4-2024

An Intelligent Framework for Efficiently Utilizing Distributed Heterogeneous Resources to Improve HPC Application Performance

Moiz Arif
ma3890@rit.edu

An Intelligent Framework for Efficiently Utilizing Distributed Heterogeneous Resources to Improve HPC Application Performance

by

Moiz Arif

A dissertation submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
in Computing and Information Sciences

B. Thomas Golisano College of Computing and
Information Sciences

Rochester Institute of Technology
Rochester, New York
April 2024

An Intelligent Framework for Efficiently Utilizing Distributed Heterogeneous Resources to Improve HPC Application Performance

by

Moiz Arif

Committee Approval:

We, the undersigned committee members, certify that we have advised and/or supervised the candidate on the work described in this dissertation. We further certify that we have reviewed the dissertation manuscript and approve it in partial fulfillment of the requirements of the degree of Doctor of Philosophy in Computing and Information Sciences.

Dr. M. Mustafa Rafique Dissertation Advisor	Date
Dr. Sudharshan Vazhkudai Dissertation Committee Member	Date
Dr. Fawad Ahmad Dissertation Committee Member	Date
Dr. Minseok Kwon Dissertation Committee Member	Date
Dr. Zachary Butler Dissertation Defense Chairperson	Date

Certified by:

Dr. Pencheng Shi Ph.D. Program Director, Computing and Information Sciences	Date

An Intelligent Framework for Efficiently Utilizing Distributed Heterogeneous Resources to Improve HPC Application Performance

by
Moiz Arif

Submitted to the
B. Thomas Golisano College of Computing and Information Sciences Ph.D. Program in Computing and Information Sciences
in partial fulfillment of the requirements for the
Doctor of Philosophy Degree
at the Rochester Institute of Technology

Abstract

High-Performance Computing (HPC) workloads are being widely used to solve complex problems in scientific applications from diverse domains, such as weather forecasting, medical diagnostics, and fluid dynamics simulation. HPC workloads are traditionally executed on baremetal HPC systems, containers, functions, or as workflows or ensembles. These workloads consume a large amount of data and have large memory and storage requirements that typically exceed the limited amount of main memory and storage available on an HPC system. HPC workloads such as deep learning (DL) are executed on platforms such as TensorFlow or PyTorch, are oblivious to the availability and performance profiles of the underlying HPC systems, and do not incorporate resource requirements of the given workloads for distributed training. Function-as-a-Service (FaaS) platforms running HPC functions impose resource-level constraints, specifically fixed memory allocation and short task timeouts, that lead to job failures, thus making these desirable platforms unreliable for guaranteeing function execution and ensuring performance requirements for stateful applications such as DL workloads. Containerized workflow execution of HPC jobs requires several terabytes of memory that exceed node capacity, resulting in excessive data swapping to slower storage, degraded job performance, and failures. Similarly, co-located bandwidth-intensive, latency-sensitive, or short-lived workflows suffer from degraded performance due to contention, memory exhaustion, and higher access latency due to suboptimal memory allocation. Recently, tiered memory systems comprising persistent memory and compute express link (CXL) have been explored to provide additional memory capacity and bandwidth to memory-constrained systems and applications. However, current memory allocation and management techniques for tiered memory subsystems are inadequate to meet the diverse needs of colocated containerized jobs in HPC systems that run workflows and ensembles at scale concurrently.

In this research, we propose a framework that makes HPC platforms, workflow management systems (WMS), and HPC schedulers aware of the availability and capabilities of the underlying heterogeneous datacenter resources and optimize the performance of HPC workloads. We propose architectural improvements and new software modules leveraging the latest advancements in the memory subsystem, specifically CXL, to provide additional memory and fast scratch space for HPC workloads to reduce the overall model training time while enabling HPC jobs to efficiently train models using data that is much larger than the installed system memory. The proposed framework manages the allocation of additional CXL-based memory, introduces a fast intermediate storage tier, provides intelligent prefetching and caching mechanisms for HPC workloads. We leverage tiered memory systems for HPC execution and propose efficient memory management policies including intelligent page placement and eviction policies to improve memory access performance. Our page allocation and replacement policies incorporate task characteristics and enable efficient memory sharing between workflows. We integrate our policies with the popular HPC scheduler, SLURM, and container runtime, Singularity, to show that our approach improves tiered memory utilization and application performance. Similarly, we also integrate our framework with a popular DL platform, TensorFlow, and Apache OpenWhisk to introduce infrastructure-aware scheduling, performance optimization of DL workloads, introduce resilience and fault-tolerance to FaaS platforms. The evaluation of our proposed framework reveals improved system utilization, throughput, and performance, as well as reduced training time, failure rate, recovery time, latency, and cold-start time for large-scale deployments.

Acknowledgments

I extend my heartfelt appreciation to my advisor for his exceptional guidance, unwavering support, and invaluable insights throughout my doctoral journey. His expertise, encouragement, and patience have profoundly influenced my research and academic development. Additionally, I am grateful to my dissertation committee members for their constructive feedback, valuable suggestions, and scholarly contributions to my work. I am deeply thankful to my fellow researchers for their collaboration, support, and camaraderie during this journey. Their insights, discussions, and encouragement have been pivotal in overcoming challenges and reaching milestones. Moreover, I wish to acknowledge the financial assistance provided by the National Science Foundation (grant no. 2106635), which enabled me to pursue my research aspirations and contribute to advancing knowledge in my field.

Finally, I would like to express my deepest gratitude to my loving wife and family for their unwavering support, understanding, and encouragement throughout my journey to completing this dissertation. Their patience, love, and sacrifices have been the cornerstone of my success, providing me with the strength and motivation to overcome challenges and pursue my academic goals. I am truly blessed to have such a supportive and caring family by my side, and I am forever grateful for their endless love and encouragement. Additionally, I extend heartfelt gratitude to my toddler son, whose infectious joy and boundless energy infused each day with inspiration and purpose.

Contents

List of Figures

Chapter 1

Introduction

1.1 High-level Problem Statement

Running HPC workloads across heterogeneous datacenter resources as bare-metal, containerized, functions, or workflows poses various challenges, since each execution environment has different characteristics and distinct performance profiles. To avoid execution stalls and failures new heterogeneity-aware resource management techniques need to be developed that exploit the capabilities of heterogeneous datacenter servers and leverage advancements in hardware technologies to maximize workload performance. Optimizing the performance of HPC and scientific workloads on heterogeneous datacenters has several challenges:

1. Resource heterogeneity [170] in datacenters results in varying performance profiles. Unaware of such profiles, schedulers and platforms schedule execution on resource-constrained servers, which results in a significant increase in the overall execution time [52].

2. Unavailability of system resources causes a significant increase in overall training time as the input training data and model parameters are frequently swapped to slower memory and storage tiers during the training process.

3. Latest advancements in hardware technologies introduce new features that require platforms to adapt and leverage such advancements to reduce the time-to-answer of time-sensitive applications and improve the utilization of available data center resources.

4. Serverless computing imposes resource-level constraints, specifically fixed memory allocation

and short task timeouts, that lead to job failures.

5. Failures in FaaS have not been fully investigated, which makes such platforms unreliable for guaranteeing function execution and ensuring performance requirements of HPC workloads.

6. Traditional job scheduling and memory allocation approaches for GPU-based HPC workloads leveraging tiered memory create contention, reduced throughput, and increase the overall data transfer time.

7. Current memory allocation and management techniques for co-located containerized HPC workflows on tiered memory systems are sub-optimal and inadequate to meet the resource demands of large-scale workflows running at scale.

In this research, we focus on addressing the above challenges to make HPC and DL platforms aware of the underlying resource heterogeneity, address the limitations of limited memory availability to functions by leveraging tiered memory resources, and provide on-demand memory to functions. We also aim to introduce fault tolerance to FaaS platforms and reduce recovery time for stateful time-sensitive scientific workloads.

1.2 Detailed Problem Statement

Large-scale datacenters often consist of tens of thousands of servers, with new servers added incrementally over time to meet the growing demands of modern high-performance computing (HPC) applications. As hardware technologies evolve to support the compute and I/O requirements of complex scientific workloads such as deep learning (DL), datacenters exhibit performance and architectural heterogeneity. This leads to performance and architectural heterogeneity [170], which must be addressed in managing datacenter resources. Managing resources in such heterogeneous environments poses challenges, as most software platforms are designed for homogeneous datacenters, leading to suboptimal performance. Most software platforms are designed for homogeneous datacenters and lead to sub-optimal performance when executing in heterogeneous environments [45, 70, 162, 217]. The increasing popularity of heterogeneous datacenters, equipped with diverse compute, memory, storage, and network resources, aims to address the needs of time-sensitive applications. Baremetal, containerized or Function-as-a-Service (FaaS) platforms running HPC workloads such as TensorFlow [27], PyTorch [185], Pegasus [69], and Apache OpenWhisk [23], often lack awareness of the underlying datacenter resources' availability and performance profiles, resulting in inefficient resource utilization during distributed training.

HPC workloads such as DL [72] are extensively applied across various scientific domains, including weather forecasting, medical diagnostics, and fluid dynamics simulation. DL workloads entail significant data consumption and utilize large-scale HPC systems for model training. However, these workloads often exceed the available main memory on HPC servers, leading to computational losses and time delays. Additionally, resource inefficiencies arise, with some HPC servers idling while others experience performance degradation due to congestion caused by straggling processes.

Apart from the traditional bare-metal execution of HPC workloads, serverless and FaaS execution models are rapidly gaining traction due to their seamless application deployment and scalability features. These platforms have recently been explored for running data-intensive HPC workloads [232, 241], including DL [41, 117], aiming to enhance application performance and reduce execution costs. Stateful execution, where applications produce intermediate data required for subsequent processing, contrasts with stateless execution, where application components' states or data production are independent. Many stateful applications have migrated [57] to FaaS platforms due to their ease of deployment, scalability, and minimal management overhead. However, FaaS platforms impose resource constraints such as fixed memory allocation, leading to job failures. Limited system memory availability or inadequate memory allocation to application functions can result in premature function termination. In summary, memory limitations imposed by FaaS platforms and underlying servers contribute to application-level failures associated with memory constraints.

Running HPC workloads on accelerators like Graphics Processing Units (GPUs) for running HPC workloads accelerates computation but exacerbates memory and data issues due to the faster processing speed. GPU-based HPC workloads, often data-intensive, involve large-scale simulations, complex computations, and massive data processing, necessitating efficient memory and data operations for optimal performance and scalability. Such workloads encounter limitations in memory capacity and bandwidth, primarily due to the constrained onboard High Bandwidth Memory (HBM) [248]. To address this, GPUs frequently read/write data to the main memory, leading to performance bottlenecks. In multi-GPU setups, memory pinning on the main memory restricts memory availability for other GPUs, resulting in data transfer overheads and potential job failures. The introduction of CXL-attached memory expands memory availability for GPU-based workloads. However, suboptimal memory allocations by the underlying OS or GPU drivers may hinder efficient memory mappings. These inefficient mappings can impede data movement, leading to reduced memory throughput and increased application execution times.

HPC workflows, often comprising data and memory-intensive tasks, require efficient and coordinated execution. These workflows exhibit unique memory demands based on factors like data size, computational complexity, and I/O activity. Containerized workflow execution of HPC jobs requires several terabytes of memory that exceed node capacity, leading to excessive data swapping to slower storage, degraded performance, and job failures. Similarly, co-located bandwidth-intensive or latency-sensitive workflows face performance degradation [29,77,78,108] due to contention, memory exhaustion, and suboptimal memory allocation. Tiered memory systems, featuring multiple memory tiers, aim to augment memory capacity and bandwidth for memory-constrained systems and applications. However, current memory allocation techniques for tiered memory subsystems are inadequate in addressing the diverse needs of colocated containerized jobs in large-scale HPC systems running workflows and ensembles concurrently.

1.3 Motivation

In the following, we discuss the motivation for developing capability-aware scheduling and resource management for heterogeneous datacenters and propose intelligent mechanisms to improve the performance of HPC workloads.

1.3.1 Heterogeneity in Datacenters

Large-scale datacenters feature diverse compute, software, storage, and networking resources each with unique capabilities and performance characteristics. Such heterogeneity in datacenters is inevitable and arises from ongoing technological advancements and hardware upgrades [130, 170, 244], leading to infrastructure-level performance variations. This impacts the performance [116] of complex workflows and large-scale distributed applications, which can be avoided by making them aware of the performance variations of the underlying datacenter resources. Applications often lack awareness of server performance heterogeneity, leading to task scheduling on sub-optimal servers that leads to unpredictable performance and execution failures.

1.3.2 Platform Limitations to Support Resource Heterogeneity

The TensorFlow platform supports conventional multi-core processors and computational accelerators but lacks awareness of underlying infrastructure type and resource capabilities during training job execution. As a result, training jobs are often scheduled on worker nodes already running other tasks, leading to increased overall training time. Similarly, training jobs assigned to straggler

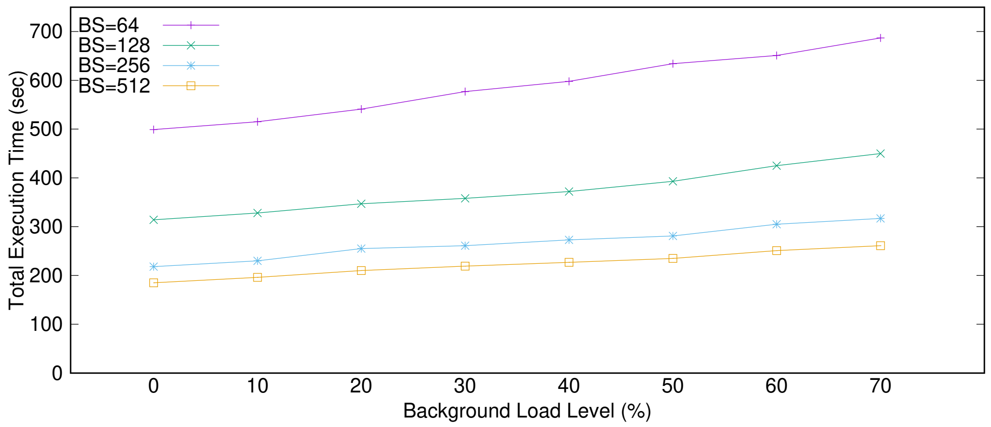


Figure 1.1: Effects of the background load on total execution time using different training batch sizes.

worker nodes further exacerbate training time [51, 81, 102, 242], as parameters of the DL model must be gathered and updated from each node after every iteration. This can cause missed deadlines for time-sensitive training processes [34, 191]. Hard-coding worker nodes and devices for model training using default TensorFlow is not scalable in large-scale datacenters shared by multiple users. Additionally, this approach results in poor resource utilization, as accurately executing training workloads on idle resources for distributed training becomes impractical for developers.

We conducted several experiments to study the impact of existing system load on the execution duration of DL jobs across various batch sizes. Figure 1.1 illustrates the outcomes of this experiment, using the ResNet322 [105] model with the CIFAR10 [139] dataset. Our experiments entailed running jobs that utilized 10% CPU, 20% GPU, 4 GB of main memory, and 2.6 GB of GPU device memory, thereby introducing background load on the worker nodes. Multiple instances of these jobs were executed to achieve varying levels of background load, with the average load reported as a combination of CPU, GPU, memory, and network resource utilization. For batch sizes 64, 128, 256, and 512, we observed an increase in training time by 37%, 43%, 45%, and 41%, respectively, as the background load increased from 0% to 70%. We observe similar performance trends with other ML models during our evaluation. This shows that it is critical to execute the training jobs on resources with minimum interference from other tasks.

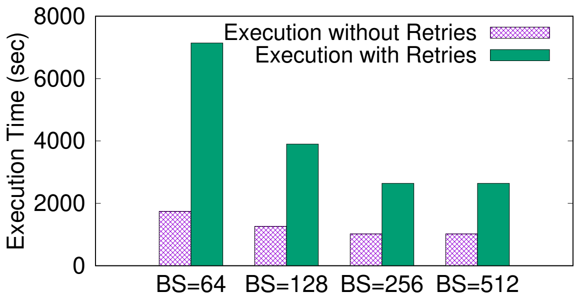


Figure 1.2: Impact of retries on execution time using MobileNet on CIFAR100; Epochs=15.

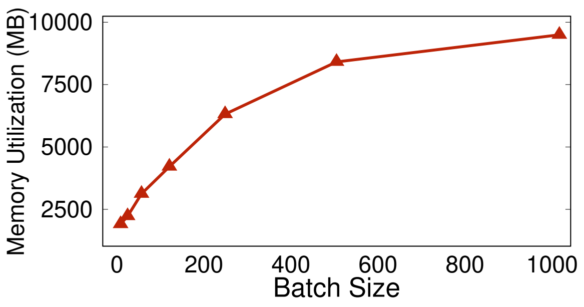


Figure 1.3: Impact of the batch size on memory using MobileNet on CIFAR10; Epochs=5.

1.3.3 Limitations of FaaS Platforms

FaaS environments [125, 199] are routinely used for running DL workloads since they reduce the provisioning and management overhead and provide an easy-to-use, scalable, flexible, and cost-effective alternative to the traditional server-centric compute model. They feature short-lived execution environments, effectively managing resource limits on function execution to control the cost and resource consumption of DL jobs. Distributed DL on serverless platforms demonstrates superior performance for training DL models compared to IaaS platforms within the same cost constraints [33, 158, 229]. Moreover, serverless computing is well-suited for periodic model training, such as continuous learning for incremental learning systems like recommendation and anomaly detection systems, where prediction models are periodically updated after acquiring new data.

Conventional serverless computing platforms are designed for short-lived tasks and impose restrictions on resource usage. For instance, Apache OpenWhisk, AWS Lambda, Azure Function, and Google Cloud Function have default timeouts of 300 sec., 900 sec., 600 sec., and 540 sec., respectively. DL tasks may fail if their training duration surpasses these default timeouts. Hence, ensuring successful execution and training of high-quality models necessitates specifying appropriate memory and timeout limits for each action.

Accurately estimating the training time, memory requirements, and epochs needed to achieve the desired accuracy in DL jobs is challenging before job submission. Allocating lower resource limits leads to job failures which must be restarted using a retry strategy with increased memory and timeout limits. Each retry consumes additional time and resources until suitable limits are identified. We implemented a simple retry strategy that increments the timeout by 240 sec. for failed actions to train MobileNet [111] with the CIFAR100 [139] dataset using batch sizes of 64, 128, 256, and 512 over fifteen epochs. Figure 1.2 shows the results. The total execution time includes failed retry attempts time as failed actions exceed the allocated time limits. The retry strategy uses 7 and 5 retry attempts for a batch size of 64 and 128 respectively, and uses 4 retry attempts for a batch size of 256 and 512 before successfully completing the job. We observe an additional latency of 5,400 sec. and 2,640 sec. for batch sizes of 64 and 128 respectively, and a latency of 1,620 sec. for batch sizes of 256 and 512. Therefore, retrying with different resource limits results in longer training times and a loss of compute cycles.

We investigate the effect of serverless computing on DL job performance by adjusting the batch size. Specifically, we train the MobileNet model for five epochs on the CIFAR10 [139] dataset to evaluate the memory consumption and show in Figure 1.3 that there is a gradual increase in memory utilization for larger batch sizes because more memory is allocated to the job to accommodate larger input data. This underscores the importance of employing data parallelism for DL jobs to ensure efficient memory management by dividing the dataset into smaller batches. Moreover, training with large batch sizes leads to memory contentions, further highlighting the advantages of serverless computing, as training can be parallelized with smaller batches, ensuring each batch fits within the allocated resources for action execution.

1.3.4 Fault Tolerance in FaaS Platforms

FaaS Execution and Failure Types: Functions in a FaaS platform are modular code units designed to execute specific tasks in response to events. When creating a function, developers provide the code, runtime environment, memory allocation, trigger, and a unique identifier. Triggers initiate function execution and the provided code runs within the specified runtime. Functions can process input data through single or multiple phases known as states, reflecting the current status of function variables and data structures. These states are referred to as the current state of function variables and data structures. They generate intermediate and final data stored in storage media for consumption by subsequent functions. After processing, functions can invoke other functions to work on the produced data, forming a workflow as depicted in Figure 1.4.

Failures in the FaaS platform can occur at various stages of function execution, typically falling into four categories: request, concurrency, function, and runtime. Request failures arise when resource requests for a function exceed the limits associated with the account. Concurrency failures occur when the number of simultaneous executions requested surpasses the maximum allowed. Function failures stem from issues within the application code, while runtime failures relate to the setup and preparation of the runtime environment. Both function and runtime failures are critical as they can result in data loss, computation errors, and financial losses for stateful applications. To ensure

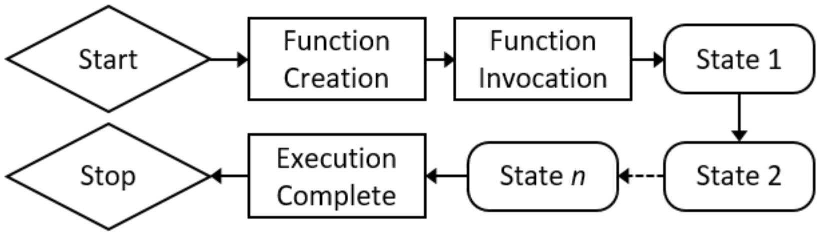


Figure 1.4: Execution flow of a function in FaaS.

fault tolerance in FaaS, proactive measures must be taken to address these failures or recover from them promptly to meet application-level SLAs.

Reliability and Fault Tolerance in FaaS Platforms: Failures occur at various levels such as hardware [93, 224, 228], platform [193, 239], software stack [154, 243], and application [120, 219]. While FaaS platforms offer best-effort reliability guarantees, failures can lead to dropped requests or repeated executions, necessitating stronger reliability assurances from applications. Implementing stronger semantics, such as *exactly once* guarantees in FaaS platforms comes with significant latency and resource overhead. Different types of failures, e.g., server, network links, and software process can result in a loss of data and inconsistent stream processing across the data center. Therefore, providing fault tolerance and reliability to stateful applications in FaaS platforms are critical yet largely unexplored area. The adoption of serverless computing continues to rise with more than 200% [10] increase in the average weekly invocations over the past years. Meanwhile, the number of failures in function execution have increased from 1% for highly maintained runtimes to about 25% for deprecated runtimes [10]. Therefore, there is a need to mitigate FaaS failures and reduce the recovery time of failed functions to improve application reliability and response time. FaaS deployed on HPC infrastructure is directly impacted by HPC failures [82, 86]. The state-of-the-art fault tolerance techniques, e.g., checkpointing [132, 184] and replication [115, 214], to mitigate HPC failures cannot be directly applied to FaaS platforms [129, 209] because of their unique characteristics, e.g., the massive scale and short lifespan of invoked functions.

Stateless and Stateful Functions: Stateless functions execute independently without any reference to previous executions, while stateful functions retain information from previous executions and the current execution may be affected by the status of previous executions. Execution failure causes the function to lose its context and execution progress and is unable to return or resume from the previous state. However, modern applications, e.g., iterative applications, are stateful and depend on application data and results from previous executions. The challenge of maintaining states exacerbates for FaaS platforms, which use containers designed to be stateless, portable, and

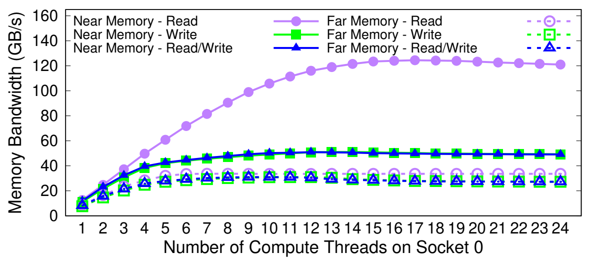


Figure 1.5: Impact of increasing Socket 0 thread count on near and far memory bandwidth.

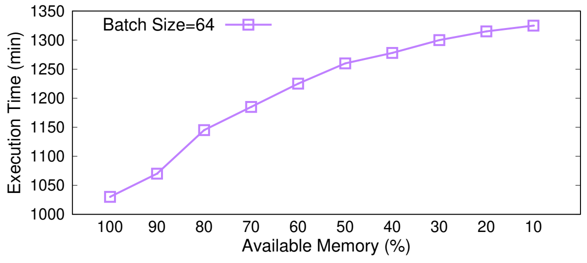


Figure 1.6: Impact of available memory on DL performance with 64 batch size and 3 epochs.

flexible. Because of its popularity and ease of use, existing applications are being migrated [56, 57] to FaaS platforms, and new stateful applications are developed using FaaS platforms. For stateful execution, an approach is to make FaaS functions stateful by default. However, this violates the basic design concepts of short-running and lightweight functions because persisting data would significantly increase function execution times. Moreover, migration of stateful applications to FaaS is inevitable. FaaS platforms must adapt to support both stateless and stateful applications. Due to the ephemeral nature of data in FaaS computing, the impact of a failure would be significant as all progress of the running function will be lost. Typically, stateful applications rely on fault tolerance approaches, e.g., replication to external storage, for maintaining their states reliably. No available end-to-end fault tolerance approach adapts to both stateless and stateful FaaS applications. Simple retry-based approaches do not address the challenges for stateful function execution that would experience computation loss and inconsistent application data upon failure.

1.3.5 Data Movement Optimizations

Slow Storage Tiers and HPC Workloads: Each memory and storage tier in modern data centers has a distinct access bandwidth and latency profiles resulting in unpredictable application performance. Distributed HPC workloads are executed over several servers with varying memory and storage capacity and performance profiles and require fast memory, low-latency I/O pipelines, and a large storage medium to store huge datasets. The performance of HPC workloads heavily depends on data transfer speed and how quickly the required data is made available to the processing threads. Typically, the processing threads process the data at a much higher rate than staging the data into the system memory. We observe that as the memory allocation moves farther away, e.g., to other NUMA nodes, from the compute threads, the performance starts to drop due to the impact of latency associated with accessing memory and storage resources over the respective interconnections. The result is shown in Figure 1.5. We ran STREAM Triad [174] and observed maximum bandwidth when the data is accessed on the same node as the compute threads. Efforts have been made to predict both optimal core allocation and memory bandwidth usage with high accuracy and low overhead for memory-intensive multi-threaded applications on large-scale clusters [231]. However, such optimizations do not directly apply to distributed HPC jobs over heterogeneous memory and storage resources.

The memory footprint of most HPC workloads such as DL increases over time [169,173], which leads to excessive swapping for servers with limited memory. The memory access patterns of a workload determine the impact of using swap space on its performance. Most modern DL applications are read-intensive and perform write operations at regular intervals and performance drops as the memory footprint spills over to the swap storage resulting in increased epoch time and an increase in the overall training time as shown in Figure 1.6. The memory footprint of the DL job is about 166 GB and fits entirely into the memory when 100% of memory is available to the DL workload. The configured system swap space is 200 GB and the increased execution time is attributed to increased reads and writes to swap. To mitigate the impact of using swap space, it is critical to explore the use of high-capacity and low-latency alternatives, such as CXL-based memory.

I/O Challenges in DL Data Pipelines: Executing a DL workload requires processing large datasets to achieve the desired training accuracy of the given DL model. The growing size of datasets emphasizes the importance of designing highly efficient I/O pipelines, especially in distributed DL environments where the dataset is distributed across multiple workers for processing. DL workloads contain various I/O stages, e.g., data loading, caching, prefetching, model fitting, and checkpointing. DL platforms provide methods and APIs to perform parallel I/O operations and improve the performance of data staging and placement. For example, TensorFlow provides *tf.data* [178] whereas PyTorch provides *DataLoader* [185] APIs to improve the performance of data staging. However, these built-in methods and APIs for data staging do not incorporate different memory tiers for storing large datasets.

During model training, the collection of data samples called batches is shuffled randomly [143] before each epoch for model convergence and to prevent overfitting. Typically, the input data is cached into memory during the first epoch to speed up the read operations in subsequent epochs. However, the cache hit rate is severely impacted when the entire dataset cannot fit into the available memory. The cached data is evicted after being processed during an epoch to load new batches into the main memory, causing thrashing and forcing the workers to fetch data from slower storage devices. Prefetching reduces I/O stalls by bringing data to a lower storage tier and moving the next

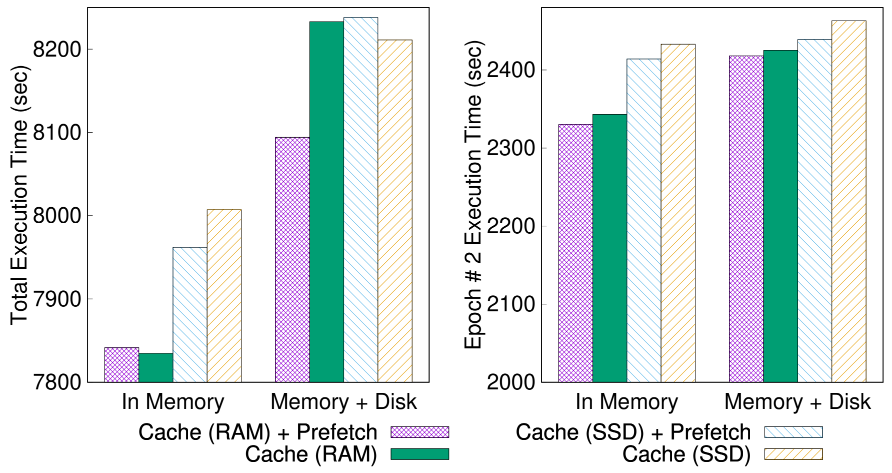


Figure 1.7: Impact of caching and prefetching on DL workload performance with MobileNetv2 and a subset of ImageNet with batch size of 64 and 3 epochs.

batch into the main memory before the next iteration. This becomes challenging for large datasets as prefetching the next batch takes longer than processing the current batch [80]. Therefore, a data staging strategy is required to incorporate the heterogeneity of underlying memory and storage tiers to orchestrate the data pipeline and reduce I/O stalls.

Limitations of Data Pre-processing in DL Platforms: The size and location of the dataset govern the creation of the data pipeline, pre-processing, and data loading into the main memory. Typically, large datasets cannot fit into the memory subsystem of a single worker due to limited memory and storage on each worker. A DL job fails to execute when available system memory is not enough to hold the entire training dataset. To avoid such failures due to limited resources, data is placed into pipelines and prefetching and caching techniques are used to efficiently manage memory and storage resources. TensorFlow caches the dataset in memory for improving I/O, however, its caching mechanism becomes ineffective if the memory is not large enough to host the entire dataset and it does not yield any performance benefit because the dataset cannot be cached in memory. TensorFlow also allows caching to disk, which becomes beneficial when caching to local NVMe devices as compared to reading the batch from network-attached storage. Another important factor that improves performance is prefetch, which ensures that the dataset is loaded in the main memory before the training job has finished processing the previous batch of data.

We analyze the impact of caching and prefetching techniques for two scenarios. First, the training data is greater than the available system memory resulting in excessive swapping, and second, the training data is smaller than the available system memory and it can fit entirely into the memory without swap utilization. We used TensorFlow's optimized data pipelines with limited memory and observed performance degradation due to frequent disk accesses. Figure 1.7 shows the results. We observe that prefetching and caching into memory yield better performance as compared to using the swap storage. Moreover, the impact of data prefetching reduces when the available system memory is less than the dataset size. With limited memory available, disk I/O becomes a bottleneck as compute units consume data at a much higher rate. We observe this during data loading, pre-processing, and training by analyzing the DL job's footprint, available system memory, and disk utilization with the help of PCM and SYSSTAT monitoring tools. Prefetching consumes additional memory to store data batches and speeds up the training process. Therefore, less available memory significantly increases application execution times and reduces throughput. This impact is amplified with large batch sizes despite TensorFlow's caching and prefetching mechanisms. For a worker where less amount of memory is available, caching to the local storage is useful since it reduces access latency as compared to the network storage to fetch the same data for subsequent epochs. Moreover, caching to the local storage is beneficial if faster storage, such as NVMe [126] and CXL devices are available at the worker nodes.

1.3.6 Tiered Memory Management

HPC Workflows and Workflow Management Systems: HPC workloads are composed of a series of tasks, organized as workflows, that work in tandem to run larger scientific applications such as (1) scientific simulations, which run in embarrassingly parallel or tightly coupled fashion; (2) surrogate computations, which typically generate a deep-learning-based approximation to assist the scientific simulation for faster convergence; (3) real-time data analysis, which includes on-the-fly data manipulation and visualization based on which experiments and/or algorithms are steered; (4) producer-consumer workflow patterns, where workflows consume data generated by other workflows; and (5) checkpointing for fault-tolerance, posthoc analysis, supporting out-of-core adjoint computations, or explaining the evolution of data and scientific model. Several Workflow Management Systems (WMS), e.g., Pegasus [69], Cromwell [225], and Nextflow [75], facilitate the orchestration and automation of such complex computational workflows. WMSs interact with sophisticated schedulers to efficiently allocate computing resources, optimize task dependencies, and balance workflows. However, they face challenges in managing diverse workflows with varying resource demands, adapting to dynamic system conditions, and ensuring optimal resource utilization amidst changing priorities and constraints [44, 87]. Additionally, optimizing memory allocation in tiered memory systems, efficient data movement between memory tiers, optimal data placement, catering for data locality, memory requirements, and inter-task communication further complicates the scheduling process and is not supported in modern WMSs [69, 201, 225].

Memory Characteristics for HPC Jobs: HPC jobs pose diverse requirements to memory subsystems, such as combinations of large memory tiers, low latency, and high bandwidth. These requirements can change dynamically during job execution. Moreover, HPC jobs are often composed of several workflows with diverse memory requirements [114, 187, 222] causing memory starvation, contention, and degraded performance. The basic allocation unit for HPC jobs is a compute node that leads to reduced resource utilization and fragmentation. The available memory is limited by the job-level allocations and the total physical memory installed on each server. To improve the performance of HPC jobs, in-memory computation is becoming increasingly popular [203] leading to higher memory demands in HPC clusters.

In containerized execution, memory is allocated at the start based on the memory requirement of the job and does not support dynamic memory allocation based on different execution phases of HPC workflows. Typically, HPC jobs are deployed as separate workflows [164], each catering to a diverse range of resource profiles, e.g., compute and memory-intensive tasks, I/O, bandwidth-intensive operations, and capacity- and latency-sensitive operations. Given the varying demands of different resource profiles, accurately identifying the memory requirements associated with each workflow is challenging. Similarly, colocated containerized HPC workflows and ensembles have additional resource limitations, e.g., CPU, memory, storage, I/O, and network, which are specified by the workflow and negatively impact its performance. These restrictions limit the performance of highly parallel memory and data-intensive workflows where most tasks require a large amount of memory to store the input, intermediate, and output data of various tasks of HPC workflow. Similarly, it is challenging to accurately estimate the memory requirements of workflow tasks and allocate enough memory, resulting in a loss of critical computation during failures [42].

Tiered Memory Systems: Tiered memory systems utilize the latest advancements in memory subsystems to provide large memory to servers and workflows. It allows workflows to scale by utilizing additional memory available beyond the total available DRAM on each server. In tiered memory systems, the DRAM tier is utilized for high-speed, low-latency access to frequently accessed data, whereas PMem [121] bridges the gap between volatile and non-volatile memory to provide a balance between speed and persistence. Recently CXL [155, 226] has been explored to provide high-speed, low-latency I/O between the host processor and devices while expanding memory capacity and bandwidth [30, 227]. CXL memory also enables direct access to additional memory resources and optimizes data movement across the system by providing byte-addressable, cache-coherent memory in the same physical address space and allowing transparent memory allocation using

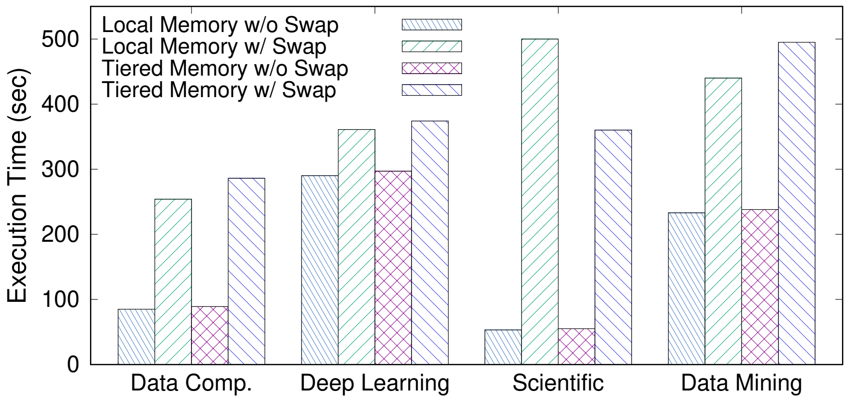


Figure 1.8: Impact of tiered memory on workflows with SSD-based swap.

standard memory allocation APIs. Even with colocated memory-intensive tasks, HPC jobs rarely use the entire allocated memory and often leave a large amount of unused memory during their life cycles. For instance, our evaluations (Section 8.2) demonstrate that in the case of BERT [74] model training, during the initial 120 seconds of application execution, ~55%-80% of the allocated memory remains idle, thereby becoming *cold memory* pages. Moving these cold memory pages to a slower memory tier can allow hot memory pages to reside in fast memory tiers and improve application-level performance. Furthermore, fast memory tiers reduce the reliance on slow swap storage. Optimizing access to different memory tiers based on data access patterns ensures that frequently used data remains in high-speed memory, minimizing the need for costly swaps to slower persistent storage.

Figure 1.8 shows the impact of allocating tiered memory to different containerized workflows. The performance of all workflows significantly drops when onboard system memory is limited and memory pages are swapped to disk-based swap storage. Allocating memory from different tiers improves the performance of each workflow regardless of the workload type and memory access pattern, however, bandwidth-intensive tasks benefit more due to additional bandwidth available over the CXL interface. Moreover, the performance is further improved when the memory pages are actively swapped out to CXL-based swap space instead of disk-based swap storage.

With the popularity of containerized HPC workflows, there is a need to rethink the management of tiered memory to support granular memory allocation for workflow tasks, intelligent data placement techniques for latency-sensitive tasks, and enable fast data sharing between local and remote tasks from the same or different workflows to increase the resource utilization and reduce the execution time of HPC jobs. To the best of our knowledge, we are the first to explore tiered memory for containerized HPC jobs and propose specialized memory allocation and management policies to meet workflow tasks' latency, bandwidth, and capacity requirements. Similarly, HPC jobs scheduled to execute on GPUs face memory contention, lower throughput, and degraded performance due to suboptimal scheduling and tiered memory management.

1.4 Objectives and Approach

1.4.1 Infrastructure-Aware Distributed Systems

To address datacenter heterogeneity, we propose architectural improvements and new software modules in the default TensorFlow platform to make it aware of the availability and capabilities of the underlying datacenter resources. This will enable TensorFlow to make efficient resource management decisions, and will result in reduced training time and improve datacenter utilization by scheduling jobs on idle resources. The proposed Infrastructure-Aware TensorFlow efficiently schedules the training tasks on the best possible resources for execution, isolates and limit the impact of busy and straggler worker nodes in large datacenters on the performance of distributed training. This will significantly improve the performance of the training process by reduces the overall training time. The proposed design alleviates application developers from managing the underlying datacenter resources and enables full utilization of the available resources.

1.4.2 HPC Application Support for FaaS Platforms

We address the fixed memory and timeout constraints by developing an effective runtime framework to determine the appropriate memory and timeout limits for executing serverless functions that improve the performance of DL jobs by leveraging data splitting techniques, and ensuring that an appropriate amount of memory is allocated to containers for storing application data and a suitable timeout is selected for each job based on its complexity in serverless deployments. We implement our approach using Apache OpenWhisk and TensorFlow platforms and evaluate it using representative DL workloads to show that it eliminates DL job failures and reduces action memory consumption and total training time. The proposed design eliminates function execution failures in FaaS to improve the utilization of datacenter resources and improves the performance of DL workloads by reducing their function execution time on serverless platforms.

1.4.3 Fault Tolerant FaaS Execution

We propose a highly resilient and fault-tolerant framework for FaaS that mitigates the impact of failures and reduces the overhead of function restart. This enables FaaS platforms to tolerate faults in function execution and invocation guaranteeing exactly once execution and reducing the total recovery time for failed functions. The proposed design utilizes replicated container runtimes and application-level checkpoints to reduce application recovery time over FaaS platforms by enabling faster recovery of serverless applications from faults by using intelligent and dynamic checkpointing and replication techniques. The replication of container runtimes ensures faster function execution after a failure by restoring the saved state and data in the replicated runtimes.

1.4.4 Leverage Latest Advancements in Memory Technology

We use the latest advancements in the memory subsystem, specifically Compute Express Link (CXL), to provide additional memory and fast scratch space for DL workloads to reduce the overall training time while enabling DL jobs to efficiently train models using data that is much larger than the installed system memory. We propose a framework, that manages the allocation of additional CXL-based memory, introduces a fast intermediate storage tier, and provides intelligent prefetching and caching mechanisms for DL workloads. We implement and integrate the proposed framework with TensorFlow, to show that our approach reduces read and write latencies, improves the overall I/O throughput, and reduces the training time.

1.4.5 Leverage Tiered Memory

We leverage tiered memory that includes various memory types categorized into distinct memory tiers to propose application-attuned intelligent memory management policies and incorporate the access latency associated with memory tiers to optimize the performance of workflows while incorporating the performance characteristics, i.e., sensitivity to latency, bandwidth, and capacity, of each workflow task. Our policies leverage workflow memory access patterns and system memory utilization to evict data from memory tiers. Our approach improves tiered memory utilization and application performance and reduces the cold-start time for large-scale deployments. Similarly, for GPU-based HPC workloads leveraging tiered memory, we propose an algorithm to mitigate the contention on the CXL memory, maximize throughput, and reduce the overall data transfer time. The algorithm addresses the performance bottlenecks of default memory allocation on CXL-enabled systems when running multiple jobs on a single multi-GPU system. Our schedule-aware memory allocation approach incorporates memory requirements on each socket of a multi-GPU system and provides an efficient memory placement map to mitigate memory contention.

1.5 Publication List

This dissertation is based in part on the following publications:

① **Moiz Arif**, M. Mustafa Rafique, Seung-Hwan Lim, and Zaki Malik, “Infrastructure-Aware TensorFlow for Heterogeneous Datacenters,” in Proc. IEEE MASCOTS, 2020.

② **Moiz Arif**, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai, “Exploiting CXL-based Memory for Distributed Deep Learning,” in Proc. ACM ICPP, 2022.

③ **Moiz. Arif**, Kevin Assogba, and M. Mustafa Rafique, “Canary: Fault-tolerant FaaS for Stateful Time-sensitive Applications,” in Proc. IEEE SC, 2022.

④ Kevin Assogba, **Moiz Arif**, M. Mustafa Rafique, and Dimitrios S. Nikolopoulos, “On Realizing Efficient Deep Learning Using Serverless Computing,” in Proc. IEEE CCGrid, 2022.

⑤ **Moiz Arif**, A. Maurya, and M. Mustafa Rafique, ”Accelerating Performance of GPU-based Workloads Using CXL,” in Proc. ACM FlexScience, 2023.

⑥ **Moiz Arif**, A. Maurya, M. Mustafa Rafique, Dimitrios S. Nikolopoulos, Ali R. Butt, ”Application-Attuned Memory Management for Containerized HPC Workflows,” in Proc. IEEE IPDPS, 2024.

Chapter 2

Literature Review

The focus of this research is to improve the performance of HPC workloads by introducing heterogeneity-aware scheduling and resource management to HPC platforms on baremetal and FaaS platforms, leverage advancements in hardware technologies, use tired memory, and introduce fault tolerance to FaaS platforms. This section summarizes the prior work that is closely related to these areas.

2.1 DL Platform Optimizations

Many recent efforts have focused on improving the performance of the TensorFlow [27] platform. Horovod [204] extends the capabilities of using TensorFlow in a distributed setting by improving inter-GPU communication via the ring reduction method. Similarly, MARBLE [101] proposes an approach to select an optimal number of GPUs per node for an ML workload. While these approaches significantly improve TensorFlow, they rely on the node placement approach of TensorFlow that does not avoid executing ML jobs on busy or straggler nodes. Another effort [124] addresses the challenge of straggler and heterogeneity for distributed training, but it heavily relies on the parameter server approach and is not applicable in other distributed training methods.

A recent effort [223] proposes using the message passing interface (MPI) primitives for multi-node TensorFlow deployments to improve portability without requiring application-level changes. These efforts address the performance of the TensorFlow platform from a communication perspective but do not address challenges related to scheduling ML jobs in heterogeneous datacenter environments with worker nodes having diverse performance profiles.

Tensor-tracing [104] proposes collecting application-level run-time metrics and exchanging them between ML job executions to improve the execution performance of the TensorFlow applications. However, this approach incurs additional run-time overhead. Another approach [176] extends the TensorFlow platform and introduces a hierarchical model for efficient graph placement by incorporating the heterogeneity in worker nodes. Although it incorporates various compute resources, i.e., CPU and GPU, it does not incorporate other performance metrics, e.g., GPU capabilities, memory, and network latency, while making the placement decisions.

Previous efforts have explored different aspects to improve the performance of the TensorFlow platform, however, they do not holistically address the challenges of resource and performance heterogeneity.

2.2 FaaS Optimizations

Serverless computing provides an efficient, reliable, flexible, and scalable infrastructure to a variety of HPC applications. FaaS enables applications development by using granular functions, offering benefits similar to modern workflow management systems [221]. This is aligned with recent efforts to design distributed systems for DL by leveraging data parallelism, model parallelism, or hybrid strategies [92] to optimize resource utilization and enable resource sharing in HPC clusters.

Open-source FaaS: The widespread adoption of open-source software has become a driving force for cloud computing [96], and many of these systems benefit from serverless computing which provides simplified deployment and management for a variety of applications. Open-source FaaS platforms, e.g., Apache OpenWhisk, Kubeless [16], Fn Project [9], SAND [32], and $f$uncX [60] offer flexible options for private deployments. However, these platforms do not support dynamic memory allocation or timeout adaptation.

FaaS for DL Applications: Existing efforts to use serverless computing for DL applications mainly target lightweight computations, specifically on edge devices [182] and inference engines [21]. Lin and Glikson [159] deploy a cat/dog image classification model on Knative [21] for inference using TensorFlow, while Ishakian et al. [119] use AWS Lambda to serve large DL models using TensorFlow. Rausch et al. [195] explore the use of an Edge AI workflow on serverless platforms and propose a serverless model using edge devices as cluster resources for edge-cloud platforms. Palade et al. [182] explore the hypothesis that incorporating serverless computing into IoT devices for small tasks reduces processing time.

There is a growing interest to deploy serverless functions for tensor-parallel operations and for end-to-end model training to achieve higher parallelism [125, 218]. Feng et al. [147] argue that serverless is ideal for training small models, and minimizing data transfer between subsequent actions improves the performance of the platform. Cirrus [55] expands the design of serverless architecture to support ML systems. It integrates a stateless server-side back-end and addresses challenges of resource constraints and workers' scalability. It addresses memory resource limits by streaming batches of training data from storage, however, jobs are exposed to failures when training large models. Our proposed memory estimation strategy addresses this by proactive allocations of memory before starting model training. Furthermore, Cirrus cannot run TensorFlow workloads in serverless environments due to resource constraints, whereas, we fully supports DL training using TensorFlow over serverless platforms. SIREN [229] proposes a distributed ML framework over AWS Lambda. It deploys cloud actions at each epoch to process training jobs and the scheduler selects the number of actions and handles memory allocation. This approach is similar to our approach, however, we assigned actions per mini-batch instead of epoch to eliminate system overload, and minimize additional data access latency. Moreover, controlling the number of actions eliminates CPU over/under utilization.

Existing efforts have explored different aspects to ensure the execution of DL workloads with serverless resources. These efforts propose problem-specific or job-specific designs and do not focus on improving the resource utilization of serverless platforms.

2.3 Performance Optimizations in Tiered Memory Systems

Many recent efforts have focused on improving the I/O performance of the TensorFlow platform. In this section, we provide an overview of efforts that are closely related to our contributions.

Several other efforts have been made to optimize the I/O path in DL, such as, [65] which studies the impact of the BeeGFS filesystem on DL workload performance. NoPFS [80] predicts data access patterns and performs prefetching and caching based on these patterns. It provides a distributed caching policy using local and distributed memory to improve the I/O performance of DL jobs. However, NoPFS does not support CXL-based memory or storage devices that introduce additional tiers in the memory and storage layer. In [62], the authors study the impact of multi-threading on the I/O pipelines on improving the performance of DL jobs. Recent efforts also explore optimizing data loading in the I/O pipelines and pre-processing to accelerate DL applications by utilizing Nvidia's Data Loading Library (DALI) [25]. Prisma [166] decouples storage I/O optimization using software-defined storage that is composed of a control plane that maintains user-defined caching policies and a data plane that implements parallel data prefetching. However, it does not leverage CXL-based memory and storage devices.

Informed Prefetching Data Loader (IPDL) [200] prefetches data from remote data stores to reduce the I/O wait times in PyTorch based DL and edge computing environments. Similarly, in [138] the authors employ caching and prefetching techniques to improve the performance of DL training in cloud environments. Quiver [143] is an informed storage cache designed to improve the performance of DL jobs on GPU-enabled clusters using secure hash-based addressing to reuse cached data across jobs and avoid cache thrashing. The approaches are tailored for GPUs and cannot be applied to all stages of DL jobs. Moreover, they do not explore the use of CXL-based memory and storage tiers. PreFAM [136] improves the performance of fabric-attached memory architectures by predicting future data access and prefetching data blocks from fabric-attached memories to node-local memory resulting in improved access latency. While this approach is similar to ours in leveraging the latest advancements in memory subsystem to provide additional memory and optimize data access, however, our approach reduces the uncertainty in predicting future data access and improves the accuracy of prefetching by integrating with the DL frameworks. We also propose a caching mechanism to maintain prefetched blocks that will be accessed in the future in the closest memory tier.

Distributed remote memory accesses can be performed by using fast low latency networks and protocols involving RDMA, NVMEoF [100] and SEMERU [230]. Remote memory paging system over RDMA called Infiniswap [99] that provides memory disaggregation. In [126], the authors use prefetching over NVRAM and DRAM to bridge the I/O gap between hard disk to RAM. RAMCloud [181] aggregates server memories into a single coherent key-value store and provides low-latency access to large-scale datasets enabling faster access to large datasets for various applications including DL workloads. Fanstore [247] provides a runtime file system to optimize DL I/O on existing hardware and software architecture by distributing datasets to all compute nodes, and maintains a global namespace. DLFS [249] provides I/O services on top of an emerging industrial standard NVMeOF leveraging storage disaggregation.

Previous efforts have explored different aspects of DL I/O to improve training and optimize input pipelines by introducing middleware, runtimes, file system abstractions, and utilizing caching and prefetching techniques. However, these efforts do not holistically incorporate multiple memory and storage tiers and do not leverage emerging technologies to optimize the data pre-processing and input pipelines in DL platforms. In this paper, we propose a holistic framework that improves the performance of DL workloads by incorporating and utilizing CXL-based memory and storage devices in the TensorFlow platform.

2.4 Stateful Serverless Fault Tolerance

Modern applications are composed of several closely connected components launched as functions. These components communicate and share states using an additional storage layer that requires fine-grained state management at a low cost [47]. A well-orchestrated state sharing technique is required to avoid issues with non-atomic updates, concurrency control, duplication, etc. [7]. Existing research has addressed the design of such data layer for stateful FaaS in three main directions, i.e., function composition, external storage, and low latency shared memory layer. State sharing between FaaS functions is achieved through function composition when two consecutive functions are executed such that the output of the first function is the input of the second function. This sharing technique is solely applicable when the output size remains within the quotas of the FaaS platform. When large data transfer is required between functions, FaaS applications rely on external storage such as AWS S3 [1], Google Cloud Storage [24], IBM Cloud Object Storage [12], etc. Despite their high data access latency, these solutions are used to facilitate the design of stateful applications and ensure data persistence in FaaS. In-memory KV stores such as Redis [18], MemcacheDB [175], etc. are used to provide low latency, high bandwidth, but non-persistent data storage. To facilitate function auto-scaling, FaaS systems, such as Cloudburst [213] maintain states in auto-scaling and fault tolerant KV stores as Anna [234]. Distributed shared memory layer provides a trade-off between latency and data size while improving state management [47, 49]. However, concerns regarding memory address space isolation [208] are not addressed in such approaches. Similarly, Faaslets [208] employs WebAssembly software-fault isolation tool to provide isolation while sharing memory regions between FaaS functions.

Failures are addressed in cloud computing using approaches such as replication [168], checkpoint [35], checksum [63], self-healing [73], retry [177], safety-bag checks [186], task re-submission [190], etc. Fault tolerance techniques are grouped into two categories, i.e., proactive and reactive. Proactive fault tolerance [160] involves preemption migration [85], self-healing [73], periodically reboot with a clean copy [43], or load balance when resource utilization threshold is reached [58]. Reactive fault tolerance [28] includes techniques, such as, retries [177], task re-submission to the same or a different node [190], reconfiguration [186], etc. Proactive and reactive fault tolerance techniques are often used together to improve system reliability [194]. We combines both techniques by pre-emptively saving checkpoints and maintaining runtime replicas during execution, and completing the remainder of the workload execution in a replica after failure.

The most targeted types of failures in serverless computing are related to resource limitations on FaaS platforms [94]. Existing research addresses hardware and network issues but hardware failures are mainly explored from the perspective of cloud providers [224]. Serverless platforms have built-in fault tolerance techniques such as check-pointing [246], retries [212], object replication [48], etc. However, function failure can still occur due to memory, timeout, network, concurrency, and user quotas. These failures are more detrimental to stateful applications due to the cost associated with loss of computation. Existing efforts provide fault tolerance and reliability to stateful applications by integrating object storage, KV stores or optimizing file systems [202] to facilitate function retries or re-submission. Moreover, log-based techniques that monitor execution logs are explored for fault tolerance and data consistency [123]. Monitoring logs facilitate the detection of function states and the coordination of chained FaaS applications. To further optimize state management for data consistency guarantees, transaction processing techniques are used to control read and write operations on intermediate data [245]. These techniques involve data staging and commit after transaction validation. Nevertheless, transaction processing techniques add an overhead to the system [68]. Similarly, as more FaaS applications depend on network-based services, node failures cause requests to be re-executed multiple times [142]. Two strategies, i.e., request replication and active standby are proposed [54] to improve fault tolerance. Request replication involves having multiple replicas to execute the same request and returning results to the client once any of the replicas successfully returns. Active standby refers to maintaining one passive instance whenever there is an active function. The passive function is activated when the function fails and triggers the creation of a new passive instance. These approaches proved better than function retries, but can yield high expenses as more requests are submitted. Request replication results in multiple unused function instances, and one passive instance becomes a bottleneck with multiple consecutive function failures.

Our dynamic replication and checkpointing approach adjusts the replication factor and checkpointing frequency to provide improved fault tolerance and reduced cost as compared to these approaches. These shortcomings in existing approaches necessitate further exploration efforts to improve fault tolerance and reliability in FaaS.

2.5 Tiered Memory Systems

Tiered memory systems in HPC address the increasing memory capacity, bandwidth, and latency requirements of HPC workflows. These systems leverage different memory technologies, e.g., DRAM, PMem, and CXL-based memory, where each memory type offers distinct performance characteristics [89, 149]. DRAM provides high-speed and low-latency access while PMem offers non-volatile memory and bridges the gap between DRAM and storage, enabling data persistence even during power loss [113, 137, 145]. CXL memory provides fast, high-capacity, and low-latency access to applications enhancing scalability and resource pooling in tiered memory systems for improved HPC performance [97, 153, 155, 237]. Tiered memory systems also improve overall memory utilization by intelligently allocating data to the most appropriate tier based on access patterns and performance requirements [133, 161, 206]. However, neither the applications nor the platforms are optimized to leverage the true potential of tiered memory systems resulting in degraded application performance and system utilization.

Tiered memory management approaches have been extensively explored by several studies such as Nimble [235], TPP [171], HeMem [196], Pond [155], AutoTM [110] etc. These approaches perform application-agnostic memory allocations and page movement across various memory tiers. However, these techniques result in degraded performance for colocated HPC workflows with diverse memory requirements. Moreover, they perform strictly hierarchical page movement and do not perform concurrent tiered memory allocation to optimize bandwidth through parallel interconnects. Other efforts [127, 134, 144, 197, 205] either solve the challenge of memory management for terabyte-scale applications (e.g., HM-Keeper [197]) or partially optimize and automate memory management across multiple memory tiers. Similarly, MTM [198] performs application-transparent page management based on profiling, multi-tiered page migration policy, and huge page awareness. Our approach extends on the general design ideas of the above state-of-the-art tiered memory approaches, and incorporates applications' memory characteristics for efficient memory management. Lastly, none of these approaches are optimized for GPU-based workloads and do not cater for pinned memory allocated on CXL-enabled multi-GPU setups, and the bandwidth bottleneck of CXL memory connected over PCIe lanes.

Chapter 3

Infrastructure-Aware TensorFlow for Heterogeneous Datacenters

3.1 System Design

To make the TensorFlow platform aware of the underlying datacenter resource heterogeneity we extend the platform that enables it to catalog and monitor datacenter resources. Our approach [40] enables the TensorFlow platform to utilize resource metrics from worker nodes such as CPU, GPU, memory, and network utilization to make informed scheduling decisions for DL jobs. Figure 3.1 shows the proposed architecture of the Infrastructure-Aware TensorFlow platform. The software modules developed as a part of this research are described as follows:

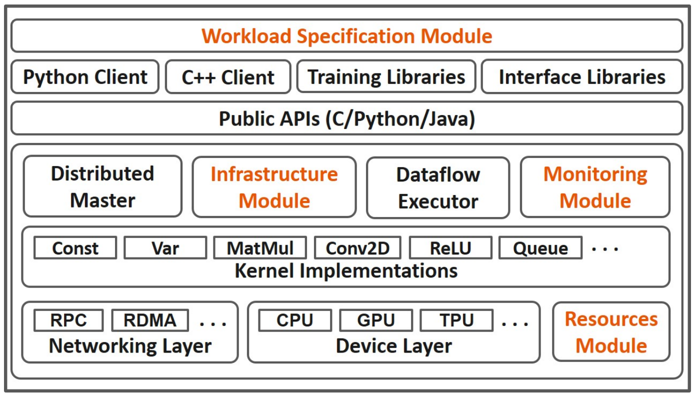


Figure 3.1: Infrastructure-Aware TensorFlow Architecture

Workload Specification Module: This module allows application developers to provide resource requirements for the DL job that ensure guaranteed execution by extending the TensorFlow configuration specification and adding a section called `resources`. Developers can specify the number of required CPU/GPU, memory, and network bandwidth. These resource requirements are processed and used by the *Infrastructure Module* for identifying appropriate worker nodes to run the given DL job.

Resources Module: This module captures the compute, accelerator, memory, and network resource information of workers and stores them in the database under distinct tables. The information from each worker table is used to build a list of available worker nodes for distributed execution. This list contains the hostname, IP address, and optional port numbers that are required for monitoring worker resources. It uses the information to populate each worker table with installed hardware resources, i.e., CPU/GPU count, capabilities, system memory, and network speed. The *Resources Module* automatically detects changes in the installed hardware on each worker and updates the record accordingly by utilizing the monitoring capabilities provided by the *Monitoring Module* which continuously monitors resources on each worker.

Infrastructure Module: This module is the core component of the Infrastructure-Aware TensorFlow tasked with making informed scheduling decisions for DL job execution. The default TensorFlow does not leverage the latest resource utilization metrics of the worker nodes and schedules DL jobs on sub-optimal worker nodes with busy CPU and GPU resources, limited available memory, and network bottlenecks. The *Infrastructure Module* enables TensorFlow to incorporate the latest resource utilization metrics to make an informed decision while selecting a suitable worker to schedule the given DL job. The *Infrastructure Module* categorizes worker nodes as a straggler if the historic and current resource utilization exceeds the predefined threshold and the information is stored back in the database and later utilized during the worker selection process.

Monitoring Module: We have developed a lightweight Python utility that periodically collects the CPU, GPU, memory, and network utilization metrics, and stores them in the configured database. The initial monitoring frequency is set to 10 seconds, however, the frequency is automatically adjusted to reduce the monitoring overhead on worker nodes. Similarly, historical monitoring data for the workers is stored in the database, which is used to identify straggler worker nodes by the *Infrastructure Module*. This information is also correlated with the current resource utilization metrics reported by the *Monitoring Module* and the job completion statistics stored in the database.

This data is also used to determine if a given worker node can experience high resource utilization or resource contention during job execution.

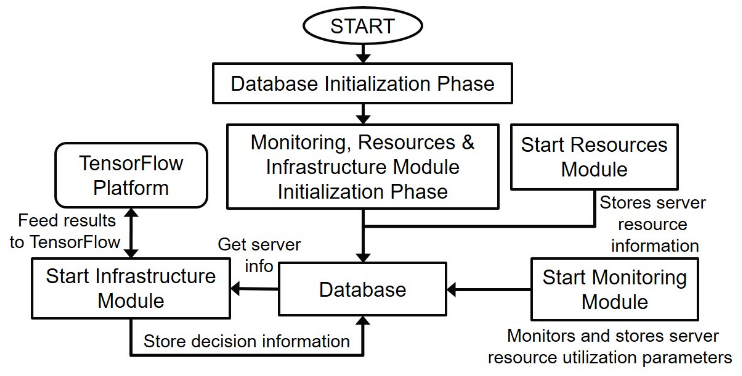

START
Database Initialization Phase
Monitoring, Resources & Infrastructure Module Initialization Phase
Start Resources Module
Stores server resource information
TensorFlow Platform
Feed results to TensorFlow
Get server info
Start Infrastructure Module
Database
Start Monitoring Module
Store decision information
Monitors and stores server resource utilization parameters

Figure 3.2: Internal workflow of modules included in our proposed design.

The internal workflow of Infrastrucutre-Aware TensorFlow is shown in Figure 3.2. Once all modules are initialized and synchronized with the database, the *Resources Module* starts capturing information of all worker nodes followed by the *Monitoring Module* capturing the current resource utilization as instructed by the *Resources Module*. The *Workload Specification Module* accepts new job submissions and the *Infrastructure Module* fetches the up-to-date information about the underlying infrastructure from the database and selects appropriate workers to execute the given workload. This decision, along with the job execution information, is stored back in the database for future use.

Algorithm 1 outlines the procedure employed by the *Infrastructure Module* to select suitable worker nodes for executing a given DL job. Initially, it retrieves the list of worker nodes from the database along with their installed resources to identify straggler nodes that are excluded from the list of available workers along with the worker nodes lacking sufficient resources to execute the DL job. Then the current resource utilization of each worker node is accessed, excluding those surpassing an adaptive threshold derived from historical job profiling data for each job. The remaining nodes undergo a weighing and scoring process based on resource utilization metrics, utilizing predefined numeric multipliers as weights. These scores are aggregated for each node, with higher scores indicating lower resource utilization. The node with the highest aggregated score, signifying the most suitable node, is selected to execute the DL job, aiming for balanced resource utilization across all nodes. Finally, the final list of worker nodes is sorted based on the aggregated scores, and the required number of nodes are selected based on the resource specifications of the DL job.

Algorithm 1: Workers Selection in the Infrastructure-Aware TensorFlow.

```
Input: worker nodes ($N$), job resource requirements ($R$)
Output: List of ideal worker nodes.
begin
    Get list of straggler nodes ($stragglers$)
    Get $cpu_{req}$, $mem_{req}$, $net_{bw}$, $gpu_{req}$
    Get $cpu_{wt}$, $mem_{wt}$, $net_{wt}$, $gpu_{wt}$
    Get $cpu_{th}$, $mem_{th}$, $net_{th}$, $gpu_{th}$, $gpumem_{th}$
    for all workers $n_j \in N$ do
        if $n_j \in stragglers$ then
            Skip this node and continue to the next node
        else
            Retain worker node
        Get $cpu_{tot}$, $mem_{tot}$, $net_{bw}$, $gpu_{tot}$, $gpu_{memtot}$
        if $cpu_{tot} \geq cpu_{req}$ AND $mem_{tot} \geq mem_{req}$ AND $net_{bw} \geq net_{req}$ AND $gpu_{tot} \geq gpu_{req}$ then
            Retain worker node
        else
            Skip this node and continue to the next node
        Get $cpu_{util}$, $mem_{util}$, $net_{util}$, $gpu_{util}$, $gpu_{memutil}$
        if $cpu_{util} \geq cpu_{th}$ OR $mem_{util} \geq mem_{th}$ OR $net_{util} \geq net_{th}$ OR $gpu_{util} \geq gpu_{th}$ OR
          $gpu_{memutil} \geq gpumem_{th}$ then
            Skip this node and continue to the next node
        else
            Retain worker node
        $score[n_j] = cpu_{util} \times cpu_{wt} + mem_{util} \times mem_{wt} + net_{util} \times net_{wt} + gpu_{util} \times gpu_{wt}$
    Sort $score$ for all workers in descending order
    Select and return top $w$ workers from sorted list, where $w$ represents the number of workers specified in $R$
```

3.2 Performance Evaluation

3.2.1 Testbed Setup

Our evaluation setup consists of eight Dell PowerEdge R730 servers having two 2.30 GHz Intel Xeon E5-2670 v3 processors, 128 GB main memory, two NVIDIA P100 GPUs, and a 10G network interconnect between the servers. We run a distributed TensorFlow environment on these servers using Ubuntu 18.04 LTS server operating system. We use multiple TensorFlow jobs to create background noise on the TensorFlow worker nodes. We use MNIST [151], ImageNet [71] and CIFAR10/100 [139] datasets and Keras [64], ResNet32/56 [105], Inception-V1 [215] and MobileNet [112] models to evaluate the performance of distributed training using different execution environments. For evaluating our proposed Infrastructure-Aware TensorFlow platform on varying load scenarios, we overload the servers by increasing the CPU, GPU, memory, and network utilization. To this end, we use `netem` [107] along with `ethtool` [8] to simulate different network constraints on the links between the master and worker nodes. Moreover, we use `stress` [19], which is a Linux utility to overload CPU and memory on the worker nodes to mimic the behavior of stragglers and busy nodes in the datacenter.

3.2.2 Execution Environments

We define three TensorFlow execution environments in our evaluation where each environment showcases various capabilities of the TensorFlow platform to address changing resource availability in datacenter settings. These environments are:

1. **Unconstrained TensorFlow Environment:** This environment is based on an ideal scenario where a single job has dedicated access to the worker node with no resource constraints. We use the default TensorFlow platform to run the given DL job in this environment.
2. **Constrained TensorFlow Environment:** This environment is subjected to resource constraints, such as increased CPU, GPU, memory, and network bandwidth utilization of the heterogeneous datacenter resources. We use the default TensorFlow platform to run the given DL job in this environment with background jobs that utilize 10% CPU, 20% GPU, 4 GB of main memory, and 2.6 GB of GPU memory to introduce constraints on the worker nodes. We run multiple instances of these background jobs to mimic different load scenarios.
3. **Infrastructure-Aware TensorFlow Environment:** This is the same as the constrained TensorFlow environment, however, we use a hand-tuned implementation of the proposed Infrastructure-Aware TensorFlow platform to run the given DL job.

3.3 Performance Results

3.3.1 Training Makespan using Multiple Nodes:

We ran several experiments using a combination of DL models and datasets with varying numbers of training iterations, batch sizes, and epochs and reported the total execution time for the studied execution environments. Figure 3.3 shows the performance comparison of the studied environments using a batch size of 64 to train the studied models. We note that training takes the longest in the constrained environment for all models, primarily due to resource limitations on worker nodes. Additionally, the default TensorFlow platform lacks awareness of resource constraints, often utilizing nodes with limited resources for job execution. The Infrastructure-Aware TensorFlow utilizes resource availability and constraints to avoid scheduling jobs on sub-optimal worker nodes. We observe similar performance trends when using the training batch size of 128 and 256, as shown in Figure 3.4 and Figure 3.5, for the models used in our evaluation. Figure 3.6 shows the performance comparison using the batch size of 512. Here, we observe that the Inception-V1 model fails to execute using the constrained TensorFlow environment because of the resource constraints. On average, the execution time of the Infrastructure-Aware TensorFlow for the studied models is 37% more than the unconstrained (dedicated) environment. However, on average, the execution time for the Infrastructure-Aware TensorFlow is 24% less than the constrained TensorFlow environments for the studied models. This occurs because the straggler node delays the training process, leading to prolonged training times in the constrained TensorFlow environment. As other worker nodes wait for the straggler node to aggregate model parameters, they experience execution stalls, ultimately diminishing the overall performance of DL jobs.

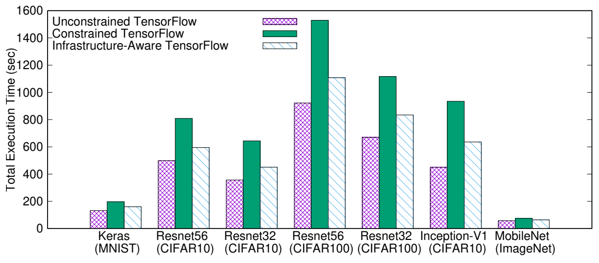


Figure 3.3: Training performance of studied execution environments with 64 batch size.

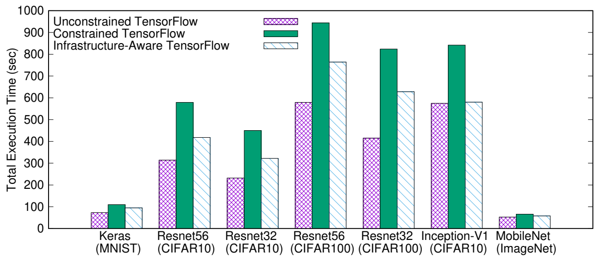


Figure 3.4: Training performance of studied execution environments with 128 batch size.

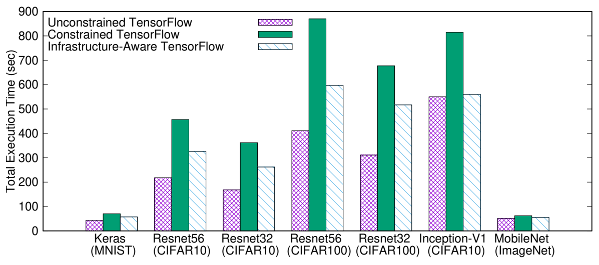


Figure 3.5: Training performance of studied execution environments with 256 batch size.

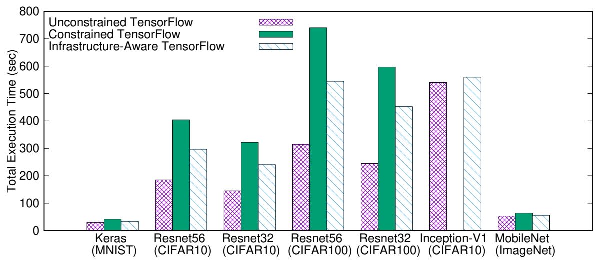


Figure 3.6: Training performance of studied execution environments with 512 batch size.

Overall, Infrastructure-Aware TensorFlow reduces the overall execution time by up to 54% as compared to the default TensorFlow platform for scenarios when a limited amount of compute, graphics processing unit (GPU), memory, and network resources are available at the worker nodes.

3.3.2 Impact of Available Network Bandwidth:

To study the impact of network bandwidth on the training performance for the studied execution environments, we limit the available bandwidth on the network link from 1 Gbps to 100 Mbps between workers. Figure 3.7 shows the result of this experiment using ResNet32 with the CIFAR10 dataset. We observe that the overall training time increases significantly as the available network bandwidth decreases. The default TensorFlow platform does not incorporate busy network links in scheduling the training jobs to the worker nodes increasing the overall model training time as the network link gets fully congested. However, the *Infrastructure Module* of our proposed platform addresses this constraint by incorporating the available bandwidth metric reported by the *Monitoring Module* assigning appropriate weights during the scoring phase. This enables our TensorFlow platform to exclude a possibly powerful worker from executing the DL job as it will increase the overall training time due to network congestion between the worker and the master nodes. On average, the proposed TensorFlow platform results in a 42% reduction in the overall training time when the available network bandwidth is between 100 Mbps and 700 Mbps.

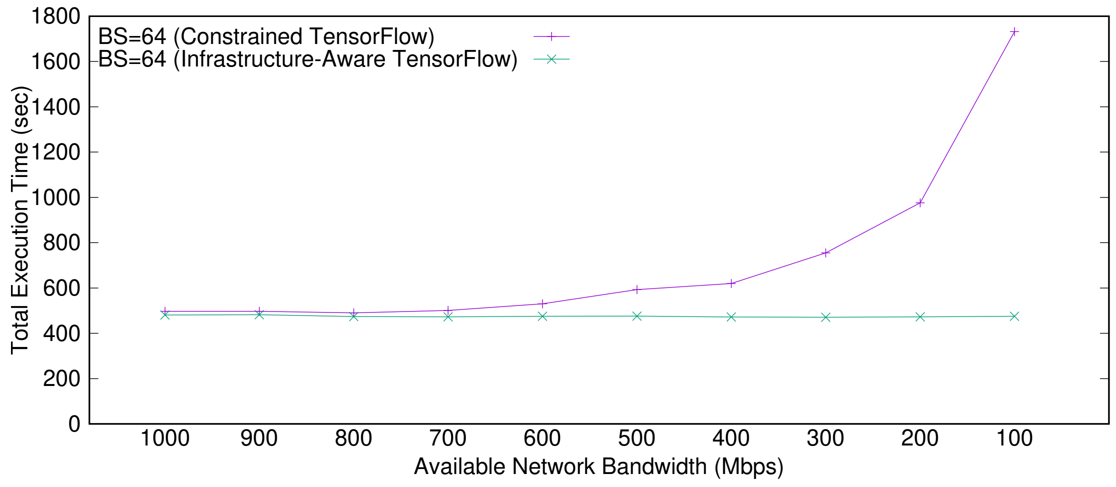


Figure 3.7: Impact of decreasing network bandwidth on execution time.

3.3.3 Impact of Increased Network Latency:

We study the impact of network latency on the performance of the studied DL models using unconstrained, constrained, and Infrastructure-Aware TensorFlow execution environments. Workers that are physically located far away from the master node add additional latency, which leads to an increase in the overall execution time. Figure 3.8 shows the result of this experiment where additional network latency is added to one of the worker nodes using the `netem` tool to mimic the behavior of a straggler node for a ResNet32 model using the CIFAR10 dataset. An induced network latency of 0 millisecond for the constrained TensorFlow represents the unconstrained TensorFlow environment. As the network latency increases, the overall training time increases proportionally showing that the distributed training is highly sensitive to variations in the network latency between the worker and the master nodes. Infrastructure-Aware TensorFlow handles heterogeneous network latencies between the worker nodes and avoids scheduling jobs on workers with increased network latencies. The *Infrastructure Module* accounts for the latency between the nodes and excludes straggler nodes from the list of available workers. We also observe that increased latency causes additional time to fetch the updated model parameters from the worker nodes. However, in the unconstrained TensorFlow environment, the worker nodes are selected regardless of their current network latencies, which leads to increased overall model training time. Overall, the proposed Infrastructure-Aware TensorFlow performs 52%, 54%, 41%, and 36% better than the constrained TensorFlow environment for model training when using the batch size of 64, 128, 256, and 512, respectively, with increased network latency between the worker and the master node.

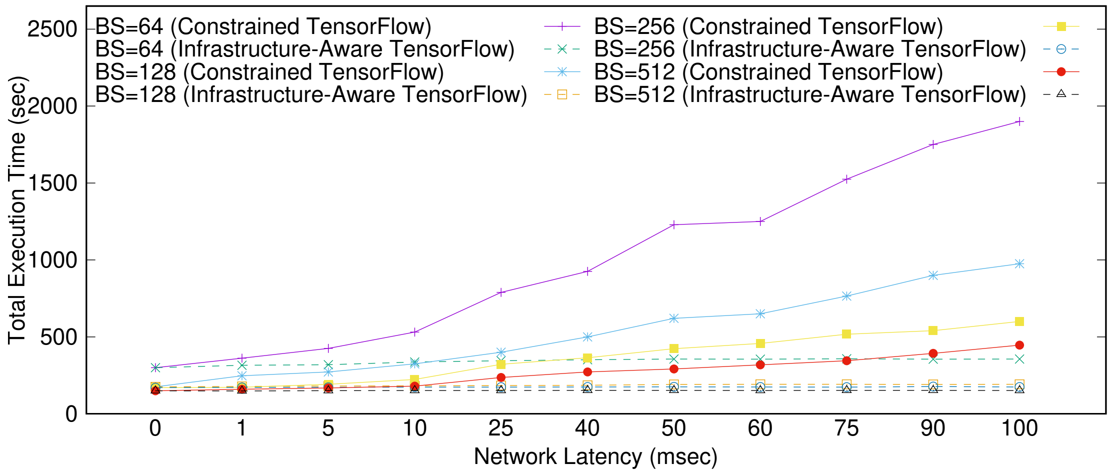


Figure 3.8: Impact of increasing network latency on training time.

3.3.4 Impact of GPU Utilization:

To study the impact of available GPU resources on training, we run resource-intensive TensorFlow jobs in the background to overload the GPU resources at the worker nodes simulating a shared environment. We use `nvidia-smi` [17] to monitor the GPU core and memory utilization to achieve specific levels of resource saturation. Figure 3.9 shows the result as we compare the constrained and Infrastructure-Aware TensorFlow platforms with ResNet32 model with the CIFAR10 dataset. A GPU utilization of 0% represents the unconstrained TensorFlow environment since no jobs are running in the background. For the constrained TensorFlow environment, the total execution time increases as the GPU is overloaded until enough GPU memory is available to successfully run the given DL job. We also observe that as the amount of available GPU resources is reduced, the training process slows down increasing the overall execution time. However, the *Infrastructure Module* of the Infrastructure-Aware TensorFlow accounts for the GPU resource availability by ensuring that the training job is not scheduled on the worker nodes with overloaded GPU resources. Overall, the Infrastructure-Aware TensorFlow performs 5%, 7%, 9%, and 8% better than the constrained TensorFlow environment on average for model training using the batch size of 64, 128, 256, and 512, respectively, when the GPU resources are shared between multiple DL workloads.

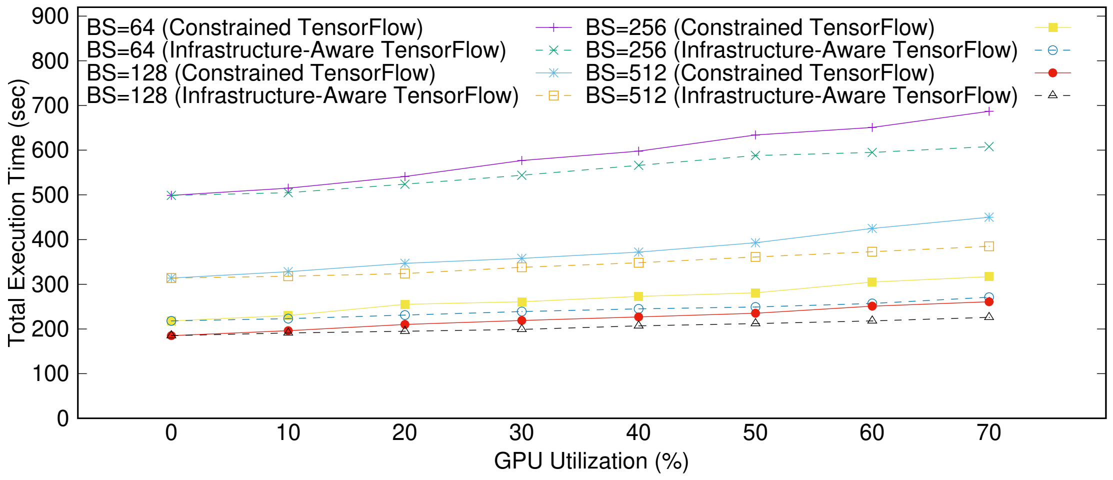


Figure 3.9: Impact of background GPU utilization on training time.

3.4 Summary

To summarize, we introduced an Infrastructure-Aware TensorFlow platform, that enhances the TensorFlow framework by integrating new software modules that enable developers to specify resource requirements for DL jobs, capture heterogeneous resources on worker nodes, consider real-time monitoring data, and schedule DL jobs accordingly. Our evaluation demonstrates that Infrastructure-Aware TensorFlow reduces overall execution time by up to 54% over the default TensorFlow execution environment in scenarios with limited compute, GPU, memory, and network resources.

Chapter 4

On Realizing Efficient Deep Learning Using Serverless Computing

4.1 System Design

To address the limitation of fixed memory allocation and static timeout limit of each function execution in serverless platforms, we propose *Distributed Serverless Deep Learning* (*DiSDeL*) [42], which is an efficient runtime framework for running long-running DL jobs on serverless platforms. *DiSDeL* ensures that the appropriate memory and timeout limits are allocated to each function. *DiSDeL* improves performance by leveraging data parallelism to assign jobs to concurrent actions, uses an in-memory data store to maintain the intermediate states, and aggregates intermediate outputs to generate the final parameters of the trained model.

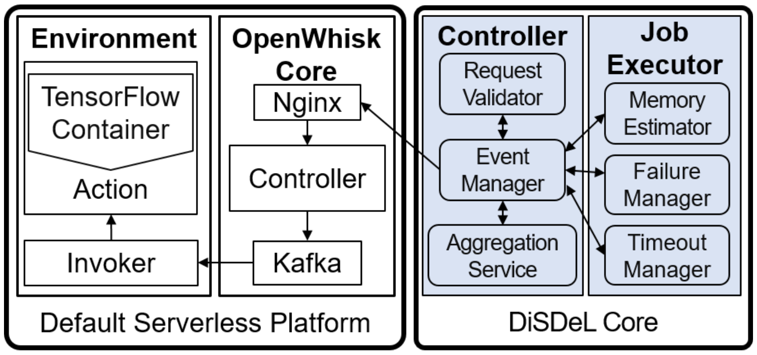


Figure 4.1: High-level architecture of *DiSDeL*.

A high-level architecture of *DiSDeL* is shown in Figure 4.1. The system interacts with two main external components, i.e., the core and the container runtime of the serverless platform i.e. Apache OpenWhisk. *DiSDeL* includes self-contained modules exposing the core functionality e.g., $configure$, $schedule$, and $process\_requests$, as APIs for future work. *DiSDeL* includes two core components, i.e., a controller and a job executor, which drive the execution flow as shown in Figure 4.2. The submitted request contains the model name, training dataset, batch size, and the number of epochs. The controller validates the request, fetches the dataset attributes, creates a package to wrap the entire composition, and starts the training process.

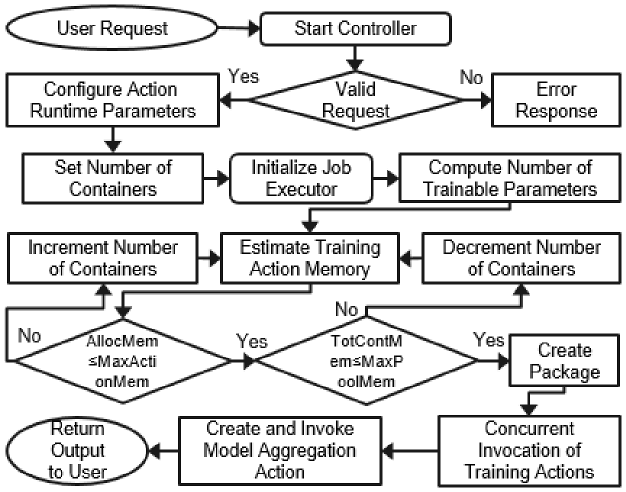

User Request
Start Controller
Valid Request
Yes
No
Configure Action Runtime Parameters
Error Response
Set Number of Containers
Initialize Job Executor
Compute Number of Trainable Parameters
Increment Number of Containers
Estimate Training Action Memory
Decrement Number of Containers
No
AllocMem ≤MaxActionMem
Yes
No
TotContMem≤MaxPoolMem
Yes
Create Package
Return Output to User
Create and Invoke Model Aggregation Action
Concurrent Invocation of Training Actions

Figure 4.2: Execution flow of our framework for distributed DL using serverless.

Controller: It is the main component of *DiSDeL* and contains three main sub-components, i.e., request validator, event manager, and aggregation service. All operations, e.g., request validation, application parameters configuration, and collection of execution results are coordinated by the controller. The controller splits the target dataset into smaller chunks to fit into individual actions and estimates memory for each action from the job executor based on the required number of actions. Similarly, the input dataset is divided into equal parts for balanced workload distribution. The number of deployed containers $c_d > 1$ is initially set to its minimum possible value 2, which is dynamically adjusted based on the estimated action memory. Whenever the estimated memory $m_e > M$, where M represents the configured action memory limit, the number of containers is increased by $\lceil (m_e - M)/M \rceil$, and is resubmitted for memory estimation. Once a valid allocation scheme is determined, the controller orchestrates fork-join operations and returns the execution results to the user.

Algorithm 2: Execution workflow of Event Manager.

```
Input: model, data, batch size (bch), #. of epoch (epc).
Output: Status of the execution (Succeeded OR Failed OR Error).
1  begin
2      parse DL job request request
3      if is_valid(request) then
4          Retrieve model, data, bch, and epc from request
5      else
6          Return Error
7      Get job execution history from file execution.log
8      if job not in execution.log then
9          Estimate memory $m_a$ and timeout $e_{time}$
10         Determine aggregation scheme agg
11         Get #. of training $n_{tr}$ and #. of aggregation $n_{agg}$ actions
12     else
13         Load $m_a$, $e_{time}$, and agg from execution.log
14     Invoke $n_{tr}$ training actions concurrently
15     for all cluster $n_{cst} \in n_{agg}$ do
16         for all action $n_{idx} \in n_{cst}$ do
17             join $n_{idx}$
18         Invoke aggregation action of cluster $n_{cst}$
19     Return status Succeeded OR Failed
```

Request Validator validates incoming requests, validity, completeness, and verification of request parameters, and whether the arguments contain required information such as model name, training dataset, batch size, and the number of epochs. Once validation is complete the request is forwarded to the Event Manager.

Event Manager coordinates all events between the controller and the job executor as illustrated in Algorithm 2. It interacts with other components to request, assign, and collect responses to various tasks, e.g., memory estimation, timeout assignment, number of containers to launch, job submission to OpenWhisk, and coordinates responses back to the user.

Aggregation Service determines the number of aggregation actions to launch to meet the memory requirements of training actions. Starting with level-1 aggregation, this service determines if one level-1 container is sufficient to handle the workload from all level-0 containers. If the memory requirement to aggregate all level-0 containers exceeds the current allocation to level-1 containers, then two or more level-1 containers are launched along with one additional level-2 container for all level-1 containers. This process continues until the entire aggregation process is completed.

The training containers are divided into logical clusters based on the total number of deployed containers with each cluster assigned to an aggregation action launched as soon as the containers in that cluster complete the training process. Each aggregation action updates its copy of the model before the controller launches the highest-level aggregation container to complete the aggregation service and store the final trained model in the data storage.

Job Executor is tasked with estimating the memory requirement of each action by considering model and dataset attributes, determining appropriate timeout, and analyzing various failure scenarios. It handles failures associated with insufficient memory allocation to run the job. The Job Executor contains a Memory Estimator, Timeout Manager, and Failure Manager to perform these tasks. The memory estimation depends on the configuration of the serverless platform, i.e., maximum action memory, and container pool memory. Therefore, in addition to parameters e.g., the batch size, and input data dimensions which are used to determine the total number of activations and parameters generated during DL jobs, our memory estimator correlates the action memory and the container pool memory limits to ensure that the estimated memory does not exceed the system memory limit. Similarly, the memory required by a DL job also depends on the model, dataset, runtime environment, and execution logs stored in the container.

Timeout Manager analyzes the job and utilizes the historical job execution information to assign an appropriate timeout limit and stores the execution time of submitted jobs to build execution history. This ensures that the job is executed once without failure due to insufficient timeout. If historical execution data is not available the Timeout Manager considers the expected computation cost e_{cost} provided by the user. e_{cost} is used along with the estimated memory to determine an expected maximum execution time $e_{time} = (e_{cost}/c_d)/(\mu \times m_e)$. e_{time} is used for training and aggregation actions. This does not guarantee successful execution since the timeout is only based on the user's expected computation cost. If this computation cost is not provided, the Timeout Manager applies the maximum action timeout of the serverless framework. DL frameworks, e.g., TensorFlow, provide a mechanism to track the execution time of each epoch, which can be recorded by profiling one or two initial epochs and used to estimate the timeout for a particular job. However, the profiled execution time varies depending on many factors, such as the load on each server [40] and leads to inaccurate estimations. Moreover, large training jobs require a significant amount of time for profiling. Our approach avoids this overhead by estimating timeout before an action's execution.

Failure Manager collects errors during execution to detect anomalies due to the predicted memory and timeout allocations. It analyzes job execution logs and the physical resource utilization to identify the cause of a failure, i.e., container out-of-memory (OOM) and out-of-time (OOT) errors. Based on the root cause the memory and timeout allocations are adjusted and stored for future executions of the same model.

Following the initial memory estimation, the Event Manager triggers DL job training actions on the provided dataset. To ensure timely completion and to meet the accuracy targets, a bi-objective optimization approach minimizes the loss function cost is used. Each action incurs a cost denoted by μ, representing memory consumption multiplied by execution duration. With c_d containers deployed, total memory consumption sums the memory used by each action. The execution duration is determined by the time difference between the latest and earliest container's finish and start times. Both memory consumption and execution time are multiplied by the unit cost μ to obtain the total job cost. Each action is independently run and the performance of ongoing jobs is evaluated at the end of each epoch to determine if the required loss is achieved. A custom callback function is developed to define an early stopping criterion for each action. The training actions check if the loss value l_a has reached a certain threshold ϵ at the end of each epoch. The designed optimization problem minimizes the total cost f_{cost} and the mean loss obtained by averaging the loss l_a of each action. Next, we apply the ϵ-constraint method [172] incorporating the average loss objective function as a constraint.

4.2 Performance Evaluation

4.2.1 Testbed Setup

Our testbed consists of a cluster of 8 bare-metal servers from the Chameleon testbed [131] with each server having two Intel Xeon Gold 6126/6240R/6242 processors containing 192 GB of main memory and running the Ubuntu 18.04 LTS server operating system. We deploy OpenWhisk on a Kubernetes cluster along with Docker, OpenWhisk CLI (wsk), CouchDB [36], and Redis to store the model weights.

Models: We use popular DL models including InceptionV3 [216], ResNet50/152 [106] and VGG-16 [210]. InceptionV3, developed by Google, is a Convolutional Neural Network (CNN) renowned for object classification in computer vision. ResNet, an Artificial Neural Network (ANN), integrates identity shortcut connections via skip connections, allowing for faster training. Various ResNet variants exist, differing in layer count and weight training. VGG-16, another CNN architecture, boasts 16 layers and approximately 138 million parameters. Each model is compiled with categorical cross-entropy loss function [167] and Adam optimizer [135], serving as standard benchmarks for assessing TensorFlow platform optimizations and larger DL models.

Datasets: We used three popular datasets from the TensorFlow catalog: 1) MNIST [152] dataset of size 33.55 MB containing handwritten digits used for image classification jobs. 2) CIFAR10 [140] of size 308.28 MB containing images from ten categories commonly used to train machine learning and computer vision models. 3) DMLAB [6] of size 3221.22 MB contains 360×480 color images used to evaluate the distance between an agent and objects in a 3D environment.

4.2.2 Execution Environments

We use the following three execution environments to analyze the performance of *DiSDeL*:

- ***Default Serverless TensorFlow*:** This is the default TensorFlow running over the OpenWhisk platform. This scenario directly executes the user's request on OpenWhisk without any optimization or middleware to control the deployment of action containers. We assume that appropriate memory and timeout values are selected using multiple retries to successfully execute DL jobs in a single attempt using one container. We consider this environment as a baseline serverless environment because this environment avoids run-time failures due to inadequate memory and timeout allocation.
- ***DiSDeL*:** This implementation of OpenWhisk includes our proposed modules. It dynamically selects suitable memory and timeout allocation for each action.
- ***Bare-metal TensorFlow*:** In this environment, we run DL jobs on a dedicated bare-metal cluster where no limit is imposed on the amount of memory, and DL jobs are allowed to run till completion. This is the ideal scenario where the entire server is available to run the given DL job. Distributed Training on a single server uses mirrored strategy [183] on multiple local CPUs concurrently. Distributed training on a cluster uses a multi-worker mirrored strategy [5] that utilizes multiple distributed CPUs.

4.3 Performance Results

4.3.1 Impact on Memory Footprint and Execution Time

We train the studied DL models using popular datasets and report the memory footprint Figure 4.3 of all execution environments for the studied models and datasets. *DiSDeL* successfully executes the submitted jobs staying within 11.5%, and 8.9% of the total memory consumption of *Bare-metal TensorFlow* and *Default Serverless TensorFlow* approaches, respectively. *DiSDeL* consumes more memory due to the replication of DL models in each container and the batch splitting approach of *DiSDeL* allows each action to consume less than half of the total memory of *Default Serverless TensorFlow* and *Bare-metal TensorFlow*. For example, using *DiSDeL*, the ResNet50 model is successfully trained with CIFAR10 using two independent containers and consumes 44% and 39% less memory over the *Default Serverless TensorFlow* and *Bare-metal TensorFlow* approaches, respectively.

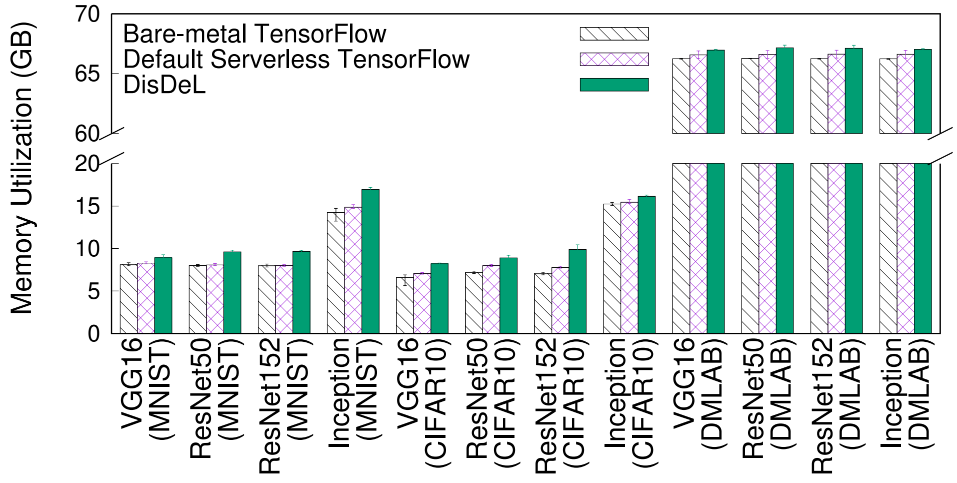


Figure 4.3: Memory footprint for the three execution environments; Batch size=64, Epochs=50.

Figure 4.4 shows the total execution time of DL jobs on all execution environments. *DiSDeL* completes the DL job in significantly less time than *Bare-metal TensorFlow* and *Default Serverless TensorFlow* because of the efficiency of concurrent DL job executions on a serverless platform. The memory estimation for each action in *DiSDeL* is fine-tuned based on the model and dataset type, enabling simultaneous execution of multiple actions. For example, *Bare-metal TensorFlow* and *Default Serverless TensorFlow* trained the InceptionV3 model over 60,000 data records of the MNIST dataset, but *DiSDeL* launched two concurrent containers for training on 30,000 data records

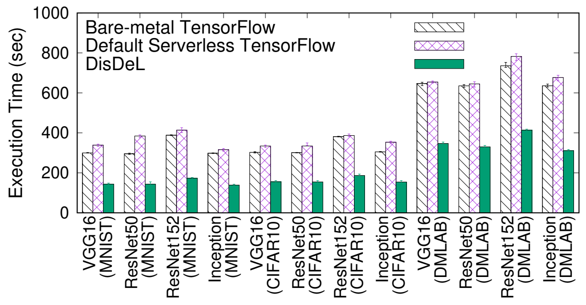


Figure 4.4: Total execution time for studied execution environments; Batch size=64, Epochs=50.

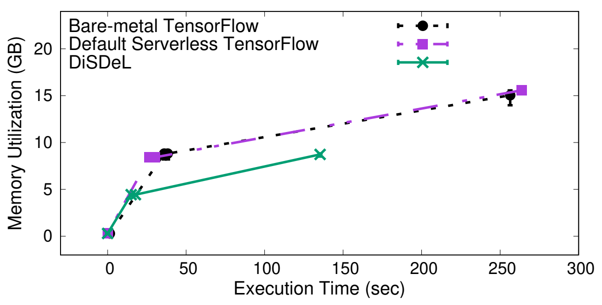


Figure 4.5: Training makespan of InceptionV3 on MNIST for the three execution environments.

each to reduce the overall training time. With a reduced data size for processing, containers in *DiSDeL* complete data pre-processing within 16 seconds while *Bare-metal TensorFlow* and *Default Serverless TensorFlow* take 30 and 20 seconds, respectively. Overall, both training actions complete their processing in 143.7 and 144.3 seconds. Despite the additional overhead caused by the aggregation containers, *DiSDeL* yields an average training time reduction of 46%, and 40% over *Default Serverless TensorFlow* and *Bare-metal TensorFlow*, respectively.

4.3.2 Impact on the Training Makespan

We evaluate the makespan of a DL training job on the three execution environments to examine their memory usage as training progresses. Figure 4.5 shows the amount of memory used at job submission, after data pre-processing, and after model fitting. We observe a memory consumption of approximately 280 MB across all environments at job submission due to the memory consumption of different software modules. While the data pre-processing phase completes within the same time and consumes the same memory for *Default Serverless TensorFlow* and *Bare-metal TensorFlow*, *DiSDeL* requires less time and memory due to high parallelism. For all environments, we observe a significant increase in memory consumption during the training phase for storing weights, biases, and other hyper-parameters required by the training process. *DiSDeL* requires less memory per action and leads to a reduced overall execution time by using concurrent actions.

4.3.3 Impact of Memory Utilization on Execution Time

Apache OpenWhisk imposes memory limits on the action and the container pool and assigns a default memory limit of 2 GB to container pools, which cannot be adjusted dynamically for each

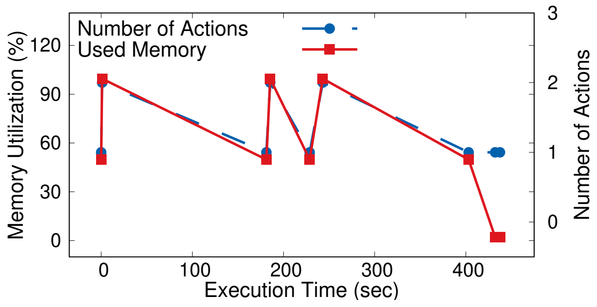


Figure 4.6: Impact of system load on training time of ResNet152 on DMLAB dataset.

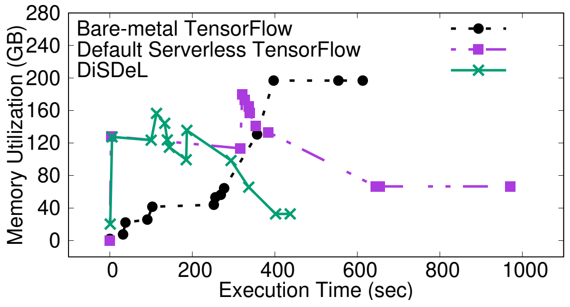


Figure 4.7: Memory footprint of a batch of eight jobs in all three execution environments.

job. Typically DL jobs require much more memory compared to the container pool limit. Figure 4.6 shows the result of running a DL training job with DMLAB using four training actions in *DiSDeL*. An action consumes 32 GB of memory and the container pool memory limit was set to 70 GB. Once the DL job is submitted an action container is launched, which consumes 49% of the container pool limit. At this point, another container is launched which consumes the entire pool memory, and further actions are queued until one of the previous actions is completed. Hence, lower pool limits result in queuing of action increasing the overall execution time. Increasing the pool limit to 140 GB enables more actions to run concurrently reducing the total execution time by 52.3%. *DiSDeL* efficiently allocates the appropriate number of containers and ensures that the underlying hardware resources are not exhausted while concurrently running multiple actions.

4.3.4 Impact of Batch Jobs on Execution Time

To evaluate the behavior of the three execution environments in a shared multi-tenant setup we submit a batch of eight DL jobs and show its impact on memory and total execution time. The action memory limit was set to 70 GB and the container pool memory limit to 190 GB to analyze the performance of *DiSDeL* when the available memory is the same as of *Bare-metal TensorFlow*. In Figure 4.7, sufficient memory resources allow *DiSDeL* to deploy several actions to execute DL jobs. A higher variation in memory footprint was observed with *DiSDeL* due to data parallelism that enables concurrent invocation of short-lived functions. Overall, *DiSDeL* achieves 55% and 29% faster execution of batch jobs compared to *Default Serverless TensorFlow* and *Bare-metal TensorFlow*, respectively. This shows higher scaling capabilities and confirms the effectiveness of *DiSDeL* in a multi-tenant environment as compared to *Bare-metal TensorFlow* and *Default Serverless TensorFlow*.

4.4 Summary

In summary, the evaluation *DiSDeL* highlights the advantages of dynamically adjusting resource allocations, particularly memory allocation and timeout limits, to enhance the performance of DL workloads in serverless settings. While *Default Serverless TensorFlow* runs actions concurrently, it doesn't consider the impact on system resources, resulting in over-utilized and unresponsive servers. Conversely, *DiSDeL* limits concurrent container launches, employs batch splitting to distribute less work per action, and reduces overall memory consumption, effectively optimizing the container pool. Moreover, *DiSDeL* reduces the training time by up to 40% as compared to *Bare-metal TensorFlow*. In a shared multi-tenant setting, *DiSDeL* reduces the training time by 55% and 29% on average as compared to *Default Serverless TensorFlow* and *Bare-metal TensorFlow*, respectively.

Chapter 5

Exploiting CXL-based Memory for Distributed Deep Learning

5.1 System Design

To improve the performance of DL workloads, we present a framework, *DeepMemoryDL* [39], that efficiently leverages storage and memory tiers to prefetch and cache training data proactively. We emulate CXL memory to provide additional memory and fast scratch storage space to workloads to reduce overall training time. *DeepMemoryDL* is integrated with the TensorFlow platform to improve the performance of its I/O requests for data pre-processing and training stages.

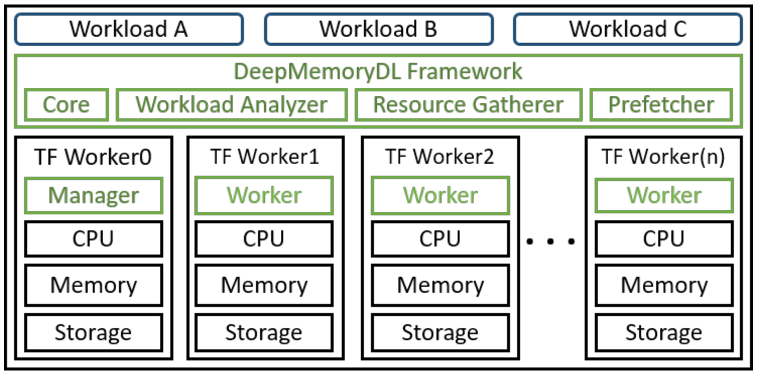


Figure 5.1: Proposed architecture of *DeepMemoryDL*.

Figure 5.1 shows the high-level architecture of the *DeepMemoryDL* framework. We develop a lightweight *Resource Gatherer Module* that collects the compute, memory, network, and storage resources of all servers included in the cluster. We implement a *Workload Analyzer Module* that analyzes the submitted DL workload and breaks down the job in I/O and compute phases. Moreover, we develop a *Prefetcher Module* that prefetches data and loads it in the main memory before it is required by the processing thread to minimize I/O stalls.

***Workload Analyzer Module*:** This module analyzes the submitted DL jobs to isolate the I/O operations from the computation phases. It is responsible for the following tasks:

- **Analyze DL Job:** The submitted DL jobs are analyzed to capture the DL model, parameters, dataset, epochs, batch size, data pre-processing stage, and model training steps. It also identifies if TensorFlow's native checkpoint or data caching option is enabled for the submitted job.
- **Separate Data Processing from Execution:** The submitted DL jobs are divided into data processing and model execution phases. In the data processing phase, the dataset is loaded and pre-processed, while in the model execution phase, the model is trained, validated, and evaluated.
- **Analyze Dataset and Batches:** The dataset shards assigned to each TensorFlow worker are tracked at the start of the training process and used to estimate memory and storage allocations on each worker to accurately determine the completion time for prefetching the required data in memory tiers.

***Core Module*:** It is the main component of *DeepMemoryDL* and divided into **manager** and **workers**. The manager resides on the same node as the master node in TensorFlow and supervises all operations of *DeepMemoryDL*. The *Core Module* computes the batching schedule for each worker and shares it with each worker along with metadata that specifies the location of each batch for the corresponding epochs. It instructs workers on when to launch the prefetching threads and the location of each batch based on the memory and storage tiers available at each worker for proactive prefetching to memory tiers. The workers reside at the worker nodes and perform tasks such as monitoring local buffers, prefetching, and caching the required data. The *Core Module* exposes an API that is used by the DL workloads to request memory space and once the request is received, *DeepMemoryDL* fetches the latest resource availability data, computes the ideal execution strategy, and services the request.

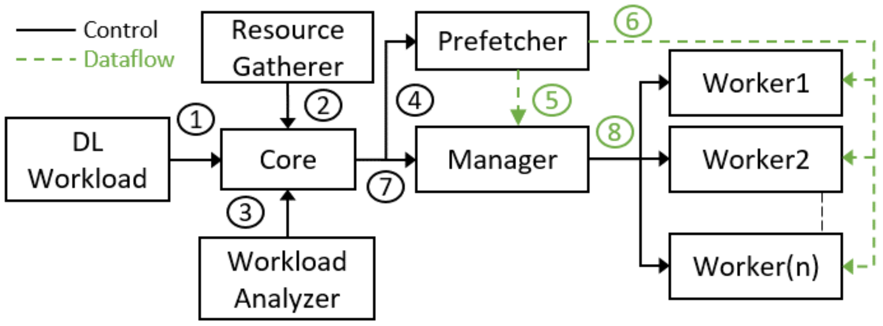


Figure 5.2: Control flow between *DeepMemoryDL* components.

Figure 5.2 shows the interactions between various components of *DeepMemoryDL*. Once the DL job is submitted to the *Core Module*, it collects information on the existing memory and storage tiers from the *Resource Gatherer Module*. Concurrently, the *Workload Analyzer Module* analyzes the submitted DL workload to identify data processing and training operations. The *Core Module* forwards all the information on the DL job to the prefetcher module for creating prefetching and caching schedules. The *Prefetcher Module* executes the schedule on the manager node and on all the worker nodes to ensure that the data is available on the fastest memory and storage tiers for optimized I/O. The *Core Module* forwards the DL job to the manager for execution followed by the manager sharing the batching schedule with the workers and coordinating the execution of the DL job with all the worker nodes.

The *Core Module* defines prefetching and caching buffer sizes at each memory and storage tier on the worker nodes. The allocated buffers are adaptive to handle batches with varying sizes depending on the available system resources. A training batch contains n elements of width x_i, height y_i, and depth z_i stored in memory as arrays of d_i bytes objects. The memory size in bytes of one element is the product of the width, height, depth, and the number of bytes consumed per pixel. Therefore, the size S_b of a batch b is computed as $S_b = \sum_{i=1}^{n} x_i \times y_i \times z_i \times d_i$. The manager works closely with the *Prefetcher Module* on all the workers to ensure appropriate buffer sizes. *DeepMemoryDL* starts with reserving 20% of available space at the memory or storage tier and adjusts the allocation of buffer S_t based on the available space at tier t and the total number of batches B scheduled to be loaded onto t such that the total number of elements at the memory or storage tier does not exceed the space allocated to the buffer ($B \times S_b \leq S_t$). The buffer sizes increase as we traverse from the fastest to the slowest tier based on the assumption that the fastest tier is the most expensive and with limited storage space.

Proactive Data Prefetching and Scheduling The manager is tightly integrated with TensorFlow's core coordinating dataset preparation and pre-processing, to ensure that each training

Algorithm 3: Data prefetching and caching scheduling.

```
1: for each worker in cluster do
2:   determine location for $n$ batches
3:   if tier $t$ avail. buffer space $\geq$ space for $n$ batches then
4:     prefetch $n$ batches from location $x$ to tier $t$
5:   else
6:     prefetch $(n - k)$ batches from location $x$ to tier $t$
7:     prefetch $k$ batches from location $x$ to tier $t - 1$
8:   end if
9:   if buffer space in tier $t \geq$ util. threshold at tier $t$ then
10:    if batch $b$ is needed in upcoming $i$ iterations then
11:      cache batch $b$ to tier $t - 1$
12:    else
13:      evict batch $b$
14:    end if
15:  end if
16: end for
```

batch is loaded into the memory before the next iteration. The manager gets information about the DL job from the *Workload Analyzer Module* and determines a schedule and deadlines for I/O operations to stage the required data in the main memory of the worker nodes. Initially, the dataset resides in a cold storage tier, e.g., network-attached storage accessible from each server. The manager locates the dataset and creates a schedule to ensure that the initial dataset required for pre-processing is loaded into the main memory to minimize I/O stalls. The data prefetching and caching approach in *DeepMemoryDL* is shown in Algorithm 3. Given the dataset, the target batch size, and available memory and storage space at each tier, it determines the initial location of each data batch and defines a prefetching and caching schedule. The schedule includes instructions to stage the pre-processed data in the memory subsystem. If the pre-processed data is larger than the available system memory, the additional data is cached in the CXL-based memory instead of slower local storage.

Figure 5.3 shows the flow of data to the prefetching and caching buffers. The prefetching schedule follows priority rules for storing the prefetched data. The priority is: 1) main memory; 2) CXL-based memory; 3) storage tier 0, i.e., CXL scratch storage; and 4) storage tier 1. The prefetching strategy begins with the fastest storage tier and progresses through slower tiers, working alongside the caching mechanism to manage data evictions. Caching operates in reverse order of prefetching to ensure essential data remains accessible in main memory. However, if dataset sizes exceed main

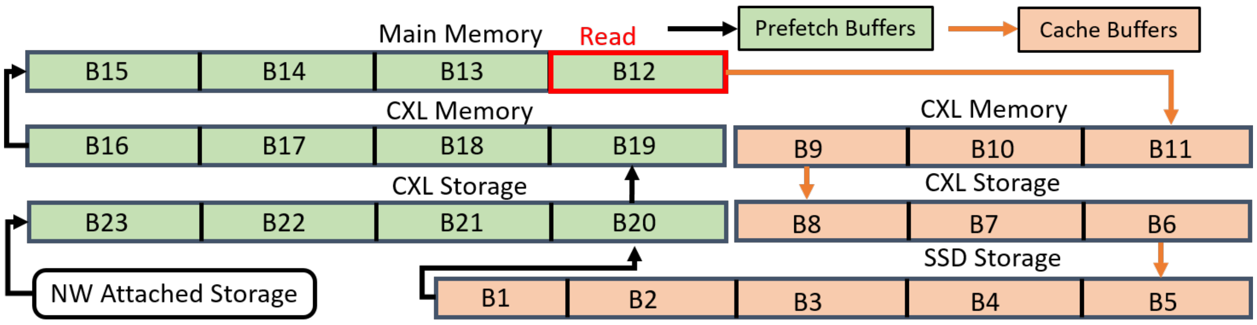


Figure 5.3: Dataflow for prefetching and caching in *DeepMemoryDL* using CXL-based memory and storage subsystem.

memory capacity, eviction coordination halts, allowing prefetching buffers to utilize memory and storage fully. DL tasks can directly access CXL-based memory to enhance I/O performance, despite slightly higher latency compared to prefetching from the local storage.

Allocation of CXL-based Memory and Storage The manager tracks CXL-based memory allocations on all worker nodes and increases the CXL memory allocation in chunks of 512 MB once a request for additional memory is received to avoid using swap space after consuming the entire system memory. Throughout the training phase, each worker node allocates memory to hold both the model parameters and the training dataset. However, as the system's memory availability fluctuates based on concurrent job activity, this can result in memory shortages for DL tasks, thereby impeding training progress. To mitigate this, the manager ensures sufficient memory allocation to accommodate the model's growth over the designated training epochs.

Allocation of Fast Scratch Storage The manager oversees the allocation of fast scratch storage space via CXL-based storage which is crucial for mitigating I/O wait times caused by slower storage tiers within the data processing workflow. Leveraging CXL-based storage for storing intermediate data proves beneficial when system memory is insufficient to hold cached data, a common scenario in DL workloads with sizable datasets. Similarly, it proves beneficial when processed data needs to be written back to local storage.

Prefetcher Module

The *Prefetcher Module* is a part of the manager and worker components taking instructions from the *Core Module* to ensure that the data is prefetched and available to DL workloads before execution begins. The schedule contains information about the worker nodes, assigned chunks of the dataset, memory and scratch space allocation, and a resource map for data placement. For extremely large datasets, the size of a single batch becomes substantially large causing the *Core Module* to define large buffer sizes and launch the prefetching threads ahead of schedule. The *Prefetcher Module* executes the prefetching schedule and reports the prefetching latency back to the *Core Module*. This information is used to dynamically adjust prefetching buffer sizes and the number of prefetching threads to further improve the I/O throughput. The *Prefetcher Module* tracks the memory footprint of each sample and the size of the entire training batch. This gives the *Prefetcher Module* the total size of a single prefetch block which is used to estimate the time it takes to prefetch a batch. For data transferred over the network, *DeepMemoryDL* incorporates the available link bandwidth and the latency to compute an estimated time to prefetch a given batch. This information is subsequently used to launch the prefetching threads and execute the prefetching schedule. The prefetching threads execute the schedule concurrently with the DL training job to ensure that the batch required in the next iteration is prefetched in the main memory.

The manager also defines a caching policy that is implemented by all worker nodes to ensure quick access to the training data not in the main memory. Figure 5.4 illustrates the caching policy of *DeepMemoryDL*. The policy uses the resource map provided by the *Prefetcher Module*. Once the main memory is fully used, the workers run the eviction policy to free up the main memory. The I/O buffers at each memory and storage tier hold the prefetched data. The size of I/O buffers are dynamic to incorporate the variations in the size of each batch. The manager defines and uses an eviction strategy to evict data from these buffers to make space for new data for the next iterations. The eviction strategy in *DeepMemoryDL* works closely with the TensorFlow training schedule and the *Prefetcher Module*. The policy is based on the following rules: 1) data is evicted in FIFO order; and 2) samples within a batch that are marked for prefetching will be cached to a lower memory tier. The eviction policy in *DeepMemoryDL* ensures that enough space remains available in the buffers of each tier and unnecessary expansion of a buffer is avoided at each tier. The worker nodes cache the evicted data to a lower memory and storage tier if the data is required by subsequent training iterations. The caching policy of *DeepMemoryDL* ensures that: 1) data is cached until the buffers are full; 2) the cached data is evicted in the first-in-first-out (FIFO) order; 3) data is always cached from a higher (faster) tier to a lower (slower) tier based on the prefetching schedule. The data that is needed first by the *Prefetcher Module* is kept in the CXL-based memory. Once the buffers in the CXL-based memory are full, the lower priority batches are cached into the storage tiers. *DeepMemoryDL* is more effective for workloads with high data re-use, such as DL jobs, due to prefetching and caching policies that ensure data to be prefetched is available in the fastest tier. However, for workloads with low data re-use, the prefetcher ensures

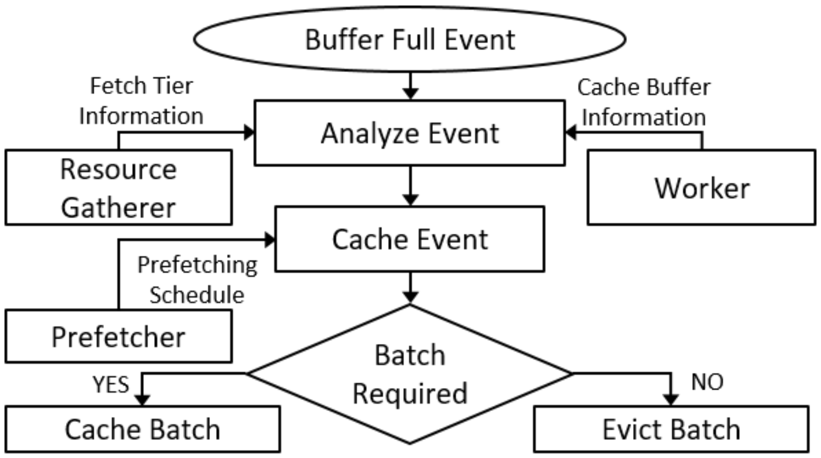


Figure 5.4: Caching workflow in *DeepMemoryDL*.

that the required data is available in local/CXL-based memory before it is required for processing. For such workloads, *DeepMemoryDL*'s caching policy avoids aggressive caching to lower memory and storage tiers because data is not re-used by the workload.

By default, the Linux operating system employs a caching mechanism to store application data read from local storage, assuming that this data will be accessed again soon. Subsequent reads are then served from this cache, reducing I/O latency by avoiding disk access. However, as the memory demands of a DL job increase, requiring more memory, data from the cache is evicted to accommodate the needed data. Hence, *DeepMemoryDL*'s caching policy plays a crucial role in ensuring that essential data remains in the main memory. This is because Linux's caching policy may evict data crucial for upcoming training iterations. Through its caching policies, *DeepMemoryDL* minimizes the need to read training batches from cold storage, thereby enhancing I/O throughput.

5.2 Performance Evaluation

5.2.1 Testbed Setup

Our evaluation setup consists of eight servers running Ubuntu 20.04 LTS server operating systems each with two 2.40 GHz Intel Xeon Gold 6240R processors, with 192 GB main memory, out of which 96 GB of the main memory is reserved for emulating CXL-based memory and CXL-based storage scenarios, and 10 Gbps Ethernet between servers. We emulate the provisioning of CXL-based memory by allocating memory from the remote NUMA domain and CXL-based storage devices by creating a RAMDisk [88] on the remote NUMA domain. We stress the memory and storage subsystem with large datasets as we focus on realistic scenarios where HPC servers have less memory available to store the entire dataset. To evaluate *DeepMemoryDL*, we use ImageNet [71] dataset and ResNet50 [105], Inception-V3 [215], and MobileNetV2 [112] models. We use Intel PCM [15], and `sysstat` [26] to monitor the memory, disk, swap, NUMA domains, and network activity during the execution of DL workloads. We develop a memory hogger to hog system memory on the worker nodes to mimic the behavior of background jobs in production data centers. We investigate the impact of limited system memory, the availability of CXL-based memory and storage tiers, and the impact of proactive prefetching and caching on the performance of DL jobs. Existing state-of-the-art prefetching and caching approaches [80, 166] are either developed for a single server or do not incorporate the characteristics of additional memory and storage tiers, specifically, CXL-based devices.

5.2.2 Execution Environments

We analyze the performance of *DeepMemoryDL* using five realistic TensorFlow environments depending on the availability of memory and storage subsystems. These environments are:

(1) ***Unconstrained Baseline Environment*:** This environment represents an ideal scenario with no resource constraints or sharing between DL jobs.

(2) ***Constrained Baseline Environment*:** This environment represents a realistic scenario where resources are shared and limited memory is available for DL jobs.

(3) ***CXL-based Storage Environment*:** This environment has CXL-based storage available with limited system memory.

(4) ***CXL-based Memory Environment*:** This environment has CXL-based memory available with limited system memory.

(5) ***CXL-based Memory and Storage Environment*:** This environment has CXL-based memory and CXL-based storage available with limited system memory.

(6) ***DeepMemoryDL*:** This environment has our proposed framework, *DeepMemoryDL*, integrated with TensorFlow, which manages system resources including CXL-based memory and CXL-based storage to run DL jobs.

5.3 Performance Results

5.3.1 Total Execution Time of the DL Job

We evaluate the effectiveness of *DeepMemoryDL* in reducing the overall training time and compare it with the studied environments. Figure 5.5 shows the result for training a DL job over 3 epochs with a batch size of 64. For all models, we observe that *Constrained Baseline Environment* takes the longest time due to limited memory availability on worker nodes and excessive swapping of pages to the underlying SSD-based storage. The *CXL-based Memory Environment* enables DL jobs to train using a larger working set by allowing access to CXL-based memory, however, the additional latency of the CXL-based memory increases the training time by 14% on average compared to the *Unconstrained Baseline Environment*. The *CXL-based Storage Environment* provides fast storage space to DL jobs to read the input data from CXL-based storage which results in a significant performance increase over reading data from the SSD. However, due to the limited system memory, the training time increases by 10% on average compared to the *Unconstrained Baseline Environment* but reduces the training time by 9% as compared to the *Constrained Baseline Environment*. Overall, we observe that *DeepMemoryDL* reduces the training time by up to 20%, 34%, 27%, and 25% as compared to the *Unconstrained Baseline Environment*, *Constrained Baseline Environment*, *CXL-based Memory Environment*, and *CXL-based Storage Environment*, respectively. The performance improvement of *DeepMemoryDL* is attributed to the allocation of CXL-based resources, prefetching of data batches to main memory, caching data into CXL-based storage instead of the underlying SSD-based storage, and tailored data eviction policies.

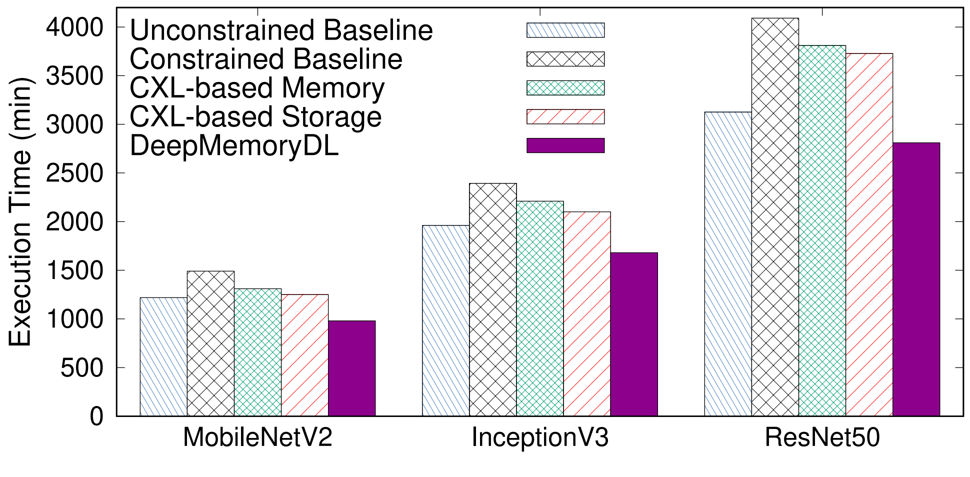


Figure 5.5: Total execution time of DL job with batch size of 64 and 3 epochs.

5.3.2 Data Pre-Processing Phase

Optimizing the performance of the input data pipeline is crucial to the performance of DL jobs. To evaluate the impact of *DeepMemoryDL* on the pre-processing stage we pre-process 40 GB of images from the ImageNet dataset. The pre-processing phase consists of downloading, extracting, generating training and validation data, shuffling, and reshaping images. Figure 5.6 shows the results. Before pre-processing, *DeepMemoryDL* proactively prefetches the input dataset to CXL-based storage to significantly reduce loading time and then caches the data in CXL-based memory to move data closer to compute threads allowing for faster prefetching onto main memory. These policies defined by the *Core Module* of *DeepMemoryDL* yield better I/O performance as compared to *Constrained Baseline Environment* and reduce overall data pre-processing time by 56%, 43%, and 23% as compared to *Constrained Baseline Environment*, *CXL-based Memory Environment*, and *CXL-based Memory and Storage Environment*, respectively. The impact of *DeepMemoryDL* is further observed while training a model with the pre-processed data as data batches are prefetched to reduce the training time for subsequent iterations. We note that the experiment results shown in Figure 5.6 only involve data pre-processing, therefore, the change in the batch size does not impact pre-processing time.

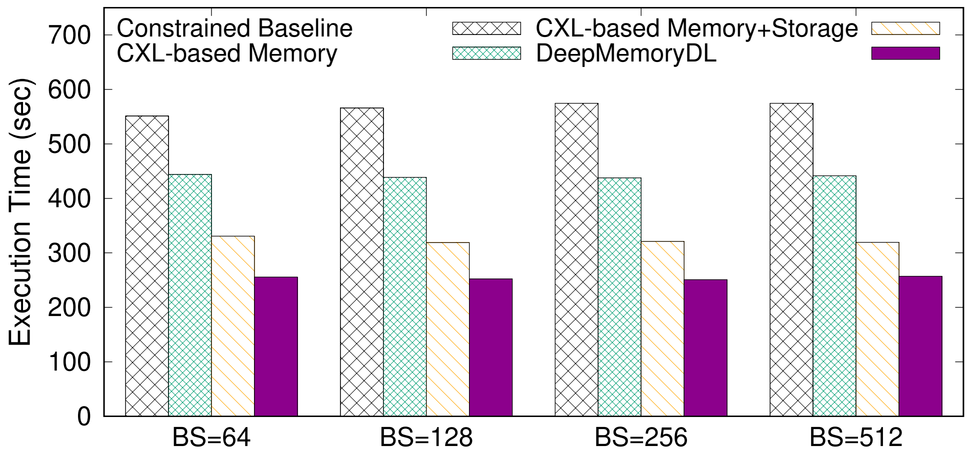


Figure 5.6: Data pre-processing time for ImageNet dataset with limited main memory.

5.3.3 Impact of Using CXL-based memory on DL Job

Large memory systems allow DL jobs to train large models and datasets with sufficient system resources, however, limited memory leads to premature termination of DL job causing computation loss. In such a scenario, a DL job is either restarted or resumed from the last checkpoint. We conduct experiments with varying CXL-based memory allocation to DL jobs to study the impact on the execution time using a batch size of 64. Figure 5.7 shows that as memory footprint of DL jobs increases, *DeepMemoryDL* allocates CXL-based memory to expand its working set size to the CXL-based memory and as the CXL-based memory footprint increases, the total training time is reduced due to the usage of a faster memory tier as compared to the SSD-based swap storage.

We evaluated the impact of available CXL-based memory and dynamic buffer sizes on the total execution time of DL jobs. Figure 5.8 shows that as the buffer size increases, the execution time reduces due to the increased prefetching and caching capacity at the CXL-based memory tier. We observe that as the batch size increases the execution time increases proportionally, however, *DeepMemoryDL* adjusts the I/O buffer sizes based on the footprint of a data batch for prefetching and caching. *DeepMemoryDL* mitigates the impact of using large batch sizes on the execution time of a DL job and also enables TensorFlow to manage a much larger working set size.

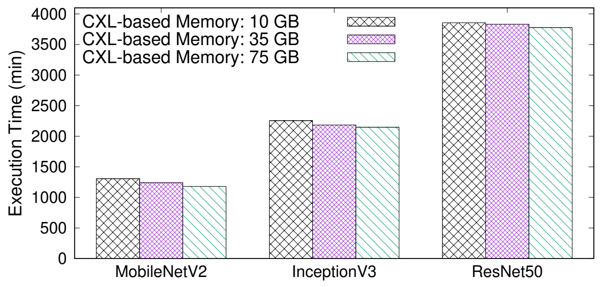


Figure 5.7: Impact of CXL memory allocation on DL job with batch size of 64 and 3 epochs.

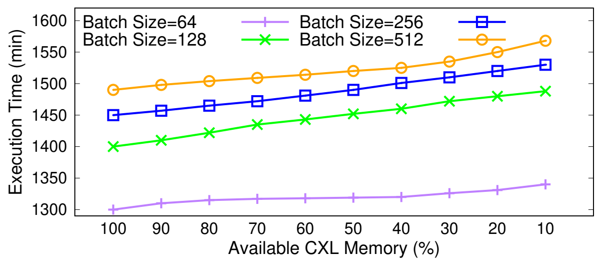


Figure 5.8: Impact of CXL-based memory on total execution time of DL job with 3 epochs.

5.3.4 Impact for Using CXL-based Storage for Staging Data on DL Job

We conduct experiments to study the impact of using CXL-based storage on read and write operations for staging large datasets. Figure 5.9 shows that *CXL-based Storage Environment* reduces the execution time by up to 30% as compared to the *Constrained Baseline Environment* due to the improved I/O performance of the CXL-based storage. Overall, *DeepMemoryDL* reduces the execution time by up to 20%, 32%, and 24% as compared to the *Unconstrained Baseline Environment*, *Constrained Baseline Environment*, and *CXL-based Storage Environment*, respectively. *DeepMemoryDL* stages data in the CXL-based storage resulting in improved performance as compared to the other execution environments since storing the entire dataset in the staging area ideally yields higher read bandwidth and IOPS.

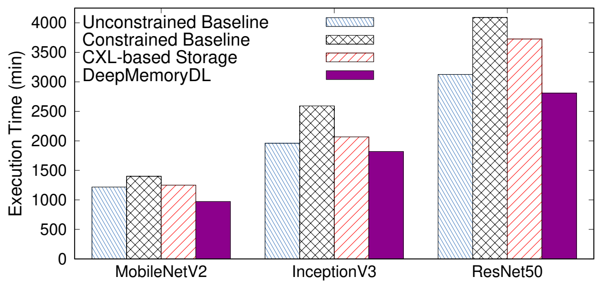


Figure 5.9: Impact of data staging storage on DL job with batch size of 64 and 3 epochs.

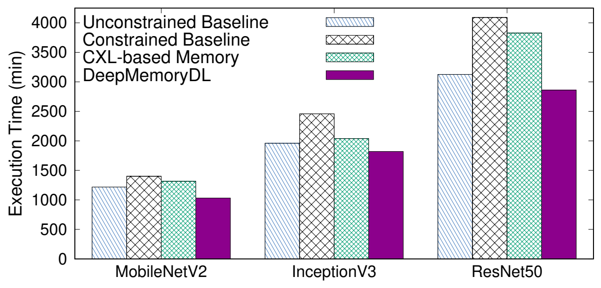


Figure 5.10: Impact of proactive prefetching & caching on DL job with 64 batch size & 3 epochs.

5.3.5 Impact of Proactive Data Prefetching and Caching on DL Job

We study the impact of proactive data prefetching and caching on the execution time of DL jobs while training MobileNetV2, InceptionV3, and ResNet50 models using the ImageNet dataset over 3 epochs and a batch size of 64. Figure 5.10 shows that by effectively managing the CXL-based memory, *DeepMemoryDL* reduces the execution time of DL jobs by 15%, 30%, 25% as compared to the *Unconstrained Baseline Environment*, *Constrained Baseline Environment*, and *CXL-based Memory Environment*, respectively. Moreover, the *Unconstrained Baseline Environment* performs better than the *CXL-based Memory Environment* when enough space is available in the main memory by an average overhead of up to 10%. The *CXL-based Memory Environment* provides additional CXL-based memory to DL jobs that reduces the execution time by up to 17% as compared to the *Constrained Baseline Environment*. Data eviction contributed to the worst performance of the *Constrained Baseline Environment*. The manager in *DeepMemoryDL* optimizes the caching mechanism by preparing a caching policy that is implemented on all worker nodes running the training job. Additional memory is pooled from CXL-based memory to cache processed data to ensure that the required data is always available in the faster available tier. In tandem with prefetching, caching improved the performance of *DeepMemoryDL* and optimized memory resource utilization.

5.3.6 Scalability Analysis of ***DeepMemoryDL*** on DL Job Performance

To evaluate the performance of *DeepMemoryDL* we use a combination of real-world use cases and vary the available system resources and workers. Single worker training is resource-intensive since the entire dataset has to be processed on a single node requiring more memory and storage resources. Multi-worker training divides the dataset between workers to reduce the resource requirement on each worker. However, for large datasets, memory and I/O remain the bottlenecks. We studied the impact of increasing the number of workers on *DeepMemoryDL* by training MobileNetV2 and compare its execution time with *Constrained Baseline Environment*. Figure 5.11 shows that *DeepMemoryDL* outperforms the *Constrained Baseline Environment*, and the respective performance gap remains similar as we increase the number of worker nodes. Moreover, the performance of a DL job improves linearly using *DeepMemoryDL* as we increase the number of worker nodes.

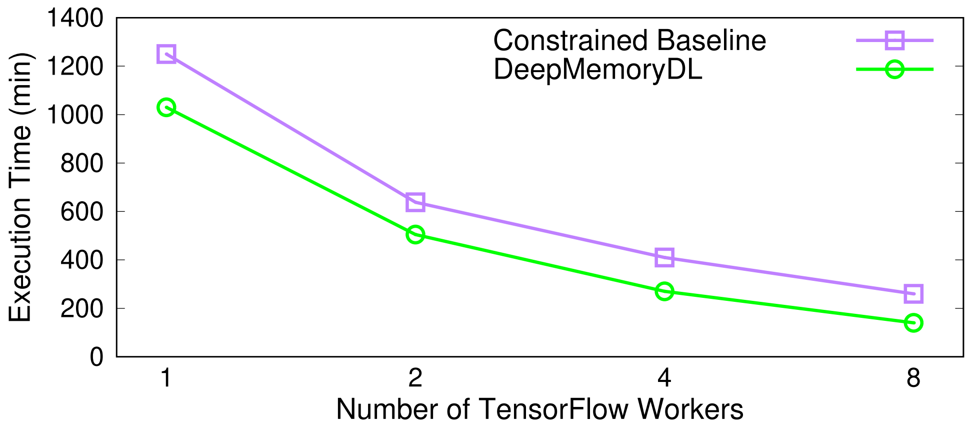


Figure 5.11: Scalability analysis on total execution time of DL job with 64 batch size & 3 epochs.

5.4 Summary

To summarize, *DeepMemoryDL* improves DL performance by efficiently leveraging storage and memory tiers to proactively prefetch and cache training data. We emulate CXL memory to provide additional memory and fast scratch storage space to DL workloads and reduce the overall training time. Overall, *DeepMemoryDL* reduces the overall training time of a DL job by up to 34% and 27% as compared to the default TensorFlow and CXL-based memory expansion approaches, respectively. In our future work, we will extend *DeepMemoryDL* to support other DL platforms, specifically the PyTorch platform, to improve its performance by eliminating I/O stalls.

Chapter 6

Fault-tolerant FaaS for Stateful Time-sensitive Applications

6.1 System Design

To provide fault tolerance to stateful time-sensitive applications we present a fault-tolerant and resilient stateful FaaS framework, *Canary* [38], that extends existing FaaS platforms by adding new software modules that store function states, critical checkpoint data and replicate function runtimes for faster failure recovery. *Canary* ensures that functions execute *exactly once* on FaaS platforms for achieving minimal application execution time. *Canary* achieves this by proposing a modular architecture, as shown in Figure 6.1.

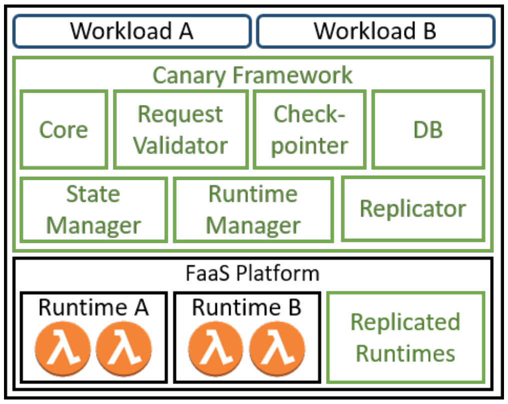


Figure 6.1: High-level architecture of *Canary*.

Canary consists of a *Core Module* that handles end-to-end job execution, failure recovery, and coordination between various components. The *Request Validator Module* is used by the *Core Module* to avoid failures regarding the submitted job request. The *Checkpointing Module* handles the state and critical data checkpointing. The *Runtime Manager Module* keeps track of all the function runtimes deployed and the runtime replicas for the jobs created by the *Replication Module*. The database stores the function states and the checkpoints.

***Core Module*:** It orchestrates execution between various components and modules, as shown in Figure 6.2. It exposes a lightweight listener that receives incoming user requests and forwards them to the *Request Validator Module* for validation and generates a set of unique IDs for the submitted jobs functions, checkpoints, and replicas used to identify functions, corresponding applications, location of functions, identification of failed functions, and the associated checkpoints.

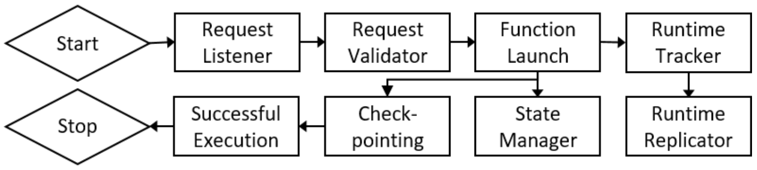


Figure 6.2: High-level orchestration of *Canary* modules.

The *Core Module* handles the creation and maintenance of the required database tables. The five main tables created in the database are *worker_info*, *job_info*, *function_info*, *checkpoint_info*, and *replication_info*. The *worker_info* table stores information about the platform, including the number of nodes in the cluster and worker-specific information, i.e., assigned roles and system specifications. The *job_info* table stores information about the submitted job, its unique ID, the number of functions launched for each job, and other critical information required by the *Core Module*. The *function_info* table stores information about all the functions launched for the submitted jobs, their unique IDs, the job ID to which they belong, runtime for each function, and the worker on which the function is deployed. The *checkpoint_info* table stores information about the checkpoints of each function, its unique ID, job ID, function ID, and the state information related to the checkpoint. Finally, the *replication_info* table stores information about the replicated runtimes deployed on the FaaS platform. It also includes the runtime information, job ID, and the worker information where the replicated runtime is deployed.

The *Core Module* forwards the validated requests for scheduling and creates database entries based on the type of job, its runtime, number of scheduled functions, checkpointing frequency, and the replication factor. The *Core Module* forwards the information to the *Checkpointing Module*, which

Algorithm 4: State and Critical Data Checkpointing.

```
Input: Func. ID $f_{id}$, Job ID $j_{id}$, State $st$, Checkpoint $ckpt$
1  begin
2      for each $st$ do
3          if user ckpt then
4              get $ckpt_{data}$, $ckpt_{name}$, $ckpt_{loc}$
5              if $ckpt_{data} > db_{limit}$ then
6                  $ckpt_{data} \rightarrow disk$
7                  $ckpt \leftarrow \{ckpt_{name}, ckpt_{loc}\}$
8              else
9                  $ckpt \leftarrow \{ckpt_{data}\}$
10         else
11             $ckpt \leftarrow \{st, data_{cric}\}$
12         if $ckpt_{cur} > ckpt_{thresh}$ then
13             remove $ckpt_{oldest}$ from $db$
14         push $\{j_{id}, f_{id}, ckpt_{id}, ckpt\}$ to $db$
```

stores the checkpoint metadata in the database. It coordinates between different components of *Canary*'s runtime, which includes scheduling of function runtimes, usage details, and the entire life cycle, through the *Runtime Manager Module*. It also keeps track of all scheduled functions and their current states. Upon function failure, the *Core Module* detects the failure, identifies the function runtime, gathers checkpoint information, and initiates the recovery process. The recovery process restores the function from its latest checkpoint available on the runtime associated with the failed function.

Request Validation Module: Its primary purpose is to prevent request failures before *Canary* starts processing the request. It accepts requests from the *Core Module* and uses the job information and the resources requested from the FaaS platform for validating the job request. The *Request Validator Module* verifies if the requested resources are within the resource limits of the FaaS platform, and the user has not reached the associated maximum concurrent function limit. For example, if invoking a new function would result in a concurrency failure because the requested functions, if launched, will exceed the maximum limit, the *Request Validator Module* notifies the *Core Module* which queues the job until there is enough limit available to launch new functions.

Runtime Manager Module: It tracks all runtimes used by the running functions in the cluster and works alongside the *Replication Module* to replicate these runtimes. It maintains information about the used runtimes and their corresponding replicated runtimes and enables the *Core Module* to map the failed functions to the replicated runtimes in the event of a function failure. Moreover, the *Runtime Manager Module* stores the location information of replicated runtimes that are deployed in the cluster.

Checkpointing Module: Stateful functions produce data that must be stored and persisted during and after the function execution along with the state information. Functions that belong to the same application require the state information and data from the previous functions for their successful execution. *Canary* supports fault-tolerant stateful function by maintaining the state information of all functions along with the application data. The *Checkpointing Module* exposes its core functionality via an API that interacts with other modules to monitor and record the state of running functions. Application states can be defined in the application code that will be used by the *Checkpointing Module* for checkpointing. With minimum modification to the function code, application states are registered by calling the *Canary* APIs. The specified states are stored throughout application execution and are used to recover a failed function.

The *Checkpointing Module* allows the definition of critical data within the application code that should be replicated and persisted after the successful function execution. This functionality is critical when an application must store its critical data structures along with the function state. This data is added to the state information. We show *Canary*'s approach of checkpointing application states and critical data in Algorithm 4. The location for storing critical datasets is determined by the total size of the dataset. Checkpoints in *Canary* are primarily maintained in an in-memory key-value (KV) data store. We use Apache Ignite [22] as the KV store for storing the state information. However, in-memory databases limit the size of data stored per key. The *Checkpointing Module* transfers the checkpoint data to a faster storage tier available in the system such as persistent memory, Ramdisk, or to a shared storage accessible to all cluster nodes. The storage hierarchy is determined at the deployment phase of the FaaS platform and can be overwritten by a custom storage endpoint, such as an S3 bucket. The *Checkpointing Module* executes in a linear time to checkpoint the state of each function in a given job. Algorithm 4 yields $\mathcal{O}(S)$ complexity, where S denotes the number of states within a function.

Canary records a series of state checkpoints throughout the function execution and stores the latest n checkpoints in an in-memory data store. The initial value of n is set to 3, which is dynamically adjusted throughout the execution based on the application data to be checkpointed and the frequency of states produced during function execution. An application state is comprised of current values of its critical data structures that are registered with the *Checkpointing Module*.

The critical data remains available in the persistent storage or a KV store and is used to restore the corresponding failed function. For enabling quick lookup, application states are stored in a KV store where the key corresponds to the function ID and the value corresponds to its states. When a new function is assigned the task of a failed one, the *Checkpointing Module* issues a query to the KV store to retrieve the state of the given function ID. When the size of the checkpoint and the data exceed the database limits, the data is then stored in a fast storage tier or external storage, e.g., an in-memory storage or a distributed persistent memory, and the location of the checkpoint is pushed to the database along with the state information. For a DL workload, the checkpoint also includes a copy of pre-processed data, model weights, and other data required to resume the training process from the failed epoch.

By default, *Canary* implements an implicit checkpointing strategy, which has coarse-grained control over checkpoint intervals, location of stored checkpoints, and restoring function state from the stored checkpoint in the event of a failure. *Canary* also supports explicit checkpointing where the application can specify its state and data for creating checkpoints, thus reducing the checkpoint size and the associated overhead while increasing the programming complexity. In both of these approaches, checkpoints are first stored in either the KV-store or written in-memory and then flushed asynchronously to the shared storage that is available to all nodes in the cluster.

The *Checkpointing Module* handles the recovery of failed functions by restoring the function state and data to a new function. Checkpointing provides the record of previous states of the function to avoid restarting the function from the beginning. The *Core Module* detects failed functions in the cluster and handles the end-to-end recovery process. It identifies the execution runtime required by the failed function, the latest checkpoint available, and the location of the checkpoint data. The *Core Module* ensures that the best possible replicated runtime is selected to minimize the recovery time. Once the replicated runtime is located, the function is deployed on it along with the checkpointed function state and data. The *Core Module* notifies the *Runtime Manager Module* about the runtime utilized during the recovery process. Once the function state has been recovered, the function resumes normal execution and continues execution from its previous state. In the event of multiple function failures, the default retry-based strategy concurrently restarts all the failed functions which leads to resource contention and further increases the recovery time.

Replication Module: It ensure that *Core Module* quickly recovers failed functions, the *Replication Module* replicates the runtimes used for launching functions of the scheduled jobs. The runtimes used at any given point are replicated throughout the cluster to enable faster recovery by reducing initialization and cold-start latencies by providing warm function runtime. Instead of

Algorithm 5: Runtime Replication at Job Submission

```
Input: Act. funcs. ($func_{act}$), Act. repl. ($rep_{act}$), Repl. loc. ($rep_{loc}$), Sched. funcs. ($func_{sch}$), Sched.
        runt. ($run_{sch}$)
Output: Req. repl. ($rep_{req}$), Repl. loc. ($rep_{loc}$), Repl. thresh. $rep_{th}$
1  begin
2      compute $func_{tot}$ given ($func_{act}$, $func_{sch}$)
3      for each $run_{sch}$ do
4          compute $rep_{req}$
5          if $rep_{req} \geq 1$ then
6              compute $cur\_rep_{factor}$ given ($func_{act}$, $rep_{act}$)
7              compute $new\_rep_{factor}$ given ($func_{tot}$, $rep_{act}$)
8              if $cur\_rep_{factor} < new\_rep_{factor}$ then
9                  determine $rep_{loc}$
10                 launch $rep_{req}$ at $rep_{loc}$
```

creating a replica of each running function's runtime, the *Runtime Manager Module* detects the runtime of an invoked function and verifies whether a corresponding replica is active. The *Runtime Manager Module* only triggers the replication when a function is created with a runtime that is not already replicated in the cluster. Once a replica is assigned to a failed function, the *Runtime Manager Module* creates a new replica if an active function is deployed with the same runtime to replace the existing replica. Therefore, throughout the execution of a function, an active replicated runtime is available to use for failure recovery.

Algorithm 5 explains the runtime replication workflow in *Canary*. Once a new job is submitted to the FaaS platform, the *Core Module* determines the number of functions $func_{sch}$ to launch for the job and the function runtimes run_{sch} to schedule. The replication module uses a linear-time method to compute the total number of functions $func_{total}$, including active functions $func_{act}$, and iterate through the scheduled runtimes for replication. For each run_{sch}, the replication module computes the required number of replicas rep_{req} for a given job. The current replication factor cur_rep_{factor} is the ratio of $func_{act}$ and rep_{act} and the new replication factor new_rep_{factor} includes the $func_{total}$ and rep_{act}. The replication factors determine if enough runtime replicas are available for all running functions. The runtime replication module keeps the current and new replication factors consistent and if the cur_rep_{factor} is less than the new_rep_{factor}, a new runtime replica is launched at the replica location rep_{loc} which is determined to avoid a single point of failure for the submitted job as well as for the FaaS platform. The rep_{loc} is crucial to recover failed functions as it provides enough replicas to the *Core Module* to select a suitable replica to ensure minimal recovery time on heterogeneous resources.

The runtime replication factor maps running functions to the replicated runtimes. A higher factor value shows that the number of replicated runtimes for each runtime is higher. This provides redundancy and allows faster recovery for large function failures but results in higher operating costs. A lower value of the replication factor means that less number of replicated runtimes are launched. This results in lower cost, but, in the event of large function failures, the initialization time of *Canary* for launching new functions becomes the same as the default retry-based strategy. The *Replication Module* dynamically adjusts the replication factor to achieve an optimal operating point which results in less frequent restarts and lower operating costs. The *Replication Module* handles the placement of runtime replicas in the cluster. The replica placement follows a set of rules that determines the ideal location for a replica based on the location of the running functions. The first replica is placed on any worker that hosts the job function. Further replicas are placed away from the worker hosting the first replica to avoid a single point of failure for the replicated runtimes. The placement decisions are locality aware and take into account the location of worker nodes in the data center.

6.2 Performance Evaluation

6.2.1 Testbed Setup

Our testbed consists of a cluster of 16 bare-metal servers from the Chameleon Cloud testbed [131] with two Intel Xeon Gold 6126/6240R/6242 processors, contains 192 GB of main memory, runs Ubuntu 20.04 LTS server operating system, and connected using 10G Ethernet. We deploy OpenWhisk [23] on a Kubernetes [109] cluster along with Docker [220], OpenWhisk CLI (wsk), and CouchDB [36]. We deploy Apache Ignite [22] to store data in the highly scalable distributed cluster using replicated caching mode which ensures that the data is available in the entire cluster. We also enable Ignite native persistence to provide data persistence. The underlying storage for storing large files is shared over NFS [207] across the cluster nodes. We also enable the option to use Intel Optane persistent memory [122] in AppDirect mode [236] or Ramdisk [88] for storing large files and to avoid I/O bottlenecks.

Workloads: To evaluate *Canary*, we use five classes of application workloads: deep learning (DL), web service, Spark [2] data mining, data compression, and graph search. These applications are developed using Python, Node.js, and Java programming languages and use their corresponding execution runtimes. We used these workloads as these are the most widely used function runtimes in FaaS based on the current FaaS adoption trends [10], and include representative serverless HPC applications from the SeBS [66] benchmark. The DL workload is a TensorFlow [27] application that trains ResNet50 [106] model on the MNIST [152] dataset over 50 epochs. Checkpoint data for a DL application include weights and biases collected after every successful completion of an epoch. Web service workload is composed of responding to 50 requests from a web front-end to a database, i.e., PostgreSQL [3]. Each request is composed of five queries and checkpoints include queries and responses after each request. Spark data mining workload entails extracting, transforming, loading, and analyzing the given dataset to get meaningful insights, where each part of the computation is implemented as serverless functions. Specifically, it computes the diversity index at the local and national levels over the US census data [20]. A checkpoint is collected when the output for each location is computed and aggregated with the existing results. Data compression workload is a modified version of the SeBS *311.compression* benchmark that performs zip compression [103] on 50 input files (∼1 GB each). The input and output files are stored in the local storage instead of S3. Each function processes multiple input files and a checkpoint is performed after compressing an input file. Finally, the Graph search workload is based on the SeBS *501.graph-bfs* benchmark which performs Breadth-First Search (BFS) using igraph [13] in a binary tree with 50 million vertices. Each function is checkpointed after 1 million vertices have been traversed. Depending upon the experiment, these workloads require invoking one or more functions where each function is invoked in a separate container.

Performance Metrics: We consider the total execution time, i.e., the time required to complete the submitted application including the time consumed in recovering from failures to study the effectiveness of the studied approaches. We also measure the failure recovery time and perform a cost-benefit analysis of *Canary* in terms of dollar cost incurred to quickly recover from failures by leveraging additional resources, e.g., for function runtime replication.

6.3 Performance Results

6.3.1 Impact of Runtime Replication on Recovery Time

We run the workloads as functions on OpenWhisk and report the impact of replicated runtimes on the failure recovery time for the given workload runtimes. Figure 6.3 shows the impact of replicated runtimes on the workload execution time with varying failure rates for 100 invocations of Python, Node.js, and Java container runtimes. We observe that the replicated runtimes reduce the recovery

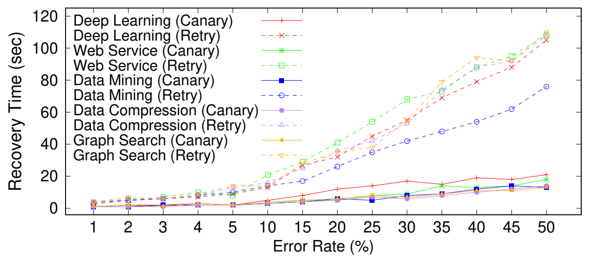


Figure 6.3: Impact of replicated runtimes on recovery time for 100 function invocations.

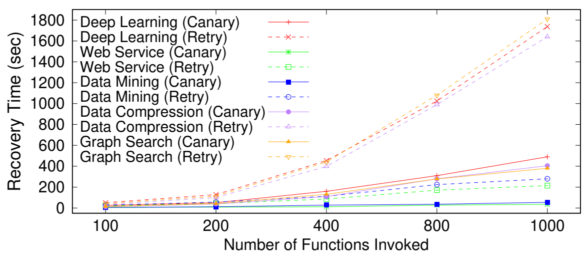


Figure 6.4: Impact of replicated runtimes on recovery time with 15% failure rate.

time by up to 81% as compared to the default retry-based recovery strategy. Moreover, we observe that as the failure rate increases, the recovery time of the default retry-based strategy increases almost linearly due to the increasing number of failed functions. However, due to the replicated runtimes, *Canary* keeps the recovery time fairly constant and stays close to the ideal scenario where there are no function failures. Similarly, the replica placement also incorporates resource heterogeneity to mitigate the impact of variation in recovery time on application performance. As more functions fail, the replicated runtimes are utilized effectively and *Canary* dynamically increases the replication factor to cope with the failures and reduces function initialization time. Overall, *Canary* reduces the recovery time by 76%, 81%, 78%, 79%, and 80% on average as compared to the default retry-based approach for DL, web service, Spark data mining, data compression, and graph search workloads, respectively.

We also run experiments to observe the performance of replicated runtimes for a large number of function invocations in a cluster setup with a fixed failure rate of 15%. The functions fail at random intervals during function execution. The results are shown in Figure 6.4. We observe that the runtime replication strategy performs better than the default retry-based strategy by up to 82%. The recovery time of *Canary* remains close to the ideal scenario where there are no failed functions. The additional time as compared to the ideal scenario is due to the time required to migrate the function to the replicated runtime and includes cases where the platform has to wait for the replicated runtimes to be ready where large numbers of functions fail simultaneously and there are not enough replicated runtime to host the failed functions. *Canary* strategically places the replicated runtimes based on the job, locations of the functions, types of runtime containers used by the failed functions, and the current resource availability in the cluster. Overall, we observe that for this experiment, *Canary* reduces the recovery time by 63%, 82%, 80%, 70%, and 71% on average as compared to the default retry-based approach for DL, web service, Spark data mining, data compression, and graph search workloads, respectively.

6.3.2 Impact of Checkpointing on Recovery Time

We study the impact of checkpoints in recovering from function failures by increasing the failure rate for a fixed number of function invocations. To simulate failures at the given failure rate, the functions are killed at random times during the job execution. The result is shown in Figure 6.5. We observe that the recovery time depends on the function failure rate and the time at which the failure occurs during the function execution. The recovery time for the retry-based strategy is large when a failure occurs close to the function completion. Moreover, we observe that the total execution time of a failed job remains close to the ideal execution scenario of failure-free execution specifically when the failure rate is low. *Canary* reduces the recovery time of the failed function by up to 83% as compared to the default retry-based recovery strategy.

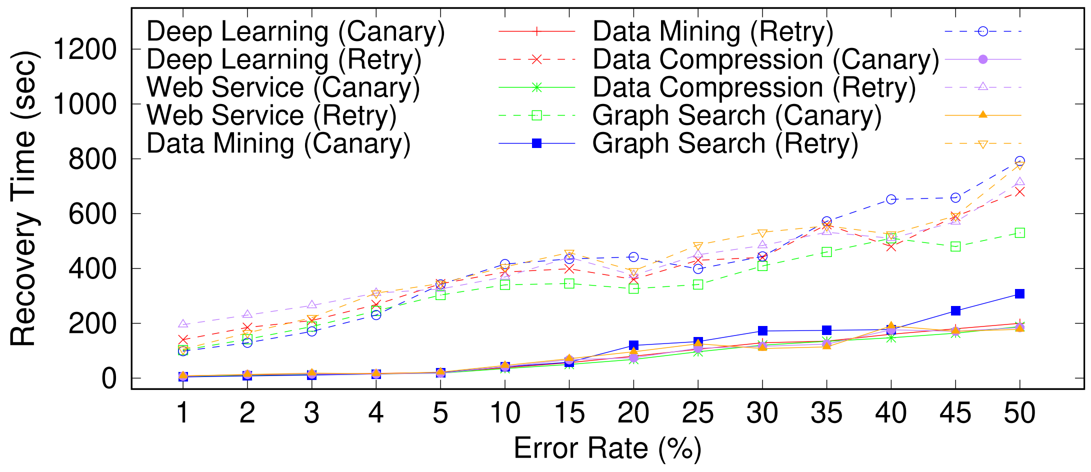


Figure 6.5: Impact of checkpoints on recovery time for 100 function invocations.

Overall, we observe that for this experiment *Canary* reduces the recovery time by 82%, 81%, 79%, 83%, and 82% on average as compared to the default retry-based approach for DL, web service, Spark data mining, data compression, and graph search workloads, respectively. *Canary* ensures that the function is recovered from the latest checkpoint, thus reducing the recovery time and keeping it consistent regardless of when the failure occurs during the function execution.

6.3.3 Impact of Using *Canary* on the Workload's Makespan

We study the impact of *Canary* on the total execution time of the studied workloads. Figure 6.6 shows the result of the total execution time of *Canary* with the default retry-based approach for various failure rates using the studied DL workload. The replicated runtimes provide a quick way to restore the latest checkpoints of the failed functions. We observe that *Canary* is more effective than the retry-based approach as the failure rate increases and when a failure occurs towards the end of workload execution. We also compare the execution time with the ideal scenario where there is no function failure. The retry-based strategy diverges from the ideal execution time as the failure rate increases, however, we observe that the execution time using *Canary* is comparable to the ideal execution time. Overall, *Canary* increases the execution time by 14% on average as compared to the ideal scenario without any failure. The overhead associated with *Canary* is because of the worst case scenario where the function fails right before a checkpoint is taken and recovers from the previous saved checkpoint. The retry-based recovery strategy performs the worse because of the loss of the entire computation of the failed function and restarting of the execution from the start upon failure. Our evaluation shows that *Canary* reduces the total execution time by up to 83% with a failure rate of 50% over the default retry-based recovery strategy. We observe similar performance trends in terms of the execution time for the web service and Spark data mining applications.

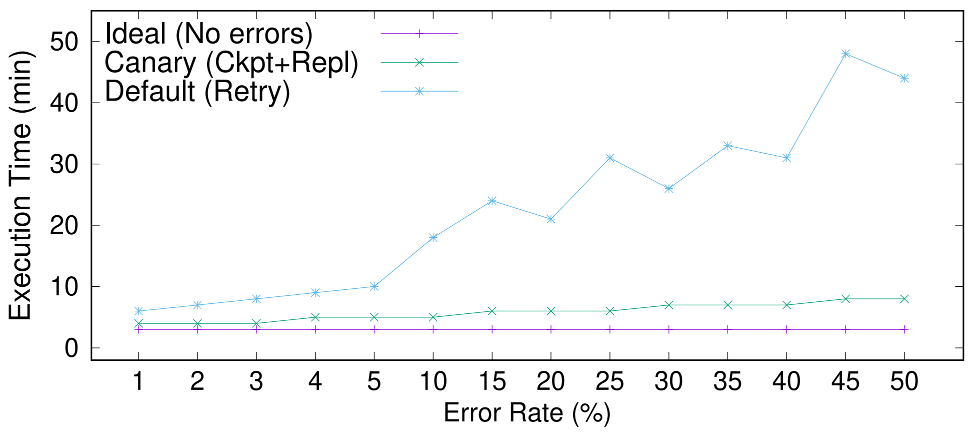


Figure 6.6: Execution makespan of 100 function invocations for the DL workload with replication and checkpointing.

6.3.4 Impact of Using *Canary* on Dollar Cost

We perform a dollar cost analysis of using *Canary* by calculating the total cost of the launched functions and the replicated runtimes. We consider the pricing model of \$0.000017 per second of execution, per GB of memory allocated from IBM Cloud Functions [11] as it is based on Apache OpenWhisk which we use for prototyping *Canary*. However, the pricing model of AWS Lambda [4] is comparable, i.e., ∼\$0.0000167 per second of execution, per GB of memory allocated. We correlate the cost with the total job execution time. For our analysis, we consider the total execution time as the time from the first launched function to the completion of the last function. Moreover, the cost of concurrent functions is aggregated to represent the overall dollar value of a workload's execution.

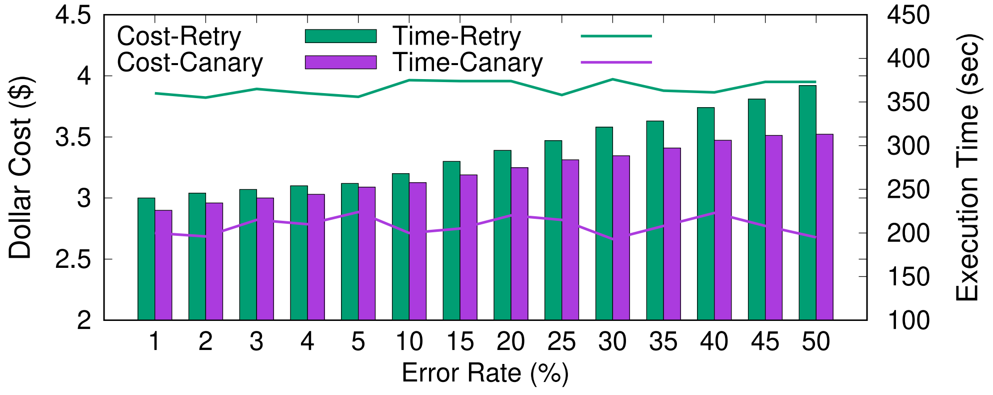


Figure 6.7: Impact of failure on cost and time of training ResNet50 on CIFAR10 over 50 epochs.

Figure 6.7 shows the cost and execution time of *Canary* as compared to the retry-based recovery strategy. We observe that as the failure rate increases, the total cost for both *Canary* and the default strategy increases proportionally. Moreover, the difference between the cost of the retry-based strategy and *Canary* becomes larger with the increase in error rate. *Canary* has a lower cost by up to 12% as compared to the default retry-based strategy due to the replicated runtimes and checkpointing strategy. Overall, *Canary* improves application availability at an average additional cost of 8% as compared to the ideal scenario without any failure. We also observed that the cost of the retry-based strategy is much higher as compared to *Canary* for high failure rates. For the retry-based recovery strategy, functions failing close to the end of their execution incur much higher costs as they have to redo the entire execution from the beginning. In the case of *Canary*, the function is recovered from the latest available checkpoint and completes the remaining execution. We observe that the execution time for *Canary* is 43% less on average as compared to the retry-based technique demonstrating the benefits of *Canary* at a reduced cost even with the additional overhead of replicating the runtimes. The overhead of function checkpointing and data replication in *Canary* results in an additional cost that depends on the error rate, checkpoint location, network congestion, and the number of replicated runtimes.

We also evaluate the impact of replication on the cost and execution time of functions. We evaluate three replication strategies: dynamic, aggressive, and lenient replication. Dynamic replication (DR) is the default strategy of *Canary* in which the replication factor is dynamic and adjusted based on the failure rate. Aggressive replication (AR) uses a higher replication factor for each running job. Lenient replication (LR) maintains one active replica throughout the execution of each job. The results of this experiment are shown in Figure 6.8. We observe that when *Canary* increases the replication factor, the associated cost also increases because of concurrently running the additional runtimes with the application functions. LR strategy results in slightly lower cost as compared to the DR strategy, however, the job execution time with LR increases at a higher rate with the increase in the failure rate. Moreover, we observe a slower increase in the execution time using AR as compared to LR and DR. This trend shows that dynamic replication used by *Canary* scales better and provides better reliability as compared to the LR approach. AR yields a higher overall cost but has the lowest execution time. As more functions fail, the number of deployed replicas under the AR approach increasingly matches the number of failed functions. This trend results in fewer unused instances and a lower cost per replica. Overall, the DR approach outperforms AR and LR approaches by 25% and 2% on-average dollar cost savings, respectively.

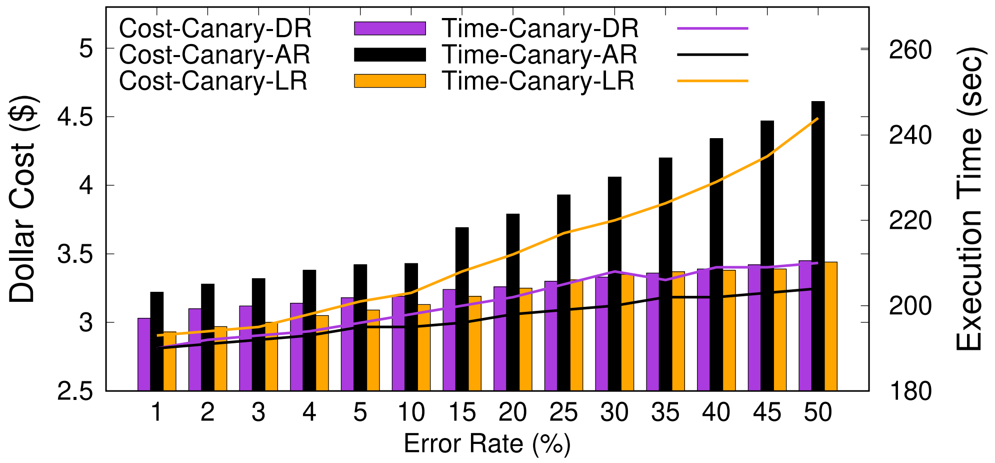


Figure 6.8: Impact of replication on cost and time of training ResNet50 on CIFAR10 over 50 epochs with aggressive replication (AR), lenient replication (LR), and dynamic replication (DR).

6.3.5 Comparison of *Canary* with State-of-the-Art Fault Tolerance Techniques

We compare the performance of *Canary* with state-of-the-art fault tolerance techniques i.e., request replication (RR) [84] and active-standby (AS) [54]. RR launches multiple replicated functions for each given function based on the given replication factor. Similarly, AS creates two function instances; one for serving all requests and the other as standby. In our evaluation, we launch one replica per request. The incoming requests are forwarded to all functions and the first successful response is accepted and the rest are discarded. Figure 6.9 shows the comparison of *Canary* with RR and AS approaches. We observe that both RR and AS result in higher costs than *Canary* by up to 2.7× and 2.8×, respectively, because of launching additional functions as replicas or standby. As the error rate increases, the probability of active, standby, and replicas functions being killed at the same time increases, thus increasing the overall execution time and cost as failed functions must be restarted from the beginning. Similarly, we observed that the standby function stays dormant while replicas process incoming requests, hence, consuming additional system resources, resulting in resource contention, and increasing the resource requirements for functions execution. We observe that as the error rate increases, the execution time of *Canary* increases by 5% on average as compared to RR due to the checkpoint restore approach of *Canary*. The execution time of AS increases with the error rate because an increased number of failed functions are redeployed on the standby instances. Overall, the execution time of AS is up to 34% higher than *Canary*. This is because stateful functions depend on previous states for correct operation and functions are restarted as there is no checkpoint in the AS technique.

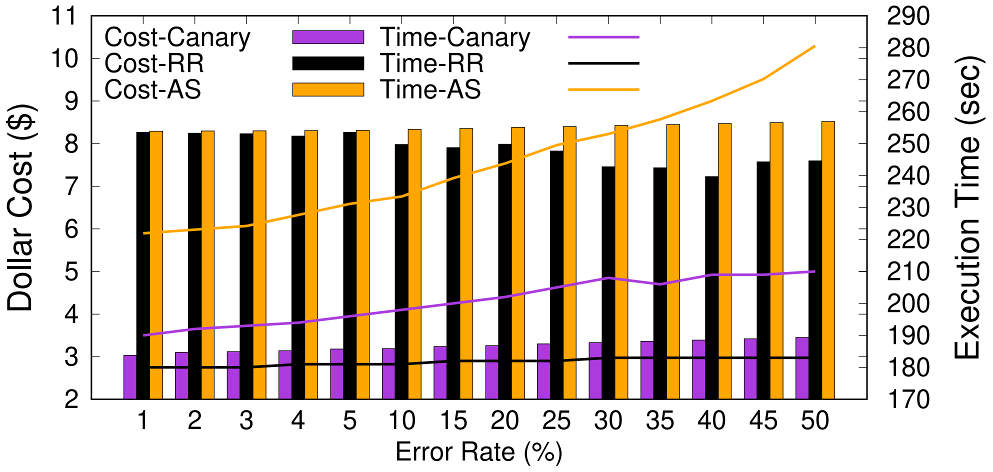


Figure 6.9: Comparison of *Canary* with active-standby (AS) and request replication (RR).

6.3.6 Impact of Scaling on *Canary* Performance

We increase the size of the computing cluster and the number of submitted jobs to observe the performance of *Canary*. We concurrently launch several hundred, i.e., 200, 400, 800, and 1000, functions and randomly kill the running functions that belong to various FaaS jobs and observe the total execution time of the submitted jobs. We also increase the failure rate proportional to the number of functions launched. Figure 6.10 shows the results of this experiment. We observe that as the number of functions increases, the total recovery time of the submitted batch of FaaS jobs remains fairly constant as compared to the default retry-based approach. The recovery time of *Canary* stays close to zero which matches the optimal failure-free scenario. However, with increased failure rates *Canary* experiences a slight increase in the recovery time due to recovery overhead. Our experiments include cases with node-level failures that lead to total loss of computation for the jobs scheduled on the failed node. For large function invocations, the retry-based strategy shows an interesting behavior, i.e., the recovery time depends on the time at which the function fails and if there is any node-level failure. For node-level failures, a large number of functions are restarted at the same time, hence, the recovery time for these functions overlap and is equal to the longest recovery time of any single failed function. Node-level failures in *Canary* are treated differently due to the availability of checkpoints in the shared storage system that is accessible throughout the cluster. Overall, in this experiment, we observe that *Canary* reduces the average recovery time by up to 80% as compared to the retry-based approach.

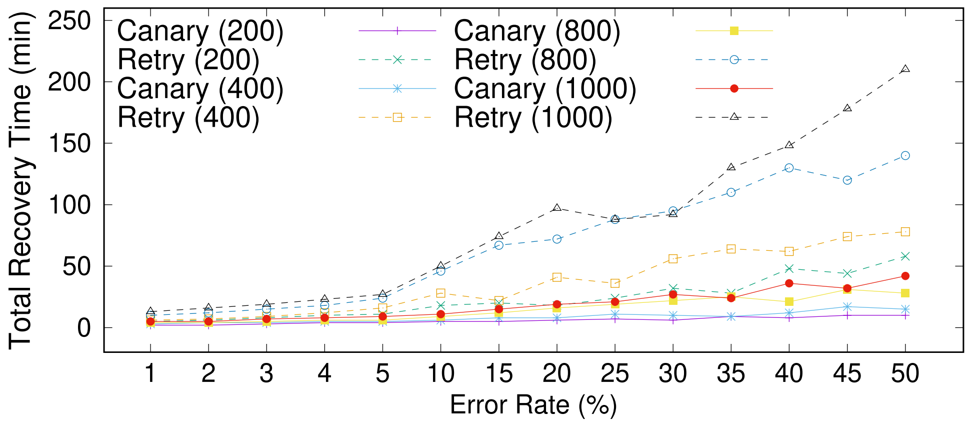


Figure 6.10: Impact of *Canary* on recovery time with a cluster size of 16 and increased function invocations.

Next, we study the scalability of the studied approaches by increasing the cluster size from 1 to 16 nodes. In this experiment, we use a failure rate of 15% and a fixed number of function invocations, i.e., 5000. Figure 6.11 shows the results. We observe that as the cluster size increases, the total execution time of batch jobs decreases for all three execution scenarios. The performance of *Canary* is close to the ideal case, with an average increase of 2.75% in the execution time, when increasing the number of nodes from 1 to 16. However, *Canary* reduces the overall execution time by up to 17% as compared to the retry-based approach. Overall, we observe the scalability of 1.2×, 1.18×, and 1.10× for the ideal, *Canary*, and the default retry-based approach, respectively, when increasing the number of nodes from 1 to 16.

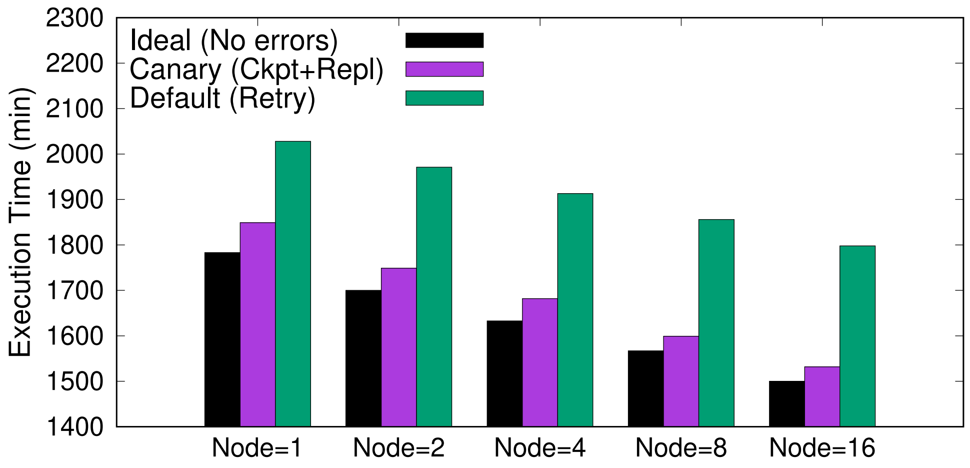


Figure 6.11: Impact of *Canary* on recovery time with 5000 function and a 15% failure rate.

6.4 Discussion

Existing FaaS platforms implement a retry-based recovery strategy for all failed functions which may not guarantee successful function execution. Moreover, using a retry-based approach re-executes the failed functions multiple times leading to significantly higher execution time and associated cost as compared to the proposed *Canary* framework. The performance of retry-based failure recovery is worse when functions fail frequently and towards the end of a function's execution. In this section, we discuss the implications of the replication and checkpointing techniques on the performance of *Canary* and analyze its potential benefits for both cloud and FaaS service providers.

Replicating Function Runtimes: The replication of execution runtime and critical application data significantly improves the function recovery time. Replication provides warm containers to resume the execution of the failed functions, however, replication performance depends on the number of replicas per job. Our analysis of dynamic, aggressive, and lenient replication strategies shows that *Canary* achieves better performance by determining the number of replicas based on the error rate. Consequently, *Canary* yields a trade-off between time and cost of execution. Lenient replication incurs less computation overhead for the same cost as compared to the dynamic replication approach when the failure rate is low. This performance gap reduces as the number of failed functions increases because a workload with the lenient function replication strategy spends more time on starting the new replicas of function runtimes. The aggressive replication strategy spends slightly higher time for a significantly higher cost as compared to dynamic replication. Therefore, dynamic replication with *Canary* performs better than the aggressive and lenient replication strategies.

Checkpointing Function States and Data: The retry-based fault tolerance strategy forces re-executing a function from its first instruction. The use of checkpoints to store function states and critical data significantly reduces the job execution time in case of function failures. The frequency of checkpoints adds overhead to FaaS computing, however, it addresses the challenges of unpredictable system failures, such as network failures. The ideal scenario is to checkpoint function states and data right before a failure or state completion, and restart the failed function using the latest checkpoint. However, it is challenging to accurately predict these events. *Canary* maintains up-to-date checkpoints after successful completion of function states to ensure quick recovery of failed functions.

Benefits of *Canary* for FaaS Platforms: Fault tolerance and resiliency are key features for measuring the quality of services provided by the cloud platforms. *Canary* integrates replication and checkpointing techniques to ensure a reduced execution time. The traditional retry-based approach employed by the existing FaaS platforms leads to a higher job execution time, which negatively impacts time-sensitive applications and may violate their SLAs. Moreover, longer function execution requires occupying the same resources for the same job for a longer period of time. *Canary* addresses these issues and alleviates the challenges of resource scheduling of incoming jobs by significantly reducing the impact of failures thus freeing up expensive data center resources.

Benefits of *Canary* for FaaS Users: FaaS offers an attractive computing model for reducing the cost of using cloud resources without negatively impacting application performance. However, the cost benefits of deploying application on FaaS platforms is undermined by unexpected failures. The default retry-based strategies used at large by cloud providers significantly increase FaaS costs. *Canary* alleviates the burden of extending the expected duration of a job and enables FaaS users to reduce the function completion time as compared to the retry-based approach. Specifically, *Canary* improves the reliability of time-sensitive applications by reducing their failure recovery time.

6.5 Summary

In summary, *Canary* provides fault-tolerance and resilience to stateful FaaS framework and extends the existing FaaS platforms by adding new software modules for storing function states, replicating function runtimes, and checkpointing critical data for faster failure recovery. *Canary* can tolerate large failures and reduces the recovery time and dollar cost by up to 83% and 12%, respectively over the default retry-based recovery strategy. Moreover, *Canary* provides improved application availability at the additional average execution time and cost overhead of 14% and 8%, respectively over the ideal scenario that does not incur any failure.

Chapter 7

Accelerating GPU-based Workloads Using Tiered Memory

7.1 System Design

To improve the performance of GPU-based HPC workloads on tiered memory systems, we envision a reference CXL-enabled multi-GPU system architecture as shown in Figure 7.1. We extend this architecture from the Nvidia DGX-A100 system, which consists of 8 GPUs distributed evenly across the two sockets. Using PCIe switches, a pair of GPUs share the available PCIe bandwidth to connect with the main and CXL memory. All PCIe links are composed of $\times16$ lanes each. GPUs are interconnected to each other using a hybrid mesh-cube topology using NVLinks and NVSwitches (which we omit from this figure for simplicity). Next, we mount a CXL memory on each socket using a dedicated PCIe link ($\times16$ lanes) to expand the capacity of the main memory. Although the processors in the DGX-A100 system can support up to 128 PCIe lanes, we map a limited number of lanes to each CXL device since most commercially available CXL expansion cards are based on PCIe $\times8$ configuration. Similarly, multiple CXL cards can be attached to the system PCIe interface. Therefore, when all GPUs are actively reading/writing data to/from the CXL memory of a single socket, the bandwidth of the CXL memory gets evenly distributed across all 8 GPUs, thereby creating contention on the PCIe interconnect of the CXL memory.

To alleviate memory contention, throughput bottlenecks and sub-optimal memory allocation we propose an algorithm that leverages heuristics from the job scheduler, system configuration, and statistics to generate efficient memory placement maps for main and CXL memory on multi-GPU

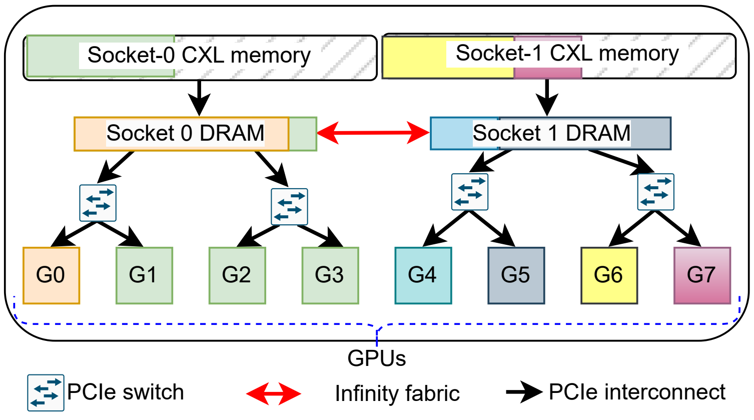


Figure 7.1: CXL-enabled multi-GPU system architecture.

systems. Our memory placement approach leverages tiered memory to identify optimal memory sources to maximize the data transfer rate and reduce the total execution time. Our proposed approach is shown in Algorithm 6. We consider a series of batch jobs J enqueued on the scheduler ready for execution. The job configuration enlists the number of GPUs required and the total memory footprint which is either known in advance or can be estimated using predictors [180]. Additionally, the system-level statistics, such as the amount of available memory per tier and data movement bandwidth, are provided to the scheduler using resource monitoring tools, micro-benchmarks, and node specifications.

Our proposed algorithm, listed in Algorithm 6 works as follows: select a list of jobs S for execution on the available GPU resources (Lines 2-5). Next, the scheduler determines the excess amount of memory required by each GPU, referred to as *spill* based on the scheduled jobs and available memory on the CXL and DRAM cache tiers (Line 8). The `calc_spill` function computes the fraction of DRAM memory requested by the GPU g of job j which exceeds the DRAM capacity when all the scheduled jobs on *socket*(g) are allocated fair proportions of the DRAM memory. Based on the *spill*, CXL memory available on that socket, and bandwidth of DRAM, PCIe, and CXL, respectively, the routine `calc_cxl` computes the amount of memory that can be efficiently allocated on the CXL device, such that none of the jobs scheduled on the peer-GPUs face DRAM starvation (Line 9-10). Once the efficient memory allocations are computed, they are mapped to the job j, and deducted from the available DRAM (D) and CXL (C) memory for the next set of jobs (Line 12-15). Finally, the algorithm outputs an efficient multi-tier memory allocation plan for the scheduled S jobs.

Algorithm 6: Our proposed memory allocation approach.

```
Input  : $N$: # sockets per node, $J$: list of jobs containing tuples $\langle j\_id, total\_mem, n\_gpus\rangle$, $G$: list of
         vacant GPUs IDs, $D$: List of DRAM memory available per socket, $C$: List of CXL memory
         available per socket, $BW_D$: DRAM bandwidth per socket, $BW_P$: PCIe bandwidth, $BW_C$:
         CXL bandwidth
Output: $S$: Amount of main and CXL memory to be allocated
begin
    $S \leftarrow [j \in J$ if $j[n\_gpus] < avail\_gpus]$
    for $j \in S$ do
        $j[gpus] \leftarrow allocate\_gpus(G, j[n\_gpus])$
        $j[mpg] \leftarrow j[total\_mem]/j[n\_gpus]$                    // req_mem/GPU
    for $j \in S$ do
        for $g \in j[gpus]$ do
            $spill \leftarrow calc\_spill(j, N, socket(g), D, C)$
            $cxl\_pull \leftarrow calc\_cxl(spill, C, BW_D, BW_P, BW_C)$
            $on\_cxl \leftarrow min(cxl\_pull, j[mpg])$
            $on\_dram \leftarrow min(j[mpg] - on\_cxl, D[socket(g)])$
            $j[`dram`][socket(g)] += on\_dram$
            $j[`cxl`][socket(g)] += on\_cxl$
            $D[socket(g)] -= on\_dram$
            $C[socket(g)] -= on\_cxl$
    return $S$
```

7.2 Performance Evaluation

7.2.1 Testbed Setup

We simulate a series of different testbed profiles using the aforementioned simulation. We vary the profiles of the testbed starting from the default configuration of the Nvidia DGX-A100 machine, with the exception of considering 64 GB memory available per socket instead of the default 512 GB. Multiple GPUs are connected to the host system using PCIe Gen 5.0 as per the topology shown in Figure 7.1. The idle memory access latency for local memory is approximately 71 ns and 136 ns for remote memory. Similarly, the loaded latency for such a system is 228 ns. Similarly, the maximum attainable memory bandwidth is 243 GB/s for read-only traffic.

7.2.2 Compared Approaches

We compare the following approaches for pinned memory allocation on CXL-enabled multi-GPU devices:

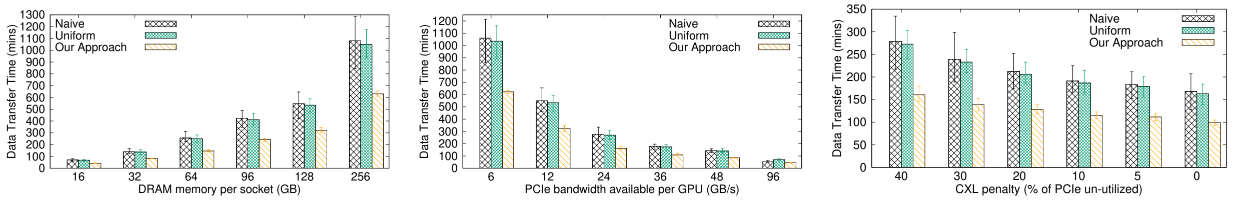
- **Naive**: This is the default approach adopted for memory allocation where the system starts allocating memory from the main memory followed by the CXL memory tier. In this approach, jobs that get scheduled first end up consuming all the available main memory, forcing the later jobs to allocate memory from the CXL device.
- **Uniform**: In this approach, the scheduler attempts to uniformly distribute the available main memory across all GPUs. This approach ensures that all jobs get an equal portion of the main memory.
- **Our Approach**: This approach is detailed in § 7.1.

7.3 Performance Results

We evaluate the performance of various compared approaches by measuring the total amount of time taken by the job to perform data transfer across the main and CXL memory allocations. In our evaluations, GPUs access data concurrently to the host memory tiers, as observed in GPU-bound HPC and DL applications. We measure the data transfer time for an increasing amount of main memory available per socket, varying PCIe bandwidth available (the GPU PCIe switches and CXL connected through $\times 8$ lanes), and varying degrees of CXL penalty.

7.3.1 Increasing Available Main Memory per Socket

Our first set of experiments evaluates the data transfer times for an increasing main memory capacity. As observed in Figure 7.2a, our approach yields faster data transfer times with increasing capacity. This is because with increased main memory capacity our approach can perform better memory placement and load distribution across both main and CXL memory. For varying job profiles, our approach demonstrates a reduction in data transfer overheads from 15.4% to 61.2% as compared to the naive memory allocation approach.



(a) Increasing memory capacity (b) Variable PCIe bandwidth (c) Varying degrees of CXL penalty

Figure 7.2: Data transfer time for varying memory, PCIe bandwidth per GPU, and CXL penalties.

7.3.2 Varying PCIe Bandwidth

Our next set of experiments measures data transfer overheads of varying available PCIe bandwidth for both GPUs and the CXL memory. This experiment studies the impact of various PCIe generations (starting from PCIe 3.0). As shown in Figure 7.2b, our approach performs 65.35% and 21.3% better on average compared to naive and uniform allocation-based allocation approaches, respectively. The bandwidth reported on the x-axis is the actual share of PCIe bandwidth available to each GPU when two GPUs share a single PCIe bus using the PCIe switch. In real-world testbeds, we achieve only ∼75% of the theoretical transfer throughput from the GPU to the host memory. We use this to estimate the PCIe bandwidth of the next-generation PCIe protocols.

7.3.3 Varying Degrees of CXL Penalty

As specified in the CXL 3.0 specification, the CXL protocol is capable of achieving only 60%-90% of actual PCIe bandwidth, which we refer to as the CXL penalty. Therefore, in our last set of experiments, we evaluate data transfer times for the compared memory allocation approaches for different degrees of CXL penalties. As observed in Figure 7.2c, our approach demonstrates 17.7% to 67% lower data transfer overheads over the naive and uniform memory allocation policies.

7.4 Summary

To summarize, CXL offers promising benefits in terms of main memory expansion, increased data transfer throughput, and low latency, however, the limited PCIe bandwidth connecting these CXL devices can become a bottleneck when the memory allocation on multi-GPU systems is done using the default schedulers. To address this challenge, we propose a reference architecture for enabling CXL on the Nvidia DGX A100 system and propose an efficient memory allocation approach that leverages job schedules and additional memory tiers to mitigate contention at the CXL memory tier and maximize the performance of HPC workloads. Our evaluations show up to 65% lower data transfer overheads compared to the default memory allocation approach.

Chapter 8

Intelligent Memory Management for HPC Workflows

8.1 System Design

To improve the performance of containerized HPC workflows, we propose memory management policies and runtime that minimize the execution time of HPC workflows by mitigating the impact of inefficient memory allocations, replacement, and movement policies of existing tiered memory approaches. Such policies are designed for heterogeneous memory systems that include at least two memory tiers including the DRAM, PMem, and CXL memory supported by NVMe, SSD-based storage, and similar technologies in memory and storage subsystems.

The intelligent memory management policies fully utilize distributed heterogeneous memory subsystems to improve the overall memory utilization and reduce workflow failures due to limited memory [141, 179], thus improving the overall system throughput. They mitigate the impacts of using tiered memory on workflow performance by using intelligent page allocation and replacement policies that leverage the access latencies of different memory tiers, the interconnection bandwidth, and local memory availability. Our proposed runtime manages the allocation and movement of additional memory requests from HPC workflows and transparently moves memory pages between memory tiers to maximize the overall system performance.

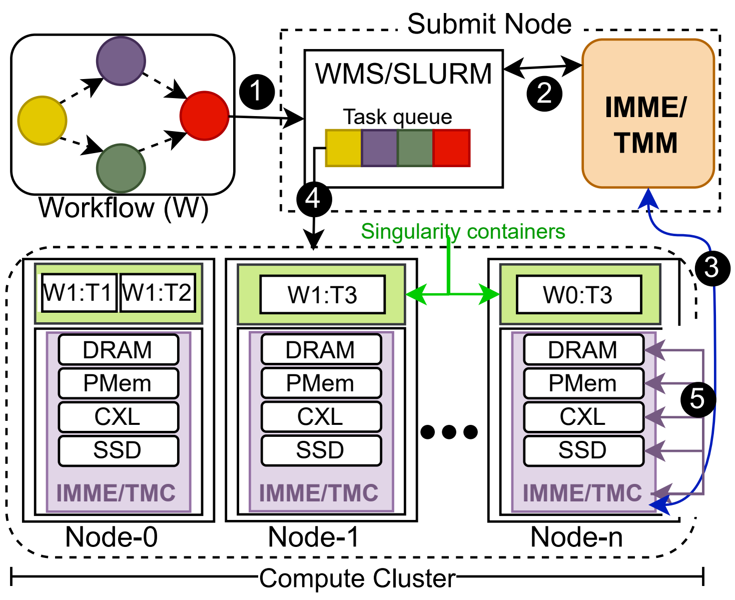


Figure 8.1: High-level system architecture with IMME leveraging tiered memory for containerized HPC workflows.

The high-level architecture of our proposed runtime is shown in Figure 8.1. The workflow is first submitted to the WMS where it is converted to an executable workflow represented by a DAG. Our proposed runtime ensures that the HPC workflows optimally leverage additional memory from the memory tiers and enable workflow-aware memory allocation to jobs. Workflow containers can request memory from specific memory tiers which can be different from the initial memory allocation. Our allocation policy serves such memory requests by efficiently allocating memory pages from the requested memory tier. It identifies the best memory tier based on the workflow characteristics, i.e., latency sensitivity, bandwidth and capacity intensive, and execution makespan, and allocates either the entire block from a single tier or from multiple memory tiers including the local and CXL memory. Our target capacity-intensive jobs, such as training DL models [39, 146] and large-scale simulations [118], require large memory capacity for continued execution and are independent of their latency and bandwidth requirements. If enough local memory is not available, then our page replacement policy and proactive swapping mechanism move existing memory pages to the appropriate lower memory tiers to provide large contiguous memory space for workflows.

The *Tiered Memory Manager* is the main component of our runtime, that handles coordination between components of our proposed runtime by using a manager and a client deployed on the cluster nodes. The main responsibilities of *Tiered Memory Manager* are: 1) identify various memory types; 2) categorize memory into tiers; 3) create staging buffers on each tier; 4) dynamically adjust buffers based on utilization; and 5) track the hotness/coldness of workflow pages. The *Tiered*

Memory Manager identifies various memory types available on the HPC systems and classifies them into tiers with the primary tier being the DRAM memory. The classification of memory into tiers depends on the available memory capacity, access latency, maximum attainable bandwidth, and the interconnect type. It also creates staging buffers on each tier based on the fair-share approach, tier characteristics, and available memory. These buffers are dynamically adjusted based on the memory utilization on each tier and the workflow requirements. Moreover, staging buffers required for transparent data movement across memory tiers are created for each compute node. Lastly, *Tiered Memory Manager* also tracks the hotness of each page of the workflows. The heatmaps are used to identify frequently accessed pages and least frequently accessed pages for efficient page movement between the memory tiers.

The memory allocation, deallocation, and management are done transparently by the runtime based on the workflow requirements and the memory access patterns of the given application. The *Tiered Memory Manager* exposes APIs that can be used by workflows to request tiered memory for expansion, staging input data, or storing intermediate and output data beyond the initial memory allocation. These APIs are used to allocate and deallocate memory from a specific tier by setting the appropriate flag and for creating shared memory regions between workflows. For example, HPC workflows can use the APIs to request memory from the PMem tier to store data structures that need to be retained. Similarly, for frequently accessed data memory from the CXL tier can be requested to store the prefetched data for caching purposes. The APIs are designed for seamless integration, allowing them to be incorporated into the existing workflow code with minimal modifications.

Once a request is received, the *Tiered Memory Manager* services the request by identifying the ideal memory tier and returning the address space. The requested memory size is in bytes and the flag accepts a combination of the following values: LAT, BW, CAP, SHL which represent latency-sensitive, bandwidth-intensive, capacity, and short-lived, respectively. The LAT flag represents memory that is extremely sensitive to access latency and the page placement necessitates the use of the fastest memory tier. Similarly, the BW flag represents a memory access pattern that requires the highest access bandwidth from either a single or multiple memory tiers. The CAP flag represents memory that is not susceptible to access latency or bandwidth and is primarily use to store pages that are not actively accessed. Lastly, the SHL flag represents memory that is shared between multiple workflows. The flags passed through these APIs allow the user to pass hints regarding the memory resource requirement of workflows. However, these flags are purely advisory and are not mandatory for successful execution. If no flags are provided, then the *Tiered Memory Manager* assigns either single or multiple flags to each workflow based on the previous

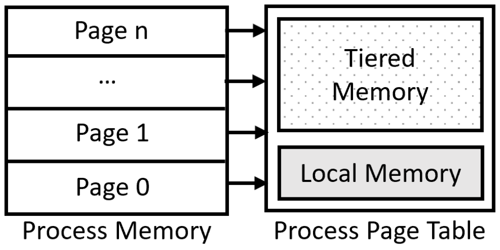


Figure 8.2: Tiered memory layout for HPC workflows.

execution logs, heuristics, and predictor [42]. The *Tiered Memory Manager* also monitors page access patterns and uses this monitoring data for efficient memory allocation and moving data across memory tiers.

Figure 8.2 shows the layout of the tiered memory system, which consists of CXL-based memory and PMem resources and can span across a cluster of servers. *Tiered Memory Manager* handles all memory access from each workflow to the tiered memory and keeps track of the memory allocations to workflows. It also monitors the memory allocations on each server and dynamically adjusts the memory allocation based on the current memory utilization on each server.

Page Allocation Policy

By default, memory is allocated for workflows from the local system memory to maximize performance and reduce the total execution time. However, memory pages are excessively swapped to the slower tiers, e.g., swap space, when the system memory runs out which degrades the performance of running jobs [50, 163]. Our page allocation policy maximizes job performance and reduces the impact of swapping to slower tiers by efficiently utilizing tiered memory and by considering workflow characteristics and the execution sequence. Similarly, it also handles the allocation of additional memory from the tiered memory once the DRAM memory runs out of available space. The policy ensures that the additional memory allocated from the tiered memory has minimal overhead and takes into account the latency requirements of HPC workflows.

Our proposed page allocation policy is shown in Algorithm 7. It takes workflow attributes as input, which includes a unique workflow identifier (w_id), the size of the requested memory (s), and an optional list of flags regarding the memory characteristics of the workflow (f). We predict the amount of memory required for each flag using previous execution logs, heuristics, and existing memory predictors [42, 157]. Specifically, the heuristics generate page temperatures by analyzing the page access frequency on each memory tier. For instance, if a job allocates 40 GB of memory

Algorithm 7: Page Allocation Policy for HPC workflows with Tiered Memory.

Input : w_id: unique workflow identifier, s: requested mem size, f: list of flags to denote mem characteristics (LAT: lat-sensitive, BW: bw-intensive, CAP: cap-intensive, SHL: short-lived), $alloc\_map$: global mem alloc maps for all workflows, ev: global map of evictable mem available on each tier-($local$, $pmem$, and cxl)

Output: A: memory allocation plan containing a map of memory allocation required from each memory resource

```
1  Function TierAlloc($w\_id, s, \langle f \rangle$):
2      if $f == NULL$ then
3          $f = predict\_flags(w\_id, s)$
4      if $type(f) == list$ then
5          $f_{first} = f.pop()$
6          $size_{first} = predict\_flag\_mem\_size(f_{first}, w\_id)$
7          TierAlloc($w\_id, size_{first}, f_{first}$)
8          TierAlloc($w\_id, s - size_{first}, f$)
9      $A \leftarrow \langle local : 0, pmem : 0, cxl : 0 \rangle$
10     if $alloc\_map.find(w\_id)$ then
11         $A \leftarrow alloc\_map[w\_id]$                         // Find prev.  alloc
12     $m = A[local] + A[pmem] + A[cxl]$                            // Alloc'd memory
13     while $m < s$ do
           // Prioritize local memory for lat-sensitive and short-lived tasks
14         if $f == LAT$ or $f == SHL$ then
15             if $ev[local] > 0$ then
16                 $A[local] += \min(s - m, ev[local])$
17             else if $ev(pmem) > 0$ then
18                 $A[pmem] += \min(s - m, ev[pmem])$
19             else if $m < s$ then
20                 $A[cxl] += s - m$                                // Unlimited CXL mem
           // Tiered memory allocation for high-bw
21         else if $f == BW$ then
22             $r \leftarrow 0$                                     // Remainder for the next tier
23             for $tier \in [local, pmem, cxl]$ do
24                 $frac[tier] = r + s \times (BW[tier]/BW[total])$
25                 $curr\_max = \min(frac[tier], ev[tier])$
26                 $A[tier] += curr\_max$
27                 $r = curr\_max - frac[tier]$
           // Addn.  memory capacity through CXL
28         else if $f == CAP$ then
29             $A[cxl] += s - m$
30         $m = A[local] + A[pmem] + A[cxl]$
31     $alloc\_map \leftarrow alloc\_map \cup A$
32     $update\_evictable(A)$
33     return $A$
```

and only 512 MB of pages are accessed 80% of the time during the first 20 seconds of execution, then 512 MB of memory is determined to be latency-sensitive (LAT) while the remaining memory is classified as capacity-sensitive (CAP) for the first 20 seconds of execution. To look up execution logs, we utilize workflow configuration information, parameters, flags, etc. For cases where logs are not available or the exact match is not found, we utilize the nearest match as hints for the predictor.

Once the flags are recursively decomposed in atomic values with their corresponding sizes, the current memory allocation of the function on each memory tier (Lines 9-12) is fetched. The total size required is updated for the given function based on previous allocations (Line 13). Next, we iteratively allocate suitable memory pages from each memory tier based on the function requirements (Lines 14-33). For latency-sensitive (LAT) and short-lived (SHL) workflows, the policy attempts greedy allocation of memory starting from the fastest to the slowest tier in a cascading fashion (Lines 15-21). This approach mitigates the challenge of higher access latency for such workflows. Part of the memory belonging to the latency-sensitive and short-lived workflows is pinned to guarantee the required performance and the remaining portion is tagged as a pageable region that can be used for swapping and replacement as shown in Figure 8.3. For simplicity, our policy assumes that an unlimited memory is available over the CXL interconnect and the remaining memory can be directly allocated from CXL. Although such greedy-based decomposition leads to suboptimal initial allocations for workflows that are launched at a later time by forcing them to allocate memory from high-latency slower memory tiers (e.g., CXL), our page replacement algorithm (discussed in Section 9) effectively mitigates this overhead. For latency-sensitive workflows, our runtime pre-faults [79] the memory addresses to reduce the overhead of page faults during memory access.

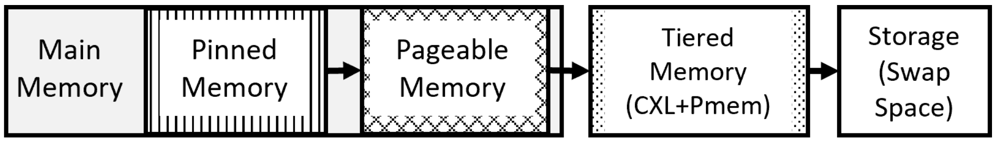


Figure 8.3: Memory allocation and page movement for workflows.

For bandwidth-intensive workflows (BW), we use a multi-path memory access approach that allocates memory on each available tier (*local*, *pmem*, *cxl*) to provide maximum available bandwidth to the workflow. The memory allocated on each tier is directly proportional to the available read/write throughput observed from that tier. For cases where faster memory tiers experience higher contention levels, and the required memory is not available (Line 26), only partial memory is allocated (Line 27), and the remainder of memory from the next fastest memory tier (Line 28).

Algorithm 8: Page Replacement Policy to manage hot/cold pages across multiple memory tiers.

```
Input  : r: Number of pages to replace from memory
Output: None
1 begin
2     $t \leftarrow 0$                                        // Number of replaced pages
3     while $t < r$ do
4         $victim\_pages \leftarrow lru\_pagable(r - t)$
5         $victim\_pages \leftarrow remove\_lat\_or\_shl(victim\_pages)$
6         $t += victim\_pages$
7         $move\_out(victim\_pages)$
8         $update\_pg\_table(victim\_pages)$
9     $update\_alloc\_map(r)$
```

Finally, for capacity-intensive (CAP) workflows, the entire memory is allocated directly from the CXL memory tier. Finally, based on the amount of memory allocated on each tier, the corresponding allocation entry in the global allocation and eviction maps are updated (Lines 34-35) and the memory allocation plan A is returned.

We note that the algorithmic complexity of the proposed page allocation policy is a linear function of the number of memory tiers. However, since we consider the case of only three memory tiers, the complexity becomes constant $\mathcal{O}(1)$. Such low complexity is particularly important for time-sensitive HPC workflows.

Page Replacement Policy

Many page replacement techniques have been extensively studied for conventional memory subsystems [61, 91, 211, 235] to create space in DRAM for pages that have been swapped out to slower storage tiers. The default behavior of the Linux kernel is to select a set of candidate pages based on various heuristics [61], such as least-recently-used, most-recently-used, and optimal page replacement, that can be evicted to a disk-based swap partition to be replaced with the requested page. However, this approach is agnostic to the underlying heterogeneous memory tiers and results in suboptimal page replacements to slower disk-based storage tiers, leading to resource underutilization, performance degradation due to major page faults, and low system throughput.

To address the above challenges, we propose a page replacement policy, shown in Algorithm 8, to mitigate the impact of suboptimal page faults to accommodate bandwidth-intensive and time-critical HPC workflows. We adopt a dynamic memory eviction model based on the characteristics of the function such as latency-sensitivity or short-lived function. Our page replacement does not depend on the input flags or predictor output, however, these flags enable fine-tuned page replacement for specific workflow types. Note that the predictor is only used for estimating initial allocation using previous execution logs or heuristics in the absence of flags. The replacement policy also considers page temperatures and memory access patterns for all colocated workflows to identify and prioritize the eviction of cold pages. The algorithm takes the number of pages to be replaced (r) as input based on the system-level page faults and filters out the memory pages belonging to the above class of applications (Lines 4-5) based on the victim pages identified by the Linux kernel. The filtered pages are tracked and moved to the lower memory tier rather than swapped out to the underlying disk-based swap space (Line 7). Once the victim pages are identified, they are swapped to the swap space and replaced with the requested page by the application. Finally, the allocation map is updated with the replaced pages (Line 8). Our page replacement policy ensures that the memory pages belonging to the latency-sensitive and short-lived workflows are not blindly swapped out by the Linux kernel resulting in major page faults that eventually degrade application performance.

Intelligent Page Movement Policy

To improve application performance and reduce the latency of accessing memory pages, we propose an intelligent page movement policy that proactively moves memory pages between various memory tiers and implements a proactive page-swapping mechanism that swaps out memory pages to the CXL memory. To mitigate the negative impacts of proactive swapping, the swapped-out memory pages are cached in the page cache if there is enough memory available on the main memory and are marked as dispensable and the corresponding page table entry is updated. If enough system memory is not available, then the memory pages are simply moved to the CXL memory tier. Once the system memory runs out, instead of swapping pages to the swap space, the pages in the page cache are first swapped out and then the workflow memory pages are swapped. The page movement from the main memory is based on workflow characteristics, e.g., latency-sensitivity, to the CXL memory and then eventually to the local disk. The proactive page swapping also performs memory compaction to reduce fragmentation and enable contiguous memory blocks to be allocated to workflows for colocating more workflows on the system, thus improving system utilization.

The proposed page movement policy also moves pages between persistent and CXL-attached memory tier based on the available page access heatmaps. This enables the runtime to effectively move pages to faster memory tiers that were previously identified as cold but later categorized as hot pages. Our application-aware intelligent page movement policy prioritizes application pages that do not belong to latency-sensitive or short-lived applications. If a page belonging to the above classes of applications must be moved, then the policy prioritizes pages belonging to the pageable memory region as defined in the page allocation map. Our intelligent page movement policy minimizes the impact of page swapping by enabling the swapped pages to be available in the fastest available memory tier. Finally, our page movement policy reduces the number of major page faults and subsequently increases the number of minor page faults as the page is accessible on other memory tiers or the page cache.

Management of Shared Memory Across Workflows

CXL memory provides a fast backend to improve the performance of shared memory regions for HPC workflows. Input or read-only data shared between workflows can be staged in the CXL memory, which can be leveraged by the HPC job scheduler e.g. SLURM, to launch workflows at scale and minimize the scale-up time and data transfers between workflows. For example, launching thousands of HPC workflows using a custom Singularity container image requires the image to be moved to all the servers that will run the job workflows. This creates a network and I/O bottleneck when a large number of workflows access the same data resulting in an increased execution time to prepare the runtime and increase the cold-start latency for containers. For simplicity, we assume that the workflow manages the shared memory and handles locking mechanisms as offered by several libraries [53,76] to block read or write operations during an ongoing write to the shared memory region. We provide three strategies for efficiently managing shared memory between workflows at the workflow and platform levels. First, shared memory pages are made locality-aware by incorporating the location of workflows accessing the shared memory by the HPC job scheduler. Such memory pages are hosted on the CXL memory accessible to both workflows, and the memory pages are cached in the local buffers for fast access on each server. Second, to improve the capability of the HPC job scheduler to scale up workflows and reduce the cold start latency, we leverage the CXL memory to host container images and application data. Third, our proposed runtime keeps track of the memory tagged as shared memory and ensures that during a scale-down event, the shared memory is not deallocated. The shared memory is freed when all references in the corresponding page tables have been removed. These approaches ensure that the shared memory is effectively allocated, managed, and utilized for large-scale containerized HPC workflow deployments.

8.2 Performance Evaluation

In this section, we present the evaluation of the proposed memory management policies for HPC workflows using tiered memory. We explain our prototype implementation, evaluation methodology, testbed, workflows, and performance metrics that we use to analyze and compare our proposed runtime with baseline and other alternative execution approaches.

8.2.1 Evaluation Methodology

We compare our runtime with the baseline scenario where HPC workflows are colocated and frequently run out of memory resulting in swapping out of memory pages. We also compare its performance with a more realistic scenario where workflows memory is allocated from CXL memory without considering the workflow performance characteristics. In our evaluation, we study the following metrics to demonstrate the effectiveness of our proposed approach: total workflow execution time, number of page faults, total execution makespan of HPC workflows submitted as batch jobs, and workflow and cluster scalability. The total execution time is the time required to complete the scheduled workflows and return the results. The bandwidth and latency numbers are reported for the CXL memory allocated to the workflows and compared to the local memory and swap space. Lastly, we use the number of memory accesses, the amount of data swapped to disk, and CXL memory to gauge the performance of the memory management policies. To evaluate workflows that have varying memory access patterns we randomly select workflows and substitute them with versions that request additional memory during execution using our APIs and incorporating specific flags. This approach ensures that the experimentation environment remains dynamic, facilitating the exploration of various memory access patterns that may evolve during execution. We run each experiment 10 times and report the average. Overall, we observe a negligible variance, i.e., less than 5% between different executions of the same experiment in our evaluation.

8.2.2 Evaluation Setup

Testbed Setup

Our evaluation setup consists of a cluster of 8 bare-metal servers connected using 10G Ethernet. Each server has two Intel Xeon Gold 6126/6240R/6242 processors, contains 512 GB of main memory, 1 TB of Intel Optane DC persistent memory, and runs Ubuntu 22.04 LTS server operating system. We deploy SLURM along with Singularity on all servers in our evaluation setup. We provision the tiered memory using the local DRAM, persistent, and CXL memory available on the servers via the CXL interconnection. The CXL memory is emulated [31, 226, 238] using the remote NUMA socket as advocated by POND [156] and CXLMemSim [238]. In our testbed, we observe the local and remote NUMA latencies to be ∼80 ns and ∼140 ns, respectively, which represent the approximate latency of a CXL-attached memory [39, 156].

8.2.3 Evalation Workflows

Modern HPC workflows [67, 98, 165, 189] typically consist of core scientific computing (SC) simulations [188], surrogate deep-learning (DL) tasks that assist the core simulation [37, 59, 240], data compression/decompression (DC) [46,148,192] for collective communications and storage, and data mining (DM) [95,150,233] required by analytics engines to steer the experimental trajectory in real-time. In our evaluations, we consider HPC workflows composed of these where each workflow represents jobs with unique characteristics, i.e., computing (requires powerful CPUs), data (processes large volumes of data), bandwidth-intensive (requires large bandwidth), latency-sensitive (requires fast access), and short-lived. DL is a data and bandwidth-intensive workflow in which we train the popular NLP model, i.e., Bert [74], over the IMDB dataset [14] for a total of 5 epochs. The DM workflow is a latency-sensitive workflow running a task on Spark that performs ETL [83] over the US census data [20] and computes the diversity index. The DC workflow is a compute and data-intensive workflow in which we run Zip [90] compression on a set of 50 GB input files. The SC workflow runs BFS using igraph [128] on a binary tree.

8.2.4 Execution Environments

To study the impact of our memory management policies, we define four realistic execution environments for running HPC workflows based on the availability of memory and storage subsystems. These execution environments are:

1. ***Ideal Environment*** **(IE)** represents an ideal baseline environment with enough local memory.
2. ***Constrained Baseline Environment*** **(CBE)** represents a more realistic environment with limited system memory and memory pages are frequently swapped out.

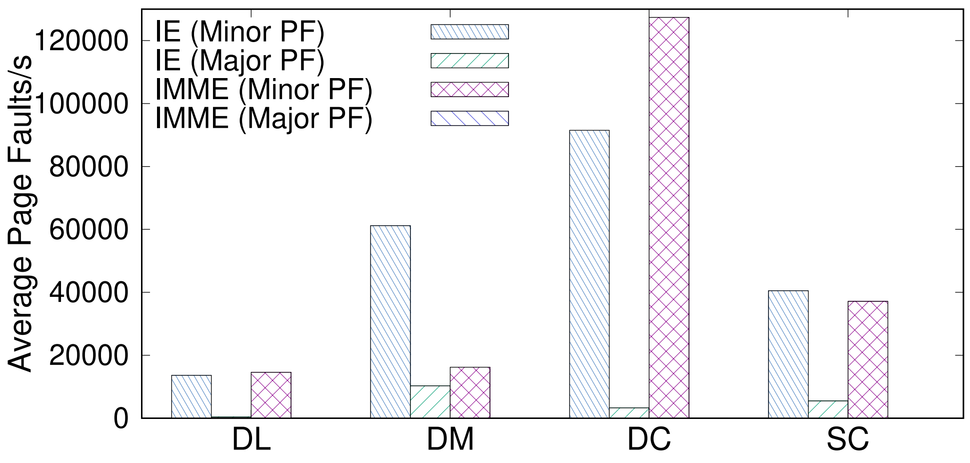


Figure 8.4: Impact of our runtime on the studied execution environments.

3. ***Tiered Memory Environment*** **(TME)** is based on the *Constrained Baseline Environment* but uses tiered memory for memory allocation with default Linux page promotion and demotion based on page temperatures.

4. ***Intelligent Memory Management Environment*** **(IMME)** is based on the *Tiered Memory Environment* and uses our intelligent memory management policies.

8.3 Performance Results

In this section, we present the performance results of our proposed approaches by executing the workflows on the studied execution environments and comparing their performance.

8.3.1 Impact of Tiered Memory on Total Execution Time of HPC Workflows

We study the impact of allocating tiered memory to HPC workflows and report the total execution time for the studied execution environments. The results are shown in Figure 8.4. We observe that the *Ideal Environment* takes the least execution time for all studied workflows because sufficient system memory is available to host the entire footprint of HPC workflows in memory. We observe degraded performance for the *Constrained Baseline Environment* as compared to *Ideal Environment* due to the limited system memory availability and frequent swapping of workflow memory pages to slower tiers. Similarly, the performance of latency-sensitive and short-lived, i.e., the DM workflows, drops significantly due excessive swapping and contention. However, the availability of tiered

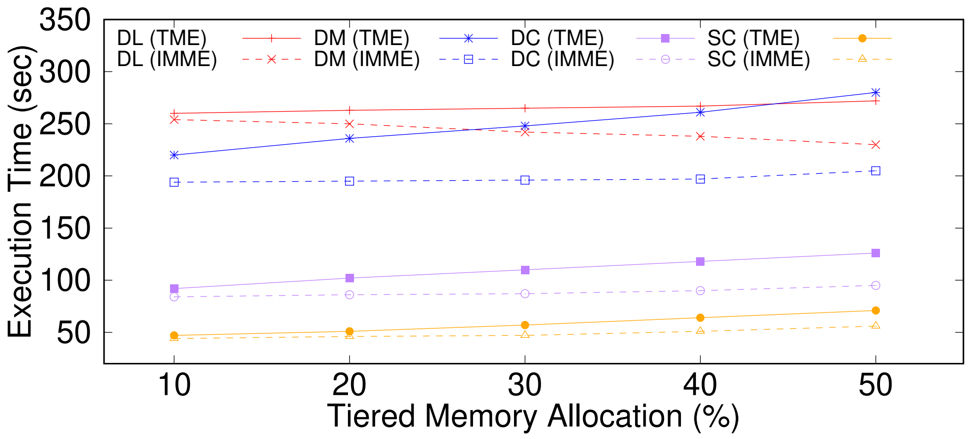


Figure 8.5: Impact of *IMME* on the workflow performance with varying tiered memory availability.

memory in the *Tiered Memory Environment* reduces this impact by providing a faster alternative and performs better than the *Constrained Baseline Environment*. Similarly, for *Intelligent Memory Management Environment*, we observe that our runtime utilizes tiered memory to improve the performance of workflows by allocating memory to appropriate workflows, intelligently moving pages between memory tiers, and proactive swapping memory pages to the CXL memory tier. Overall, we observe that the *Intelligent Memory Management Environment* reduces the execution time of studied workflows by up to 7%, 87%, and 25% as compared to the *Ideal Environment*, *Constrained Baseline Environment*, and *Tiered Memory Environment*, respectively.

We also study the impact of varying tiered memory allocations on the execution time. Figure 8.5 shows the results. Here, we vary the tiered memory allocation from 10% to 50%, where each data point represents the percentage of workflow memory allocated from the CXL memory tier. In the *Tiered Memory Environment*, we observe that as we increase the allocation of CXL memory to the workflows, the execution time increases due to the additional latency associated with accessing the CXL memory. We also observe that the *Tiered Memory Environment* does not manage tiered memory efficiently and causes bandwidth-intensive workflows to not fully utilize the additional available bandwidth, and latency-sensitive workflows to experience additional latency over the CXL interconnect. Since our proposed runtime allocates tiered memory based on workflow requirements and characteristics, we observe a reduced execution time for the studied workflows. Moreover, workflows that require additional memory continue to execute by expanding their memory footprint on the tiered memory which would otherwise crash due to limited local memory or fixed memory allocations. Overall, we observe that our memory management policies improve workflow performance by up to 80% as compared to the *Tiered Memory Environment* by efficiently allocating and managing memory tiers based on workflow characteristics and requirements.

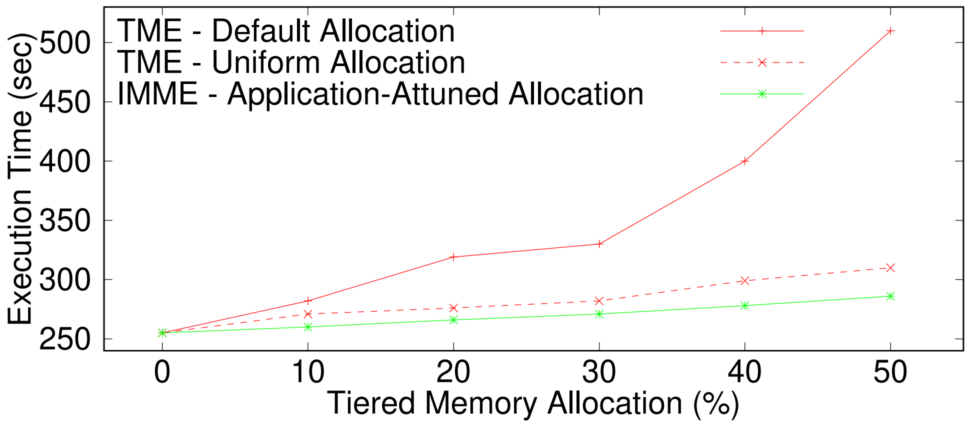


Figure 8.6: Impact of our memory allocation policy on execution time.

8.3.2 Impact of Page Allocation Policy on Workflow Performance

We study the impact of our page allocation policy on workflow performance by launching multiple instances of the studied workflows on the HPC cluster. To evaluate the effectiveness of our allocation policy, we report the total execution time of each workflow in Figure 8.6. We compare our page allocation policy with two approaches: 1) the *Default Allocation* policy where the system memory and CXL memory are allocated to workflows regardless of its requirements; 2) the *Uniform Allocation* policy allocates CXL memory to all workflows in a uniform fashion regardless of the workflow requirements. We observe that the *Default Allocation* policy allocates CXL memory to workflows based on its demand without catering to the class it belongs to and results in degraded performance for latency-sensitive and short-lived workflows. This approach is beneficial for latency-sensitive workflows and capacity-intensive workflows, but the performance of latency-sensitive workflows degrades as soon as the memory footprint overflows to tiered memory. The *Uniform Allocation* policy results in the worst performance for latency-sensitive workflows as they experience additional access latency of the tiered memory due to interleaving. However, interleaving results in improved performance for bandwidth-intensive workflows due to the availability of additional bandwidth. Overall, the *Uniform Allocation* outperforms the *Default Allocation*, however, the memory allocation is not aware of the workflow characteristics. The performance of *Uniform Allocation* can be further improved with weighted interleaving, however, setting weights does not consider the characteristic for all workflow types. We also observe that our memory allocation policy reduces the total workflow execution time by intelligently allocating CXL memory to workflows to minimize the impact of additional access latency. Overall, we observe that our allocation policy reduces the execution time by 44% and 8% on average as compared to the *Default Allocation* and *Uniform Allocation* strategies, respectively.

We also study the impact of our memory allocation policy on each class of workflow by varying the percentage of available DRAM to each workflow as a function of its working set size (WSS). The results are shown in Figure 8.7. We observe that as the amount of DRAM available to latency-sensitive workflows decreases, the memory access time increases resulting in a significant impact on makespan and performance. Similarly, for bandwidth-intensive workflows, we observe that our memory allocation policy leverages the available CXL memory to improve the overall throughput by leveraging the additional memory tiers.

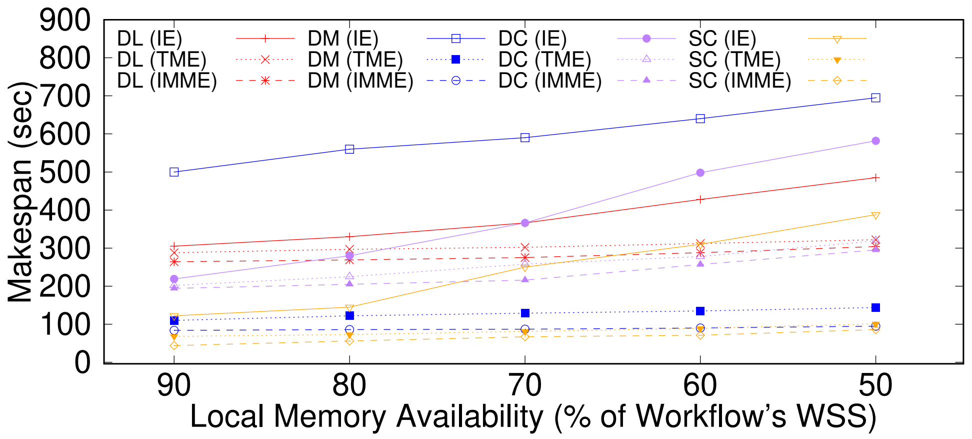


Figure 8.7: Impact of our memory allocation policy on the execution makespan of the studied workflows.

For *Tiered Memory Environment*, we observe that as the memory available to workflows decreases, the hot pages are promoted to DRAM reducing the impact of additional latency of CXL memory. Moreover, the speedup is achieved as the additional memory availability reduces the impact of swapping to slower storage for the *Ideal Environment*. Moreover, workflows that require large memory capacity to successfully execute, benefit from potentially unlimited memory availability from the CXL memory. Overall, we observe that our memory allocation policy reduces the overall makespan by 25%, 85%, 35%, and 71% on average compared to *Ideal Environment* for deep learning, data mining, data compression, and scientific workflows, respectively. Similarly, we observe that our memory allocation policy reduces the overall makespan by 8%, 31%, 9%, and 22% on average compared to *Tiered Memory Environment* for deep learning, data mining, data compression, and scientific workflows, respectively.

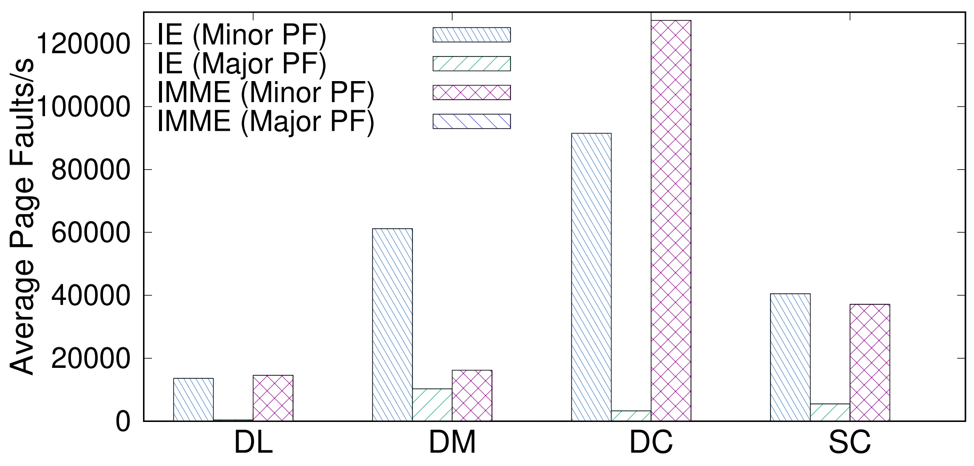


Figure 8.8: Impact of our page movement policy on workflow page faults.

8.3.3 Impact of Page Movement Policy on Workflow Performance

We study the impact of our intelligent page movement policy by observing the page fault statistics for the studied workflows. The results are shown in Figure 8.8. We observe that in the *Ideal Environment*, the Linux kernel swaps out memory pages based on the least recently used (LRU) policy regardless of the workflow requirements or characteristics. This causes a performance drop in latency-sensitive workflows which are most susceptible to additional latency when pages are swapped back in by the Linux kernel. We observe that with the availability of CXL memory, our page movement policy reduces the number of pages that are swapped to the disk by reducing the major page faults, thereby, improving workflow performance. However, workflows that are extremely sensitive to latency suffer additional latency when reading and writing from CXL memory. Our intelligent page movement policy reduces the number of major page faults by moving pages to the CXL memory which in turn increases minor page faults for each workflow. Furthermore, Linux swapping increases workflow execution time even with CXL memory. We observe that our intelligent page movement policy performs workflow-attuned page movement and ensures that the memory pages are available in the fastest tier and pages of latency-sensitive and short-lived workflows are protected from swapping. Our intelligent memory movement also performs proactive swapping in the background in addition to moving memory pages between various memory tiers. Our proactive swapping moves out workflow memory pages that are less sensitive to the overhead of moving pages back into the memory. This enables keeping more pages of latency-sensitive and short-lived workflows in the memory. Overall, we observe that our workflow-attuned page movement and proactive page-swapping improve workflow performance by 46% as compared to the default swapping policy.

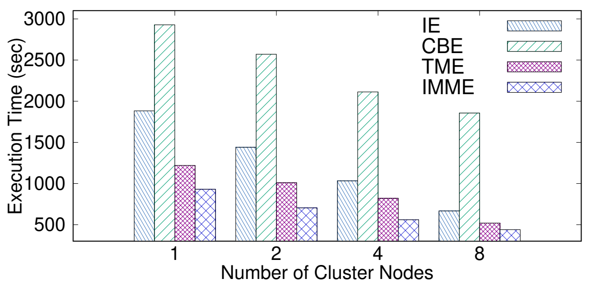


Figure 8.9: Impact of our runtime on execution time of 3000 workflows on an 8-node cluster.

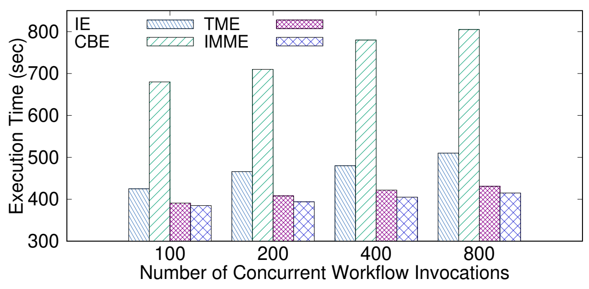


Figure 8.10: Impact of our runtime on execution time on an 8-node cluster.

8.3.4 Scalability Analysis of our Proposed Runtime on Workflow Performance

We increase the size of the HPC cluster and concurrent workflows to study the impact of our proposed runtime on a large HPC cluster. We launch 2000 instances of the studied workflows (150 for DL, 1100 for DM, 150 for DC, 600 for SC workflows) concurrently and observe the workflow execution time. Figure 8.9 shows the results of this experiment. We observe that the execution time is significantly reduced with the increasing number of cluster nodes thanks to the overall memory allocation and page movement on each server leveraging the CXL memory effectively. With the *Constrained Baseline Environment*, the execution time is the highest due to the limited resource availability and the contention at each node of the cluster. As memory utilization of the system increases due to colocation, the *Tiered Memory Environment* efficiently utilizes the tiered memory to promote hot pages to faster tiers improving the overall workflow performance. Moreover, we observe that for large-scale invocations, the overall execution time and the workflow startup time are reduced with *Intelligent Memory Management Environment* due to the effective placement of shared files on the CXL memory that is accessible to all the nodes in the cluster. Overall, we observe a performance improvement of up to 51%, 76%, and 32% compared to the *Ideal Environment*, *Constrained Baseline Environment*, and *Tiered Memory Environment*, respectively.

We also study the impact of concurrent workflow invocations on the overall execution time of batch HPC jobs containing all studied workflows with varying, i.e., 100, 200, 400, and 800, instances. The results are shown in Figure 8.10. We observe that as the number of concurrent workflows increases, the execution time also increases due to resource contention at servers. We observe a negligible overhead, i.e., 4%, of our proposed runtime as the workflows are scaled up due to efficient multi-tiered memory allocation policy and intelligent page movement to ensure that the workflow startup time is reduced. Overall, we observe that our proposed runtime reduces the execution time by up to 19%, 48%, and 4% compared to the *Ideal Environment*, *Constrained Baseline Environment*, and *Tiered Memory Environment*, respectively.

8.4 Summary

In summary, we explore tiered memory systems for running containerized HPC workflows and propose application-attuned intelligent page allocation, movement, and replacement policies to improve performance. We integrate our proposed runtime with popular HPC scheduler (SLURM) and container runtime (Singularity) and evaluate its performance using diverse HPC workflows with various computing, capacity, bandwidth, and latency requirements. Our evaluation shows that our proposed runtime reduces workflow execution times by up to 51%, 87%, and 35% as compared to the ideal, realistic, and optimized tiered execution environments, respectively.

Chapter 9

Conclusion

High-performance computing (HPC) workloads are pivotal in solving complex scientific challenges across various domains, including weather forecasting, medical diagnostics, and fluid dynamics simulation. Traditionally executed on bare-metal systems, containers, functions, or workflows, these workloads pose substantial memory and storage demands that often surpass available resources. Platforms like TensorFlow or PyTorch, used for executing HPC tasks such as deep learning (DL), lack awareness of underlying system performance and resource availability, hindering efficient distributed training. Similarly, Function-as-a-Service (FaaS) platforms impose fixed memory allocation and short task timeouts, leading to unreliable performance and job failures, especially for stateful HPC applications. Lastly, containerized workflows require extensive memory, causing data swapping and degraded performance, while bandwidth-intensive or latency-sensitive tasks suffer from sub-optimal memory allocation. Tiered memory systems, like persistent memory and compute express link (CXL), offer potential solutions by enhancing memory capacity and bandwidth. However, existing memory management techniques fail to adequately address the diverse needs of colocated containerized jobs in HPC systems that run workflows and ensembles at scale concurrently.

In this research, we introduce a comprehensive framework aimed at enhancing the efficiency of HPC platforms, workflow management systems (WMS), and HPC schedulers. By leveraging advancements in memory subsystems, particularly CXL, our framework optimizes HPC workloads' performance by ensuring that heterogeneous datacenter resources are efficiently utilized. Our proposed architectural enhancements and software modules manage the allocation of additional CXL-based memory and introduce a fast intermediate storage tier, alongside intelligent prefetching and caching mechanisms tailored for HPC tasks. We integrate tiered memory systems and implement efficient memory management policies, including intelligent page placement and eviction policies, to enhance memory access performance. These policies consider task characteristics and facilitate efficient memory sharing between workflows. Integration with popular HPC schedulers, such as SLURM, and container runtimes like Singularity, demonstrates improved tiered memory utilization and application performance. Furthermore, integration with TensorFlow and Apache OpenWhisk enables infrastructure-aware scheduling and performance optimization for DL workloads, enhancing resilience and fault tolerance in FaaS platforms. Evaluation results showcase enhanced system utilization, throughput, and performance, along with reduced training time, failure rate, recovery time, latency, and cold-start time across large-scale deployments.

9.0.1 Future Work

In the future, we aim to enhance our *Infrastructure-Aware TensorFlow* platform by addressing the challenges posed by a large number of straggler nodes in scheduling and executing DL jobs on heterogeneous resources. Additionally, we will explore techniques to adapt the platform to different distributed training approaches, such as parameter-server-based distributed training. Expanding our focus on *DiSDeL*, we intend to minimize the cold start latency of containers and incorporate model parallelism to cater to various DL applications. Furthermore, refining the scheduling of DL jobs in multi-tenant environments will further improve serverless resource utilization. For *DeepMemoryDL*, extending support to other DL platforms, particularly PyTorch, to eliminate I/O stalls and enhance overall performance. Additionally, we plan to leverage accelerator-based systems, such as GPUs, to utilize their High Bandwidth Memory (HBM) interconnects, creating an additional tier for prefetching and caching training data, thereby enhancing DL workload performance in distributed settings. Expanding the capabilities of the *Canary* framework, we aim to proactively predict and mitigate failures and explore advanced techniques like request and function replication for robust failure recovery. Moreover, integrating user requirements into the failure recovery strategy will be a key focus to maximize performance and cost benefits when utilizing FaaS platforms. In terms of memory allocation, we plan to introduce dynamic memory resizing and intelligent data movement between different memory tiers to optimize resource utilization. Furthermore, extending our page allocation policy to support variable latency and bandwidth will enable more efficient page replacement and movement. Additionally, we aim to incorporate accelerator memory into our implementation to further enhance system performance.

Bibliography

[1] Amazon s3. https://aws.amazon.com/s3/. Last Accessed: August 26, 2022.

[2] Apache spark. https://spark.apache.org/. Last Accessed: August 26, 2022.

[3] Apache spark. https://www.postgresql.org/. Last Accessed: August 26, 2022.

[4] Aws lambda pricing. https://aws.amazon.com/lambda/pricing/. Last Accessed: August 26, 2022.

[5] Distributed training with tensorflow. https://www.tensorflow.org/guide/distributed_training. Last Accessed: April 17, 2020.

[6] Dmlab dataset. https://www.tensorflow.org/datasets/catalog/dmlab. Last Accessed: October 15, 2021.

[7] Durable functions: Semantics for stateful serverless. https://angelhof.github.io/files/papers/durable-functions-2021-oopsla.pdf. Last Accessed: August 26, 2022.

[8] ethtool(8) - linux man page. https://linux.die.net/man/8/ethtool. Last Accessed: April 17, 2020.

[9] Fn Project - The Container Native Serverless Framework. https://fnproject.io/. Last Accessed: October 15, 2021.

[10] For the love of serverless. https://newrelic.com/sites/default/files/2021-08/serverless-benchmark-report-aws-lambda-2020.pdf. Last Accessed: August 26, 2022.

[11] Ibm cloud functions. https://www.ibm.com/cloud/functions. Last Accessed: August 26, 2022.

[12] Ibm cloud object storage. https://www.ibm.com/cloud/object-storage. Last Accessed: August 26, 2022.

[13] igraph – network analysis software. `https://igraph.org/`. Last Accessed: August 26, 2022.

[14] Imdb dataset. `https://www.imdb.com/interfaces/`. Last Accessed: April 07, 2023.

[15] Intel pcm. `https://www.intel.com/content/www/us/en/developer/articles/technical/performance-counter-monitor.html`. Last Accessed: January 10, 2022.

[16] Kubeless. `https://kubeless.io/`. Last Accessed: October 15, 2021.

[17] Nvidia system management interface. https://developer.nvidia.com/nvidia-system-management-interface.

[18] Redislabs - redis. `https://redis.io/`. Last Accessed: August 26, 2022.

[19] stress(1) - linux man page. `https://linux.die.net/man/1/stress`. Last Accessed: April 17, 2020.

[20] Us census data. `https://www2.census.gov/programs-surveys/popest/technical-documentation/file-layouts/2010-2017/cc-est2017-alldata.pdf`. Last Accessed: August 26, 2022.

[21] *Google Cloud - Knative*, 10 2021. Last Accessed: October 15, 2021.

[22] Apache ignite. `https://ignite.apache.org/`, August 2022. Last Accessed: August 26, 2022.

[23] Apache openwhisk. `https://openwhisk.apache.org/`, August 2022. Last Accessed: August 26, 2022.

[24] Google cloud storage. `https://cloud.google.com/storage`, March 2022. Last Accessed: August 26, 2022.

[25] Nvidia data loading library (dali). `https://developer.nvidia.com/DALI`, January 2022. Last Accessed: January 10, 2022.

[26] Sysstat. `http://sebastien.godard.pagesperso-orange.fr/`, January 2022. Last Accessed: January 10, 2022.

[27] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *Proc. USENIX OSDI*, 2016.

[28] Eman AbdElfattah, Mohamed Elkawkagy, and Ashraf El-Sisi. A reactive fault tolerance approach for cloud computing. In *2017 13th International Computer Engineering Conference (ICENCO)*, pages 190–194, 2017.

[29] Dong H. Ahn, Ned Bass, Albert Chu, Jim Garlick, Mark Grondona, Stephen Herbein, Helgi I. Ingólfsson, Joseph Koning, Tapasya Patki, Thomas R.W. Scogland, Becky Springmeyer, and Michela Taufer. Flux: Overcoming scheduling challenges for exascale workflows. *Future Generation Computer Systems*, 110:202–213, 2020.

[30] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling cxl memory expansion for in-memory database management systems. In *Data Management on New Hardware*, DaMoN'22, New York, NY, USA, 2022. Association for Computing Machinery.

[31] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. Enabling cxl memory expansion for in-memory database management systems. In *Data Management on New Hardware*, pages 1–5. 2022.

[32] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. SAND: Towards high-performance serverless computing. In *Proc. USENIX ATC*, 2018.

[33] Ahsan Ali, Syed Zawad, Paarijaat Aditya, Istemi Ekin Akkus, Ruichuan Chen, and Feng Yan. Smlt: A serverless framework for scalable and adaptive machine learning design and training, 2022.

[34] R. E. Allen, A. A. Clark, J. A. Starek, and M. Pavone. A machine learning approach for real-time reachability analysis. In *Proc. IEEE/RSJ IROS*, 2014.

[35] Mohammed Amoon, Nirmeen El-Bahnasawy, Samy Sadi, and Manar Wagdi. On the design of reactive approach with flexible checkpoint interval to tolerate faults in cloud computing systems. *Journal of Ambient Intelligence and Humanized Computing*, 10(11):4567–4577, 2019.

[36] J Chris Anderson, Jan Lehnardt, and Noah Slater. *CouchDB: the definitive guide: time to relax.* O'Reilly Media, Inc., 2010.

[37] Rushil Anirudh, Jayaraman J Thiagarajan, Peer-Timo Bremer, and Brian K Spears. Improved surrogates in inertial confinement fusion with manifold and cycle consistencies. *Proceedings of the National Academy of Sciences*, 117(18):9741–9746, 2020.

[38] Moiz Arif, Kevin Assogba, and M. Mustafa Rafique. Canary: Fault-tolerant faas for stateful time-sensitive applications. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2022.

[39] Moiz Arif, Kevin Assogba, M. Mustafa Rafique, and Sudharshan Vazhkudai. Exploiting cxl-based memory for distributed deep learning. In *2022 51st International Conference on Parallel Processing (ICPP)*, 2022.

[40] Moiz Arif, M. Mustafa Rafique, Seung-Hwan Lim, and Zaki Malik. Infrastructure-aware tensorflow for heterogeneous datacenters. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 1–8, 2020.

[41] Kevin Assogba, Moiz Arif, M. Mustafa Rafique, and Dimitrios S. Nikolopoulos. On realizing efficient deep learning using serverless computing. In *2022 IEEE/ACM 22nd International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022.

[42] Kevin Assogba, Moiz Arif, M. Mustafa Rafique, and Dimitrios S. Nikolopoulos. On realizing efficient deep learning using serverless computing. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 220–229, 2022.

[43] Salma M. A. Ataallah, Salwa M. Nassar, and Elsayed E. Hemayed. Fault tolerance in cloud computing - survey. In *2015 11th International Computer Engineering Conference (ICENCO)*, pages 241–245, 2015.

[44] Rosa Maria Badia Sala, Eduard Ayguadé Parra, and Jesús José Labarta Mancho. Workflows for science: A challenge when facing the convergence of hpc and big data. *Supercomputing frontiers and innovations*, 4(1):27–47, 2017.

[45] Wei-Hua Bai, Jian-Qing Xi, Jia-Xian Zhu, and Shao-Wei Huang. Performance analysis of heterogeneous data centers in cloud computing using a complex queuing model. *Mathematical Problems in Engineering*, 2015.

[46] Carlos HS Barbosa, Liliane NO Kunstmann, Rômulo M Silva, Charlan DS Alves, Bruno S Silva, MS Djalma Filho, Marta Mattoso, Fernando A Rochinha, and Alvaro LGA Coutinho. A workflow for seismic imaging with quantified uncertainty. *Computers & Geosciences*, 145:104615, 2020.

[47] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architec-

tures. In *Proceedings of the 20th International Middleware Conference*, Middleware '19, page 41–54, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Daniel Barcelona-Pons, Marc Sánchez-Artigas, Gerard París, Pierre Sutra, and Pedro García-López. On the faas track: Building stateful distributed applications with serverless architectures. In *Proceedings of the 20th International Middleware Conference*, 2019.

[49] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. Stateful serverless computing with crucial. *ACM Trans. Softw. Eng. Methodol.*, 31(3), mar 2022.

[50] Salman Abdul Baset, Long Wang, and Chunqiang Tang. Towards an understanding of oversubscription in cloud. In *Hot-ICE*, 2012.

[51] Kamalakant Laxman Bawankule, Rupesh Kumar Dewang, and Anil Kumar Singh. Early straggler tasks detection by recurrent neural network in a heterogeneous environment. *Applied Intelligence*, pages 1–21, 2022.

[52] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Comput. Surv.*, 52(4), August 2019.

[53] Tim Blechmann. Boost. lockfree. *Boost C++ Libraries*, 2013.

[54] Yasmina Bouizem, Nikos Parlavantzas, Djawida Dib, and Christine Morin. Active-standby for high-availability in faas. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, page 31–36, New York, NY, USA, 2020. Association for Computing Machinery.

[55] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proc. ACM SoCC*, 2019.

[56] Dheeraj Chahal, Pradeep Gameria, Rajesh Kulkarni, and Amit Kalele. Isesa: Towards migrating hpc and ai workloads to serverless platform. In *Proceedings of the 12th Workshop on AI and Scientific Computing at Scale Using Flexible Computing Infrastructures*, FlexScience '22, page 1–8, New York, NY, USA, 2022. Association for Computing Machinery.

[57] Dheeraj Chahal, Ravi Ojha, Manju Ramesh, and Rekha Singhal. Migrating large deep learning models to serverless architecture. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 111–116, 2020.

[58] Sayantan Chakravorty, Celso Mendes, and Laxmikant V Kale. Proactive fault tolerance in large systems. In *HPCRI Workshop in conjunction with HPCA*, volume 2005, pages 1–7. Citeseer, 2005.

[59] Matthew Chantry, Sam Hatfield, Peter Dueben, Inna Polichtchouk, and Tim Palmer. Machine learning emulation of gravity wave drag in numerical weather forecasting. *Journal of Advances in Modeling Earth Systems*, 13(7):e2021MS002477, 2021.

[60] Ryan Chard, Yadu Babuji, Zhuozhao Li, Tyler Skluzacek, Anna Woodard, Ben Blaiszik, Ian Foster, and Kyle Chard. Funcx: A federated function serving fabric for science. In *Proc. ACM HPDC*, 2020.

[61] Amit S Chavan, Kartik R Nayak, Keval D Vora, Manish D Purohit, and Pramila M Chawan. A comparison of page replacement algorithms. *International Journal of Engineering and Technology*, 3(2):171, 2011.

[62] Steven W. D. Chien, Stefano Markidis, Chaitanya Prasad Sishtla, Luis Santos, Pawel Herman, Sai Narasimhamurthy, and Erwin Laure. Characterizing deep-learning i/o workloads in tensorflow. In *Proceedings of the 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems (PDSW-DISCS)*, pages 54–63. IEEE, 2018.

[63] Sathya Chinnathambi, Agilan Santhanam, Jeyarani Rajarathinam, and M Senthilkumar. Scheduling and checkpointing optimization algorithm for byzantine fault tolerance in cloud clusters. *Cluster Computing*, 22(6):14637–14650, 2019.

[64] François Chollet et al. Keras. `https://keras.io`. Last Accessed: April 17, 2020.

[65] Fahim Chowdhury, Yue Zhu, Todd Heer, Saul Paredes, Adam Moody, Robin Goldstone, Kathryn Mohror, and Weikuan Yu. I/o characterization and performance evaluation of beegfs for deep learning. In *Proceedings of the 48th ACM International Conference on Parallel Processing (ICPP)*, New York, NY, USA, 2019. ACM.

[66] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, page 64–78, New York, NY, USA, 2021. Association for Computing Machinery.

[67] Rafael Ferreira da Silva, Henri Casanova, Kyle Chard, Ilkay Altintas, Rosa M Badia, Bartosz Balis, Tainã Coleman, Frederik Coppens, Frank Di Natale, Bjoern Enders, Thomas

Fahringer, Rosa Filgueira, Grigori Fursin, Daniel Garijo, Carole Goble, Dorran Howell, Shantenu Jha, Daniel S. Katz, Daniel Laney, Ulf Leser, Maciej Malawski, Kshitij Mehta, Loïc Pottier, Jonathan Ozik, J. Luc Peterson, Lavanya Ramakrishnan, Stian Soiland-Reyes, Douglas Thain, and Matthew Wolf. A community roadmap for scientific workflows research and development. In *2021 IEEE Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 81–90, 2021.

[68] Martijn de Heus, Kyriakos Psarakis, Marios Fragkoulis, and Asterios Katsifodimos. Distributed transactions on serverless stateful functions. In *Proceedings of the 15th ACM International Conference on Distributed and Event-Based Systems*, DEBS '21, page 31–42, New York, NY, USA, 2021. Association for Computing Machinery.

[69] Ewa Deelman, Rafael Ferreira da Silva, Karan Vahi, Mats Rynge, Rajiv Mayani, Ryan Tanaka, Wendy Whitcup, and Miron Livny. The pegasus workflow management system: translational computer science in practice. *Journal of Computational Science*, 52:101200, 2021.

[70] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proc. ACM ASPLOS*, 2013.

[71] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Proc. IEEE CVPR*, 2009.

[72] Li Deng and Dong Yu. Deep learning: Methods and applications. *Foundations and Trends in Signal Processing*, 7(3–4):197–387, 2014.

[73] R. Kanniga Devi and M. Muthukannan. Self-healing fault tolerance technique in cloud datacenter. In *2021 6th International Conference on Inventive Computation Technologies (ICICT)*, pages 731–737, 2021.

[74] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[75] Paolo Di Tommaso, Maria Chatzou, Evan W Floden, Pablo Prieto Barja, Emilio Palumbo, and Cedric Notredame. Nextflow enables reproducible computational workflows. *Nature biotechnology*, 35(4):316–319, 2017.

[76] Dave Dice and Nir Shavit. Tlrw: return of the read-write lock. In *Proceedings of the twenty-second annual ACM symposium on Parallelism in algorithms and architectures*, pages 284–293, 2010.

[77] Tu Mai Anh Do, Loïc Pottier, Rafael Ferreira da Silva, Silvina Caíno-Lores, Michela Taufer, and Ewa Deelman. Performance assessment of ensembles of in situ workflows under resource constraints. *Concurrency and Computation: Practice and Experience*, page e7111, 2022.

[78] Tu Mai Anh Do, Loïc Pottier, Rafael Ferreira da Silva, Frédéric Suter, Silvina Caíno-Lores, Michela Taufer, and Ewa Deelman. Co-scheduling ensembles of in situ workflows. In *2022 IEEE/ACM Workshop on Workflows in Support of Large-Scale Science (WORKS)*, pages 43–51, 2022.

[79] Niall Douglas. User mode memory page allocation: A silver bullet for memory allocation? *arXiv preprint arXiv:1105.1811*, 2011.

[80] Nikoli Dryden, Roman Böhringer, Tal Ben-Nun, and Torsten Hoefler. Clairvoyant prefetching for distributed machine learning i/o. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, New York, NY, USA, 2021. ACM.

[81] Haizhou Du, Sheng Huang, and Qiao Xiang. Orchestra: Adaptively accelerating distributed deep learning in heterogeneous environments. In *Proceedings of the 19th ACM International Conference on Computing Frontiers*, CF '22, page 181–184, New York, NY, USA, 2022. Association for Computing Machinery.

[82] Ifeanyi P Egwutuoha, David Levy, Bran Selic, and Shiping Chen. A survey of fault tolerance mechanisms and checkpoint/restart implementations for high performance computing systems. *The Journal of Supercomputing*, 65(3):1302–1326, 2013.

[83] Shaker H Ali El-Sappagh, Abdeltawab M Ahmed Hendawi, and Ali Hamed El Bastawissy. A proposed model for data warehouse etl processes. *Journal of King Saud University-Computer and Information Sciences*, 23(2):91–104, 2011.

[84] Christian Engelmann and Swen Böhm. Redundant execution of hpc applications with mr-mpi. In *Proceedings of the 10th IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN)*, pages 15–17, 2011.

[85] Christian Engelmann, Geoffroy R. Vallee, Thomas Naughton, and Stephen L. Scott. Proactive fault tolerance using preemptive migration. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 252–257, 2009.

[86] Kurt B. Ferreira, Scott Levy, Joshua Hemmert, and Kevin Pedretti. Understanding memory failures on a petascale arm system. In *Proceedings of the 31st International Symposium on*

High-Performance Parallel and Distributed Computing, HPDC '22, page 84–96, New York, NY, USA, 2022. Association for Computing Machinery.

[87] Rafael Ferreira da Silva, Rosa Filgueira, Ilia Pietri, Ming Jiang, Rizos Sakellariou, and Ewa Deelman. A characterization of workflow management systems for extreme-scale applications. *Future Generation Computer Systems*, 75:228–238, 2017.

[88] Michail D Flouris and Evangelos P Markatos. The network ramdisk: Using remote memory on heterogeneous nows. *Cluster computing*, 2(4):281–293, 1999.

[89] Yehonatan Fridman, Suprasad Mutalik Desai, Navneet Singh, Thomas Willhalm, and Gal Oren. Cxl memory as persistent memory for disaggregated hpc: A practical approach. *arXiv preprint arXiv:2308.10714*, 2023.

[90] Jean-loup Gailly and Mark Adler. Gnu gzip. *GNU Operating System*, 1992.

[91] Debashis Ganguly, Ziyu Zhang, Jun Yang, and Rami Melhem. Adaptive page migration for irregular data-intensive applications under gpu memory oversubscription. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–461, 2020.

[92] Amir Gholami, Ariful Azad, Kurt Keutzer, and Aydin Buluç. Integrated model and data parallelism in training neural networks. *arXiv preprint arXiv:1712.04432*, 2017.

[93] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.*, 41(4):350–361, aug 2011.

[94] Samuel Ginzburg and Michael J. Freedman. Serverless isn't server-less: Measuring and exploiting resource variability on cloud faas platforms. In *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, WoSC'20, page 43–48, New York, NY, USA, 2020. Association for Computing Machinery.

[95] William F Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, et al. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX*, 12:100561, 2020.

[96] Eugene Gorelik. *Cloud computing models.* PhD thesis, Massachusetts Institute of Technology, 2013.

[97] Donghyun Gouk, Miryeong Kwon, Hanyeoreum Bae, Sangwon Lee, and Myoungsoo Jung. Memory pooling with cxl. *IEEE Micro*, 43(2):48–57, 2023.

[98] Graphcore. AI for Simulation: How Graphcore is Helping Transform Traditional HPC. `https://www.graphcore.ai/posts/ai-for-simulation-how-graphcore-is-helping-transform-traditional-hpc`. Last Accessed: April 07, 2023.

[99] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with infiniswap. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 649–667, Boston, MA, March 2017. USENIX Association.

[100] Zvika Guz, Harry (Huan) Li, Anahita Shayesteh, and Vijay Balakrishnan. Nvme-over-fabrics performance characterization and the path to low-overhead flash disaggregation. In *Proceedings of the 10th ACM International Systems and Storage Conference (SYSTOR)*, New York, NY, USA, 2017. ACM.

[101] Jingoo Han, M. Mustafa Rafique, Luna Xu, Ali R. Butt, Seung-Hwan Lim, and Sudharshan S. Vazhkudai. Marble: A multi-gpu aware job scheduler for deep learning on hpc systems. In *Proc. IEEE/ACM CCGrid*, 2020.

[102] Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory Ganger, Phillip Gibbons, Garth Gibson, and Eric Xing. Addressing the straggler problem for iterative convergent parallel ml. In *Proc. ACM SoCC*, 2016.

[103] Danny Harnik, Ronen Kat, Dmitry Sotnikov, Avishay Traeger, and Oded Margalit. To zip or not to zip: Effective resource usage for Real-Time compression. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 229–241, San Jose, CA, February 2013. USENIX Association.

[104] Sayed Hadi Hashemi, Paul Rausch, Benjamin Rabe, Kuan-Yen Chou, Simeng Liu, Volodymyr Kindratenko, and Roy H Campbell. tensorflow-tracing: A performance tuning framework for production. In *Proc. USENIX OpML*, 2019.

[105] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE CVPR*, 2016.

[106] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. IEEE CVPR*, 2016.

[107] Stephen Hemminger. Network emulation with netem. In *Linux Conf. Au*, pages 18–23, 2005.

[108] Stephen Herbein, Ayush Dusia, Aaron Landwehr, Sean McDaniel, Jose Monsalve, Yang Yang, Seetharami R Seelam, and Michela Taufer. Resource management for running hpc applications in container clouds. In *High Performance Computing: 31st International Conference, ISC High Performance 2016, Frankfurt, Germany, June 19-23, 2016, Proceedings*, pages 261–278. Springer, 2016.

[109] Kelsey Hightower, Brendan Burns, and Joe Beda. *Kubernetes: Up and Running Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 1st edition, 2017.

[110] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. Autotm: Automatic tensor movement in heterogeneous memory systems using integer linear programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 875–890, New York, NY, USA, 2020. Association for Computing Machinery.

[111] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[112] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[113] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. Halo: A hybrid pmem-dram persistent hash index with fast recovery. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1049–1063, 2022.

[114] Dan Huang, Zhenlu Qin, Qing Liu, Norbert Podhorszki, and Scott Klasky. A comprehensive study of in-memory computing on large hpc systems. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 987–997. IEEE, 2020.

[115] Zaeem Hussain, Taieb Znati, and Rami Melhem. Partial redundancy in hpc systems with non-uniform node reliabilities. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 566–576, 2018.

[116] J. Hwang, S. Zeng, F. y. Wu, and T. Wood. Benefits and challenges of managing heterogeneous data centers. In *Proc. IFIP/IEEE IM*, 2013.

[117] Khaled Z. Ibrahim, Tan Nguyen, Hai Ah Nam, Wahid Bhimji, Steven Farrell, Leonid Oliker, Michael Rowan, Nicholas J. Wright, and Samuel Williams. Architectural requirements for deep learning workloads in hpc environments. In *2021 International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 7–17, 2021.

[118] Connor Imes, Steven Hofmeyr, Dong In D. Kang, and John Paul Walters. A case study and characterization of a many-socket, multi-tier numa hpc platform. In *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*, pages 74–84, 2020.

[119] V. Ishakian, V. Muthusamy, and A. Slominski. Serving deep learning models in a serverless platform. In *Proc. IEEE IC2E*, 2018.

[120] Tariqul Islam and Dakshnamoorthy Manivannan. Predicting application failure in cloud: A machine learning approach. In *2017 IEEE International Conference on Cognitive Computing (ICCC)*, pages 24–31, 2017.

[121] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amir Saman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the intel optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[122] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[123] Zhipeng Jia and Emmett Witchel. Boki: Stateful serverless computing with shared logs. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 691–707, New York, NY, USA, 2021. Association for Computing Machinery.

[124] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *Proc. ACM SIGMOD*, 2017.

[125] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. Towards demystifying serverless machine learning training. *Proc. ACM SIGMOD*, 2021.

[126] Wenbin Jiang, Pai Liu, Hai Jin, and Jing Peng. An efficient data prefetch strategy for deep learning based on non-volatile memory. In Zhiwen Yu, Christian Becker, and Guoliang Xing, editors, *Green, Pervasive, and Cloud Computing*, pages 101–114, Cham, 2020. Springer.

[127] Tong Jin, Fan Zhang, Qian Sun, Hoang Bui, Melissa Romanus, Norbert Podhorszki, Scott Klasky, Hemanth Kolla, Jacqueline Chen, Robert Hager, et al. Exploring data staging across deep memory hierarchies for coupled data intensive simulation workflows. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 1033–1042. IEEE, 2015.

[128] Wuyang Ju, Jianxin Li, Weiren Yu, and Richong Zhang. Igraph: An incremental data processing system for dynamic graph. 10(3):462–476, jun 2016.

[129] Pekka Karhula, Jan Janak, and Henning Schulzrinne. Checkpointing and migration of iot edge functions. In *Proceedings of the 2nd International Workshop on Edge Systems, Analytics and Networking*, EdgeSys '19, page 60–65, New York, NY, USA, 2019. Association for Computing Machinery.

[130] Avita Katal, Susheela Dahiya, and Tanupriya Choudhury. Energy efficiency in cloud computing data center: A survey on hardware technologies. *Cluster Computing*, 25(1):675–705, 2022.

[131] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. Lessons learned from the chameleon testbed. In *Proc. USENIX ATC*. 2020.

[132] Kai Keller and Leonardo Bautista-Gomez. Application-level differential checkpointing for hpc applications with dynamic datasets. In *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 52–61, 2019.

[133] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *USENIX Annual Technical Conference*, pages 715–728, 2021.

[134] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander. *IEEE Micro*, 43(2):20–29, 2023.

[135] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proc. ICLR*, 2015.

[136] Vamsee Reddy Kommareddy, Jagadish Kotra, Clayton Hughes, Simon David Hammond, and Amro Awad. Prefam: Understanding the impact of prefetching in fabric-attached memory architectures. In *Proceedings of the ACM International Symposium on Memory Systems (MEMSYS)*, page 323–334, New York, NY, USA, 2020. ACM.

[137] Jan Kończak and Paweł T Wojciechowski. Failure recovery from persistent memory in paxos-based state machine replication. In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*, pages 88–98. IEEE, 2021.

[138] Nicholas Krichevsky, Renee St Louis, and Tian Guo. Quantifying and improving performance of distributed deep learning with cloud storage. In *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E)*, pages 99–109. IEEE, 2021.

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

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

[141] Jörn Kuhlenkamp, Sebastian Werner, Maria C Borges, Karim El Tal, and Stefan Tai. An evaluation of faas platforms as a foundation for serverless big data processing. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*, pages 1–9, 2019.

[142] Sameer G Kulkarni, Guyue Liu, K. K. Ramakrishnan, and Timothy Wood. Living on the edge: Serverless computing and the cost of failure resiliency. In *2019 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN)*, pages 1–6, 2019.

[143] Abhishek Vijaya Kumar and Muthian Sivathanu. Quiver: An informed storage cache for deep learning. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*, pages 283–296, Santa Clara, CA, February 2020. USENIX Association.

[144] Sandeep Kumar, Aravinda Prasad, Smruti R Sarangi, and Sreenivas Subramoney. Radiant: efficient page table management for tiered memory systems. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*, pages 66–79, 2021.

[145] Miryeong Kwon, Junhyeok Jang, Hanjin Choi, Sangwon Lee, and Myoungsoo Jung. Failure tolerant training with persistent memory disaggregation over cxl. *IEEE Micro*, 43(2):66–75, 2023.

[146] Miryeong Kwon, Junhyeok Jang, Hanjin Choi, Sangwon Lee, and Myoungsoo Jung. Training resilience with persistent memory pooling using cxl technology. In *Heterogeneous and Composable Memory Workshop at HPCA, 2023*. IEEE, 2023.

[147] Feng Lang, Kudva Prabhakar, Da Silva Dilma, and Hu Jiang. Exploring serverless computing for neural network training. In *Proc. IEEE CLOUD*, 2018.

[148] Rubén Langarita Benítez. Evaluation of genome alignment workflows on hpc processors. Master's thesis, Universitat Politècnica de Catalunya, 2021.

[149] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. Cost modelling for optimal data placement in heterogeneous main memory. *Proceedings of the VLDB Endowment*, 15(11):2867–2880, 2022.

[150] Michael Laufer and Erick Fredj. High performance parallel i/o and in-situ analysis in the wrf model with adios2. *arXiv preprint arXiv:2201.08228*, 2022.

[151] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. IEEE*, 86(11):2278–2324, 1998.

[152] Yann LeCun, Corinna Cortes, and CJ Burges. Mnist handwritten digit database. *ATT Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2010.

[153] KyungSoo Lee, Sohyun Kim, Joohee Lee, Donguk Moon, Rakie Kim, Honggyu Kim, Hyeongtak Ji, Yunjeong Mun, and Youngpyo Joo. Improving key-value cache performance with heterogeneous memory tiering: A case study of cxl-based memory expansion. *IEEE Micro*, pages 1–11, 2024.

[154] Chao Li, Changhai Zhao, Haihua Yan, and Jianlei Zhang. Event-driven fault tolerance for building nonstop active message programs. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 382–390, 2013.

[155] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 574–587, 2023.

[156] Huaicheng Li, Daniel S Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark Hill, Marcus Fontoura, et al. First-generation memory disaggregation for cloud platforms. *arXiv preprint arXiv:2203.00241*, 2022.

[157] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. First-generation memory disaggregation for cloud platforms, 2022.

[158] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. Tetris: Memory-efficient serverless inference through tensor sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.

[159] Ping-Min Lin and Alex Glikson. Mitigating cold starts in serverless platforms: A pool-based approach. *arXiv preprint arXiv:1903.12221*, 2019.

[160] Jialei Liu, Shangguang Wang, Ao Zhou, Sathish A. P. Kumar, Fangchun Yang, and Rajkumar Buyya. Using proactive fault-tolerance approach to enhance cloud service reliability. *IEEE Transactions on Cloud Computing*, 6(4):1191–1202, 2018.

[161] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. Hierarchical hybrid memory management in os for tiered memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2223–2236, 2019.

[162] Petra Loncar and Paula Loncar. Scalable management of heterogeneous cloud resources based on evolution strategies algorithm. *IEEE Access*, 10:68778–68791, 2022.

[163] Xinjian Long, Xiangyang Gong, Bo Zhang, and Huiyang Zhou. An intelligent framework for oversubscription management in cpu-gpu unified memory. *Journal of Grid Computing*, 21(1):11, 2023.

[164] Sebastian Lührs, Daniel Rohe, Alexander Schnurpfeil, Kay Thust, and Wolfgang Frings. Flexible and generic workflow management. In *Parallel Computing: On the Road to Exascale*, pages 431–438. IOS Press, 2016.

[165] Jakob Lüttgau, Shane Snyder, Philip Carns, Justin M Wozniak, Julian Kunkel, and Thomas Ludwig. Toward understanding i/o behavior in hpc workflows. In *2018 IEEE/ACM 3rd international workshop on parallel data storage & data intensive scalable computing systems (PDSW-DISCS)*, pages 64–75. IEEE, 2018.

[166] Ricardo Macedo, Cláudia Correia, Marco Dantas, Cláudia Brito, Weijia Xu, Yusuke Tanimura, Jason Haga, and João Paulo. The case for storage optimization decoupling in deep

learning frameworks. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, pages 649–656. IEEE, 2021.

[167] Shie Mannor, Dori Peleg, and Reuven Rubinstein. The cross entropy method for classification. In *In Proc. ICML*, 2005.

[168] Najme Mansouri. Adaptive data replication strategy in cloud computing for performance improvement. *Frontiers of Computer Science*, 10(5):925–935, 2016.

[169] Ying Mao, Vaishali Sharma, Wenjia Zheng, Long Cheng, Qiang Guan, and Ang Li. Elastic resource management for deep learning applications in a container cluster. *IEEE Transactions on Cloud Computing*, pages 1–13, 2022.

[170] J. Mars, L. Tang, and R. Hundt. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters*, 10(2):29–32, 2011.

[171] 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.

[172] George Mavrotas. Effective implementation of the ϵ-constraint method in multi-objective mathematical programming problems. *Applied Mathematics and Computation*, 213(2):455–465, 2009.

[173] Ruben Mayer and Hans-Arno Jacobsen. Scalable deep learning on distributed infrastructures: Challenges, techniques, and tools. *ACM Comput. Surv.*, 53(1), feb 2020.

[174] John D McCalpin et al. Memory bandwidth and machine balance in current high performance computers. *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, 2(19-25), 1995.

[175] Memcached. Memcached - a distributed memory object caching system. `https://memcached.org/`. Last Accessed: October 15, 2021.

[176] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V. Le, and Jeff Dean. A hierarchical model for device placement. In *Proc. ICLR*, 2018.

[177] Mukosi Abraham Mukwevho and Turgay Celik. Toward a smart cloud: A review of fault-tolerance methods in cloud systems. *IEEE Transactions on Services Computing*, 14(2):589–605, 2021.

[178] Derek Gordon Murray, Jiri Simsa, Ana Klimovic, and Ihor Indyk. tf.data: A machine learning data processing framework. *CoRR*, abs/2101.12127, 2021.

[179] Djob Mvondo, Mathieu Bacou, Kevin Nguetchouang, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, and Alain Tchana. Ofc: An opportunistic caching system for faas platforms. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 228–244, New York, NY, USA, 2021. Association for Computing Machinery.

[180] Md Nahid Newaz and Md Atiqul Mollah. Memory usage prediction of hpc workloads using feature engineering and machine learning. In *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*, HPC Asia '23, page 64–74, New York, NY, USA, 2023. Association for Computing Machinery.

[181] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The ramcloud storage system. *ACM Trans. Comput. Syst.*, 33(3), aug 2015.

[182] A. Palade, A. Kazmi, and S. Clarke. An evaluation of open source serverless computing frameworks support at the edge. In *Proc. IEEE SERVICES*, 2019.

[183] Bo Pang, Erik Nijkamp, and Ying Nian Wu. Deep learning with tensorflow: A review. *J. Educ. Behav. Stat.*, 2020.

[184] Konstantinos Parasyris, Kai Keller, Leonardo Bautista-Gomez, and Osman Unsal. Checkpoint restart support for heterogeneous hpc applications. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 242–251, 2020.

[185] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[186] Prasenjit Kumar Patra, Harshpreet Singh, and Gurpreet Singh. Fault tolerance techniques and comparative implementation in cloud computing. *International Journal of Computer Applications*, 64(14), 2013.

[187] Ivy Peng, Ian Karlin, Maya Gokhale, Kathleen Shoga, Matthew Legendre, and Todd Gamblin. A holistic view of memory utilization on hpc systems: Current and future trends. In *The International Symposium on Memory Systems*, pages 1–11, 2021.

[188] J Luc Peterson, K Athey, PT Bremer, V Castillo, F Di Natale, JE Field, D Fox, J Gaffney, D Hysom, SA Jacobs, et al. Merlin: enabling machine learning-ready hpc ensembles. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2019.

[189] J Luc Peterson, Ben Bay, Joe Koning, Peter Robinson, Jessica Semler, Jeremy White, Rushil Anirudh, Kevin Athey, Peer-Timo Bremer, Francesco Di Natale, et al. Enabling machine learning-ready hpc ensembles with merlin. *Future Generation Computer Systems*, 131:255–268, 2022.

[190] Kassian Plankensteiner, Radu Prodan, and Thomas Fahringer. A new fault tolerance heuristic for scientific workflows in highly distributed environments based on resubmission impact. In *2009 Fifth IEEE International Conference on e-Science*, pages 313–320, 2009.

[191] G. Plastiras, M. Terzi, C. Kyrkou, and T. Theocharidcs. Edge intelligence: Challenges and opportunities of near-sensor machine learning applications. In *Proc. IEEE ASAP*, 2018.

[192] Franz Poeschel, Juncheng E, William F Godoy, Norbert Podhorszki, Scott Klasky, Greg Eisenhauer, Philip E Davis, Lipeng Wan, Ana Gainaru, Junmin Gu, et al. Transitioning from file-based hpc workflows to streaming data pipelines with openpmd and adios2. In *Smoky Mountains Computational Sciences and Engineering Conference*, pages 99–118. Springer, 2021.

[193] Rahul Potharaju and Navendu Jain. Demystifying the dark side of the middle: A field study of middlebox failures in datacenters. In *Proceedings of the 2013 Conference on Internet Measurement Conference*, IMC '13, page 9–22, New York, NY, USA, 2013. Association for Computing Machinery.

[194] Alexander Power and Gerald Kotonya. A microservices architecture for reactive and proactive fault tolerance in iot systems. In *2018 IEEE 19th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pages 588–599, 2018.

[195] Thomas Rausch, Waldemar Hummer, Vinod Muthusamy, Alexander Rashed, and Schahram Dustdar. Towards a serverless platform for edge AI. In *Proc. USENIX HotEdge*, 2019.

[196] 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, page 392–407, New York, NY, USA, 2021. Association for Computing Machinery.

[197] Jie Ren, Dong Xu, Ivy Peng, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Hm-keeper: Scalable page management for multi-tiered large memory systems. *arXiv preprint arXiv:2302.09468*, 2023.

[198] Jie Ren, Dong Xu, Ivy Peng, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Rethinking memory profiling and migration for multi-tiered large memory systems, 2023.

[199] Mauro Ribeiro, Katarina Grolinger, and Miriam A.M. Capretz. Mlaas: Machine learning as a service. In *Proc. IEEE ICMLA*, 2015.

[200] Xiaojun Ruan and Haiquan Chen. Informed prefetching in i/o bounded distributed deep learning. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 850–857. IEEE, 2021.

[201] Michael A Salim, Thomas D Uram, J Taylor Childers, Prasanna Balaprakash, Venkatram Vishwanath, and Michael E Papka. Balsam: Automated scheduling and execution of dynamic, data-intensive hpc workflows. *arXiv preprint arXiv:1909.08704*, 2019.

[202] Johann Schleier-Smith, Leonhard Holz, Nathan Pemberton, and Joseph M. Hellerstein. A faas file system for serverless computing. *CoRR*, abs/2009.09845, 2020.

[203] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. Memory devices and applications for in-memory computing. *Nature nanotechnology*, 15(7):529–544, 2020.

[204] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

[205] Harald Servat, Antonio J Peña, Germán Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesús Labarta. Automating the application data placement in hybrid memory systems. In *2017 IEEE International Conference on Cluster Computing*, pages 126–136. IEEE, 2017.

[206] Sai Sha, Chuandong Li, Xiaolin Wang, Zhenlin Wang, and Yingwei Luo. Hardware-software collaborative tiered-memory management framework for virtualization. *ACM Trans. Comput. Syst.*, 42(1–2), feb 2024.

[207] Spencer Shepler, Brent Callaghan, David Robinson, Robert Thurlow, Carl Beame, Mike Eisler, and David Noveck. Rfc3530: Network file system (nfs) version 4 protocol, 2003.

[208] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 419–433. USENIX Association, July 2020.

[209] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery.

[210] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[211] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. Eelru: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27(1):122–133, 1999.

[212] Vikram Sreekanti, Chenggang Wu, Saurav Chhatrapati, Joseph E. Gonzalez, Joseph M. Hellerstein, and Jose M. Faleiro. A fault-tolerance shim for serverless computing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[213] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst. *Proceedings of the VLDB Endowment*, 13(12):2438–2452, Aug 2020.

[214] Omer Subasi, Gulay Yalcin, Ferad Zyulkyarov, Osman Unsal, and Jesus Labarta. Designing and modelling selective replication for fault-tolerant hpc applications. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 452–457, 2017.

[215] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proc. IEEE CVPR*, 2015.

[216] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proc. IEEE CVPR*, 2016.

[217] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In *Proc. IEEE ICDCS*, 2017.

[218] John Thorpe, Yifan Qiao, Jonathan Eyolfson, Shen Teng, Guanzhou Hu, Zhihao Jia, Jinliang Wei, Keval Vora, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Dorylus: Affordable, scalable, and accurate GNN training with distributed CPU servers and serverless threads. In *Proc. USENIX OSDI*, 2021.

[219] Laszlo Toka, Gergely Dobreff, David Haja, and Mark Szalay. Predicting cloud-native application failures based on monitoring data of cloud infrastructure. In *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, pages 842–847, 2021.

[220] James Turnbull. *The Docker Book: Containerization is the new virtualization.* James Turnbull, 2014.

[221] Erwin van Eyk, Alexandru Iosup, Simon Seif, and Markus Thommes. The spec cloud group's research vision on faas and serverless architectures. In *Proc. ACM WoSC*, 2017.

[222] Ranjan Sarpangala Venkatesh, Tony Mason, Pradeep Fernando, Greg Eisenhauer, and Ada Gavrilovska. Scheduling hpc workflows with intel optane persistent memory. In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 56–65. IEEE, 2021.

[223] Abhinav Vishnu, Joseph Manzano, Charles Siegel, and Jeff Daily. User-transparent distributed tensorflow. *arXiv preprint arXiv:1704.04560*, 2017.

[224] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 193–204, New York, NY, USA, 2010. Association for Computing Machinery.

[225] Kate Voss, Geraldine Van Der Auwera, and Jeff Gentry. Full-stack genomics pipelining with gatk4 + wdl + cromwell, 2017.

[226] Jacob Wahlgren, Maya Gokhale, and Ivy B Peng. Evaluating emerging cxl-enabled memory pooling for hpc systems. *arXiv preprint arXiv:2211.02682*, 2022.

[227] Jacob Wahlgren, Maya Gokhale, and Ivy B. Peng. Evaluating emerging cxl-enabled memory pooling for hpc systems. In *2022 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 11–20, 2022.

[228] Guosai Wang, Lifei Zhang, and Wei Xu. What can we learn from four years of data center hardware failures? In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–36, 2017.

[229] H. Wang, D. Niu, and B. Li. Distributed machine learning with a serverless architecture. In *Proc. IEEE INFOCOM*, 2019.

[230] Qing Wang, Youyou Lu, and Jiwu Shu. Sherman: A write-optimized distributed b+tree index on disaggregated memory. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, page 1033–1048, New York, NY, USA, 2022. ACM.

[231] Wei Wang, Jack W. Davidson, and Mary Lou Soffa. Predicting the memory bandwidth and optimal core allocations for multi-threaded applications on large-scale numa machines. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 419–431. IEEE, 2016.

[232] Yinzhi Wang, R. Todd Evans, and Lei Huang. Performant container support for hpc applications. In *Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (Learning)*, PEARC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[233] Christian Witzler, J Miguel Zavala-Aké, Karol Sierociński, and Herbert Owen. Including in situ visualization and analysis in pdi. In *International Conference on High Performance Computing*, pages 508–512. Springer, 2021.

[234] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 33(2):344–358, 2021.

[235] 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, page 331–345, New York, NY, USA, 2019. Association for Computing Machinery.

[236] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, February 2020. USENIX Association.

[237] Qirui Yang, Runyu Jin, Bridget Davis, Devasena Inupakutika, and Ming Zhao. Performance evaluation on cxl-enabled hybrid memory pool. In *2022 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pages 1–5. IEEE, 2022.

[238] Yiwei Yang, Pooneh Safayenikoo, Jiacheng Ma, Tanvir Ahmed Khan, and Andrew Quinn. Cxlmemsim: A pure software simulated cxl. mem for performance characterization. *arXiv preprint arXiv:2303.06153*, 2023.

[239] Keun Soo Yim. Evaluation metrics of service-level reliability monitoring rules of a big data service. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 376–387, 2016.

[240] Junqi Yin, Feiyi Wang, and Mallikarjun Shankar. Strategies for integrating deep learning surrogate models with hpc simulation applications. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 01–10. IEEE, 2022.

[241] Andrew J. Younge, Kevin Pedretti, Ryan E. Grant, and Ron Brightwell. A tale of two systems: Using containers to deploy hpc applications on supercomputers and clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 74–81, 2017.

[242] Huihuang Yu, Zhu Zongwei, XiangLan Chen, Yuming Cheng, Yahui Hu, and Xi Li. Accelerating distributed training in heterogeneous clusters via a straggler-aware parameter server. In *Proc. IEEE HPCC*, 2019.

[243] Iman I. Yusuf and Heinz W. Schmidt. Parameterised architectural patterns for providing cloud service fault tolerance with accurate costings. In *Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*, CBSE '13, page 121–130, New York, NY, USA, 2013. Association for Computing Machinery.

[244] Zili Zha, An Wang, Yang Guo, and Songqing Chen. Towards software defined measurement in data centers: A comparative study of designs, implementation, and evaluation. *IEEE Transactions on Cloud Computing*, pages 1–12, 2022.

[245] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204. USENIX Association, November 2020.

[246] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, 2020.

[247] Zhao Zhang, Lei Huang, Uri Manor, Linjing Fang, Gabriele Merlo, Craig Michoski, John Cazes, and Niall Gaffney. Fanstore: Enabling efficient and scalable I/O for distributed deep learning. *CoRR*, abs/1809.10799, 2018.

[248] Maohua Zhu, Youwei Zhuo, Chao Wang, Wenguang Chen, and Yuan Xie. Performance evaluation and optimization of hbm-enabled gpu for data-intensive applications. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(5):831–840, 2018.

[249] Yue Zhu, Weikuan Yu, Bing Jiao, Kathryn Mohror, Adam Moody, and Fahim Chowdhury. Efficient user-level storage disaggregation for deep learning. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–12. IEEE, 2019.