its estimation-based approach across all the evaluated inference workloads and scenarios.

5.6 Summary

This chapter presents HERTI, a reinforcement learning (RL)-augmented system for efficient real-time inference on heterogeneous embedded systems. HERTI employs the accurate and efficient execution and communication cost estimators to significantly accelerate the training process. HERTI builds on the state-of-the-art RL algorithm (i.e., deep Q-network) to eliminate the state space explosion problem. HERTI efficiently explores the state space with heterogeneity and constraint awareness and robustly generates the efficient state for the target inference workload with a strong deadline guarantee through RL. Our quantitative evaluation based on the widely-used inference workloads and heterogeneous embedded system demonstrates the effectiveness of HERTI in the sense that it achieves high efficiency and significantly outperforms the deadline-conscious state-of-the-art technique (i.e., AutoScale) in multiple metrics (i.e., energy, energy-delay product) with a strong deadline guarantee, robustly satisfies the deadline constraint in contrast to the deadline-oblivious state-of-the-art technique (i.e., MOSAIC), delivers larger efficiency gains as the inference deadline and the system heterogeneity increase, exhibits the strong generality for hyper-parameter tuning, and significantly reduces the training time based on its estimation-based approach across all the evaluated inference workloads and scenarios.

VI Hotness- and Lifetime-Aware Data Placement and Migration for High-Performance Deep-Learning on Heterogeneous Memory Systems

6.1 Introduction

Heterogeneous memory systems that consist of memory nodes with a wide range of architectural characteristics are rapidly emerging as a promising solution in various computing domains ranging from embedded [48] to high-performance computing [87]. The key idea behind the heterogeneous memory systems is to provide useful properties such as performance, energy efficiency, durability, and cost efficiency that cannot be achieved by solely employing a single type of memory.

For instance, the Intel Xeon Phi KNL architecture consists of high-bandwidth memory (HBM) and DRAM nodes that are optimized for high bandwidth (i.e., HBM) and large capacity (i.e., DRAM), respectively. It is the responsibility of the underlying system software and/or hardware to effectively manage the heterogeneous memory nodes in the underlying system to maximize the metric of interest such as performance and energy efficiency.

Because deep learning (DL) is one of the representative workloads in a variety of computing domains, it is crucial to investigate efficient system software support to optimize the performance of DL on heterogeneous memory systems. In particular, as a large number of widely-used DL algorithms employ advanced network architectures where tensors exhibit significantly different characteristics [158], the system software support that robustly performs optimizations considering the characteristics of the tensors and heterogeneous memory nodes is important for high-performance DL. Despite the extensive prior works [26, 45, 75, 108, 158, 171] on the system software and architectural support for high-performance DL, it still remains unexplored to investigate the design and implementation of the effective memory management techniques to improve the performance of DL on heterogeneous memory systems.

To bridge this gap, this chapter[7] proposes HALO, hotness- and lifetime-aware data placement and migration for high-performance DL on heterogeneous memory systems. HALO extracts the hotness and lifetime information on the tensors of the target DL application based on its dataflow graph. HALO dynamically places and migrates the tensors on heterogeneous memory nodes based on their hotness and lifetime characteristics. We implement HALO based on the widely-used TensorFlow system [26] and demonstrate that it significantly improves the performance and energy efficiency of various DL applications on a real heterogeneous memory system without any hardware modifications.

Specifically, this work makes the following contributions:

- We present an in-depth characterization of widely-used DL applications in terms of the execution time and tensor characteristics on a full heterogeneous memory system. Our characterization results show that tensors of the evaluated DL applications exhibit disparate characteristics in terms of the capacity, hotness, and lifetime.

- Guided by the characterization results, we propose HALO, hotness- and lifetime-aware data place-

[7] The work presented in this chapter was also published in [60]

ment and migration for high-performance DL on heterogeneous memory systems. HALO analyzes the hotness and lifetime characteristics of the tensors of the target DL application without requiring any offline profiling.[8] We formulate the tensor placement and migration problem as a variant of the NP-hard knapsack problem with time intervals [34] and propose an efficient algorithm to address it. HALO dynamically places and migrates the tensors across the heterogeneous memory nodes in a hotness- and lifetime-aware manner based on the proposed algorithm.

- We present the design and implementation of HALO by extending TensorFlow [26], which is one of the most widely-used machine-learning systems. We demonstrate that the proposed techniques can be effectively implemented on top of the production-quality machine-learning system.

- We quantify the effectiveness of HALO using a full heterogeneous memory system and representative DL applications including the state-of-the-art convolutional neural networks (CNNs), a recurrent neural network (RNN), and a network with an attention mechanism [3, 10, 12, 22, 89, 142, 151, 156, 170]. Our quantitative evaluation demonstrates the effectiveness of HALO in that it significantly outperforms various memory management policies (e.g., 28.2% higher performance than the HBM-Preferred policy) supported by the underlying system software and hardware, delivers the performance comparable to that of the ideal case with infinite HBM, incurs small performance overheads, and achieves high performance across various application working-set sizes. To the best of our knowledge, our work is the first to propose, implement, and evaluate the hotness- and lifetime-aware memory management technique for high-performance DL on heterogeneous memory systems.

The rest of this chapter is organized as follows. Section 6.2 provides the background information for this work. Section 6.3 describes the experimental methodology. Section 6.4 investigates the characteristics of deep-learning applications on a full heterogeneous memory system. Section 6.5 presents the design and implementation of HALO. Section 6.6 quantifies the effectiveness of HALO. Section 6.7 concludes the chapter with a summary.

6.2 Background

6.2.1 TensorFlow Machine-Learning System

TensorFlow is a widely-used machine-learning (ML) system [26]. It allows for programmers to express their ML algorithms as dataflow graphs. A *dataflow graph* mainly consists of tensors and operations. *Tensors* are multidimensional arrays, whose elements have one of the basic primitive data types such as `float32`. An *operation* takes zero or more input tensors and produces zero or more output tensors [26].

The dataflow graph of the target TensorFlow application is a directed acyclic graph, where the nodes and edges represent the operations and tensors. The source (or destination) node of an edge represents

[8] Intuitively, the hotness and lifetime of a tensor represent the number of operations that access the tensor and the difference between its deallocation and allocation times (see Section 6.2.4).

(a) Linear (b) Non-linear

Figure 27: Networks with linear and non-linear connections

the producer (or consumer) of the corresponding tensor. The nodes in ML applications are typically executed in a sequential manner as the layers in their network architectures are sequentially connected.

The dataflow graph of the target TensorFlow application remains unchanged through the execution of the application. Therefore, static analysis and optimization techniques based on the dataflow graph of the target TensorFlow application are widely-used and effective. As discussed in Section 6.5, HALO also builds on static analysis and optimization techniques.

A recent work classifies network architecture types into *linear* and *non-linear* network architectures [158]. Figure 27 shows the network architectures with linear and non-linear architectures. With the linear network architecture, edges exist only between consecutive nodes in the sequential execution order of the operations. In contrast, with the non-linear network architecture, edges may exist between non-consecutive nodes in the sequential execution order. This indicates that the variance in the hotness and lifetime of the tensors tends to be larger with the non-linear network architecture as some tensors can be consumed by more operations than the others.

We focus on optimizing the efficiency of the training phase of deep-learning (DL) applications. In each *training epoch* (or epoch), a DL application iterates all of the training data to train its model. In each *training step* (or step), the DL application processes a batch of the training data. For example, if 70,000 images are used as the training data with a batch size of 100, an epoch consists of 700 steps.

We implement and evaluate HALO by extending TensorFlow owing to its widespread use and well-established development environment. However, we believe that the design of HALO is sufficiently generic to be applicable to other ML platforms such as Caffe [75] and PyTorch [123].

6.2.2 NUMA-Aware Memory Policies

Non-uniform memory access (NUMA) systems consist of multiple *processor nodes*, each of which having a CPU and local memory with one or more DIMMs. Processor nodes are connected via an interconnection network (ICN). Modern NUMA systems typically support cache coherence across the processor nodes to provide an abstraction of a single globally addressable memory space.

Accesses to a memory object on a NUMA system are largely classified into two categories – *local* and *remote* accesses. A local (or remote) access occurs when a processor node accesses a memory object that is located on the same (or different) processor node. Remote accesses incur a longer latency than that of local accesses because they generate packets that are transmitted over the ICN. Thus, frequent remote accesses can significantly degrade the overall performance of NUMA systems [55].

The Linux kernel provides several NUMA-aware memory placement policies including the follow-

ing three representative policies – the *local*, *interleave*, and *preferred* policies. The local policy places a page in the local memory of the processor node where the task that first touches the page is running. The interleave policy places pages across the processor nodes in a round-robin manner without considering the physical location of the tasks that access the pages. The preferred policy first attempts to place pages in the memory of user-specified processor nodes. If there is no sufficient space in the user-specified nodes, it places pages in other nodes. The local policy generally achieves higher locality, whereas the interleave policy generally achieves better load balancing across the nodes.

6.2.3 Heterogeneous Memory Systems

Heterogeneous memory systems consist of two or more types of memory nodes (e.g., non-volatile memory (NVM) and DRAM [9, 52] high-bandwidth memory (HBM) and DRAM [11, 144]) which exhibit widely-different characteristics. The more detailed background on heterogeneous memory systems can be found in Chapter II.

In this work, we assume that the underlying system is similar to the Intel Xeon Phi KNL. This is done because it represents advanced and generic heterogeneous memory systems[9] in which the computation units can directly and simultaneously access the data from each of the heterogeneous memory nodes and the access to the full software stack (e.g., OS, DL framework) is publicly available.

The Xeon Phi KNL architecture provides the `Flat` mode (i.e., flat memory organization) in which the HBM and LBM nodes are mapped to a single physical address space. Among the various `Flat` modes supported by the Xeon Phi KNL architecture, we employ the `SNC4` mode because it consistently achieves high performance across a wide range of applications [129]. In the `SNC4` mode, the Xeon Phi KNL architecture exposes four HBM nodes and four LBM nodes to the operating system (OS). The OS manages the HBM and LBM nodes as eight NUMA nodes.

In addition, the Xeon Phi KNL architecture provides the `Cache` mode (i.e., tiered memory organization). In the `Cache` mode, HBM is fully managed by hardware as a last-level hardware cache and transparent to the OS. In this work, along with the other representative software-based memory management policies, we compare the performance of HALO with the hardware-based memory management policy based on the `Cache` mode.

6.2.4 Terminology

In this section, we provide the definitions of the two key concepts (i.e., tensor hotness and lifetime). The hotness of a tensor is defined as the sum of its consumer count and one (i.e., the producer count). Intuitively, a tensor with a larger hotness value is expected to generate a larger amount of data transfers (per byte).

To define the lifetime of a tensor, we consider the linearized execution order of the operations in the target DL application and assume that each instruction is assigned an index in the execution order. The

[9]Currently-available GPUs can be considered as a restricted type of heterogeneous memory systems in that the computation units in GPUs can only access the data from the device memory and the data must be always copied from the host memory to the device memory before it can be accessed by the computation units in GPUs.

Table 8: System specification

Component	Description
Processors	Intel Xeon Phi Processor 7230 @ 1.3GHz, 64 cores per CPU
L1 I-cache	Private, 32KB, 8 ways
L1 D-cache	Private, 32KB, 8 ways
L2 cache	Private (per tile), 1MB, 16 ways
Memory	192GB DDR4 (LBM) + 16GB MCDRAM (HBM)
TensorFlow	1.4.0
System software	Intel MKL 0.9, GCC 5.5.0, Intel Python 2.7.4, CentOS 7, Linux Kernel 3.10.0

Table 9: Evaluated deep-learning applications

Application	Dataset	Network	Depth	N_{Ops}	N_{Tensors}	Batch size (S, M, L)
AN	Synthetic [3]	Linear	8	61	112	2950, 11000, N/A
DN	CIFAR-10 [88]	Non-linear	40	2075	2783	2450, 4000, 8000
GN	Synthetic [3]	Non-linear	22	366	877	1160, 1800, 2650
INv4	ImageNet [135]	Non-linear	75	5185	7750	185, 300, 600
LSTM	PTB [102]	Non-linear	(2, 2400)	3672	2655	150, 4200, 6100
RN	ImageNet [135]	Non-linear	50	2210	3158	380, 600, 1000
RNv2	ImageNet [135]	Non-linear	50	2019	2830	440, 700, 1000
TR	WMT14 [21]	Non-linear	28	17505	15904	4000, 26000, 47000
VGG	Synthetic [3]	Linear	11	77	160	205, 720, 820

allocation (i.e., start) time of a tensor is defined as the index of the producer. The deallocation (i.e., end) time of a tensor is defined as the index of the last consumer (among all the consumers) in the linearized execution order. The lifetime of a tensor is then defined as the difference between the deallocation and allocation times.

6.3 Experimental Methodology

6.3.1 System Configuration

To quantify the effectiveness of HALO and various software- and hardware-based memory management policies, we employ a 64-core server system equipped with a single Xeon Phi KNL many-core CPU [144]. Table 8 summarizes the configuration of the evaluated server system. To ensure the high performance of mathematical primitives that are frequently executed by deep-learning (DL) applications, we employ the Intel Math Kernel Library (MKL), which is highly optimized for the Xeon Phi KNL architecture.

6.3.2 Deep-Learning Applications

Table 9 summarizes the evaluated DL applications. Specifically, we use nine widely-used DL applications including CNNs, an RNN, and a network with an attention mechanism – AlexNet (AN), DenseNet (DN), GoogLeNet (GN), Inception-v4 (INv4), RNN with LSTM (LSTM), ResNet (RN), ResNet-v2 (RNv2),

Transformer (TR), and VGG (VGG) [3, 10, 12, 22, 89, 142, 151, 156, 170]. They exhibit widely-different characteristics in terms of the network linearity, the operation count, and the tensor count.

For each application, we set the parallelism parameter of TensorFlow to 64, which determines the number of threads that execute each operation in a parallel manner. This is done because the use of this value consistently provides the highest performance across the applications. Further, we set the Intel MKL parallelism parameter, which controls the number of threads used to execute the mathematical primitive operations implemented in the Intel MKL, to 64 for a similar reason. Our parameter settings are in line with earlier findings [24].

Table 9 summarizes the batch sizes used for each application. In general, as the batch size increases, the working-set size of the application also increases. We use three batch sizes (i.e., small, medium, and large) for the applications. For the performance characterization studies, we use the small batch size to investigate the performance sensitivity of the applications when most of the data is allocated to LBM or HBM. To quantify the effectiveness of HALO and the other memory policies, we use all the three batch sizes.

We collect and analyze various performance and energy consumption data for each DL application through the performance event counters provided by TensorFlow, Linux, and the Performance Monitoring Counters (PMCs) available in the Intel Xeon Phi KNL architecture.

6.4 Characterization of DL Applications

This section investigates the characteristics of the deep-learning (DL) applications on the evaluated heterogeneous memory system in terms of the execution time and tensor characteristics. We present more detailed data with VGG and GN, which represent DL applications that employ the linear and non-linear networks, respectively. While omitted for conciseness, the other DL applications exhibit the data trends similar to those of the aforementioned applications.

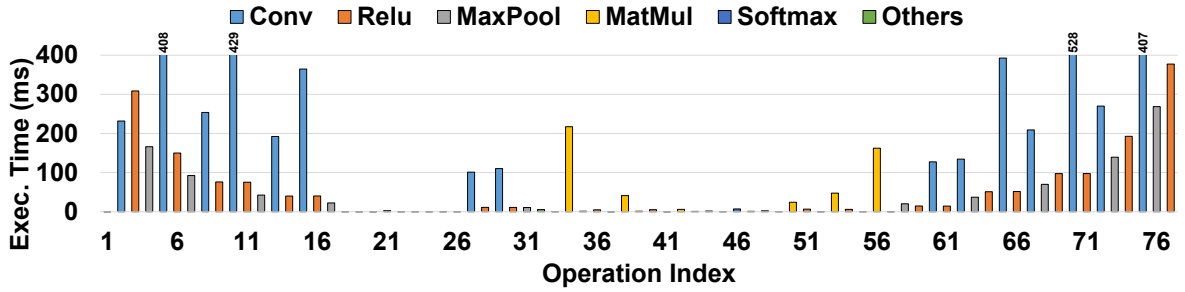
6.4.1 Execution Time Characteristics

We first analyze the per-operation execution time of the DL applications. Figures 28a and 28b show the per-operation execution time of VGG when all of the tensors are allocated to LBM and HBM, respectively.[10] In addition, Figure 28c shows the per-operation speedup of HBM over LBM when executing VGG. We observe the following data trends.

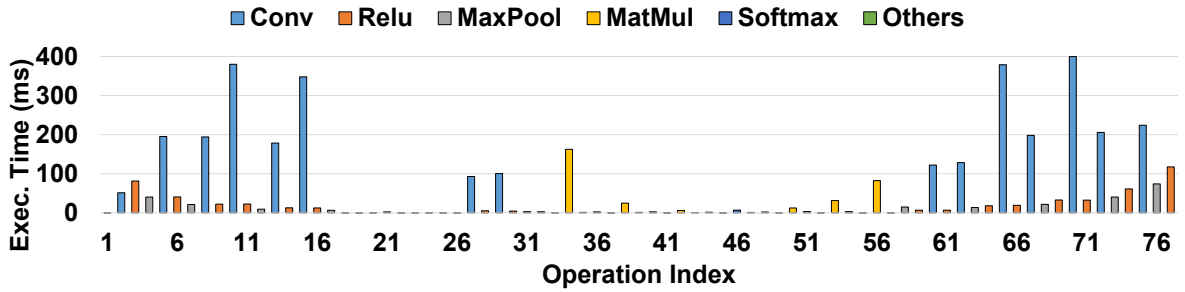
First, a small number of operation types account for a majority of the total execution time, which is in line with the findings from a recent work [27]. To gain a deeper insight into this data trend, Figure 29 shows a breakdown of the execution time of each application, normalized to the execution time with all the tensors allocated to LBM. We observe that the operations such as the convolution (Conv), MaxPool (MaxPool), and rectifier (Relu) operations account for a significant portion of the total execution time.

Second, a few operation types (e.g., FusedBatchNorm, Relu, MaxPool) achieve higher speedup with

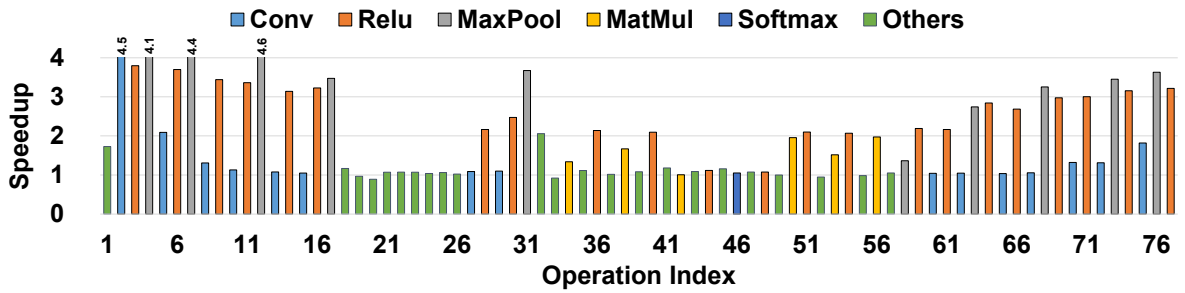
[10] In Figure 28 (and also in Figure 30), the top five time-consuming operations are highlighted with different colors and patterns.



(a) LBM



(b) HBM



(c) Speedup

Figure 28: Per-operation execution time of `VGG`

HBM than the other operation types. Because such operations perform relatively simple computations compared with the other operations, the memory access time for such operations accounts for a larger portion of the total execution time than the other operations, making their performance highly sensitive to the memory bandwidth.

Third, the execution time data across the operations exhibits symmetric trends from the middle (Figures 28a and 28b). This occurs mainly because the first half of the operations performed during the forward path of the training and the second half of the operations are performed during the backpropagation path, which is essentially the inverse of the forward path.

Fourth, the execution time of each operation type tends to decrease towards the end of the forward path (or the beginning of the backpropagation path). This is mainly because the size of the tensors is reduced (or increased) as they are processed by operations such as `MaxPool` in the forward path (or the backpropagation path). This indicates that operations near the beginning of the forward path and the end of the backpropagation path account for a large portion of the total execution time.

Figure 30 shows the per-operation execution time of `GN`, which employs a non-linear network. We

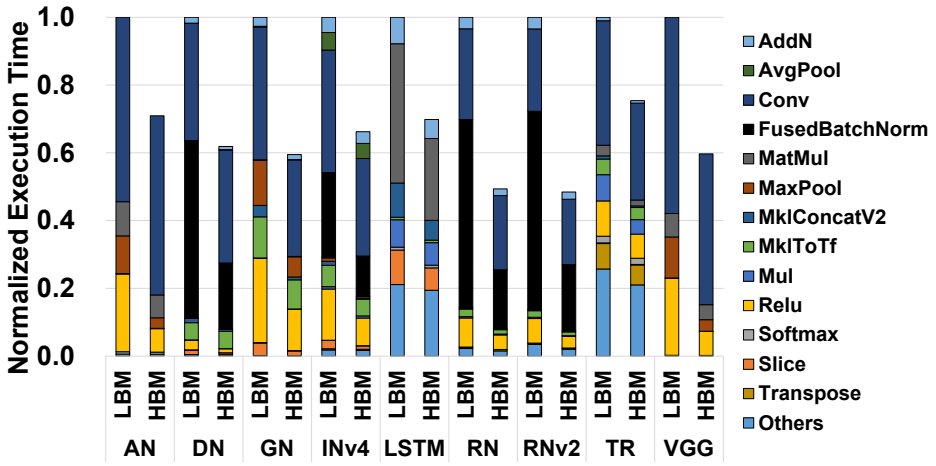


Figure 29: Execution time breakdowns

observe that GN exhibits the per-operation execution time characteristics similar to those of VGG. In addition, while omitted for conciseness, the other DL applications evaluated in this work exhibit similar characteristics.

6.4.2 Tensor Characteristics

We investigate the characteristics of tensors of the DL applications in terms of their size, hotness, and lifetime. Figure 31 shows the tensor characteristics of VGG.

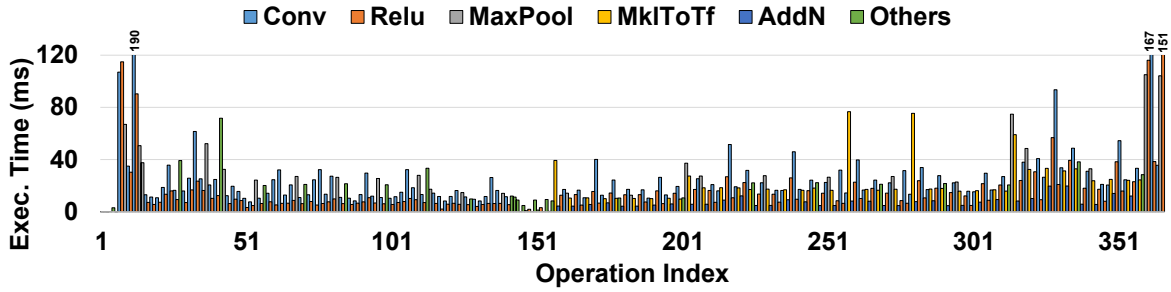
First, we observe that tensors exhibit widely-different hotness characteristics. Some tensors are only consumed by the subsequent operation in the sequential execution order of operations. In contrast, other tensors produced by operations in the forward path are consumed not only by the next operation in the execution order but also by the associated operations in the backpropagation path.

Second, tensors also exhibit great variance in their lifetime characteristics due to the aforementioned reasons. These observations indicate that data placement and migration techniques for high-performance DL on heterogeneous memory systems must judiciously consider the hotness and lifetime characteristics of each tensor to improve the overall performance.

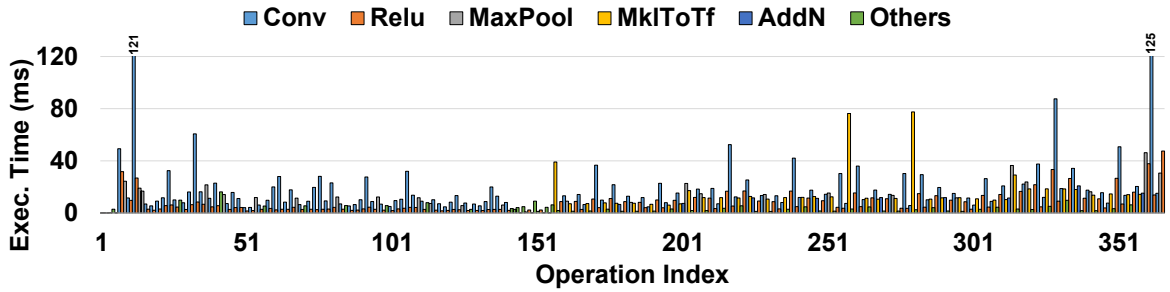
Third, similarly to the per-operation execution time results, the tensor size data shows symmetric trends from the middle (Figure 31a). This is mainly because the same set of operations in the forward path are performed in a reverse order in the backpropagation path.

Fourth, the size of the tensors tends to decrease (or increase) as they are allocated near the end (or beginning) of the forward (or backpropagation) path. This mainly arises because the size of tensors is significantly reduced (or increased) as they are processed by operations such as MaxPool in the forward path (or the backpropagation path).

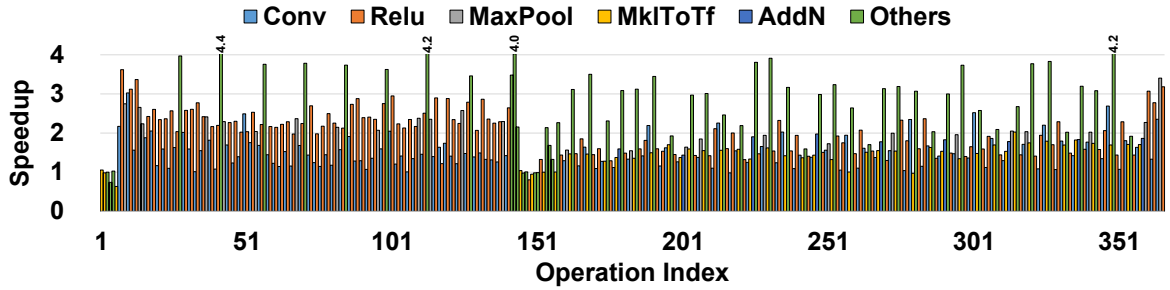
Figure 32 shows the tensor characteristics of GN, which employs a non-linear network. We observe that GN exhibits the tensor characteristics similar to those of VGG. One notable difference is that GN shows larger variations in terms of the tensor hotness and lifetime than VGG. GN employs a convolutional neural network with non-linear connections in which some tensors that are allocated in a path (i.e., forward



(a) LBM



(b) HBM



(c) Speedup

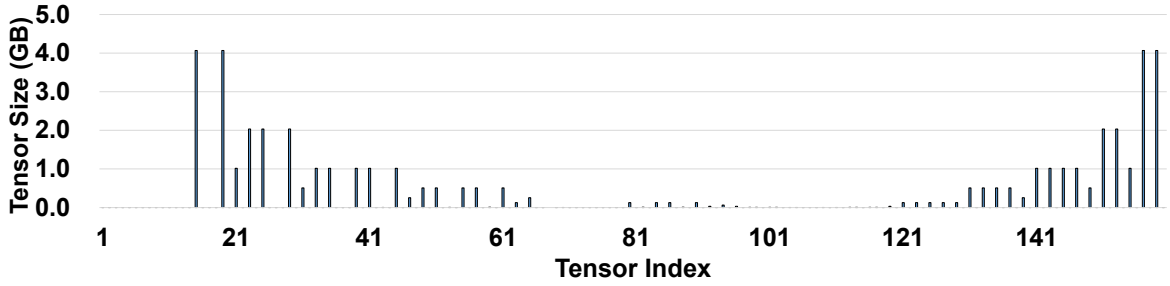
Figure 30: Per-operation execution time of GN

or backpropagation) are consumed by multiple operations in the same path, making their hotness and lifetime characteristics significantly different from those of the other tensors. This observation indicates that hotness- and lifetime-aware memory management becomes even more crucial for high-performance DL with more advanced neural network architectures on heterogeneous memory systems.

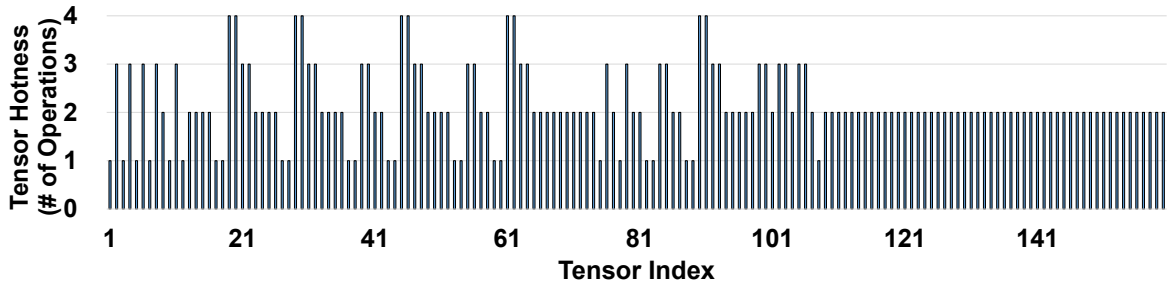
We summarize the characterization results of the DL applications as follows.

- Tensors exhibit a wide range of characteristics in terms of the hotness and lifetime. Given that HBM is a highly limited resource in heterogeneous memory systems, it is crucial to place and migrate the tensors across the heterogeneous memory nodes in a hotness- and lifetime-aware manner in order to maximize the performance of the target DL application.

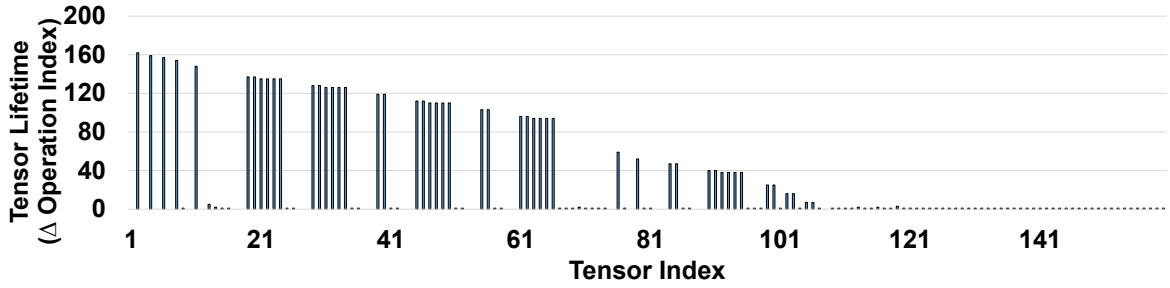
- Operations also exhibit widely-different characteristics in terms of the execution time and performance sensitivity to the memory bandwidth. Since data-intensive operations are highly sensitive to the memory bandwidth, it is crucial to identify the tensors associated with the data-intensive operations based on their hotness and lifetime characteristics and efficiently place and migrate these



(a) Size



(b) Hotness



(c) Lifetime

Figure 31: Tensor characteristics of `VGG`

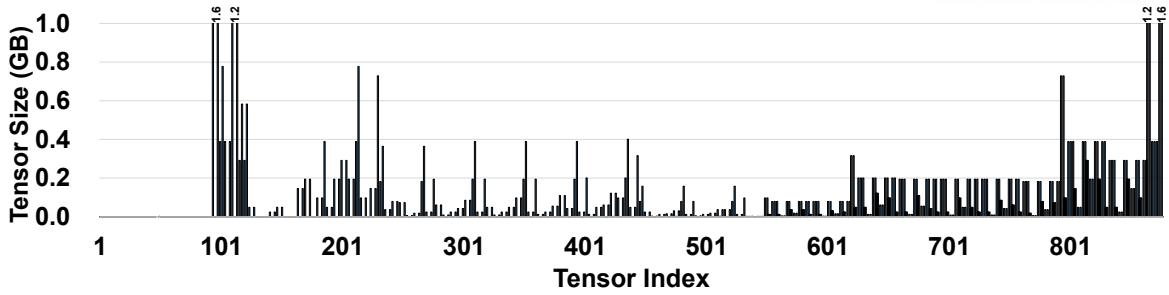
tensors across the heterogeneous memory nodes in order to improve the overall performance of the target DL application.

6.5 Design and Implementation

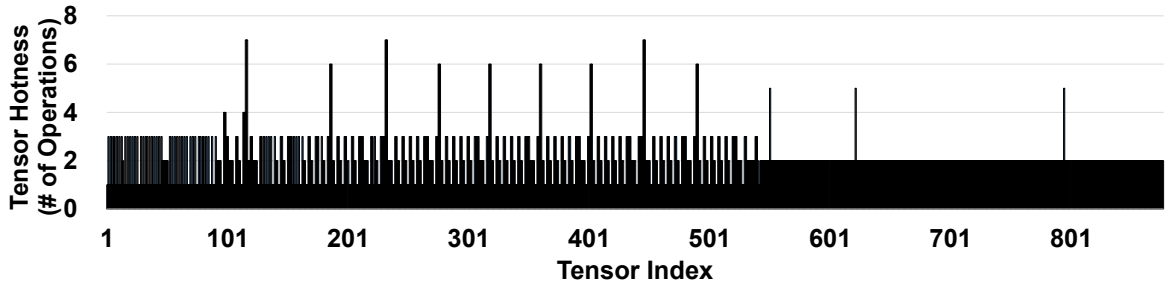
HALO is a software-based system that dynamically places and migrates tensors across the heterogeneous memory nodes based on their characteristics in order to improve the performance of the target deep-learning (DL) application. As shown in Figure 45, HALO mainly comprises the following four components – (1) the tensor hotness analyzer, (2) the tensor lifetime analyzer, (3) the tensor combiner, and (4) the tensor manager.

6.5.1 Tensor Hotness Analyzer

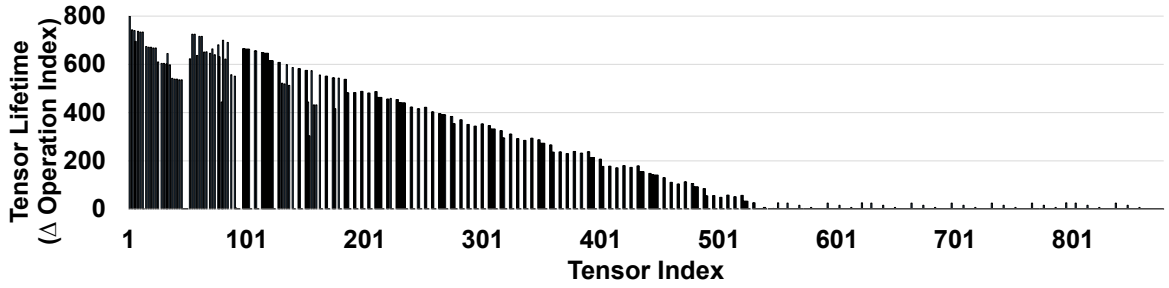
The tensor hotness analyzer estimates the access frequency (i.e., hotness) of each tensor used by the target DL application. Section 6.2.4 provides the definition of the tensor hotness.



(a) Size



(b) Hotness



(c) Lifetime

Figure 32: Tensor characteristics of `GN`

Before executing the target DL application, TensorFlow constructs an initial dataflow graph, performs a sequence of optimizations, and generates an optimized dataflow graph. We have extended the metadata for the tensor to encode the consumer count. In addition, we have added an additional dataflow graph optimization pass to compute the hotness of each tensor.

6.5.2 Tensor Lifetime Analyzer

The tensor lifetime analyzer estimates the lifetime of each tensor used by the target DL application. TensorFlow dynamically tracks the usage of each tensor using a reference counting technique. When the reference counter of a tensor becomes zero, TensorFlow deallocates the memory used by the tensor.

To estimate the lifetime of each tensor, the tensor lifetime analyzer first generates the linearized execution order of the operations of the target DL application, which is equivalent to the actual linearized execution order produced by the TensorFlow scheduler. The tensor lifetime analyzer assigns each operation a unique integer value (i.e., index) in the linearized execution order of the operations. For instance, if the index of an operation is 3, it is the third operation in the linearized execution order.

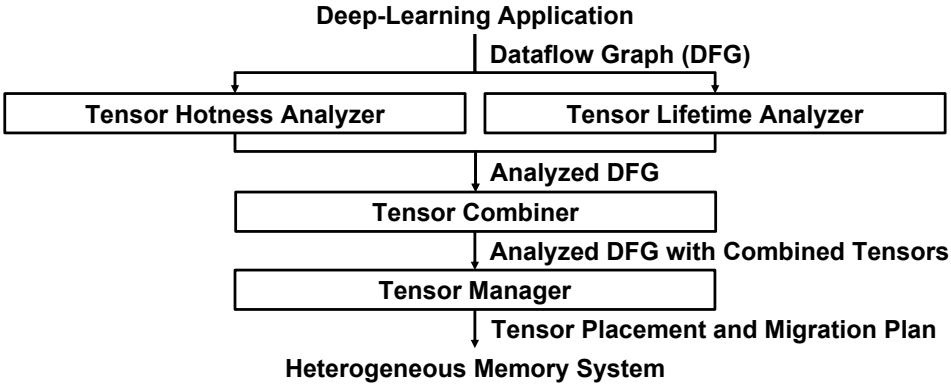


Figure 33: Overall architecture of HALO

The tensor lifetime analyzer then computes the lifetime of each tensor based on its definition provided in Section 6.2.4.

6.5.3 Tensor Combiner

TensorFlow performs an optimization that reuses the memory buffer of an input tensor as the memory buffer of an output tensor for predefined operations such as `MklInputConversion` and `Mkl2Tf`. This optimization is conducted to eliminate the overheads of deallocating the memory buffer allocated for an input tensor and allocating the new memory buffer for an output tensor if the dimensions of the two tensors are identical.

Because tensors affected by this optimization share the same memory buffer, they essentially behave as a single (virtually) combined tensor in terms of the hotness and lifetime. To accommodate this, the tensor combiner (virtually) combines all the tensors that share the same memory buffer into a single tensor and assigns the aggregated hotness and lifetime to the combined tensor.

Specifically, the tensor combiner sets the `combined` flag of tensors that share the same memory buffer to `true`, *except for* the tensor with the earliest allocation time among the tensors. The tensor combiner sets the `combined` flag of the tensor with the earliest allocation time among the tensors that share the same memory buffer to `false`. The tensor manager, which will be subsequently discussed in Section 6.5.4, only considers tensors whose `combined` flag is set to `false` to ensure that it places and migrates tensors with unique memory buffers.

The tensor combiner then aggregates all of the hotness and lifetime data across tensors that share the same memory buffer. The tensor combiner then assigns the aggregated hotness and lifetime to the tensor with the earliest allocation time (i.e., with its `combined` flag set to `false`) among the tensors. The aggregated hotness is computed as the sum of the hotness across the tensors. The aggregated allocation (or deallocation) time is determined as the earliest (or latest) of all the allocation (or deallocation) times among the tensors.

Algorithm 4 The placeAndMigrateTensors function

```
1: tensorList ← initTensorList(LBM-Only)
2: utilization ← initArray(numOperations, 0)
3: procedure PLACEANDMIGRATETENSORS
4:     generateTensorPlacementPlan()
5:     generateTensorPlacementAndMigrationPlan()
6: end procedure
```

6.5.4 Tensor Manager

Overview: The tensor manager of HALO constructs a plan for each tensor to dynamically place and migrate it across the heterogeneous memory nodes. Specifically, for each tensor, the tensor manager determines on which heterogeneous memory node it must be placed and which tensor (if any) must be migrated to the other heterogeneous memory node to secure sufficient memory space for the newly-allocated tensor.

Let $s_{\mathbf{t}}$, $h_{\mathbf{t}}$ and B_P be the size of the tensor $\mathbf{t}$, the hotness of the tensor $\mathbf{t}$, and the effective bandwidth provided by the memory allocation policy P, where $P \in \{\text{LBM-Only, interleave, HBM-Only}\}$.[11] We define the value ($v_{\mathbf{t},P}$) of the tensor $\mathbf{t}$ with the memory allocation policy P using Equation 22. Essentially, $v_{\mathbf{t},P}$ represents the reduction in the data transfer time if the tensor $\mathbf{t}$ was allocated with the memory allocation policy P instead of the LBM-Only policy.

$$v_{\mathbf{t},P} = \begin{cases} (\frac{s_{\mathbf{t}}}{B_{LBM}} - \frac{s_{\mathbf{t}}}{B_{HBM}}) \cdot h_{\mathbf{t}} & \text{if } P = \text{HBM-Only} \\ (\frac{s_{\mathbf{t}}}{B_{LBM}} - \frac{s_{\mathbf{t}}}{B_{interleave}}) \cdot h_{\mathbf{t}} & \text{if } P = \text{interleave} \\ 0 & \text{if } P = \text{LBM-Only} \end{cases} \tag{22}$$

We also define the weight ($w_{\mathbf{t},P}$) of tensor $\mathbf{t}$ with memory allocation policy P using Equation 23. In essence, $w_{\mathbf{t},P}$ denotes the amount of HBM that would be consumed if the tensor $\mathbf{t}$ was allocated with memory allocation policy P.

$$w_{\mathbf{t},P} = \begin{cases} s_{\mathbf{t}} & \text{if } P = \text{HBM-Only} \\ \frac{s_{\mathbf{t}}}{2} & \text{if } P = \text{interleave} \\ 0 & \text{if } P = \text{LBM-Only} \end{cases} \tag{23}$$

The tensor placement problem itself is a variant of the knapsack problem with time intervals, which is NP-hard [34]. Thus, we propose an efficient approximate algorithm that addresses the tensor placement and migration problem. Algorithm 4 shows the top-level function of the proposed algorithm, which mainly comprises two phases.

Phase 1. Initial Tensor Placement: During the first phase, the tensor manager aims to find an efficient allocation plan for all the tensors without considering the migration (Line 4 in Algorithm 4).

[11] On the server system evaluated in this work, $B_{LBM} = 80$GB/s, $B_{HBM} = 480$GB/s, and $B_{interleave} = 2 \cdot \min(B_{LBM}, B_{HBM}) = 160$GB/s.

Algorithm 5 The generateTensorPlacementPlan function

```
1:  procedure GENERATETENSORPLACEMENTPLAN
2:      slices ← generateSlices()
3:      sortedSliceList ← sort(slices,valPerWeight,DEC)
4:      for sl in sortedSliceList do
5:          if isFeasible(sl) = true then
6:              if sl.id = FIRST then
7:                  setAllocPolicy(sl.t, interleave)
8:              else                                          ▷ sl.id = SECOND
9:                  setAllocPolicy(sl.t, HBM-Only)
10:             end if
11:             updateUtil(sl.t.size/2, sl.t.start, sl.t.end)
12:         end if
13:     end for
14: end procedure
```

Algorithm 6 The isFeasible function

```
1:  procedure ISFEASIBLE(slice)
2:      feasibility ← true
3:      for i in [slice.start, slice.end] do
4:          if slice.size > HBM.size − utilization[i] then
5:              feasibility ← false
6:              break
7:          end if
8:      end for
9:      return feasibility
10: end procedure
```

Algorithm 5 shows the top-level function for the first phase of the tensor manager. The first phase is based on a greedy algorithm in that it considers all the tensors in the order of their value per weight (Line 3) and makes a decision to gain the maximum value from each of the considered tensors.

Because the value and weight of each tensor vary depending on which memory allocation policy (e.g., HBM-Only and interleave) is used to allocate the tensor, we introduce a concept called *tensor slices* (Line 2 in Algorithm 5). We assume that each tensor consists of two (virtual) tensor slices.

The first slice of a tensor represents the value (i.e., $v_{\mathbf{t},interleave}$) and weight (i.e., $\frac{s_\mathbf{t}}{2}$) associated with the tensor when the allocation policy for the tensor is set to the interleave policy. The second slice of the tensor represents the additional value (i.e., $v_{\mathbf{t},HBM\text{-}Only} - v_{\mathbf{t},interleave}$) and weight (i.e., $\frac{s_\mathbf{t}}{2}$) associated with the tensor if the allocation policy for the tensor is upgraded from the interleave policy to the HBM-Only policy. Note that the value per weight of the first tensor slice is always higher than that of the second slice.

The tensor manager visits each tensor slice in the (decreasing) order of the value per weight (Lines 2–13 in Algorithm 5). For each tensor slice, the tensor manager invokes (Line 5) the `isFeasible` function in Algorithm 6 to check if it can be allocated to HBM without exceeding the HBM capacity.

If the tensor slice can be allocated to HBM, the tensor manager checks whether it is the first slice

Algorithm 7 The updateUtil function

```
1: procedure UPDATEUTIL(size, start, end)
2:     for i in [start, end] do
3:         utilization[i] ← utilization[i] + size
4:     end for
5: end procedure
```

Algorithm 8 The generateTensorPlacementAndMigrationPlan function

```
1: procedure GENERATETENSORPLACEMENTANDMIGRATIONPLAN
2:     sortedTensorList ← sort(tensorList, start, INC)
3:     for t in sortedTensorList do
4:         plan_1 ← getCandidatePlan(t, interleave)
5:         plan_2 ← getCandidatePlan(t, HBM-Only)
6:         plan_best ← maxProfitPlan(plan_1, plan_2)
7:         if plan_best.profit > 0 then
8:             setAllocPolicy(t, plan_best.allocPolicy)
9:             if plan_best.migrationFlag = true then
10:                setVictim(t, plan_best.victim)
11:                setEvictPolicy(t, plan_best.evictPolicy)
12:            end if
13:            updateUtil(plan_best.sizeChange, t.start, t.end)
14:        end if
15:    end for
16: end procedure
```

of the associated tensor. If it is the first slice, the tensor manager sets the memory allocation policy of the associated tensor to the interleave policy (Lines 6–7 in Algorithm 5). Otherwise, the tensor manager upgrades the memory allocation policy of the associated tensor to the HBM-Only policy (Lines 8–9). Finally, the tensor manager accordingly updates the HBM usage information (Line 11) by invoking the `updateUtil` function in Algorithm 7 and proceeds with the second phase of the tensor manager by invoking (Line 5 in Algorithm 4) the `generateTensorPlacementAndMigrationPlan` function in Algorithm 8.

Phase 2. Refined Tensor Placement and Migration: During the second phase of the tensor manager, it refines the tensor placement plan produced during the first phase. Specifically, when the available space in HBM is insufficient to allocate a new tensor, placement of the newly allocated tensor in HBM during the first phase is disallowed. However, if there are currently some tensors in HBM that have already been accessed by most of their consumers, their values have already been significantly depreciated. Such tensors are good candidates for eviction to secure the sufficient space in HBM for a new tensor with a high value in order to increase the overall value.

To this end, the tensor manager visits each tensor in the (increasing) order of the allocation time in the sorted list (Lines 3–15 in Algorithm 8). At the allocation time of each tensor, the tensor manager reevaluates the current values of all the tensors that have already been allocated in HBM and selects the victim tensor that maximizes the profit (Lines 4–6). Specifically, the profit is defined as the value of the newly-allocated tensor minus the remaining value and migration (i.e., from HBM to LBM) cost of

the victim tensor. If the value of the newly allocated tensor is higher than the cost of the victim tensor, the tensor manager evicts the victim tensor and places the newly allocated tensor in HBM (Lines 7–12) because it increases the overall value. Finally, the tensor manager accordingly updates the HBM usage information (Line 13).

6.5.5 Discussion

Asymptotic Time Complexity Analysis: The time complexity of the first phase of the tensor manager is $O(N_T \cdot logN_T + N_T \cdot N_O)$ because the most time-consuming part consists of Line 3 (i.e., $O(N_T \cdot logN_T)$) or Lines 4 and 11 (i.e., $O(N_T \cdot N_O)$) in Algorithm 5.[12] The time complexity of the second phase of the tensor manager is $O({N_T}^2 + N_T \cdot N_O)$ because the most time-consuming part is Lines 3 and 4 (i.e., $O({N_T}^2)$) or Lines 3 and 13 (i.e., $O(N_T \cdot N_O)$) in Algorithm 8. Therefore, the overall time complexity of the tensor manager is $O({N_T}^2 + N_T \cdot N_O)$. As quantified in Section 6.6.2, HALO incurs small performance overheads.

Implementation: We have implemented HALO by extending the TensorFlow 1.4.0. Specifically, we have added or modified 39 files and 3,726 lines of code (C++ and Python) in TensorFlow to implement HALO. Most of the code for HALO is DL framework-independent as it implements high-level optimizations based on the dataflow graph and can be widely applied to other DL frameworks (e.g., PyTorch [123]).

Memory Allocation Ratios: For clarity and conciseness, we have discussed HALO with the assumption that the underlying heterogeneous memory system provides the three memory allocation ratios (i.e., LBM-Only, HBM-Only, and interleave). However, HALO can be readily extended to support heterogeneous memory systems with four or more memory allocation ratios.

Specifically, if the underlying heterogeneous memory system provides N (non-zero) memory allocation ratios, HALO can be extended to logically decompose each tensor into $N-1$ (virtual) slices. HALO then sorts the slices of all the tensors in the order of the value per weight and places each slice in the same order if there is sufficient space for the slice. If M slices ($M \leq N-1$) of a tensor are allowed to be placed in HBM, the tensor can be allocated across the heterogeneous memory nodes in a way to employ the $(N-M)$-th highest memory bandwidth that can be provided by the underlying heterogeneous memory system.

6.6 Evaluation

This section quantifies the effectiveness of HALO. Specifically, we aim to investigate the following – (1) the performance and energy consumption of HALO and various memory management policies, (2) the performance overheads of HALO, (3) the performance sensitivity of HALO to the application working-set size, and (4) the performance impact of the individual optimization techniques of HALO.

[12] N_T and N_O denote the total number of the tensors and the operations in the target DL application, respectively.

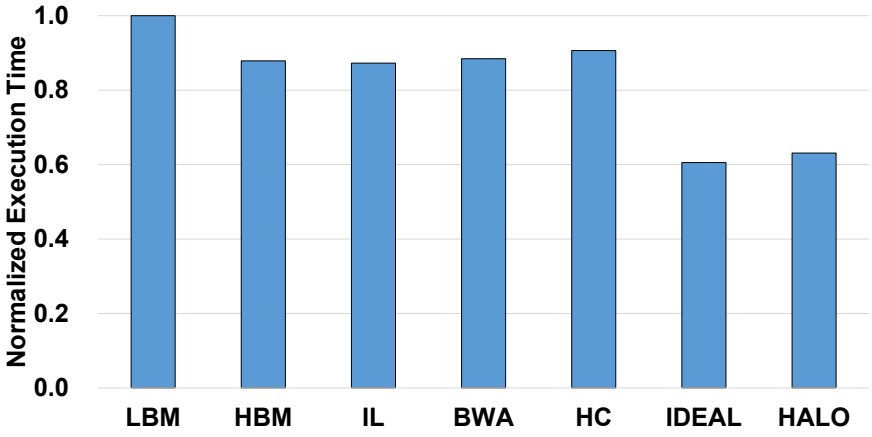


Figure 34: Overall performance results

6.6.1 Performance and Energy Results

To quantify the performance and energy consumption impact of HALO and various software- and hardware-based memory management policies, we run each of the nine deep-learning (DL) applications summarized in Table 9 with the following seven versions – (1) LBM-Preferred (`LBM`), (2) HBM-Preferred (`HBM`), (3) interleave (`IL`), (4) bandwidth aware (`BWA`), (5) hardware cache (`HC`), (6) ideal case with infinite HBM (`IDEAL`), and (7) HALO. With regard to the first three versions, we employ the corresponding memory policies in Linux with the `Flat` mode on the Xeon Phi KNL architecture.

For the `BWA` version, we employ a bandwidth-aware memory management policy similar to the ones proposed in [28, 166] with the `Flat` mode. When allocating the memory buffer for each tensor, the bandwidth-aware memory management policy aims to employ pages from the HBM and LBM nodes with the optimal allocation ratios of $\frac{B_{HBM}}{B_{HBM}+B_{LBM}}$ and $\frac{B_{LBM}}{B_{HBM}+B_{LBM}}$, which are mathematically proven to maximize the effective bandwidth across the heterogeneous memory nodes [28, 166]. If there is insufficient space in HBM required for a tensor, the `BWA` version uses all the available space in HBM that can accommodate a portion of the tensor and allocates the remaining portion of the tensor to LBM.

Regarding the `HC` version, we employ the interleave memory policy in Linux with the `Cache` mode on the Xeon Phi KNL architecture. Note that, in the `Cache` mode, HBM is fully managed by hardware as a last-level cache (i.e., transparent to the OS).

For the `IDEAL` version, we estimate the ideal performance that can be potentially achieved on an imaginary system that has HBM with infinite capacity. To this end, we execute each layer of the target DL application by allocating all the tensors of the layer to HBM and add the execution times of all the layers. Finally, regarding the HALO version, we employ HALO with the `Flat` mode. Unless specified otherwise, we use the medium batch size for each application.

We first analyze the overall performance results of HALO and the other memory management policies. Figure 34 shows the execution time of each version, which is the average across all the evaluated applications. Each bar denotes the execution time of each version, normalized to the LBM-Preferred version.

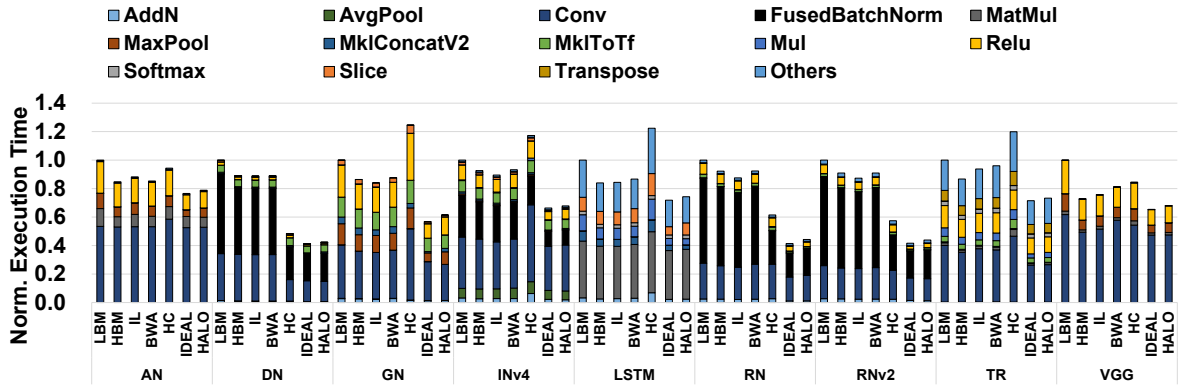


Figure 35: Execution breakdowns with HALO and various memory management policies

We observe that HALO significantly outperforms the other memory management policies. Specifically, HALO outperforms the LBM-Preferred, HBM-Preferred, `IL`, `BWA`, and `HC` versions by 36.9%, 28.2%, 27.7%, 28.7%, and 30.4%, respectively, across the evaluated applications on average (i.e., geometric mean). In addition, HALO achieves the performance comparable to the `IDEAL` version. Specifically, HALO exhibits 4.2% lower performance on average than the `IDEAL` version, which assumes an imaginary system equipped with infinite HBM. In summary, the overall performance results clearly demonstrate the effectiveness of HALO.

To gain a deeper insight into the overall performance trends, Figure 35 shows a breakdown of the execution time of each version of the applications, normalized to the LBM-Preferred version. Each bar consists of the times spent for executing the most time-consuming operations.

We observe that HALO significantly outperforms the other memory management policies across all of the evaluated DL applications. In particular, HALO provides larger performance gains over the other memory management policies with the applications (e.g., `DN` and `RNv2`) that employ non-linear neural networks. With sophisticated neural networks, the diversity of the tensors increases in terms of the hotness and lifetime. Because HALO effectively places and migrates tensors based on their characteristics, it delivers larger performance gains with the applications with non-linear networks.

We also observe that HALO significantly reduces the execution times of bandwidth-sensitive operations (e.g., `FusedBatchNorm`, `MaxPool`, `Relu`) by effectively placing the tensors in a hotness- and lifetime-aware manner. In contrast, because the other software-based memory management policies place less critical tensors to HBM since they cannot utilize the hotness and lifetime characteristics on the tensors, they are significantly outperformed by HALO.

In addition to the aforementioned reasons, the bandwidth-aware memory management policy achieves lower performance than HALO because the working-set size of the evaluated applications is larger than the capacity of HBM, making it impossible to maintain the optimal memory allocation ratio when allocating tensors across the HBM and LBM nodes. Further, since the bandwidth of LBM is significantly lower than that of HBM (i.e., $B_{LBM} = 80$GB/s, $B_{HBM} = 480$GB/s), it has relatively small impact on the overall performance.

The `HC` version achieves lower performance than HALO because it allocates the data in a hotness-

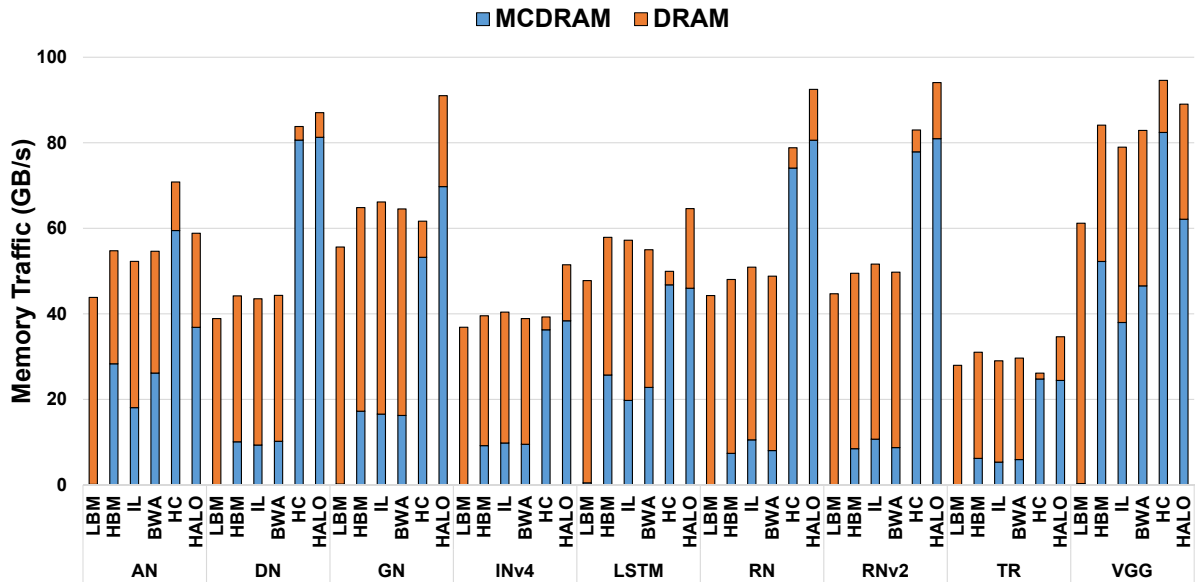


Figure 36: Memory traffic

and lifetime-oblivious manner. Further, since HBM is managed as a hardware cache with the `HC` version, it incurs extra traffic for the data to be cached and for secondary operations [46], degrading its overall performance.

The `IDEAL` version slightly outperforms HALO as it places all the data in HBM with infinite capacity. Nevertheless, HALO, which employs HBM with a finite capacity, achieves the performance comparable (i.e., 4.2% lower on average) to that of the `IDEAL` version through hotness- and lifetime-aware tensor placement and migration. This data trend clearly demonstrates the effectiveness of HALO.

As shown in Figure 36, the performance differences between HALO and the other memory management policies can also be explained in terms of memory traffic. HALO dynamically places and migrates tensors in a hotness- and lifetime-aware manner, achieving high memory traffic. In contrast, because the software-based memory management policies (i.e., LBM-Preferred, HBM-Preferred, `IL`, and `BWA`) place tensors without considering the hotness and lifetime characteristics of the tensors, they incur low memory traffic, achieving significantly lower performance than HALO.

The `HC` version on some applications incurs relatively high memory traffic (e.g., `AN`, `VGG`). This is mainly due to the extra memory traffic caused by the data placed in or evicted to LBM in a hotness- and lifetime-oblivious manner and the secondary operations for cache management [46]. The increased memory traffic incurs significant performance overheads. In contrast, HALO effectively utilizes the off-chip memory bandwidth by placing and migrating the tensors in a hotness- and lifetime-aware manner, significantly outperforming the `HC` version.

We also analyze the overall energy consumption results of HALO and the other memory management policies. Figure 37 shows a breakdown of the energy consumption of each version of the applications. Each bar denotes the energy consumption normalized to the LBM-Preferred version and consists

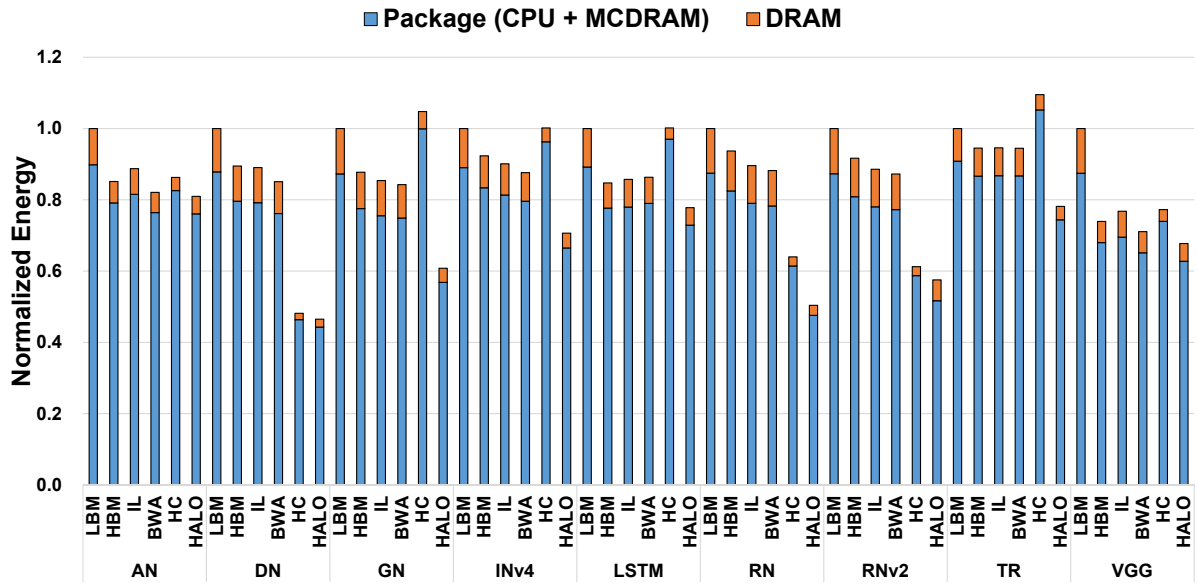


Figure 37: Energy consumption breakdowns

of the energy consumption of the package (i.e., CPU and MCDRAM) and DRAM.[13]

First, similarly to the performance results, HALO consumes significantly less energy than the other memory management policies, demonstrating the effectiveness of HALO. Specifically, HALO consumes 34.6%, 25.8%, 24.9%, 23.3%, and 24.0% less energy than the LBM-Preferred, HBM-Preferred, `IL`, `BWA`, and `HC` versions, respectively. Given that one of the major factors determine the overall energy consumption of each version is its execution time, the energy consumption data trends are similar to the performance data trends.

Second, interestingly, the DRAM energy consumption accounts for a smaller portion of the total energy consumption across all the DL applications with HALO than the software-based memory management policies (i.e., LBM-Preferred, HBM-Preferred, `IL`, and `BWA`). Since HALO reduces the traffic to LBM (i.e., DRAM) by allocating frequently-accessed tensors to HBM, it reduces the DRAM energy consumption. In contrast, as the software-based memory management policies send more traffic to LBM by allocating tensors in a hotness- and lifetime-oblivious manner, they exhibit higher DRAM energy consumption.

The `HC` version exhibits relatively low DRAM energy consumption because the data is frequently copied from LBM to HBM and accessed from HBM, which is managed as a hardware cache. However, because the `HC` version achieves lower performance than HALO, its overall energy consumption is higher than HALO.

[13] Since the evaluated system lacks the capability of separately monitoring the energy consumption of the CPU and MCDRAM in the package, we present the combined energy consumption (i.e., the package energy consumption) data for the CPU and MCDRAM. Considering the DRAM energy consumption data trends and the data-intensive nature of the DL applications, we conjecture that the MCDRAM energy consumption would account for a considerable portion of the total package energy consumption.

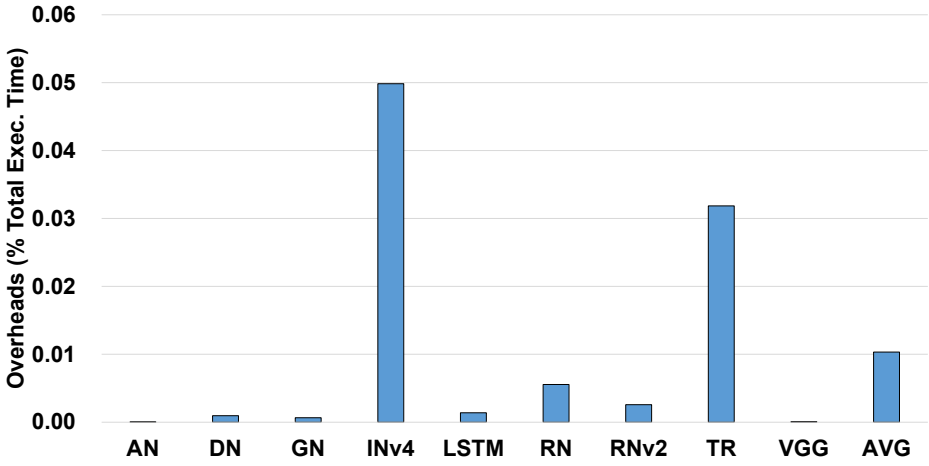


Figure 38: Performance overheads of HALO

6.6.2 Performance Overheads

We investigate the performance overheads of the tensor placement and migration planning of HALO. Figure 38 shows the percentage of the time spent to generate the tensor placement and migration plan out of the total execution time for each application.

First, the performance overheads of HALO are small across all the evaluated applications. For instance, the average performance overhead of HALO across the evaluated applications is 0.010% of the total execution time, which is small. Because HALO employs the efficient algorithm to generate the tensor placement and migration plan, it incurs small performance overheads. Second, some applications (e.g., `INv4`) incur larger (but still small) overheads than the other applications. Since their dataflow graphs in terms of the operation and tensor counts are larger than those of the other applications, HALO spends a longer time to find the efficient placement and migration plan for the tensors with such applications. Nevertheless, the overall performance overheads of HALO are small across all the evaluated applications, demonstrating its efficiency.

6.6.3 Performance Sensitivity

We investigate the performance sensitivity of HALO to the application working-set size. Figure 39 shows the execution time ratio of the HALO version of each application to the HBM-Preferred version with different batch sizes (i.e., small, medium, large).[14] Note that the application working-set size increases as the batch size increases. We observe the following data trends.

First, HALO consistently outperforms the HBM-Preferred policy across various working-set sizes, demonstrating its effectiveness. Second, the execution time ratio of HALO to the HBM-Preferred policy tends to decrease as the application working-set size increases. With a larger working-set size of the target DL application, only a small portion of the tensors can be accommodated by HBM. Because the HBM-Preferred policy places the tensors without considering their hotness and lifetime characteristics,

[14] `AN` is excluded because it fails to run with the large batch size on the evaluated server system.

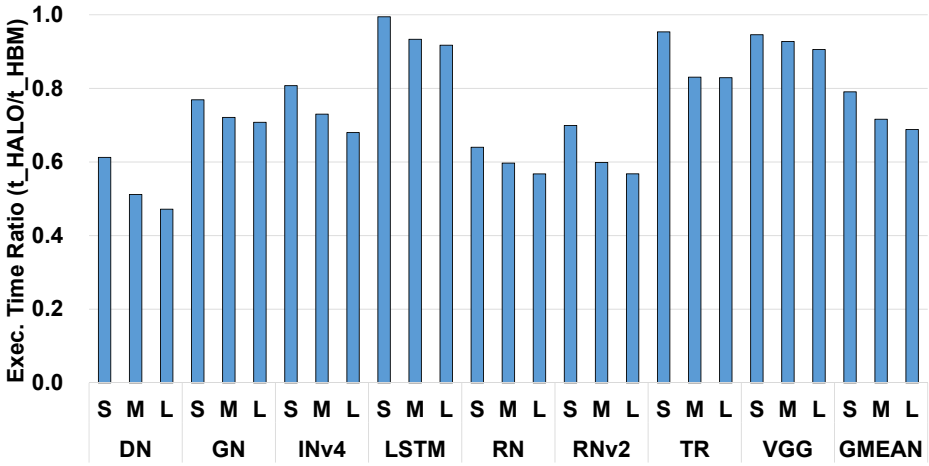


Figure 39: Sensitivity to the application working-set size

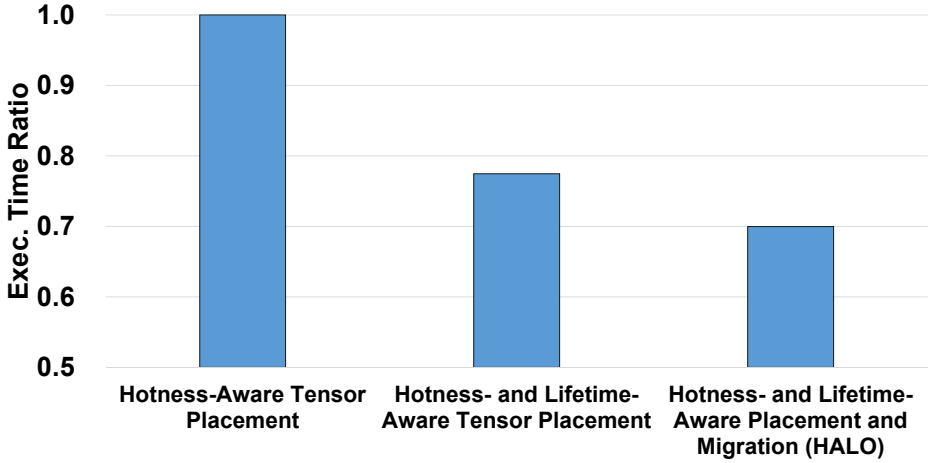


Figure 40: Impact of the optimization techniques

it frequently places inefficient tensors in HBM, resulting in low performance. In contrast, HALO dynamically places and migrates the tensors in a hotness- and lifetime-aware manner, achieving significantly higher performance than the HBM-Preferred policy.

Overall, our quantitative evaluation demonstrates the effectiveness of HALO in that it achieves significantly higher performance and lower energy consumption than the representative five software- and hardware-based memory management policies, delivers the performance comparable to the ideal case with infinite HBM, incurs small performance overheads, and exhibits high performance across a wide range of the application working-set sizes when executing various DL applications.

6.6.4 Impact of the Optimization Techniques

Finally, we quantify the performance impact of the individual optimization techniques of HALO. To this end, we create two intermediate versions of HALO that only perform hotness-aware tensor placement and hotness- and lifetime-aware tensor placement, respectively. Figure 40 shows the ratio of the average (i.e., geometric mean across the evaluated applications) execution time of the two intermediate versions

of HALO and the full version (i.e., hotness- and lifetime-aware tensor placement and migration) of HALO to the intermediate version, which only performs hotness-aware tensor placement.

We observe that each optimization technique constructively composes with the other optimization techniques in that the performance of HALO continues to improve as more techniques are applied. Further, each optimization technique considerably contributes to improving the overall performance, demonstrating the effectiveness of HALO.

6.7 Summary

In this chapter, we investigate the characteristics of various deep-learning (DL) applications on a real heterogeneous memory system. Guided by the characterization results, we propose HALO, hotness- and lifetime-aware data placement and migration for high-performance DL on heterogeneous memory systems. HALO dynamically analyzes the hotness and lifetime characteristics of the tensors of the target DL application and places the tensors across the heterogeneous memory nodes in a hotness- and lifetime-conscious manner. Our experimental results demonstrate the effectiveness of HALO in that it significantly outperforms various memory management policies supported by the underlying system software and hardware, achieves the performance comparable to the ideal case with infinite HBM, incurs small performance overheads, and delivers high performance across a wide range of the application working-set sizes.

VII Coordinated Management of Cores, Memory, and Compressed Memory Swap for QoS-Aware and Efficient Workload Consolidation for Memory-Intensive Applications

7.1 Introduction

The memory demands in cloud computing systems and datacenters are explosively growing because of the rise of emerging memory-intensive applications such as machine learning and big data applications [131, 169]. In addition, DRAM scaling has slowed down [94, 100] and there have been large fluctuations in DRAM prices [160]. As a result, DRAM has become one of the most critical and expensive components in cloud computing systems and datacenters [160].

The compressed memory swap (CMS) [134] is a promising technique to host memory-intensive applications without increasing the memory capacity of the underlying server system [93,160]. With CMS, pages selected as victim pages by the memory reclaim algorithm in the OS are compressed and evicted to the CMS instead of the disk swap. The CMS incurs overheads when compressing and decompressing pages when pages are transferred between the memory and CMS. However, since the page compression and decompression operations are performed using CPU cores and memory, the CMS is significantly faster than the disk swap, which incurs expensive disk I/O operations [160]. The CMS is supported by widely-used OSes such as Windows [18], Linux [25], and macOS [19] and employed in commercial cloud computing systems and datacenters [93, 160].

Workload consolidation is an effective technique to improve the resource efficiency of cloud computing systems and datacenters [40,99]. Without workload consolidation, dedicated servers are allocated to latency-critical (LC) applications that have soft or hard deadlines in order to satisfy their quality-of-service (QoS) requirements, drastically degrading the resource efficiency of cloud computing systems and datacenters. Workload consolidation significantly improves the resource efficiency by colocating the LC and batch applications on the same physical server. The key challenge for the resource manager with regard to workload consolidation is to find the right amounts of resources allocated to each of the LC and batch applications in order to maximize the resource efficiency while providing QoS guarantees.

Prior works have extensively investigated system software support for workload consolidation [40, 59, 65, 99, 111, 113, 119, 121, 163, 172]. However, most of the prior works present system software techniques that mitigate the performance interference caused by the contention on cores, caches, and memory bandwidth [59, 65, 99, 111, 113, 119, 121, 163, 172] but lack the capability of controlling the contention on memory capacity. While the resource manager proposed in [40] provides memory capacity partitioning, it lacks the capability of dynamic management of the CMS, which is crucial for meeting the QoS requirements of the LC application and improving the throughput of the consolidated applications with the limited memory capacity.

To bridge this gap, this chapter[15] characterizes the impact of cores, memory, and CMS on the QoS

[15]The work presented in this chapter was also published in [62]

and the throughput of the consolidated applications. Based on the characterization results, we propose a system called COSMOS for coordinated management of cores, memory, and CMS for QoS-aware and efficient workload consolidation for memory-intensive applications. We quantify the effectiveness of COSMOS with various LC and batch applications in various scenarios.

Specifically, this work makes the following contributions:

- We present the in-depth characterization of the impact of cores, memory, and CMS on the LC application's QoS and throughput of the consolidated memory-intensive LC and batch applications. Through our characterization study, we derive guidelines to effectively find an efficient system state that can significantly improve the overall throughput of the consolidated applications while satisfying the QoS.

- Guided by the characterization results, we propose COSMOS, a software-based system for coordinated management of cores, memory, and CMS for QoS-aware and efficient workload consolidation for memory-intensive applications. COSMOS dynamically collects the runtime data from the consolidated applications and the underlying system and allocates cores, memory, and CMS in a way that significantly improves the throughput of the consolidated applications while satisfying the LC application's QoS.

- We design and implement a prototype of COSMOS as a user-level runtime system on Linux. COSMOS is lightweight and readily applicable to various commodity systems without requiring the specialized hardware support or the modifications of the underlying operating systems.

- We quantify the effectiveness of COSMOS using widely-used memory-intensive LC and batch benchmarks on a real server system. Our experimental results demonstrate that COSMOS provides strong QoS guarantees and achieves high throughput across all the workload mixes with various loads for the LC application and memory overcommit ratios. In addition, COSMOS significantly reduces the number of explored system states by skipping the inefficient system states. To the best of our knowledge, our work is the first to present the in-depth characterization of the impact of cores, memory, and CMS on the QoS and throughput of the consolidated applications and design, implement, and evaluate a coordinated resource manager for cores, memory, and CMS for QoS-aware and efficient workload consolidation for memory-intensive applications on a full commodity server system.

The rest of this chapter is organized as follows. Section 7.2 provides background information. Section 7.3 describes the experimental methodology. Section 7.4 characterizes the impact of cores, memory, and CMS on the QoS and throughput of consolidated applications. Section 7.5 presents the design and implementation of COSMOS. Section 7.6 quantifies the effectiveness of COSMOS. Section 7.7 concludes the chapter with a summary.

7.2 Background

7.2.1 Memory Reclaim and CMS

When the memory usage exceeds one of the thresholds, the operating system (OS) performs memory reclaim operation to secure free space in memory. During memory reclaim, the OS determines a set of victim pages that need to be evicted from memory based on various metrics (e.g., recency, hotness) and moves the victim pages from memory to the swap area to reserve free space.

Memory reclaim can be conducted in a foreground or background manner. When a user-level process requests the OS to allocate a free page but the current memory usage exceeds a threshold, the OS conducts a foreground memory reclaim. Because the requesting process is blocked during the foreground memory reclaim, this process can drastically degrade the latency-critical (LC) application's QoS because of the significantly increased tail latency.

When the memory usage exceeds another threshold (typically set to a smaller value than that for the foreground memory reclaim), a background memory reclaim can be triggered by the OS or user-level processes even without any pending page allocation requests. One of the major advantages of background memory reclaim is that user-level processes can continue their execution during the background memory reclaim.

With the conventional disk swap, victim pages that are evicted from memory are stored in the swap area in the disk. The disk swap incurs significant performance overheads due to the I/O operations that are executed to transfer pages between the memory and disk.

The compressed memory swap (CMS) mitigates the page swapping overhead by storing the victim pages in the in-memory swap area instead of the disk. While the CMS incurs overheads for compressing and decompressing pages, it is still significantly faster than the disk swap because it does not require expensive disk I/O operations. Widely-used OSes (e.g., zswap in Linux [25], memory compression in Windows [18], and compressed memory in macOS [19]) support CMS. While this work focuses on Linux and zswap, we believe that the findings of this work can be applied to other OSes and implementations of CMS.

zswap provides various compression and decompression algorithms and memory pools for victim pages [25]. Among the compression and decompression algorithms and memory pools, we use the Lempel–Ziv–Oberhumer (LZO) algorithm [16] and the zbud memory pool [73], which are the default algorithm and memory pool for zswap.

7.2.2 Workload Consolidation

Workload consolidation enables the colocation of latency-critical (LC) and batch applications on a single physical server. LC applications are user-facing and interactive applications with latency constraints. The target tail latency of an LC application is its inherent property, which determines its latency constraint (e.g., the 99th percentile latency must be lower than one millisecond) which must be satisfied to enable user-facing and interactive services. The load for an LC application is defined as the number of

incoming requests per second.

In contrast, batch applications are background applications without any latency constraints. The common metric used to evaluate the performance of batch applications is throughput such as the number of executed iterations of the main loop per second and the amount of the input data processed per second.

The main objective of workload consolidation is to maximize the throughput of the consolidated applications while providing strong QoS guarantees for the LC application. When resources are allocated in an unmanaged manner, the LC application is likely to violate its QoS due to the performance interference caused by the contention on the resources (e.g., cores, memory capacity) shared by the consolidated LC and batch applications.

To eliminate or mitigate the performance interference, the resource manager for workload consolidation partitions resources between the consolidated LC and batch applications. Production-quality operating systems provide support for core and memory capacity partitioning. For instance, Linux provides core and memory capacity partitioning through control groups (cgroups) [2]. A *cgroup* is a set of processes that can be allocated their own resources. Cores and memory can be partitioned between the consolidated LC and batch applications by associating each application with its own cgroup and allocating disjoint sets of cores and memory to each cgroup. Linux also allows to dynamically change the amounts of the resources allocated to each cgroup.

7.3 Experimental Methodology

7.3.1 System Configuration

In this work, we use two systems, each of which is used as the server or client system. The server system runs the consolidated latency-critical (LC) and batch applications. The server system is equipped with the 16-core Intel Xeon Gold 6226R CPU, 64 GB memory (4 $\times$ 16 GB DIMMs), a 1 TB Samsung 970 EVO Plus NVM-e SSD, and a 100 Gb NIC. 4 GB out of the 64 GB is reserved for the OS. The server system is installed with Ubuntu 22.04 and Linux kernel 6.1.11.

Cgroups is used to partition cores and memory between the LC and batch applications. In addition, `sysfs` is used to dynamically (1) control the amount of the compressed memory swap (CMS) allocated to the LC container (i.e., `/sys/module/zswap/parameters/max_pool_percent`) and (2) track the compression ratio, which is computed by collecting the number of pages stored in the CMS (i.e., `/sys/kernel/debug/zswap/stored_pages`) and the actual amount of memory used by the CMS (i.e., `/sys/kernel/debug/zswap/pool_total_size`).

The client system runs the load generator for each of the evaluated LC applications. The client system is equipped with two 32-core Intel Xeon Gold 6338 CPUs, 64 GB memory (4 $\times$ 16 GB DIMMs), a 1 TB Samsung 870 EVO SATA SSD, and a 100 Gb NIC. The client system is installed with Ubuntu 22.04 and Linux kernel 5.15.0. The client and server systems are directly connected through the 100 Gb Ethernet.

Table 10: Loads for the LC benchmarks

Benchmark	Low, medium, and high loads (QPS)
memcached	37,500, 75,000, and 150,000
silo	1,000, 2,000, and 4,000

7.3.2 Benchmarks

We use two latency-critical (LC) benchmarks – memcached [17, 54] and silo [79, 154], which are in-memory key-value store and in-memory database, respectively. The QoS target of memcached is that the 99th percentile latency must be lower than 200 microseconds [90, 133]. We use Lancet, which is an open loop-based load generator for memcached [86]. In line with the prior works [40, 90], we configure the key and value sizes to 30 and 200 bytes and the query type to read-only. In addition, we configure the key popularity to follow a Zipfian distribution [137] with a skewness parameter of 0.99 and the query inter-arrival time to follow an exponential distribution in order to emulate the access [33, 164] and traffic (e.g., micro-bursts) [79, 105] patterns commonly observed in datacenters.

The QoS target of silo is that the 99th percentile latency must be lower than one millisecond [41, 126]. We use the load generator included in TailBench for silo [79]. The load generator for silo also generates queries with an exponential inter-arrival time distribution based on the observations (e.g., micro-bursts) from the prior works [79, 105].

We also use five batch benchmarks – betweenness centrality (BC) [36], breath-first search (BFS) [36], canneal [38], connected components (CC) [36], and stream [104]. The metric used to quantify the throughput of the batch benchmarks is the number of executed iterations of the main loop per second.

We use the aforementioned LC and batch benchmarks because they are memory-intensive and are widely used for cloud computing systems and datacenter research [40, 41, 59, 65, 126, 160, 164]. The thread count of each benchmark is set to 16, which is identical to the number of cores in the CPU on the evaluated server system.[16]

In this work, we refer to a container that contains the LC application as the LC container. In addition, we refer to a container that consists of one or more batch applications as the batch container.

We investigate the impact of cores, memory, and compressed memory swap on the QoS of throughput of the consolidated containers and evaluate the effectiveness of COSMOS with various loads for the LC container. Table 10 summarizes low, medium, and high loads (in queries per second (QPS)) for the LC benchmarks.

We also conduct experiments with various memory overcommit ratios (MORs). The MOR is defined in Equation 24, where M is the total memory capacity (excluding the amount of the memory reserved for

[16]There are mainly two widely-used mechanisms to control the concurrency of applications – (1) dynamic threading [127, 149] and (2) thread packing [47, 120, 139]. With dynamic threading, the concurrency of an application is controlled by dynamically adjusting the number of threads of the application. With thread packing, the concurrency of an application is controlled by dynamically adjusting the number of cores allocated to the application. A major disadvantage of dynamic threading is the limited applicability because numerous applications lack the support for dynamic threading. In contrast, thread packing can be applied to all applications regardless of whether they support dynamic threading or not. Because of the advantage of thread packing, we consider it as a mechanism for dynamic concurrency control. To ensure the cores in the evaluated CPU can fully be utilized, we configure the thread count of each benchmark to 16.

Table 11: Working-set sizes

Benchmark	Working-set sizes with low, medium, and high MORs (GB)
`memcached`	42.1, 45.0, and 51.1
`silo`	42.1, 45.2, and 51.3
`BC`	22.5, 24.0, and 27.0
`BFS`	22.6, 24.1, and 27.3
`canneal`	22.5, 24.2, and 27.1
`CC`	22.6, 24.1, and 27.3
`stream`	22.5, 24.0, and 27.0

the OS) of the underlying server system and w_{LC} and w_{Batch} represent the working-set size of the LC and batch containers, respectively. Table 11 summarizes the working-set sizes of the evaluated benchmarks with low (i.e., 1.075), medium (i.e., 1.15), and high (i.e., 1.3) MORs in GB. For example, if we consider the workload mix of `memcached` and `BC` with the medium MOR, the working-set sizes of `memcached` and `BC` are 45.0 GB and 24.0 GB, respectively. The MOR is then computed to be 1.15 (i.e., $\frac{45.0+24.0}{M} = 1.15$, where M is 60 GB) on the evaluated server system using Equation 24.

$$\text{Memory overcommit ratio} = \frac{w_{\text{LC}} + w_{\text{Batch}}}{M} \tag{24}$$

7.4 Characterization

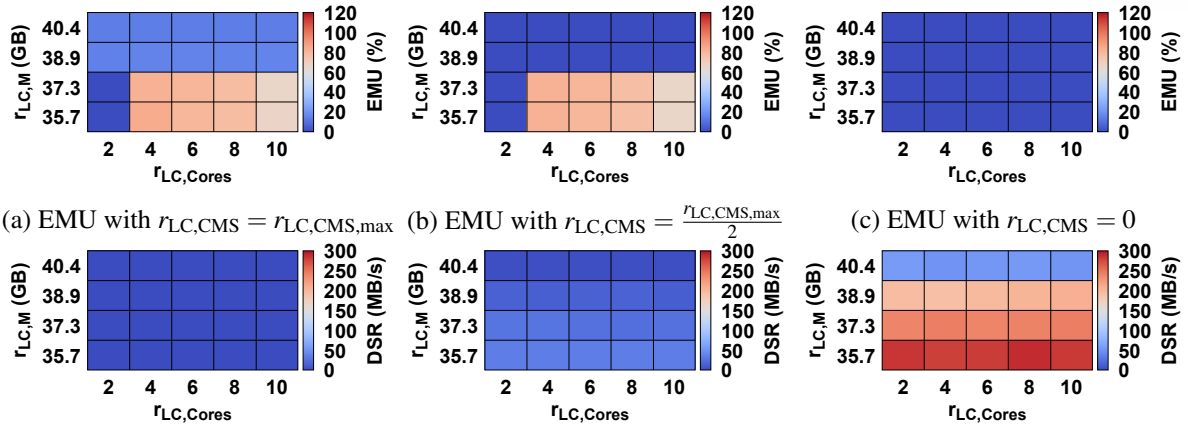
In this section, we characterize the impact of cores, memory, and compressed memory swap (CMS) on the QoS and the throughput of the consolidated containers. While we only present the experimental results with `memcached` (i.e., the latency-critical (LC) container) and `stream` (i.e., the batch container) for conciseness, other benchmarks exhibit similar data trends. We execute the consolidated containers using the following configurations – (1) low load and low memory overcommit ratio (MOR), (2) low load and high MOR, (3) high load and low MOR, and (4) high load and high MOR.

In each of the configurations, we vary the number of cores and the amounts of the memory and CMS allocated to the LC container and analyze their impact on the QoS and throughput of the consolidated containers. We denote the core count, the amount of memory, and the amount of the CMS allocated to the LC container as $r_{\text{LC,Cores}}$, $r_{\text{LC,M}}$, and $r_{\text{LC,CMS}}$, respectively. In addition, we denote the working-set size of the LC container and the average compression ratio of the pages stored in the CMS as w_{LC} and γ_{LC}.

The upper bound of $r_{\text{LC,CMS}}$ is then computed using Equation 25. Intuitively, Equation 25 indicates that there is no need to increase $r_{\text{LC,CMS}}$ further once it becomes large enough to fit w_{LC} in memory and CMS or $r_{\text{LC,CMS}}$ cannot exceed $r_{\text{LC,M}}$ because CMS consumes the memory allocated to the LC container.

$$r_{\text{LC,CMS,max}} = \min\left(\frac{w_{\text{LC}} - r_{\text{LC,M}}}{\gamma_{\text{LC}} - 1}, r_{\text{LC,M}}\right) \tag{25}$$

In line with the prior works on workload consolidation [40, 59, 99], we use *effective machine utilization* (EMU), which is a system-wide metric that quantifies the throughput of the consolidated containers.



(a) EMU with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (b) EMU with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (c) EMU with $r_{\text{LC,CMS}} = 0$

(d) DSR with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (e) DSR with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (f) DSR with $r_{\text{LC,CMS}} = 0$

Figure 41: Impact of cores, memory, and CMS allocated to the LC container with low load and low MOR

If the LC container violates its QoS, EMU becomes zero.

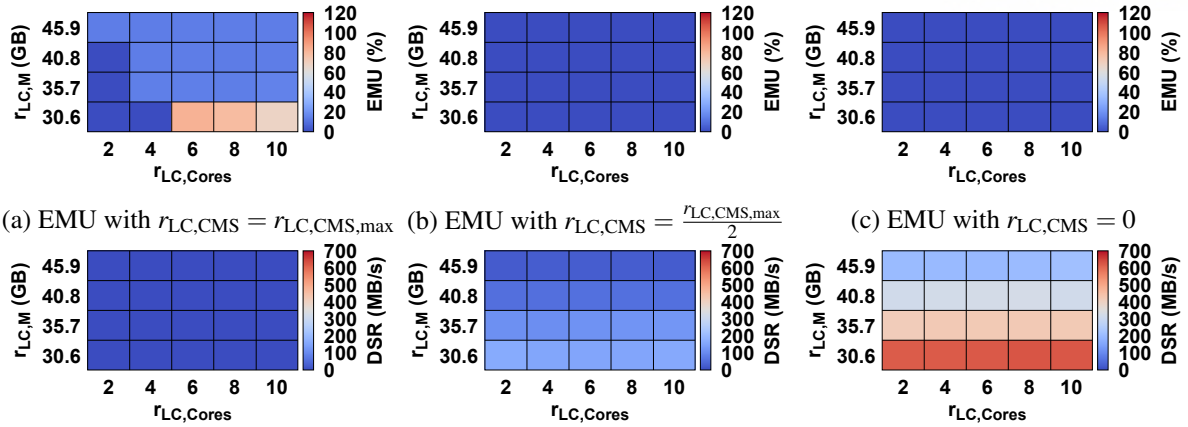
In contrast, if the LC container satisfies its QoS, EMU is computed to be a positive value. A larger EMU value indicates that the consolidated containers achieve higher throughput. To compute EMU, first we compute the normalized throughput of each of the consolidated applications by dividing the throughput of the application with workload consolidation (i.e., the hardware resources are shared by the consolidated applications) by the solo-run throughput of the application when it is allocated all of the hardware resources. We then compute EMU by summing up the normalized throughput of each of the consolidated applications.

Figure 41 shows the EMU and disk swap rates (DSRs) of `memcached` and `stream` with low load and low MOR. Each cell in the heat maps represents a single data point. Each heat map reports 20 data points collected from 20 configurations.

First, when a sufficient amount of the CMS (e.g., $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$) is allocated to the LC container, its QoS is satisfied (i.e., EMU $\neq 0$) across wide ranges of $r_{\text{LC,Cores}}$ and $r_{\text{LC,M}}$. This is mainly because the LC container requires a relatively small amount of memory with a low MOR and a relatively small number of cores with a low load.

Even when a sufficient amount of the CMS is allocated to the LC container, its QoS is violated (i.e., EMU $= 0$) with insufficient cores and memory (e.g., $r_{\text{LC,Cores}} = 2$, $r_{\text{LC,M}} = 35.7$ GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 41a). This is mainly arises due to the contention on the cores shared by the threads of the LC container and the memory reclaim threads. When a smaller amount of memory is allocated to the LC container, more of its data is transferred between the memory and CMS. This increases the CPU utilization of the memory reclaim threads, causing the contention on cores with the threads of the LC container.

Second, when a sufficient amount of the CMS (e.g., $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$) is allocated to the LC container, the EMU tends to increase as the number of cores and the amount of memory allocated to the LC container decreases. The EMU is maximized when the number of cores and the amount of



(a) EMU with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (b) EMU with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (c) EMU with $r_{\text{LC,CMS}} = 0$

(d) DSR with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (e) DSR with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (f) DSR with $r_{\text{LC,CMS}} = 0$

Figure 42: Impact of cores, memory, and CMS allocated to the LC container with low load and high MOR

memory allocated to the LC container are small yet just enough to satisfy the LC container's QoS (e.g., $r_{\text{LC,Cores}} = 4$, $r_{\text{LC,M}} = 37.3$ GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 41a).

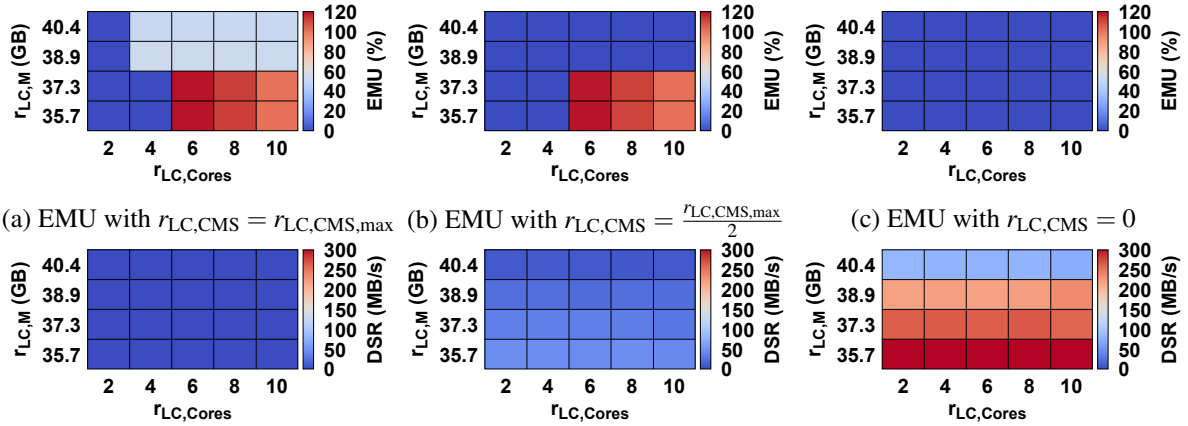
We also observe that the EMU gradually increases as the LC container's core count decreases. For example, Figure 41b shows that the EMU increases from 66.3% to 82.5% as the LC container's core count decreases from 10 to 4 when $r_{\text{LC,M}} = 37.3$ GB. This is mainly because the batch container gradually achieves higher throughput as it is allocated more cores.

In contrast, the EMU abruptly increases only when the amount of the memory allocated to the LC container is small enough to make the working-set of the batch container fit in memory. Except for this abrupt change, the EMU minimally changes as the amount of the memory allocated to the LC container decreases. This is mainly because the hotness among the data accessed by the batch container (i.e., `stream`) is uniform. With the uniform memory access pattern, the batch container exhibits high performance only when its entire working set fits in memory. The EMU changes more gradually when the batch container includes applications (e.g., `BC`) that exhibit non-uniform memory access patterns.

Third, when a relatively small amount of the CMS is allocated to the LC container, its QoS is satisfied with fewer configurations. For example, when $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$, the QoS is satisfied with only eight configurations (c.f., 18 configurations when $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$) out of 20 configurations. This mainly stems from the fact that more memory is needed to make the working set of the LC container fit in memory when a smaller amount of the CMS is allocated to the LC container. When the working set of the LC container does not fit in memory by being allocated with insufficient amounts of the memory and CMS, victim pages are evicted to the disk swap. This leads to the QoS violation of the LC container due to frequent I/O operations.

In an extreme case where $r_{\text{LC,CMS}} = 0$, the LC container's QoS is violated across all the configurations. This indicates that the use of the CMS is required to satisfy the QoS when its working-set size exceeds its allocated memory size.

Figure 42 shows the EMU and disk swap rates (DSRs) of `memcached` and `stream` with the low load



(a) EMU with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (b) EMU with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (c) EMU with $r_{\text{LC,CMS}} = 0$

(d) DSR with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (e) DSR with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (f) DSR with $r_{\text{LC,CMS}} = 0$

Figure 43: Impact of cores, memory, and CMS allocated to the LC container with high load and low MOR

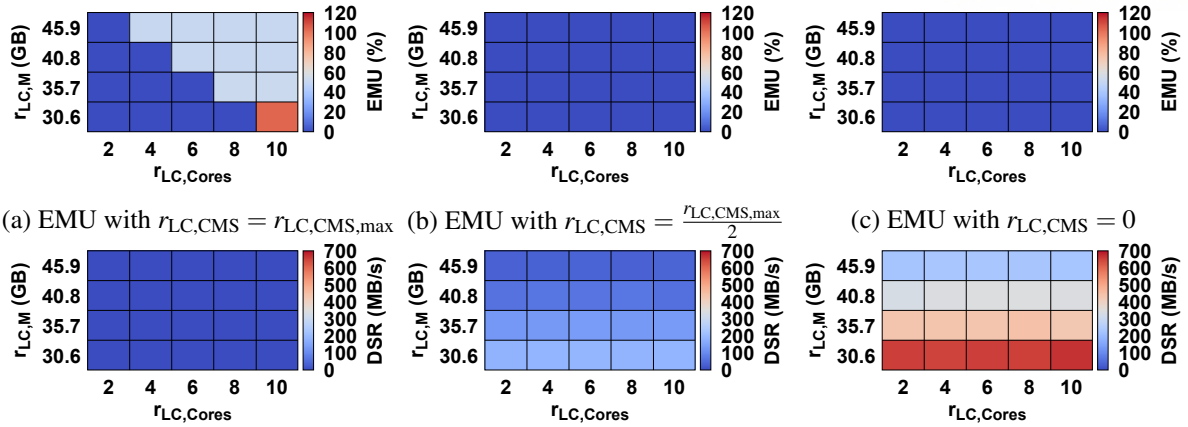
and high MOR. First, the overall EMU data trends are similar to the case with the low load and low MOR in that the EMU tends to increase when the number of cores and the amount of memory allocated to the LC container are smaller.

Second, the highest EMU achieved with the low load and high MOR (e.g., $r_{\text{LC,Cores}} = 6$, $r_{\text{LC,M}} =$ 30.6 GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 42a) is similar to the highest EMU achieved with the low load and low MOR (e.g., $r_{\text{LC,Cores}} = 4$, $r_{\text{LC,M}} = 37.3$ GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 41a). This is mainly because the load for the LC container is same and the batch container achieves similar performance when allocated a sufficient amount of memory.

Third, in comparison with the case with the low load and low MOR, the LC container needs to be allocated a larger number of cores and a smaller amount of memory to achieve the highest EMU (e.g., $r_{\text{LC,Cores}} = 4$, $r_{\text{LC,M}} = 37.3$ GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 41a vs. $r_{\text{LC,Cores}} = 6$, $r_{\text{LC,M}} =$ 30.6 GB, and $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ in Figure 42a). Because the working-set size of the batch container increases with the high MOR, the LC container needs to be allocated a smaller amount of memory in order to make the working set of the batch container fit in memory. As a result, a larger portion of the data of the LC container is transferred between the memory and CMS, increasing the CPU utilization of the memory reclaim threads. To mitigate the contention on the cores shared by the threads of the LC container and the memory reclaim threads, more cores need to be allocated to the LC container.

Fourth, in comparison with the case with the low load and low MOR, the LC container requires a larger amount of the CMS to satisfy its QoS. For instance, the LC container's QoS is violated across all the configurations when $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ with the low load and high MOR (Figure 42b), whereas it is satisfied with eight configurations when $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ with low load and low MOR (Figure 41b). Since the working-set size of the LC container increases with the high MOR, the LC container requires a larger amount of the CMS to make its working set fit in memory.

Figure 43 shows the EMU and disk swap rates (DSRs) of `memcached` and `stream` with the high load and low MOR. First, the overall EMU data trends are similar to the aforementioned cases in that



(a) EMU with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (b) EMU with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (c) EMU with $r_{\text{LC,CMS}} = 0$

(d) DSR with $r_{\text{LC,CMS}} = r_{\text{LC,CMS,max}}$ (e) DSR with $r_{\text{LC,CMS}} = \frac{r_{\text{LC,CMS,max}}}{2}$ (f) DSR with $r_{\text{LC,CMS}} = 0$

Figure 44: Impact of cores, memory, and CMS allocated to the LC container with high load and high MOR

the EMU tends to increase as the core count and the amount of the memory allocated to the LC container decrease.

Second, the highest EMU achieved with the high load and low MOR is higher than that with the cases with the low load. Since the LC container is applied with the high load, the portion of the EMU contributed by the LC container increases in comparison with the cases with the low load.

Figure 44 shows the EMU and disk swap rates (DSRs) of `memcached` and `stream` with the high load and high MOR. First, the overall EMU trends are similar to the aforementioned cases. Second, the LC container's QoS is satisfied in fewer configurations than the other cases. This occurs because the LC container requires larger amounts of resources to satisfy its QoS with the high load and high MOR.

Third, the highest EMU achieved with the high load and high MOR is higher than those achieved with the cases with the low load. Because the throughput of the LC container increases with the high load, the overall EMU also increases.

In contrast, the highest EMU achieved with the high load and high MOR is lower than that achieved with the high load and low MOR. Since the working-set size of the batch container increases with the high MOR, the LC container needs to be allocated with a smaller amount of memory. To make its working set fit in memory, the LC container requires a larger amount of the CMS. Because the CPU utilization of the memory reclaim threads increases with more frequent data transfers between the memory and CMS, the LC container requires a larger number of cores, decreasing the highest EMU achieved with the high load and high MOR.

Our characterization results clearly motivate the need for coordinated management of cores, memory, and CMS to significantly improve the EMU with strong QoS guarantees. The findings learned from the characterization study are summarized as follows.

- **Impact of resources** (C1): Cores, memory, and CMS have significant impact on the LC container's QoS and the throughput of the consolidated containers in that EMU varies greatly depending on how the resources are allocated to the consolidated containers.

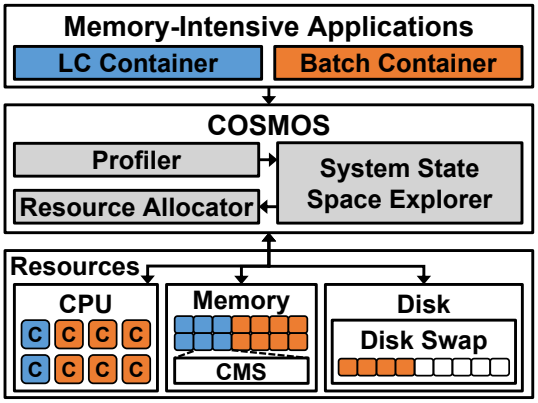


Figure 45: Overall architecture of COSMOS

- **Impact of loads and MORs** (C2): There is no single configuration of cores, memory, and CMS that delivers high EMU across various loads and MORs in that the configuration of cores, memory, and CMS resulting in high EMU widely varies across various loads and MORs.
- **Allocation of memory and CMS** (C3): The EMU of the consolidated container is significantly improved (i.e., satisfying the QoS and achieving high throughput) when the amount of the memory allocated to the LC container is small enough to make the working-set of the batch container fit in memory but the amounts of the memory and CMS allocated to the LC container are large enough to make its working-set fit in memory.
- **Allocation of cores** (C4): With a smaller amount of the memory and a larger amount of the CMS allocated to the LC container, the LC container requires a larger number of cores that are sufficient to execute not only its threads but also the memory reclaim threads for satisfying its QoS.

7.5 Design and Implementation

Our characterization study shows that resources, loads, and memory overcommit ratios have significant impact on the latency-critical (LC) container's QoS and the throughput of the consolidated containers (i.e., C1 and C2 in Section 7.4). Based on the characterization results, we propose COSMOS, a software-based runtime system that dynamically allocates cores, memory, and compressed memory swap (CMS) to the consolidated LC and batch containers to achieve high throughput while satisfying the QoS for given load and memory overcommit ratio. COSMOS consists of three components – the (1) profiler, (2) system state space explorer, and (3) resource allocator. Figure 45 illustrates the overall architecture of COSMOS.

7.5.1 Profiler

The profiler of COSMOS dynamically collects the runtime data that the system state space explorer uses to make resource allocation decisions. Specifically, it collects the load and tail latency data from the latency-critical (LC) container and the throughput data from the batch container.

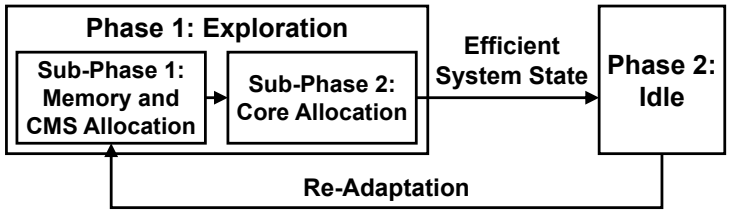


Figure 46: Execution flow of the system state space explorer

In addition, the profiler dynamically collects the working-set sizes of the LC and batch containers, which are available through procfs on Linux. Further, it collects the average compression ratio of the pages stored in the CMS through sysfs at runtime.

7.5.2 System State Space Explorer

The system state space explorer (SSSE) of COSMOS dynamically explores the system state space to discover a system state that delivers high throughput while providing strong QoS guarantees for the latency-critical (LC) container. It uses effective machine utilization (EMU), which is defined in Section 7.4, as the throughput metric.

As shown in Equation 26, we define a system state (i.e., s) as a vector of three elements (i.e., $r_{\text{LC,Cores}}$, $r_{\text{LC,M}}$, and $r_{\text{LC,CMS}}$), which denote the number of cores, the amount of memory, and the amount of CMS allocated to the LC container (see Section 7.4 for the definitions of $r_{\text{LC,Cores}}$, $r_{\text{LC,M}}$, and $r_{\text{LC,CMS}}$). Note that the remaining resources are allocated to the batch container. We then define the system state space as the set of all the valid system states.

$$s = (r_{\text{LC,Cores}}, r_{\text{LC,M}}, r_{\text{LC,CMS}}) \tag{26}$$

Figure 46 shows the execution flow of the SSSE. In addition, Algorithm 9 shows the top-level function (i.e., `exploreSystemStateSpace`) executed by the SSSE. It operates in a periodic manner. In each period, the SSSE first allocates the resources to the consolidated containers (Lines 9–10 in Algorithm 9) based on the resource allocation plan encoded in the current system state through the resource allocator (Section 7.5.3).[17] It then collects the runtime data (Lines 11–17 in Algorithm 9) – the load for the LC container, the tail latency of the LC container, the throughput of the batch container, the working-set sizes of the consolidated containers, and the average compression ratio of the pages stored in the CMS based on the profiler (Section 7.5.1). If the current system state satisfies the LC container's QoS and achieves higher throughput than the highest throughput that has been discovered so far, it updates the best system state (i.e., s_{best}) to the current system state (Lines 24–27 in Algorithm 9). It comprises two phases – the (1) exploration and (2) idle phases.

Exploration Phase: During the exploration phase, it gradually explores the system state space to find an efficient system state that achieves high throughput while satisfying the LC container's QoS. Specifically, in each period, it invokes the `getNextSystemState` function (Line 31 in Algorithm 9),

[17] The period is set to one second in this work.

Algorithm 9 The exploreSystemStateSpace function

```
1:  phase ← exploration; subphase ← memoryAndCMS
2:  L ← 0; Q ← 0; T ← 0; w_LC ← 0; w_Batch ← 0; γ_LC ← 1
3:  s_best ← s_invalid; T_best ← 0
4:  isBatchThroughputIncreased ← false
5:  procedure EXPLORESYSTEMSTATESPACE
6:      createReclaimThreads()
7:      s_next ← getInitialState()
8:      while true do
9:          s_curr ← s_next
10:         applySystemState(s_curr)
11:         sleep(τ)                                   ▷ Period: one second
12:         L ← getLoad()
13:         Q ← getTailLatency()
14:         T ← getBatchThroughput()
15:         w_LC ← getLCWorkingSetSize()
16:         w_Batch ← getBatchWorkingSetSize()
17:         γ_LC ← getCompressionRatio()
18:         if needToReadapt() = true then
19:             resetVariables()
20:             s_next ← getInitialState()
21:             phase ← exploration
22:         else
23:             if phase = exploration then
24:                 if isQoSSatisfied(Q) = true and T > T_best then
25:                     s_best ← s_curr
26:                     T_best ← T
27:                     isBatchThroughputIncreased ← true
28:                 else
29:                     isBatchThroughputIncreased ← false
30:                 end if
31:                 s_next ← getNextSystemState(s_curr)
32:                 if s_next = s_curr then
33:                     s_next ← s_best
34:                     phase ← idle
35:                 end if
36:             end if
37:         end if
38:     end while
39: end procedure
```

which is shown in Algorithm 10.

The `getNextSystemState` function determines the system state, which is explored in the next period and expected to satisfy the LC container's QoS and achieve higher throughput than the current system state. The exploration phase consists of two sub-phases – (1) memory and CMS allocation sub-phase (Lines 3–13 in Algorithm 10) and (2) core allocation sub-phase (Lines 14–24). It begins with the memory and CMS allocation sub-phase.

Algorithm 10 The getNextSystemState function

```
1:  procedure GETNEXTSYSTEMSTATE(s_curr)
2:      s_next ← s_curr
3:      if subphase = memoryAndCMS then
4:          if isInitialState(s_curr) = true then
5:              s_next ← setLCMemory(s_curr, max(M − w_Batch, w_LC/γ_LC))
6:          else
7:              if isQoSViolated(Q) = true then
8:                  s_next ← increaseLCMemory(s_curr)        ▷ Granularity: 5% of the working-set size
9:              else
10:                 subphase ← core
11:                 s_next ← decreaseLCCore(s_curr)                          ▷ Granularity: 1 core
12:             end if
13:         end if
14:     else
15:         if isQoSSatisfied(Q) = true then
16:             if isBatchThroughputIncreased = true then
17:                 s_next ← decreaseLCCore(s_curr)
18:             else
19:                 s_next ← s_curr
20:             end if
21:         else
22:             s_next ← increaseLCMemory(s_curr)
23:         end if
24:     end if
25:     if isMemoryChanged(s_curr, s_next) = true then
26:         r_LC,CMS,max ← getCMSMax(s_next, w_LC, γ_LC)
27:         s_next ← setLCCMS(s_next, r_LC,CMS,max)
28:     end if
29:     return s_next
30: end procedure
```

The SSSE builds on the third and fourth observations (i.e., C3 and C4 in Section 7.4) from the characterization study. The third observation is that the throughput of the consolidated containers is likely to significantly improve when the working-sets of the consolidated containers fit in memory through the use of the CMS. Guided by this observation, it attempts to directly allocate the memory and CMS to the LC container in a way that satisfies the following requirements – (1) R1: the amounts of the memory and CMS allocated to the LC container are sufficient to hold the working-set of the LC container and (2) R2: the amount of the remaining memory is just enough to hold the working-set of the batch container (Lines 4–5 and 25–28 in Algorithm 10).[18] This design approach has an advantage of reducing the number of explored system states by skipping inefficient system states.

The SSSE then checks if the LC container's QoS is satisfied with this state. If the QoS is satisfied,

[18] If it is impossible to satisfy both R1 and R2 because the working-set sizes of the LC and batch containers are too large for the total memory capacity even with the use of the CMS, the SSSE utilizes the entire memory allocated to the LC container as the memory pool of the CMS in order to maximize the amount of the remaining memory, which is allocated to the batch container.

it completes the memory and CMS allocation sub-phase and transitions to the core allocation sub-phase (Lines 9–12 in Algorithm 10).

If the LC container's QoS is violated, the SSSE gradually (i.e., 5% of the working-set size of the LC container) increases the amount of the memory and accordingly adjusts the amount of the CMS allocated to the LC container (Lines 7–8 and 25–28 in Algorithm 10). It repeats this process until the QoS is satisfied.

The SSSE transitions to the core allocation sub-phase after completing the memory and CMS allocation sub-phase. During the core allocation sub-phase, it gradually reclaims cores from the LC container and determines the right number of cores for the LC container, which is just enough to satisfy the QoS.

Specifically, the SSSE reduces the LC container's core count by one in each period (Lines 16–17 in Algorithm 10). If the LC container's QoS is satisfied even when allocated with the minimum number of cores or reducing the LC container's core count provides no throughput gain, it terminates the core allocation sub-phase and transitions to the idle phase (Lines 18–19 in Algorithm 10).

If the LC container's QoS is violated with the current core count, the SSSE keeps increasing the amount of the memory (Lines 21–23 in Algorithm 10) and accordingly decreasing the amount of the CMS (Lines 25–28) allocated to the LC container until the QoS is satisfied again. We have made this design decision based on the fourth observation from the characterization study. The observation is that the LC container tends to require a smaller number of cores when allocated with a larger amount of memory and a smaller amount of the CMS because of the decreased CPU utilization of the memory reclaim threads. If the size of the hot data of the batch container is smaller than its working-set, the throughput can be improved by allocating more cores to the batch container (and reducing the amount of the memory allocated to the batch container).

If the LC container's QoS is satisfied with an increased amount of memory and a decreased amount of the CMS, the SSSE attempts to reduce the LC container's core count. It repeats the aforementioned process and then transitions to the idle phase.

Idle Phase: As the SSSE enters the idle phase, it first sets the system state to the best system state (i.e., s_{best}) that achieves the highest throughput while satisfying the QoS among all the explored system states (Lines 32–35 in Algorithm 9). During the idle phase, it keeps monitoring the consolidated containers and the server system but performs no adaptation activities. If a change is detected (e.g., a significant change in the load for the LC container), it transitions to the exploration phase and re-triggers the adaptation process (Lines 18–21 in Algorithm 9).

7.5.3 Resource Allocator

The resource allocator of COSMOS dynamically allocates the resources based on the system state determined by the system state space explorer. The resource allocator builds on cgroups to dynamically allocate cores and memory to the consolidated containers. In addition, the resource allocator uses sysfs to adjust the amount of the CMS allocated to the latency-critical container at runtime.

Table 12: Evaluated workload mixes

Mix	Benchmarks	Mix	Benchmarks
1	memcached and BC	6	silo and BC
2	memcached and BFS	7	silo and BFS
3	memcached and canneal	8	silo and canneal
4	memcached and CC	9	silo and CC
5	memcached and stream	10	silo and stream

7.6 Evaluation

In this section, we evaluate the effectiveness of COSMOS. Specifically, we aim to investigate the following – (1) QoS and throughput, (2) the sensitivity to the load and memory overcommit ratio, (3) the number of explored system states, and (4) the effectiveness of dynamic resource management.

7.6.1 QoS and Throughput

We quantify the effectiveness of COSMOS in terms of QoS and throughput. To keep the discussion focused, we first report the experimental results collected with the medium load and medium memory overcommit ratio (MOR) that represent a common scenario in this section. We then report the experimental results collected with all the loads and MORs in Section 7.6.2.

Based on the two latency-critical (LC) benchmarks and five batch benchmarks discussed in Section 7.3.2, we create ten workload mixes. Each workload mix consists of the LC and batch containers, which contain the LC and batch benchmarks, respectively. Table 12 summarizes the ten workload mixes evaluated in this work.

For each of the ten workload mixes in Table 12, we execute it using the following resource allocation policies – (1) Linux default (LD), which employs the default resource allocation policy (i.e., no core or memory partitioning and compressed memory swap (CMS) enabled with the memory pool size of 20% of the total memory capacity) of Linux, (2) core allocation (CA), which dynamically allocates cores to the consolidated containers, (3) core and memory allocation (CMA), which dynamically allocates cores and memory to the consolidated containers and represents the approach adopted by PARTIES [40] with respect to memory management, (4) exhaustive, which executes the workload mix with a system state, which is discovered by exhaustively exploring the system state space through extensive offline profiling and exhibits the highest throughput while satisfying the QoS among all the explored system states,[19] and (5) COSMOS, which dynamically allocates cores, memory, and CMS to the consolidated containers based on COSMOS.

Figure 47 shows the normalized tail latency of the LC container in each of the ten workload mixes

[19]Note that the exhaustive version is impractical because it requires highly time- and resource-consuming extensive offline profiling for each workload mix. Furthermore, the exhaustive version still requires separate extensive offline profiling for each of the datasets even for the same workload mix. This is because workload mixes tend to exhibit different characteristics with different datasets. Despite the impracticality of the exhaustive version, we compare COSMOS with the exhaustive version to demonstrate that COSMOS can achieve high throughput with strong QoS guarantees without requiring extensive offline profiling.

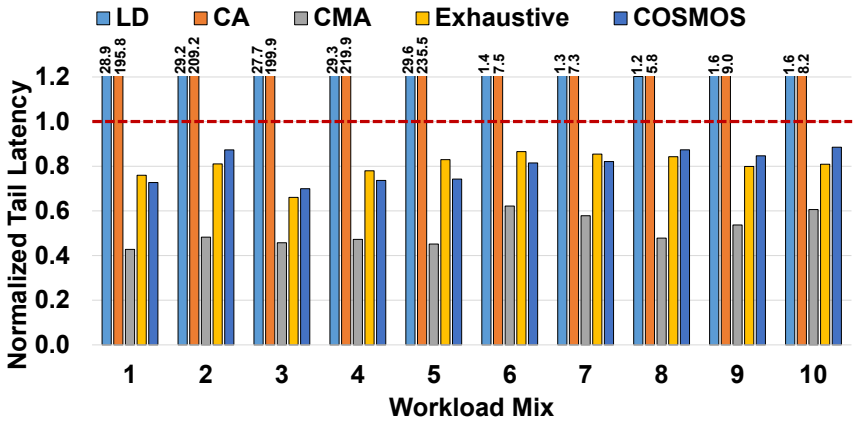


Figure 47: Quality of service

with the medium load and medium MOR. Each bar in Figure 47 reports the tail latency normalized to the target tail latency of the LC container. We observe the following data trends.

First, the `LD` and `CA` versions fail to satisfy the LC container's QoS across all the workload mixes. The `LD` version violates the QoS owing to the contention on the cores that are shared by the consolidated containers.

Since insufficient amounts of the memory and CMS are allocated to the LC container with the `CA` version, the working-set of the LC container does not fit in memory and the victim pages are evicted to the disk swap. Because of the frequent accesses to the disk swap, the `CA` version violates the LC container's QoS.

Second, the `CMA` version satisfies the LC container's QoS across all the workload mixes. Because the number of cores and the amount of memory allocated to the LC container are sufficient, it satisfies the QoS. However, achieving tail latency that is significantly lower than the target tail latency is sub-optimal because it indicates that the LC container is allocated excessive resources, some of which could have been reallocated to the batch container to improve the overall throughput. As discussed later in this section (i.e., Figure 48), the `CMA` version delivers low throughput due to the excessive amount of resources allocated to the LC container.

Third, COSMOS robustly satisfies the LC container's QoS across all workload mixes. By dynamically analyzing the resource requirements of the LC container and allocating cores, memory, and CMS in a coordinated and efficient manner, COSMOS provides strong QoS guarantees for the LC container.

Fourth, the tail latency of COSMOS is closer to the target tail latency than that of the `CMA` version. COSMOS allocates a smaller amount of memory to the LC container through the use of the CMS than the `CMA` version in order to secure a sufficient amount of memory to the batch container. While it makes the tail latency of COSMOS closer to the target tail latency, it is an effective trade-off in that COSMOS still satisfies the LC container's QoS and significantly improves the throughput (as discussed later).

Figure 48 shows the EMU of various versions of the workload mixes. The rightmost bars denote the geometric mean of each version. First, the EMU of the `LD` and `CA` versions across all the workload mixes is zero. EMU is computed to be zero for the `LD` and `CA` versions because they fail to satisfy the

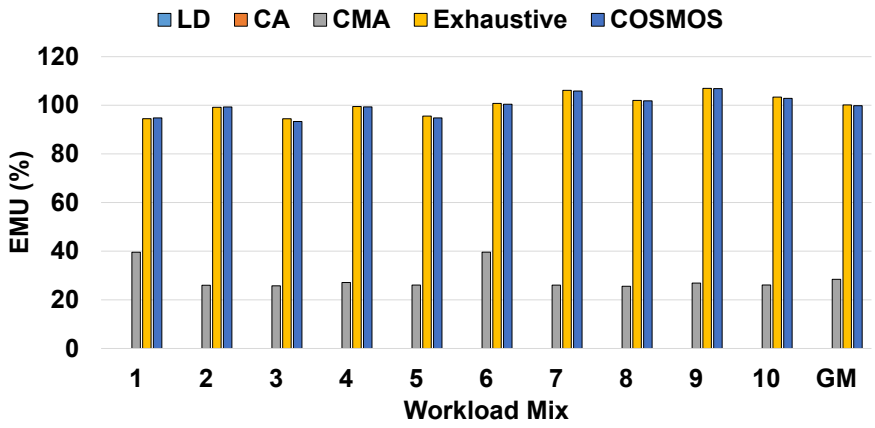


Figure 48: Effective machine utilization

LC container's QoS across all workload mixes.

Second, the `CMA` version delivers low throughput across the workload mixes. Since it lacks the capability of dynamically allocating the CMS to the LC container, it allocates a larger amount of memory to the LC container than the case in which the CMS is also used. Therefore, it allocates an insufficient amount of memory to the batch container, achieving low throughput.

The `CMA` version of the workload mixes 1 and 6, which include `BC` in the batch container, exhibits higher EMU than other workload mixes. With `BC`, some of the nodes in the graph are accessed more frequently than others. Even if the `CMA` version allocates an insufficient amount of memory to `BC`, pages that contain more frequently accessed nodes are likely to remain in the memory without being evicted to the disk swap by the memory reclaim algorithm. This makes the `CMA` version of the workload mixes 1 and 6 access the disk swap less frequently, resulting in higher EMU.

Third, COSMOS consistently achieves high throughput across all the evaluated workload mixes. Specifically, COSMOS delivers 351.1% higher (on average) throughput than the `CMA` version and the throughput similar (i.e., 0.31% lower on average) to the exhaustive version, which uses a system state that is discovered through extensive offline profiling and achieves the highest throughput while satisfying the QoS. Our experimental results demonstrate the effectiveness of COSMOS by showing that it is capable of dynamically finding an efficient system state with high throughput and QoS guarantees for each workload mix without the need for offline profiling.

COSMOS delivers higher EMU with the workload mixes that include `silo` (i.e., the workload mixes 6–10) in the LC container and `BFS` or `CC` (i.e., the workload mixes 2, 4, 7, and 9) in the batch container. As for `silo`, since it requires fewer cores to satisfy its QoS than `memcached`, COSMOS achieves higher EMU by allocating more cores to the batch container.

With regard to `BFS` and `CC`, they exhibit high throughput even when the amount of the memory allocated to them is smaller than their working-set size because some of the nodes are not accessed owing to the connectivity of the graph. COSMOS dynamically identifies their characteristics and allocates a larger amount of the memory and a smaller amount of the CMS to the LC container. This reduces the CPU utilization of the memory reclaim threads because of less frequent data transfers between the

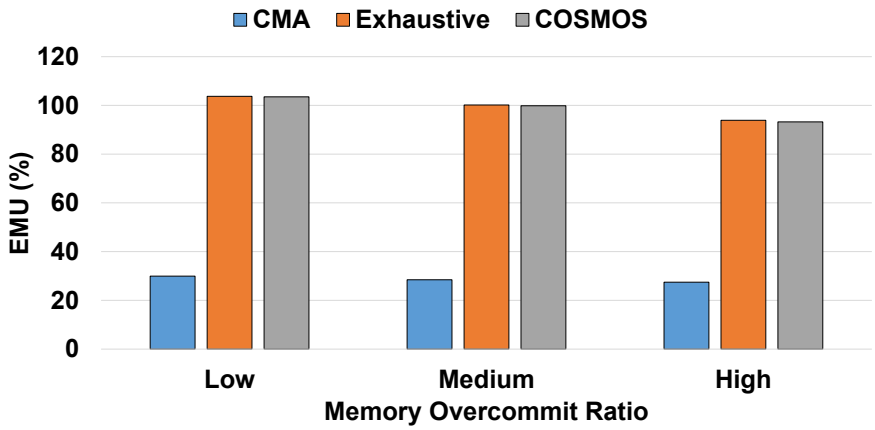


Figure 49: Sensitivity to the memory overcommit ratio

memory and CMS. Because the LC container requires a smaller number of cores with the reduced CPU utilization of the memory reclaim threads, COSMOS achieves higher EMU by allocating more cores to the batch container (i.e., `BFS` or `CC`).

7.6.2 Sensitivity

We investigate the sensitivity of the QoS and throughput achieved by COSMOS to the load for the latency-critical (LC) and the memory overcommit ratio (MOR) of the consolidated containers. We first analyze the sensitivity to the MOR. Figure 49 shows the EMU of the `CMA` and exhaustive versions and COSMOS, which is averaged (using geometric mean) across all of the 10 workload mixes. The data reported in Figure 49 is collected with the medium load and by sweeping the MOR from low to high.

First, COSMOS robustly satisfies the LC container's QoS across all the workload mixes and MORs. Note that it would be impossible to compute the average EMU using geometric mean if COSMOS failed to satisfy the QoS with any of the workload mixes and MORs because EMU would be computed as zero.

Second, COSMOS consistently achieves high throughput across all the workload mixes and MORs. Specifically, COSMOS significantly outperforms the `CMA` version and delivers the throughput similar to the exhaustive version which executes each of the workload mixes with a system state that is discovered through extensive offline profiling and achieves the highest throughput.

Third, the throughput of COSMOS gradually decreases as the MOR increases. This is mainly because COSMOS allocates a smaller amount of memory and a larger amount of CMS to the LC container with a higher MOR to make the working-set of the batch container fit in memory. With a larger amount of the CMS allocated to the LC container, relatively more cores are required because the CPU utilization of the memory reclaim threads increases. Consequently, COSMOS allocates a smaller number of cores to the batch container with a higher MOR, resulting in lower throughput.

We now investigate the sensitivity to the load for the LC container. Figure 50 shows the EMU of the `CMA` and exhaustive versions and COSMOS, which is averaged (using geometric mean) across the workload mixes. The data reported in Figure 50 is collected by sweeping the load from low to high with

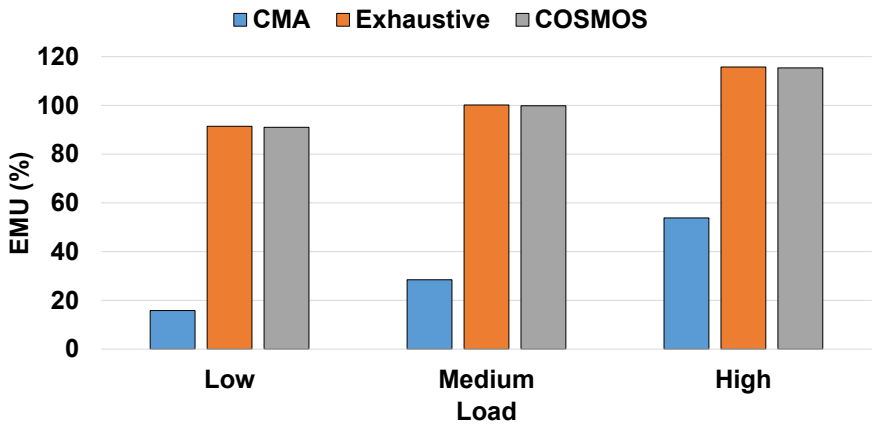


Figure 50: Sensitivity to the load for the LC container

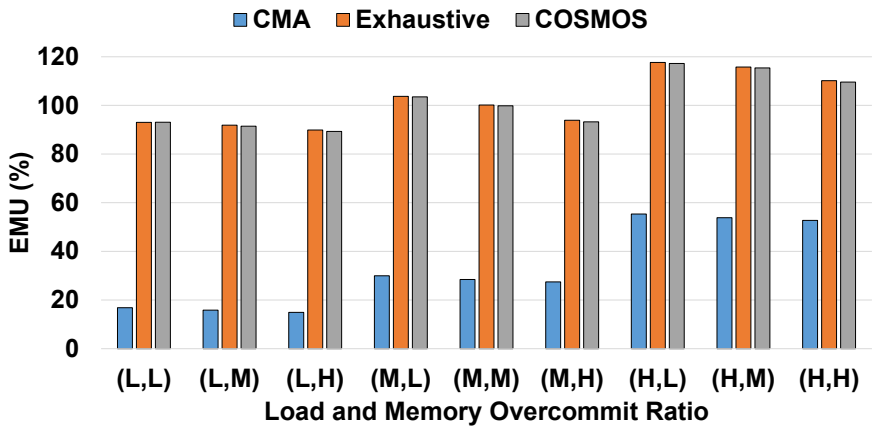


Figure 51: Sensitivity to the load and memory overcommit ratio

the medium MOR.

First, similarly to the sensitivity data trends with the MOR, COSMOS robustly satisfies the LC container's QoS and achieves high throughput across all the workload mixes and loads. Specifically, COSMOS significantly outperforms the `CMA` version and delivers the EMU similar to the exhaustive version which discovers an efficient system state through extensive offline profiling.

Second, the throughput of COSMOS gradually increases as the load for the LC container increases. Because the portion of the EMU contributed by the LC container increases with a higher load, the overall EMU also increases.

For completeness, we report the EMU of the `CMA` and exhaustive versions and COSMOS across all the loads and MORs in Figure 51. L, M, and H in Figure 51 indicate low, medium, and high, respectively. We observe that COSMOS exhibits similar sensitivity trends when the load or MOR is fixed at low or high to the ones (i.e., the load or MOR is fixed at medium) shown in Figures 49 and 50. In addition, COSMOS consistently achieves high throughput across all the evaluated loads and MORs.

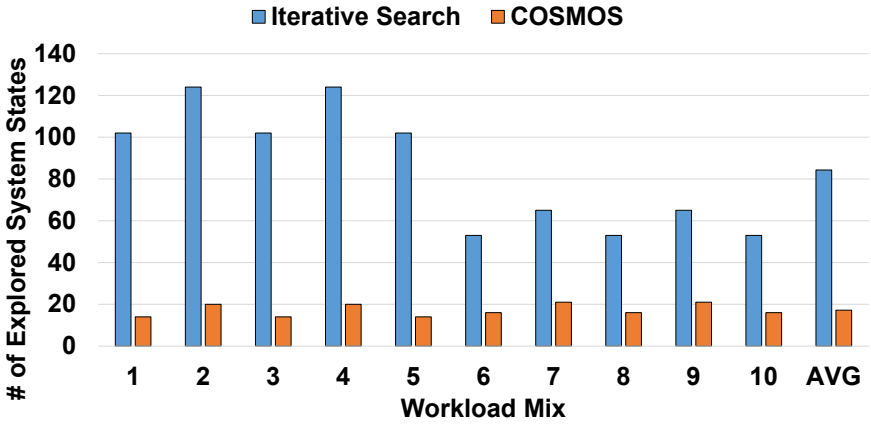


Figure 52: Number of the explored system states

7.6.3 Explored System States

We investigate the impact of the optimization (i.e., skipping inefficient system states) applied to COSMOS on the number of explored system states. To this end, we create a synthetic version of COSMOS that iteratively explores the system state space without applying the optimization (Section 7.5.2).

Figure 52 shows the number of the system states explored by the synthetic version and COSMOS with the medium load and medium memory overcommit ratio (MOR). The optimization applied to COSMOS is effective for reducing the number of the explored system states. Specifically, the full version of COSMOS explores 79.6% fewer system states on average than the synthetic version.

7.6.4 Dynamic Resource Management

We investigate the effectiveness of dynamic resource management supported by COSMOS. We use `memcached` (i.e., the latency-critical (LC) container) and `stream` (i.e., the batch container) with the medium memory overcommit ratio (MOR). The load that the load generator applies to the LC container dynamically varies over the time between the low and high loads in Table 10 by following the diurnal pattern, which is commonly observed in production datacenters [32, 98, 105, 164]. The load generator simulates a 24-hour period. Specifically, the load generator is configured to make each hour in the 24-hour diurnal pattern correspond to four minutes so as to keep the total experiment time manageable.

We execute the consolidated containers using four resource allocation policies – (1) `EX-L`, which statically allocates cores, memory, and compressed memory swap to the consolidated containers based on a system state that is discovered through exhaustive search (with extensive offline profiling) when the LC container is applied with the low load, (2) `EX-M`, which statically allocates resources based on a system state discovered through exhaustive search with the medium load, (3) `EX-H`, which statically allocates resources based on a system state discovered through exhaustive search with the high load, and (4) COSMOS, which dynamically allocates resources using COSMOS. Figure 53 shows the EMU of the four versions. We observe the following data trends.

First, the `EX-L` and `EX-M` versions fail to satisfy the LC container's QoS (i.e., EMU $= 0$) as the load

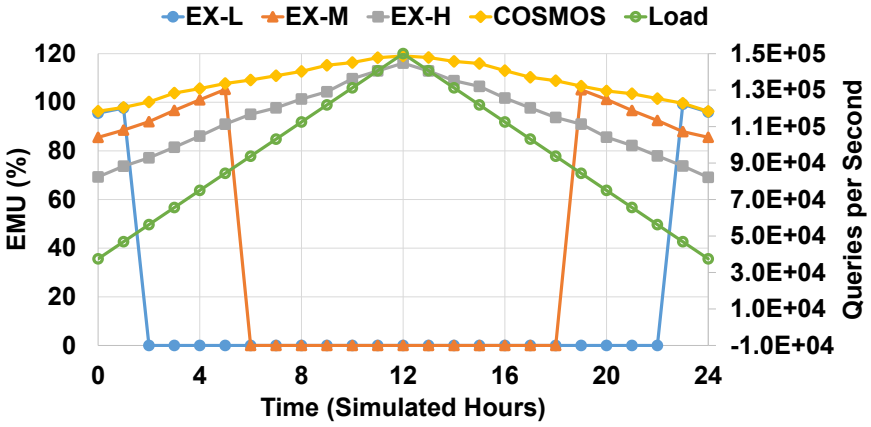


Figure 53: Effectiveness of dynamic resource management

applied to the LC container increases. Since the `EX-L` (or `EX-M`) version statically allocates resources based on a system state discovered through exhaustive search with the low (or medium) load, it allocates insufficient resources to the LC container when the load is relatively high. This results in QoS violations with relatively high loads. Second, the `EX-H` version exhibits relatively low throughput as the load applied to the LC container decreases. Because the `EX-H` version statically allocates resources based on a system state discovered through exhaustive search with the high load, it allocates excessive resources to the LC container and delivers low throughput when the load is relatively low.

Third, COSMOS robustly satisfies the LC container's QoS and achieves high throughput across all the evaluated loads. When COSMOS detects a change in the load, it re-triggers the adaptation process to find a system state that leads to high throughput while satisfying the QoS with the changed load and dynamically allocates resources to the consolidated containers based on the newly-discovered system state. With its re-adaptation capability, COSMOS achieves high throughput with strong QoS guarantees across all the evaluated loads.

Overall, our results demonstrate the effectiveness of COSMOS in that it robustly satisfies the LC container's QoS and delivers high throughput of the consolidated containers across all the evaluated workload mixes, loads, and MORs and significantly reduces the number of explored system states.[20]

7.7 Summary

In this chapter, we present the in-depth characterization of the impact of cores, memory, and compressed memory swap (CMS) on the QoS and throughput of the consolidated applications. Guided by the characterization results, we propose COSMOS, a software-based runtime system for QoS-aware and efficient workload consolidation for memory-intensive applications. Our quantitative evaluation based on a real system and widely-used benchmarks demonstrates that COSMOS robustly satisfies the QoS and achieves high throughput across all the evaluated workload mixes and scenarios and significantly reduces the number of explored system states by skipping inefficient system states.

[20] Our experimental results also show that the performance overhead of COSMOS is small. Specifically, its CPU utilization is 0.27% on average, which is low.

VIII Conclusion

In this dissertation, we investigate heterogeneity-aware resource management techniques for significantly improving the performance and efficiency of data-intensive applications by effectively exploiting heterogeneous computing and memory resources. Modern computing systems employ heterogeneous computing and memory devices to accommodate the ever-growing computational and memory demands of data-intensive applications. Heterogeneous computing and memory have great potential to lead to dramatic improvements in the performance and efficiency of data-intensive applications. Nevertheless, taking full advantage of the capabilities of heterogeneous computing and memory poses significant challenges to system software, as it is the responsibility of the underlying system software to effectively schedule computations on heterogeneous computing devices and manage data placement properly across heterogeneous memory nodes. This dissertation presents system software techniques to tackle the aforementioned challenges through heterogeneity-aware resource management.

First, we present MOSAIC, heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference on heterogeneous embedded systems. MOSAIC uses accurate models for estimating the execution and communication costs of the target inference workload and generates the efficient model slicing and execution plan with low time complexity. Our quantitative evaluation with the widely-used inference workloads and real heterogeneous embedded system shows that MOSAIC significantly reduces inference latency and energy, achieves high estimation accuracy, and incurs small overheads.

Second, we present HERTI, a reinforcement learning (RL)-augmented system for efficient real-time inference on heterogeneous embedded systems. HERTI employs accurate and efficient execution and communication cost estimators to significantly accelerate the training process. HERTI efficiently explores the state space with heterogeneity and constraint awareness and robustly generates the efficient state for the target inference workload with a strong deadline guarantee through RL. Our quantitative evaluation demonstrates the effectiveness of HERTI by showing that it achieves high efficiency with a strong deadline guarantee, delivers larger efficiency gains as the inference deadline and the system heterogeneity increase, exhibits the strong generality for hyper-parameter tuning, and significantly reduces the training time based on its estimation-based approach across all the evaluated inference workloads and scenarios.

Third, we investigate the characteristics of various deep-learning applications on a real heterogeneous memory system. Guided by the characterization results, we propose HALO, hotness- and lifetime-aware data placement and migration for high-performance deep-learning on heterogeneous memory systems. HALO dynamically analyzes the hotness and lifetime characteristics of the tensors of the target deep-learning application and places the tensors across the heterogeneous memory nodes in a hotness- and lifetime-conscious manner. Our experimental results demonstrate the effectiveness of HALO in that it significantly outperforms various memory management policies supported by the underlying system software and hardware, achieves performance comparable to the ideal case with infinite HBM, incurs small performance overheads, and delivers high performance across a wide range of the application

working-set sizes.

Finally, we investigate the system software technique for QoS-aware and efficient workload consolidation on heterogeneous memory systems based on software-defined far memory. We present the in-depth characterization of the impact of cores, memory, and compressed memory swap (CMS) on the QoS and throughput of the consolidated applications. Guided by the characterization results, we propose COSMOS, a software-based runtime system for QoS-aware and efficient workload consolidation for memory-intensive applications. COSMOS dynamically collects the runtime data from the consolidated applications and the underlying system and allocates the resources to the consolidated applications in a way that achieves high throughput with strong QoS guarantees. Our quantitative evaluation based on a real system and widely-used benchmarks demonstrates that COSMOS robustly satisfies the QoS and achieves high throughput across all the evaluated workload mixes and scenarios and significantly reduces the number of explored system states by skipping inefficient system states.

References

[1] Ai chip - aws inferentia. https://aws.amazon.com/machine-learning/inferentia.

[2] cgroups(7) - linux manual page. https://man7.org/linux/man-pages/man7/cgroups.7.html.

[3] Convnet benchmarks. https://github.com/soumith/convnet-benchmarks.

[4] Embedded systems developer kits and modules from nvidia jetson. http://www.nvidia.com/object/embedded-systems-dev-kits-modules.html.

[5] Exynos 2200 mobile processor. https://semiconductor.samsung.com/processor/mobile-processor/exynos-2200/.

[6] Hbm3 icebolt. https://semiconductor.samsung.com/dram/hbm/hbm3-icebolt/.

[7] High voltage power monitor. https://www.msoon.com/.

[8] Hikey970 - 96boards. https://www.96boards.org/product/hikey970/.

[9] Intel optane persistent memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[10] Intel optimized tensorflow wheel now available. https://software.intel.com/en-us/articles/intel-optimized-tensorflow-wheel-now-available.

[11] Intel xeon cpu max series - ai, deep learning, and hpc processor. https://www.intel.com/content/www/us/en/products/details/processors/xeon/max-series.html.

[12] Introducing dnn primitives in intel® math kernel library. https://software.intel.com/en-us/articles/introducing-dnn-primitives-in-intelr-mkl.

[13] iphone 15 pro and 15 pro max - technical specifications. https://www.apple.com/iphone-15-pro/specs/.

[14] Kirin 9000. https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-9000.

[15] Kirin 970. https://www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-970.

[16] Lzo real-time data compression library. http://www.oberhumer.com/opensource/lzo/.

[17] memcached - a distributed memory object caching system. https://memcached.org/.

[18] Memory compression in windows 10 rtm. https://learn.microsoft.com/en-us/shows/seth-juarez/memory-compression-in-windows-10-rtm.

[19] Os x mavericks core technologies overview. https://images.apple.com/media/us/osx/2013/docs/OSX_Mavericks_Core_Technology_Overview.pdf.

[20] Qualcomm snapdragon 888 5g mobile platform. https://www.qualcomm.com/products/snapdragon-888-5g-mobile-platform.

[21] Tensor2tensor - transformer. https://github.com/tbd-ai/tbd-suite/tree/master/MachineTranslation-Transformer/Tensorflow.

[22] Tensorflow benchmarks. https://github.com/tensorflow/benchmarks.

[23] Tensorflow lite. https://www.tensorflow.org/lite/.

[24] Tensorflow optimizations on modern intel® architecture. https://software.intel.com/en-us/articles/tensorflow-optimizations-on-modern-intel-architecture.

[25] zswap. https://www.kernel.org/doc/html/v4.18/vm/zswap.html.

[26] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, OSDI '16, pages 265–283, GA, 2016. USENIX Association.

[27] R. Adolf, S. Rama, B. Reagen, G. y. Wei, and D. Brooks. Fathom: reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, Sept 2016.

[28] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. Page placement strategies for gpus within heterogeneous memory systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 607–618, New York, NY, USA, 2015. ACM.

[29] Andrew Anderson and David Gregg. Optimal dnn primitive selection with partitioned boolean quadratic programming. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, pages 340–351, New York, NY, USA, 2018. ACM.

[30] Shaahin Angizi, Zhezhi He, and Deliang Fan. Dima: A depthwise cnn in-memory accelerator. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, pages 122:1–122:8, New York, NY, USA, 2018. ACM.

[31] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R. Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, Kaushik Roy, and Dejan S. Milojicic. Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 715–731, New York, NY, USA, 2019. ACM.

[32] Dan Ardelean, Amer Diwan, and Chandra Erdman. Performance analysis of cloud applications. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 405–417, Renton, WA, April 2018. USENIX Association.

[33] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery.

[34] Mark Bartlett, Alan M. Frisch, Youssef Hamadi, Ian Miguel, S. Armagan Tarim, and Chris Unsworth. The temporal knapsack problem and its solution. In *Proceedings of the Second International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, CPAIOR'05, pages 34–48, Berlin, Heidelberg, 2005. Springer-Verlag.

[35] S. Bateni, H. Zhou, Y. Zhu, and C. Liu. Predjoule: A timing-predictable energy optimization framework for deep neural networks. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 107–118, Dec 2018.

[36] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite, 2015.

[37] Richard Ernest Bellman. *Dynamic Programming*. Dover Publications, Inc., New York, NY, USA, 2003.

[38] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[39] Ruobing Chen, Haosen Shi, Yusen Li, Xiaoguang Liu, and Gang Wang. Olpart: Online learning based resource partitioning for colocating multiple latency-critical jobs on commodity computers.

In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys '23, page 347–364, New York, NY, USA, 2023. Association for Computing Machinery.

[40] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 107–120, New York, NY, USA, 2019. Association for Computing Machinery.

[41] Shuang Chen, Angela Jin, Christina Delimitrou, and José F. Martínez. Retail: Opting for learning simplicity to enable qos-aware power management in the cloud. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 155–168, 2022.

[42] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 269–284, New York, NY, USA, 2014. ACM.

[43] X. Chen, D. Z. Chen, and X. S. Hu. modnn: Memory optimal dnn training on gpus. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, DATE'18, pages 13–18, March 2018.

[44] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press.

[45] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 571–582, Berkeley, CA, USA, 2014. USENIX Association.

[46] Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. Bear: Techniques for mitigating bandwidth bloat in gigascale dram caches. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 198–210, New York, NY, USA, 2015. ACM.

[47] Ryan Cochran, Can Hankendi, Ayse K. Coskun, and Sherief Reda. Pack & cap: Adaptive dvfs and thread packing under power caps. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 175–185, 2011.

[48] Alexei Colin, Graham Harvey, Brandon Lucia, and Alanson P. Sample. An energy-interference-free hardware-software debugger for intermittent energy-harvesting systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 577–589, New York, NY, USA, 2016. ACM.

[49] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[50] Quan Deng, Lei Jiang, Youtao Zhang, Minxuan Zhang, and Jun Yang. Dracc: A dram based accelerator for accurate cnn inference. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 168:1–168:6, New York, NY, USA, 2018. ACM.

[51] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A hybrid memory page scheduler with machine intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '19, pages 37–48, New York, NY, USA, 2019. Association for Computing Machinery.

[52] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[53] Amin Firoozshahian et al. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery.

[54] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5, August 2004.

[55] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. Large pages may be harmful on numa systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 231–242, Berkeley, CA, USA, 2014. USENIX Association.

[56] J. Gaur, M. Chaudhuri, P. Ramachandran, and S. Subramoney. Near-optimal access partitioning for memory hierarchies with multiple heterogeneous bandwidth sources. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 13–24, Feb 2017.

[57] Soroush Ghodrati, Hardik Sharma, Sean Kinzer, Amir Yazdanbakhsh, Jongse Park, Nam Sung Kim, Doug Burger, and Hadi Esmaeilzadeh. Mixed-signal charge-domain acceleration of deep neural networks through interleaved bit-partitioned arithmetic. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, PACT '20, page 399–411, New York, NY, USA, 2020. Association for Computing Machinery.

[58] Myeonggyun Han and Woongki Baek. Herti: A reinforcement learning-augmented system for efficient real-time inference on heterogeneous embedded systems. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 90–102, 2021.

[59] Myeonggyun Han and Woongki Baek. Sdrp: Safe, efficient, and slo-aware workload consolidation through secure and dynamic resource partitioning. *IEEE Transactions on Services Computing*, 15(4):1868–1882, 2022.

[60] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, and Woongki Baek. Hotness- and lifetime-aware data placement and migration for high-performance deep learning on heterogeneous memory systems. *IEEE Transactions on Computers*, 69(3):377–391, 2020.

[61] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. Mosaic: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 165–177, 2019.

[62] Myeonggyun Han, Eunseong Park, Youngsam Shin, Deok-Jae Oh, Yeongon Cho, and Woongki Baek. Cosmos: Coordinated management of cores, memory, and compressed memory swap for qos-aware and efficient workload consolidation for memory-intensive applications. *IEEE Access*, 11:133199–133214, 2023.

[63] Myeonggyun Han, Jinsu Park, and Woongki Baek. Chrt: A criticality- and heterogeneity-aware runtime system for task-parallel applications. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, pages 942–945, 3001 Leuven, Belgium, Belgium, 2017. European Design and Automation Association.

[64] Myeonggyun Han, Jinsu Park, and Woongki Baek. Design and implementation of a criticality- and heterogeneity-aware runtime system for task-parallel applications. *IEEE Transactions on Parallel and Distributed Systems*, 32(5):1117–1132, 2021.

[65] Myeonggyun Han, Seongdae Yu, and Woongki Baek. Secure and dynamic core and cache partitioning for safe and efficient server consolidation. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 311–320, 2018.

[66] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 243–254, Piscataway, NJ, USA, 2016. IEEE Press.

[67] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision – ECCV 2016*, ECCV'16, pages 630–645, Cham, 2016. Springer International Publishing.

[68] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, pages 82–95, New York, NY, USA, 2017. ACM.

[69] Jeaho Hwang, Jinkyu Jeong, Hwanju Kim, Jeonghwan Choi, and Joonwon Lee. Compressed memory swap for qos of virtualized embedded systems. *IEEE Transactions on Consumer Electronics*, 58(3):834–840, 2012.

[70] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 968–981. IEEE Press, 2020.

[71] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[72] A. Jain, A. Phanishayee, J. Mars, L. Tang, and G. Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 776–789, June 2018.

[73] Seth Jennings. Transparent memory compression in linux, 2013.

[74] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. Ionn: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 401–411, New York, NY, USA, 2018. ACM.

[75] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22Nd ACM International Conference on Multimedia*, MM '14, pages 675–678, New York, NY, USA, 2014. ACM.

[76] Norman P. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[77] Norman P. Jouppi, Doe Hyun Yoon, Matthew Ashcraft, Mark Gottscho, Thomas B. Jablin, George Kurian, James Laudon, Sheng Li, Peter Ma, Xiaoyu Ma, Thomas Norrie, Nishant Patil, Sushma Prasad, Cliff Young, Zongwei Zhou, and David Patterson. Ten lessons from three generations shaped google's tpuv4i : Industrial product. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–14, 2021.

[78] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 615–629, New York, NY, USA, 2017. ACM.

[79] Harshad Kasture and Daniel Sanchez. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10, 2016.

[80] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack problems*. Springer, 2004.

[81] Jongseok Kim, Cheolgi Kim, and Euiseong Seo. *ezswap* : Enhanced compressed swap scheme for mobile devices. *IEEE Access*, 7:139678–139691, 2019.

[82] Kyu Yeun Kim and Woongki Baek. Blpp: Improving the performance of gpgpus with heterogeneous memory through bandwidth- and latency-aware page placement. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 358–365, Oct 2018.

[83] Seontae Kim, Nguyen Pham, Woongki Baek, and Young-ri Choi. Holistic vm placement for distributed parallel applications in heterogeneous clusters. *IEEE Transactions on Services Computing*, pages 1–1, 2018.

[84] Y. G. Kim and C. J. Wu. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1082–1096, 2020.

[85] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 759–773, USA, 2018. USENIX Association.

[86] Marios Kogias, Stephen Mallon, and Edouard Bugnion. Lancet: A self-correcting latency measuring tool. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 881–896, Renton, WA, July 2019. USENIX Association.

[87] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 399–411, New York, NY, USA, 2016. ACM.

[88] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[89] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc.

[90] Neeraj Kulkarni, Feng Qi, and Christina Delimitrou. Pliant: Leveraging approximation to improve datacenter resource efficiency. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 159–171, 2019.

[91] Rakesh Kumar, Keith I. Farkas, Norman P. Jouppi, Parthasarathy Ranganathan, and Dean M. Tullsen. Single-isa heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 81–, Washington, DC, USA, 2003. IEEE Computer Society.

[92] Youngeun Kwon and Minsoo Rhu. Beyond the memory wall: A case for memory-centric hpc system for deep learning. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-51, pages 148–161, Piscataway, NJ, USA, 2018. IEEE Press.

[93] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 317–330, New York, NY, USA, 2019. Association for Computing Machinery.

[94] Seok-Hee Lee. Technology scaling challenges and opportunities of memory devices. In *2016 IEEE International Electron Devices Meeting (IEDM)*, pages 1.1.1–1.1.8, 2016.

[95] Changlong Li, Liang Shi, Yu Liang, and Chun Jason Xue. Seal: User experience-aware two-level swap for mobile devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):4102–4114, 2020.

[96] Lisha Li, Kevin G. Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: Bandit-based configuration evaluation for hyperparameter optimization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[97] Chieh-Jan Mike Liang, Hui Xue, Mao Yang, Lidong Zhou, Lifei Zhu, Zhao Lucis Li, Zibo Wang, Qi Chen, Quanlu Zhang, Chuanjie Liu, and Wenjun Dai. Autosys: The design and operation of learning-augmented systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 323–336. USENIX Association, July 2020.

[98] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 301–312, 2014.

[99] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM.

[100] Chris A. Mack. Fifty years of moore's law. *IEEE Transactions on Semiconductor Manufacturing*, 24(2):202–207, 2011.

[101] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/big-data-the-next-frontier-for-innovation.

[102] Mitchell P. Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz, and Ann Taylor. Treebank-3. https://catalog.ldc.upenn.edu/ldc99t42.

[103] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[104] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[105] David Meisner, Christopher M. Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. Power management of online data-intensive services. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 319–330, New York, NY, USA, 2011. ACM.

[106] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning (ICML)*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515, Long Beach, California, USA, Jun 2019. PMLR.

[107] Charith Mendis, Cambridge Yang, Yewen Pu, Dr.Saman Amarasinghe, and Michael Carbin. Compiler auto-vectorization with imitation learning. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32, pages 14625–14635. Curran Associates, Inc., 2019.

[108] Azalia Mirhoseini, Hieu Pham, Quoc V. Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, ICML'17, pages 2430–2439, 2017.

[109] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan

Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[110] Thannirmalai Somu Muthukaruppan, Mihai Pricopi, Vanchinathan Venkataramani, Tulika Mitra, and Sanjay Vishin. Hierarchical power management for asymmetric multi-core in dark silicon era. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 174:1–174:9, New York, NY, USA, 2013. ACM.

[111] Konstantinos Nikas, Nikela Papadopoulou, Dimitra Giantsidi, Vasileios Karakostas, Georgios Goumas, and Nectarios Koziris. Dicer: Diligent cache partitioning for efficient workload consolidation. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP '19, New York, NY, USA, 2019. Association for Computing Machinery.

[112] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell. Hipster: Hybrid task manager for latency-critical cloud workloads. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, HPCA '17, pages 409–420, Feb 2017.

[113] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander. Twig: Multi-agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 167–179, 2020.

[114] Young H. Oh, Quan Quan, Daeyeon Kim, Seonghak Kim, Jun Heo, Sungjun Jung, Jaeyoung Jang, and Jae W. Lee. A portable, automatic data qantizer for deep neural networks. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 17:1–17:14, New York, NY, USA, 2018. ACM.

[115] Prasanna Pandit and R. Govindarajan. Fluidic kernels: Cooperative execution of opencl programs on multiple heterogeneous devices. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 273:273–273:283, New York, NY, USA, 2014. ACM.

[116] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 27–40, New York, NY, USA, 2017. ACM.

[117] Jinsu Park and Woongki Baek. Hap: A heterogeneity-conscious runtime system for adaptive pipeline parallelism. In *Proceedings of the 22Nd International Conference on Euro-Par 2016: Parallel Processing - Volume 9833*, Euro-Par '16, pages 518–530, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[118] Jinsu Park and Woongki Baek. Rchc: A holistic runtime system for concurrent heterogeneous computing. In *2016 45th International Conference on Parallel Processing (ICPP)*, ICPP '16, pages 211–216, Aug 2016.

[119] Jinsu Park, Seongbeom Park, and Woongki Baek. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 10:1–10:16, New York, NY, USA, 2019. ACM.

[120] Jinsu Park, Seongbeom Park, Myeonggyun Han, and Woongki Baek. Palm: Progress- and locality-aware adaptive task migration for efficient thread packing. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 330–339, 2021.

[121] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. Hypart: A hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 5:1–5:14, New York, NY, USA, 2018. ACM.

[122] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[123] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[124] Anuj Pathania, Alexandru Eugen Irimiea, Alok Prakash, and Tulika Mitra. Power-performance modelling of mobile gaming workloads on heterogeneous mpsocs. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 201:1–201:6, New York, NY, USA, 2015. ACM.

[125] A. Prakash, Siqi Wang, A.E. Irimiea, and T. Mitra. Energy-efficient execution of data-parallel applications on heterogeneous mobile platforms. In *Computer Design (ICCD), 2015 33rd IEEE International Conference on*, ICCD '15, pages 208–215, Oct 2015.

[126] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 325–341, New York, NY, USA, 2017. Association for Computing Machinery.

[127] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N. Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *2011 IEEE International Symposium on Workload Characterization (IISWC)*, pages 116–125, 2011.

[128] Hang Qi, Evan R. Sparks, and Ameet Talwalkar. Paleo: A performance model for deep neural networks. In *ICLR*, 2017.

[129] S. Ramos and T. Hoefler. Capability models for manycore memory systems: A case-study with xeon phi knl. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 297–306, May 2017.

[130] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, pages 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[131] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. ZeRO-Offload: Democratizing Billion-Scale model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564. USENIX Association, July 2021.

[132] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 18:1–18:13, Piscataway, NJ, USA, 2016. IEEE Press.

[133] Francisco Romero and Christina Delimitrou. Mage: Online and interference-aware scheduling for multi-scale heterogeneous systems. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, New York, NY, USA, 2018. Association for Computing Machinery.

[134] S. Roy, R. Kumar, and M. Prvulovic. Improving system performance with compressed memory. In *Proceedings 15th International Parallel and Distributed Processing Symposium. IPDPS 2001*, pages 7 pp.–, 2001.

[135] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vision*, 115(3):211–252, December 2015.

[136] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. A case for granularity aware page migration. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 352–362, New York, NY, USA, 2018. ACM.

[137] A.I. Saichev, Y. Malevergne, and D. Sornette. *Theory of Zipf's Law and Beyond*. Lecture Notes in Economics and Mathematical Systems. Springer Berlin Heidelberg, 2009.

[138] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR'18, June 2018.

[139] Hiroshi Sasaki, Satoshi Imamura, and Koji Inoue. Coordinated power-performance optimization in manycores. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, pages 51–61, 2013.

[140] Satyabrata Sen and Neena Imam. Machine learning based design space exploration for hybrid main-memory design. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 480–489, New York, NY, USA, 2019. Association for Computing Machinery.

[141] Ankit Sethia and Scott Mahlke. Equalizer: Dynamic tuning of gpu resources for efficient execution. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '14, pages 647–658, Washington, DC, USA, 2014. IEEE Computer Society.

[142] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[143] Avinash Sodani. Knights landing (knl): 2nd generation intel® xeon phi processor. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–24, 2015.

[144] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, March 2016.

[145] Lili Song, Ying Wang, Yinhe Han, Xin Zhao, Bosheng Liu, and Xiaowei Li. C-brain: A deep learning accelerator that tames the diversity of cnns through adaptive data-level parallelization. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 123:1–123:6, New York, NY, USA, 2016. ACM.

[146] M. Song, Y. Hu, H. Chen, and T. Li. Towards pervasive and user satisfactory cnn across gpu microarchitectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, Feb 2017.

[147] Taejoon Song, Myeongseon Kim, Gunho Lee, and Youngjin Kim. Prediction-guided performance improvement on compressed memory swap. In *2022 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–6, 2022.

[148] Z. Song, F. Wu, X. Liu, J. Ke, N. Jing, and X. Liang. Vr-dann: Real-time video recognition via decoder-assisted neural network acceleration. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 698–710, 2020.

[149] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. Feedback-driven threading: Power-efficient and high-performance execution of multi-threaded workloads on cmps. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 277–286, New York, NY, USA, 2008. Association for Computing Machinery.

[150] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

[151] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, CVPR'15, pages 1–9, June 2015.

[152] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, AAAI'17, pages 4278–4284. AAAI Press, 2017.

[153] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le. MnasNet: Platform-Aware Neural Architecture Search for Mobile. *ArXiv e-prints*, July 2018.

[154] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[155] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 213–224, Washington, DC, USA, 2012. IEEE Computer Society.

[156] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[157] H. Wang, J. Zhang, S. Shridhar, G. Park, M. Jung, and N. S. Kim. Duang: Fast and lightweight page migration in asymmetric memory systems. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 481–493, March 2016.

[158] Linnan Wang, Jinmian Ye, Yiyang Zhao, Wei Wu, Ang Li, Shuaiwen Leon Song, Zenglin Xu, and Tim Kraska. Superneurons: Dynamic gpu memory management for training deep neural networks. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '18, pages 41–53, New York, NY, USA, 2018. ACM.

[159] X. Wang, J. Yu, C. Augustine, R. Iyer, and R. Das. Bit prudent in-cache acceleration of deep convolutional neural networks. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 81–93, Feb 2019.

[160] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 609–621, New York, NY, USA, 2022. Association for Computing Machinery.

[161] Shasha Wen, Lucy Cherkasova, Felix Xiaozhu Lin, and Xu Liu. Profdp: A lightweight profiler to guide data placement in heterogeneous memory systems. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 263–273, New York, NY, USA, 2018. ACM.

[162] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 331–345, New York, NY, USA, 2019. ACM.

[163] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 607–618, New York, NY, USA, 2013. ACM.

[164] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020.

[165] Vinson Young, Chiachen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. Accord: Enabling associativity for gigascale dram caches by coordinating way-install and way-prediction. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 328–339, Piscataway, NJ, USA, 2018. IEEE Press.

[166] Seongdae Yu, Seongbeom Park, and Woongki Baek. Design and implementation of bandwidth-aware memory placement and migration policies for heterogeneous memory systems. In *Proceedings of the International Conference on Supercomputing*, ICS '17, pages 18:1–18:10, New York, NY, USA, 2017. ACM.

[167] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-efficient dram caching via software/hardware cooperation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 1–14, New York, NY, USA, 2017. ACM.

[168] Jaeyoung Yun, Jinsu Park, and Woongki Baek. Hars: A heterogeneity-aware runtime system for self-adaptive multithreaded applications. In *Proceedings of the 52Nd Annual Design Automation Conference*, DAC '15, pages 107:1–107:6, New York, NY, USA, 2015. ACM.

[169] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 2–2, Berkeley, CA, USA, 2012. USENIX Association.

[170] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *CoRR*, abs/1409.2329, 2014.

[171] Hao Zhang, Zeyu Zheng, Shizhen Xu, Wei Dai, Qirong Ho, Xiaodan Liang, Zhiting Hu, Jinliang Wei, Pengtao Xie, and Eric P. Xing. Poseidon: An efficient communication architecture for distributed deep learning on GPU clusters. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 181–193, Santa Clara, CA, 2017. USENIX Association.

[172] Haishan Zhu and Mattan Erez. Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 33–47, New York, NY, USA, 2016. ACM.

Acknowledgements

First of all, I would like to express my deepest gratitude to my advisor, Professor Woongki Baek. His dedicated guidance and generous support have always encouraged me throughout my graduate studies. His invaluable advice and inspiration guided me to dive into important research problems and pointed me in the right direction during my research. This dissertation would not have been possible without his mentorship. It was a great honor to be advised by him.

I would also like to thank the committee members of my dissertation, Professor Young-ri Choi, Professor Myeongjae Jeon, Professor Seulki Lee, and Professor Jongeun Lee. They provided valuable comments and feedback to improve the quality of this dissertation.

I am grateful to all my lab colleagues in the Intelligent System Software Lab (ISSL) for being great colleagues, supporting my research, and enabling a productive working environment. I have learned a lot from my lab colleagues and collaboration with them has been an excellent research experience. I would also like to thank my fellow graduate students in the Center for Innovative System Software Research (CISSR) group for sharing their insights on a range of systems software research topics.

I would like to express my sincere gratitude to my family for their unconditional love and support during my graduate studies. I am grateful to my parents for always believing in me and encouraging me with their optimism to pursue my dreams. They have made all of this happen.

Last but not least, I am truly grateful to my beloved wife, Gyeongjin Lee, for her deep love and unwavering support for me. She has provided the emotional support that I needed to overcome all the difficulties I had. She has always been my source of happiness. I am truly lucky to have her be with me and make my life bright. Without her, I could not have done all of this.