

FIRST LINE OF THE TITLE
SECOND LINE OF THE TITLE

by

Author
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Discipline

Committee:

_____	Dr. First Last, Thesis Director
_____	Dr. First Last, Committee Member
_____	Dr. First Last, Committee Member
_____	Dr. First Last, Department Head
_____	Dr. First Last, Dean

Date: _____ X Semester Year
George Mason University
Fairfax, VA

The Complete Title is to be Repeated Here without any Line Breaks for the Second Page
and for the Abstract Page

A thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science at George Mason University

By

Author
Bachelor of Science
My Other Former School, Year of first degree

Director: Dr. First Last, Professor
Department of Name of Department

X Semester Year
George Mason University
Fairfax, VA

Copyright © Year by Author
All Rights Reserved

Dedication

I dedicate this dissertation to ...

Acknowledgments

I would like to thank the following people who made this possible ...

Table of Contents

	Page
List of Tables	vii
List of Figures	viii
Abstract	ix
1 Introduction	1
1.1 Motivation	1
1.2 Research Questions	3
1.3 Contributions	4
2 Background	6
2.1 Intel Optane DC Persistent Memory	6
2.1.1 Overview of Intel Optane DC PMM	6
2.1.2 Performance Characterization	9
2.1.3 Operating Modes and Applications	10
2.1.4 Programming	10
2.2 Serverless Computing	11
2.2.1 Funtion-as-a-Service (FaaS)	12
2.2.2 Storage for FaaS	13
2.2.3 Service Level Agreements	14
2.3 Reinforcement Learning	15
2.3.1 Overview of Reinforcement Learning	15
2.3.2 Q-Learning	16
2.3.3 Function Approximation using Linear Regression Models	17
2.3.4 Exploration-Exploitation Tradeoff	18
2.3.5 Reward shaping	18
3 NVM Middleware: A control layer for persistent memory	20
3.1 Motivation	21
3.1.1 Concurrency Control Challenges in a serverless storage service	22
3.1.2 NVM Middleware Design Overview	23
3.2 Architecture	24

3.3	Programming Interface	26
3.4	Reinforcement Learning Component	27
3.4.1	Integration with the NVM Middleware	27
3.4.2	Reinforcement Learning Model	29
3.4.3	Training Methodology	31
3.5	Implementation	34
4	Evaluation	37
4.1	Experimental Setup	37
4.1.1	Platform	37
4.1.2	Optane DC PMem Configuration	38
4.1.3	Workload Generators	38
4.2	Efficiency of the Workload-Aware Concurrency Control Mechanism	39
4.3	Meeting SLA performance using RL	40
5	Related Work	44
6	Discussions and Future Research	47
7	Conclusion	49
A	Reinforcement Learning Algorithms	50
B	Concurrency Control Results	52
C	RL Phases	55
D	RL Agent Training	56
E	RL Agent Evaluation	64
	Bibliography	66

List of Tables

Table	Page
3.1 NVM Middleware Programming Interface	26
3.2 Reinforcement Learning State Representation	29
3.3 Reinforcement Learning Action Space	30
4.1 Experimental Platform Specifications	37
D.1 Hyperparameter Tuning Options for Polynomial Regression Models	56
D.2 Resulting Hyperparameters for Polynomial Regression Models	57
D.3 Q-Learning Parameters	57
D.4 Preliminary Measurements for Phase 1	58
D.5 Preliminary Measurements for Phase 2	58
D.6 Preliminary Measurements for Phase 3	58
D.7 Preliminary Measurements for Phase 4	59
E.1 RL Agent Reward Analysis in Long-run Test	65
E.2 RL Agent Throughput Analysis in Long-run Test	65
E.3 RL Agent Latency Analysis in Long-run Test	65

List of Figures

Figure	Page
2.1 Memory-Storage Hierarchy with Persistent Memory	6
2.2 Intel Optane Persistent Memory Module Architecture	8
2.3 Agent-Environment Interaction in Reinforcement Learning	15
3.1 NVM Middleware Architecture	25
3.2 Interaction between RL agent and NVM Middleware	28
3.3 Overview of the Reinforcement Learning Environment	35
3.4 Reinforcement Learning Agent Process Flow	36
B.1 Evaluation of NVM Middleware: Benchmark A	52
B.2 Evaluation of NVM Middleware: Benchmark B	53
B.3 Evaluation of NVM Middleware; Benchmark C	54
D.1 Learned Pattern of Agent during Phase 1	60
D.2 Learned Pattern of Agent during Phase 2	61
D.3 Learned Pattern of Agent during Phase 3	62
D.4 Learned Pattern of Agent during Phase 3	63
E.1 RL Agent Adaptation to Shifting Workloads	64

Abstract

THE COMPLETE TITLE IS TO BE REPEATED HERE WITHOUT ANY LINE BREAKS
FOR THE SECOND PAGE AND FOR THE ABSTRACT PAGE

Author, MS

George Mason University, Year

Thesis Director: Dr. First Last

Enter abstract text.

Chapter 1: Introduction

1.1 Motivation

Serverless computing is an increasingly popular cloud execution model that liberates application developers from the burden of traditional infrastructure management. With serverless platforms (e.g., AWS Lambda, Google Cloud Functions, Azure Functions), developers solely focus on writing their code as event-driven functions that will execute on-demand in response to events or triggers. Cloud providers are responsible for dynamically allocating and scaling resources to meet demands as the event triggers occur. With a pay-as-you-go pricing model, users only pay for the resource consumed during their function invocations, making serverless computing a cost-effective solution.

Cloud providers designed serverless functions to be stateless, meaning that they do not retain state between function invocations. This intentional statelessness is a fundamental aspect for achieving high elasticity. By eliminating the need to store state within the function invocation, serverless platforms promote scalability and ease of deployment. Cloud providers can execute functions in parallel, allowing for efficient resource utilization. Any data needed between function invocations must be stored in remote storage.

Although the stateless nature of serverless computing is key to achieve high elasticity, it limits the type of applications that can run efficiently on serverless platforms. Previous studies [1, 2] have found that data-intensive applications running in serverless platforms (i.e., data analytics, ML workflows, databases) are limited by the capacity and performance gaps that exist among the existing storage services. Object storage services, such as AWS S3, provide cheap long-term storage, but exhibits high access latencies. On the other hand, in-memory clusters, such as AWS ElastiCache, exhibit low access latencies and high throughput, but they are expensive and are not transparently provisioned. In between,

key-value databases, such as AWS DynamoDB, provide high throughput, but are expensive and can take a long time to scale.

Given the limitations of existing storage solutions, previous works motivate the development of a serverless storage service capable of handling the wide variety of workloads running on serverless platforms. These studies mention three requirements that such service must meet. First, it should provide low latency and high throughput for a wide range of object size and data access patterns. Second, it should be transparently provisioned and should scale to meet workload demands. Third, it must ensure isolation and predictable performance across applications and tenants.

To meet the first requirement, cloud providers must first close the capacity and performance gap between main memory and persistent storage media. As mentioned above, existing storage service have fixed tradeoffs that reflect the traditional memory hierarchy built from RAM, flash memory, and magnetic disk drives. Leveraging Non-volatile memory is a promising approach to bridge the gap between the memory and storage tiers. Non-volatile memory combines the persistence and capacity of traditional storage with the low latency and byte addressability of main memory. This technology experienced a breakthrough with the release of Intel Optane DC Persistent Memory.

Non-volatile memory technology experienced a breakthrough with the release of Intel Optane DC Persistent Memory Module (PMM). Optane PMM is an emerging technology where non-volatile media is placed in a Dual In-Line Memory Module (DIMM) and installed on the memory bus, alongside traditional DRAM (Dynamic Random Access Memory). Similar to DRAM, this technology presents a byte-addressable interface and achieves speeds comparable to DRAM (2x-3x lower). The main difference between the two is that Optane PMM has higher capacities and can retain data when the system is shutdown or loses power. This allows Optane PMM to be used as a form of persistent storage with memory-like speeds.

The unique combination of persistence and low access latency makes Optane PMM an ideal candidate to speed up data-intensive workloads running in serverless platforms. Thus,

thesis presents an analysis on how to make efficient use of Optane PMM to build a serverless storage service.

1.2 Research Questions

With the release of Intel Optane DIMM, researchers have started to understand its characteristics, capabilities, and limitations [3–5]. The initial expectation was that Intel Optane DC PMM would behave similar to DRAM, but with a lower performance (higher latency and lower bandwidth). However, recent studies suggest that it should not be treated as a “slower, persistent DRAM”. Compared to DRAM, Optane DC PMM exhibits complicated behaviors and its performance changes based on multiple factors, such as the access size, access type (read vs. write), and degree of concurrency.

Intel Optane DC PMM differs from DRAM in two ways. First, there is a mismatch between the CPU cacheline access granularity (64-byte) and the 3D-XPoint media access granularity (256-byte) in Intel Optane DC PMM. This difference can lead to write or read amplification if the data access is smaller than 256 bytes. Second, to balance the gap in access granularity, the Intel Optane DC PMM implements a small (16KB) write-combining buffer to merge small writes and reduce write amplification. However, the buffer’s limited capacity (16 KB) can cause contention within the device, limiting its ability to handle access from multiple threads simultaneously.

The complex behavior of Intel Optane DC PMM introduces interesting challenges for building a serverless storage service using this technology. Previous works have found that serverless functions vary considerable in multiple ways, including the way they access and process data, and their quality-of-service (QoS) demands. Furthermore, these workloads can spike by orders of magnitude and change dramatically over time. Knowing how these large-scale variations affect the system’s performance and QoS for applications can assist in building an efficient serverless storage service.

Consequently, this thesis addresses the following research questions:

- How does Optane PMM affect the system’s performance when used as persistent storage for serverless functions?
- How does Optane PMM performance under serverless workloads affect the (QoS) for applications?
- How can we overcome the limitations of Optane PMM to make efficient use of the device in a serverless scenario?
- How do we keep the system optimized and compliant with QoS requirements over time as workload shifts occur?

1.3 Contributions

The experiments described in Section 3 provide various helpful insights on the Optane PMM behavior when used as persistent storage for serverless workloads. First, we discover that sharing the Optane PMM among hundreds of serverless functions lead to performance loss (higher latency and lower bandwidth) in the device. This fact was expected given the contention issues experienced by Optane PMM with higher thread counts. Second, we discover that, depending on the workloads, the performance degradation in Optane PMM affects one performance metric more than the other (latency vs. bandwidth). This suggests that QoS of some applications might be affected more than others. Therefore, we conclude the success of Optane PMM should be measured by its capability of meeting the QoS requirements of the current workload.

To help alleviate the limitations of Intel Optane PMM, we introduce a control layer that runs on top of Optane and guides the efficient use of the device under dynamic workloads. Our control layer, called NVM Middleware, is designed to limit the access to persistent memory to reduce its contention. While doing so, the NVM Middleware keeps track of the type of applications running in the system and applies different optimization policies for each one to ensure that their QoS requirements are met. Using machine learning, the

NVM Middleware learns how to scale resources to meet the current demand and dynamically adapts them to changing workloads. We propose using online reinforcement learning algorithms, given that data access patterns in serverless workloads can change over time.

- We present an experimental study that describes the capabilities and limitations of Intel Optane PMM when used as persistent storage for serverless workloads. To our knowledge, Optane PMM has not been tested yet in this scenario.
- We present the NVM Middleware, a control layer promotes the efficient use of Optane PMM, while ensuring that QoS requirements for different type of applications are met.
- We propose a Reinforcement Learning model and framework that allows the NVM Middleware to learn from historical data and adapt resources to changing workloads.
- Finally, we present empirical results that demonstrate the benefits of our solution.

Chapter 2: Background

2.1 Intel Optane DC Persistent Memory

Intel Optane Persistent Memory (PMM) represents a significant advancement in persistent memory technology, bridging the gap between dynamic random-access memory (DRAM) and storage devices [6]. As a result, it has been applied to speed up data-intensive workloads in both commercial and research settings, such as in-memory databases, virtualization software, and high-performance computing [7,8]. This section provides an overview of the architecture, features, benefits, and applications of Intel Optane PMM.

2.1.1 Overview of Intel Optane DC PMM

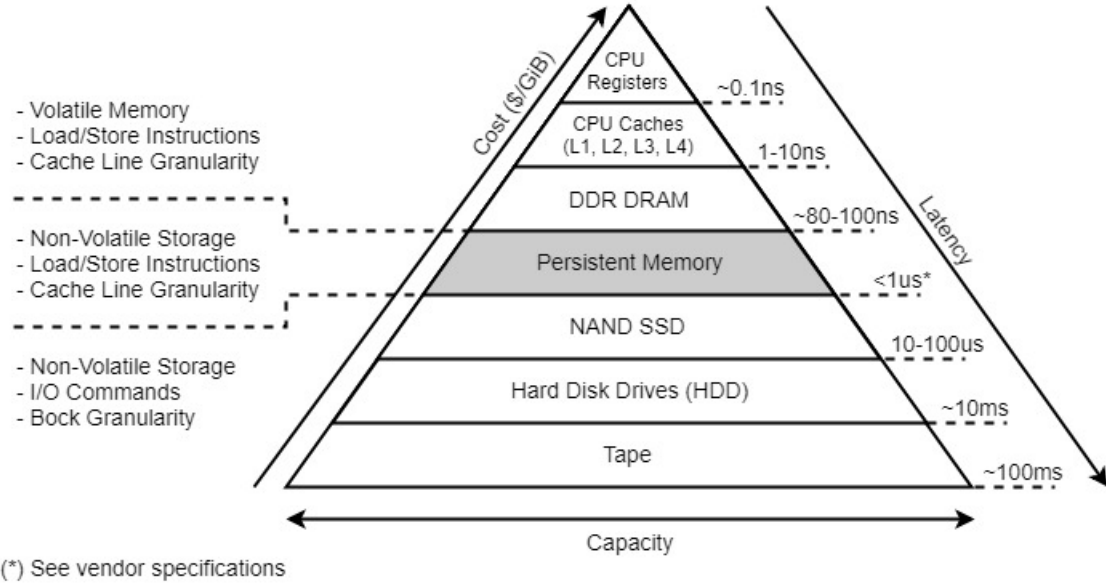


Figure 2.1: Illustration of the Memory-Storage Hierarchy with Persistent Memory. Persistent memory introduces a new tier in the hierarchy, bridging the performance and capacity gap between traditional volatile memory and persistent storage. Adapted from [9].

Persistent memory, also referred to as Non-Volatile Memory (NVM), represents a significant evolution in the memory/storage hierarchy (Figure 2.1), addressing the performance and capacity gap between dynamic random-access memory (DRAM) and traditional storage mediums. This innovative technology combines the characteristics of both DRAM and storage, offering the speed of DRAM and the non-volatile nature of storage devices [6].

Like DRAM, persistent memory is available in the form of Dual In-line Memory Modules (DIMMs), which are directly connected to the memory bus. This direct connection enables applications to access persistent memory with the same ease as traditional DRAM, eliminating the need for frequent data transfers between memory and storage. However, unlike DRAM, persistent memory DIMMs provide significantly greater capacity and retain data even when power is removed, thereby enhancing system performance and enabling fundamental changes in computing architecture [6, 10].

Intel Optane DC Persistent Memory Module (Optane PMM) stands at the forefront of commercial implementations of persistent memory technology, leveraging Intel’s innovative 3D-XPoint technology. Upon its introduction, the Optane PMM offers substantial capacities up to 512GiB and is exclusively supported by Intel Cascade Lake platform. Each processor within this platform is equipped with two integrated memory controllers (iMCs), with each iMC supporting three channels. This architecture seamlessly integrates Optane PMM with DRAM, allowing users to deploy up to one Optane PMM per channel and up to six per CPU socket, thereby enabling extensive memory capacities of potentially up to 3TiB per socket [3, 4].

Similar to conventional DRAM DIMMs, Optane PMMs are positioned on the memory bus and connect directly to the processor’s iMC. The communication protocol between the iMC and the Optane PMM is depicted in Figure 2.2. Communication between the iMC and the Optane PMM occurs via the DDR-T protocol, adapted for persistent memory and operating at cache line granularity (64B). Initial memory access to the Optane PMM is coordinated by the onboard Controller, which manages access to the 3D-XPoint media. Analogous to SSDs, the Optane PMM conducts address translation for wear-leveling and

bad block management, facilitated by the maintenance of an address indirection table (AIT). Following translation, access to the storage media occurs. Notably, with 3D-XPoint access granularity set at 256B, the controller converts 64-byte accesses into 256-byte accesses, inducing write amplification. To mitigate this, the Controller incorporates a 16KB write-combining buffer to merge adjacent writes [3–5].

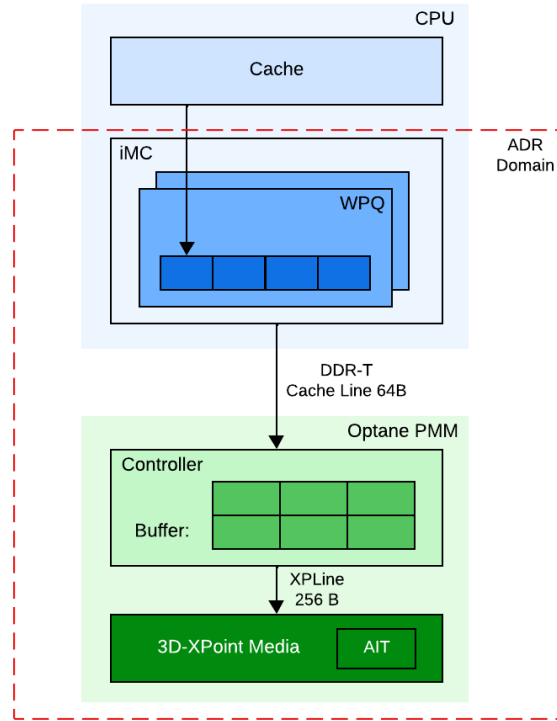


Figure 2.2: Illustration depicting the architecture of Intel Optane Persistent Memory Module (PMM). Adapted from [4].

To ensure data persistence, Intel platforms integrate the iMC and Optane PMM within the asynchronous DRAM refresh (ADR) domain. Intel’s ADR feature ensures that CPU stores that reach the ADR domain will survive a power failures [4]. The iMC manages read and write pending queues for each Optane PMM, with the ADR domain encompassing

the write pending queue. Once data reaches the write pending queue, ADR ensures its persistence within Optane PMM in the event of power failures. The ADR domain excludes the CPU caches, necessitating additional steps beyond simply executing a store instruction to ensure data persistence. To achieve this, CPU stores must be continually flushed using specialized instructions provided by Intel’s Instruction Set Architecture (ISA), including CLFLUSH, CLFLUSHOPT, and CLWB [3, 4, 10].

2.1.2 Performance Characterization

Previous studies [3, 4] conducted an empirical performance assessment of Optane PMM, revealing its nuanced behavior compared to DRAM. They observed that Optane’s performance varies significantly depending on specific access patterns, including access size, type, and concurrency level. Notably, they found that Optane’s read latency is three times slower than that of DRAM, primarily due to Optane’s longer media latency. However, sequential access patterns demonstrate notably improved latency, indicating Optane PMM’s capability to consolidate adjacent requests into single 256-byte accesses. The study also highlights that Optane PMM achieves a maximum random read bandwidth of 6.6 GB/s and a write bandwidth of 2.3 GB/s. Moreover, sequential access further enhances bandwidth performance, exhibiting up to a fourfold increase [3, 4].

An insightful observation highlighted by Izraelevitz et al. [3] is that Optane PMM’s bandwidth can become saturated when utilized in real-world multi-threaded applications, thereby introducing performance overhead. This phenomenon arises due to Optane PMM’s inability to scale performance proportionally with increased thread count, primarily due to contention occurring within the processor’s integrated memory controller (iMC) and Optane PMM’s buffer. Contentious conditions within the buffer exacerbate the frequency of evictions and write-backs to the 3D-XPoint media, resulting in Optane writing more data internally than what the application necessitates. Furthermore, given Optane PMM’s slightly slower performance compared to DRAM, the slower drainage of write pending queues by

Optane PMMs introduces head-of-line blocking effects. As the number of threads concurrently accessing Optane PMM increases, contention on the device escalates, heightening the likelihood of the processor experiencing blocking while awaiting completion of previous store operations [4].

2.1.3 Operating Modes and Applications

Intel Optane persistent memory (PMem) offers two distinct operating modes: Memory mode and App Direct mode.

In Memory mode, Optane PMem serves as a high-capacity main memory without persistence. In this configuration, DRAM is concealed from users and acts solely as a cache for Optane PMem, seamlessly managed by the operating system [4].

Conversely, in App Direct mode, Optane PMMs are directly exposed to the operating system as independent persistent memory devices, thus enabling their utilization for persistent storage [3]. Functionally, the operating system perceives DRAM and Optane PMem as distinct memory pools, with the latter offering data persistence. Applications can access Intel Optane persistent memory through direct load/store operations or via a file system configured with the dax (direct access) option. Such a file system is termed as a PM-aware file system, facilitating direct access to persistent memory without relying on the page cache [10, 11].

In the context of this thesis, Optane PMem is exclusively employed in App Direct Mode, coupled with a PM-aware file system to harness its storage capabilities.

2.1.4 Programming

In the realm of persistent memory technology, maintaining data consistency across runtime and system reboots is essential. To address this challenge, prior research underscores the necessity for applications leveraging persistent memory to implement transactions that are atomic, consistent, thread-safe, and resilient to system failures—a paradigm akin to ACID

transactions in database systems. However, achieving such robustness in real-world scenarios poses significant complexity. Recognizing this, Intel has developed the Persistent Memory Development Kit (PMDK) to tackle this challenge [6, 10].

PMDK comprises a comprehensive suite of libraries and tools tailored for both application developers and system administrators, aiming to streamline the management and utilization of persistent memory devices. Drawing on the SNIA NVM Programming model [12] as its foundation, these libraries extend its capabilities to varying extents. Some libraries offer simplified wrappers around operating system primitives, facilitating ease of use, while others provide sophisticated data structures optimized for persistent memory usage [6].

In the scope of this thesis, we leverage pmemkv [13], a persistent local key-value store provided by PMDK. Designed with cloud environments in mind, pmemkv complements PMDK’s suite of libraries with cloud-native support, abstracting the intricacies of programming with persistent memory through a familiar key-value API. Notably, pmemkv distinguishes itself from traditional key-value databases by enabling direct access to data. This means that reading data from persistent memory circumvents the need for copying it into DRAM—an approach that significantly enhances the performance of applications leveraging persistent memory [6].

2.2 Serverless Computing

Serverless computing, a prominent execution model within cloud computing, revolutionizes the deployment process by allowing developers to deploy code without the need for provisioning or managing server infrastructure. Although termed “serverless,” this model still utilizes servers provided by cloud vendors to execute developers’ code. However, the distinguishing feature lies in the abstraction of infrastructure management from the developer’s perspective. Developers no longer concern themselves with resource provisioning, scaling, fault tolerance, monitoring, or security patches; instead, they focus solely on code development. Cloud providers take on the responsibility of handling these infrastructure-related

tasks on behalf of their customers. Consequently, developers are charged based on the execution time and resources consumed during their code invocations, offering a pay-per-use billing model [1, 14, 15].

Serverless computing has witnessed remarkable growth in recent years, emerging as a prominent paradigm for cloud application development. While it was originally developed for web microservices and IoT, its elasticity and billing granularity make it appealing for other types of applications. Recent studies have highlighted the benefits of transitioning more applications to serverless computing platforms, with applications focused on video processing [16], data analytics [15, 17, 18], machine learning [19, 20], and storage applications [1, 21].

2.2.1 Function-as-a-Service (FaaS)

At the heart of serverless computing lies Function-as-a-Service (FaaS), introduced by AWS Lambda in 2015. Since then, various commercial and open-source alternatives have emerged, including Google Cloud Functions, Azure Functions, Apache OpenWhisk, and others. FaaS enables developers to express application logic as stateless functions written in high-level languages such as Java, Python, C, or C++. These functions, known as serverless functions, are packaged together with their dependencies and submitted to the serverless platform. Additionally, developers associate events with each serverless function, such as HTTP requests, file uploads, database triggers, and more. Upon the occurrence of a trigger, the cloud provider promptly executes the associated serverless function, offering a scalable and event-driven approach to application development and deployment [22–25].

Workload Characterization

Previous research has underscored the dynamic and unpredictable nature of Function-as-a-Service (FaaS) workloads within serverless computing environments. Typically, applications in serverless computing are composed of interconnected serverless functions, each serving a specific logical purpose. Analyzing these workloads often involves examining real-world

FaaS provider logs to discern patterns and characteristics. Cloud providers face significant challenges in predicting the next function invocation due to the diverse array of triggers employed by applications. Moreover, the heterogeneity of these applications results in substantial variations in invocation frequency, data access sizes, and usage patterns. Data access sizes can range from mere bytes to gigabytes, while invocation frequencies can span several orders of magnitude, with some functions being infrequently called. Learning the invocation patterns of rarely invoked functions is particularly challenging. Additionally, a significant portion of data accesses exhibit bursty behavior, leading to rapid surges in I/O requests as multiple function instances are dynamically spun up to meet application demands. This phenomenon often results in short-lived bursts of intense activity within applications [14, 15, 26].

2.2.2 Storage for FaaS

Serverless providers enforce a restriction on direct communication between serverless functions, necessitating the adoption of remote storage mechanisms for data interchange among them [1, 14, 15].

One approach to facilitate data exchange between serverless functions is through the utilization of serverless storage, a framework within cloud computing that abstracts the intricacies of managing storage infrastructure from developers. With serverless storage, developers harness storage services provided by cloud vendors like AWS S3, Google Cloud Storage, Azure Blob Storage, AWS DynamoDB, and others. These services empower developers to store and retrieve data via APIs or SDKs without the burden of managing servers or storage clusters. Moreover, serverless storage services offer features such as automatic scaling, data durability, and pay-per-use pricing models, enabling developers to seamlessly adjust their storage resources in line with demand without the need for upfront provisioning or capacity planning. This scalability is of paramount importance for applications characterized by fluctuating storage requirements or unpredictable workloads [27–33].

Despite the high scalability of existing serverless storage solutions, they are primarily optimized for durability rather than performance [1,15,21]. Studies have demonstrated that object-based storage solutions like AWS S3 may exhibit latencies of up to 10 milliseconds for small object reads or writes [34]. Similarly, key-value databases such as AWS DynamoDB, Google Cloud Datastore, and Azure Cosmos provide high throughput but may entail high expenses and lengthy scaling processes [1].

In scenarios demanding enhanced performance, developers may opt for in-memory key-value stores like AWS ElastiCache [35]. These solutions offer low access latencies and high throughput but incur higher costs associated with DRAM. Additionally, they do not provide persistence, meaning that data is not retained in the event of a system failure. However, a notable drawback is their lack of autoscaling capabilities, necessitating manual management and scaling of clusters when integrated with serverless functions [1,15,21].

2.2.3 Service Level Agreements

Service level agreements (SLAs) are integral to cloud storage as they establish the terms and conditions governing the quality of service between cloud providers and customers. These agreements, formalized through SLAs, represent negotiated contracts wherein both parties delineate system-related attributes, including the client’s anticipated request characteristics and the expected performance of the storage service under these conditions. SLAs serve to define the repercussions for any deviation from the predetermined service levels, often entailing service credits, refunds, or other forms of redress. Such measures incentivize providers to fulfill their obligations and redress any disruptions or deficiencies in service promptly [29,36,37].

In the realm of storage services, latency and throughput-related SLAs are of paramount importance. While conventional practice often involves describing performance SLAs using mean or median metrics, these measures may not adequately ensure a consistently positive user experience across all customers. Instead, a more effective approach involves assessing

performance SLAs in terms of tail (99th) percentiles, thereby prioritizing the resolution of exceptional cases and optimizing service quality for all users [29].

2.3 Reinforcement Learning

Reinforcement Learning (RL) is a subfield of machine learning concerned with learning optimal decision-making policies through interactions with an environment [38].

2.3.1 Overview of Reinforcement Learning

The fundamental concept underlying RL is the notion of an agent, which takes actions in an environment and receives feedback in the form of rewards, indicating the quality of its decisions. The agent’s objective is to learn a policy that maximizes cumulative rewards over time. Moreover, the agent is not provided with explicit instructions on which actions to take; instead, it must discover the actions that lead to the highest rewards by trying them.

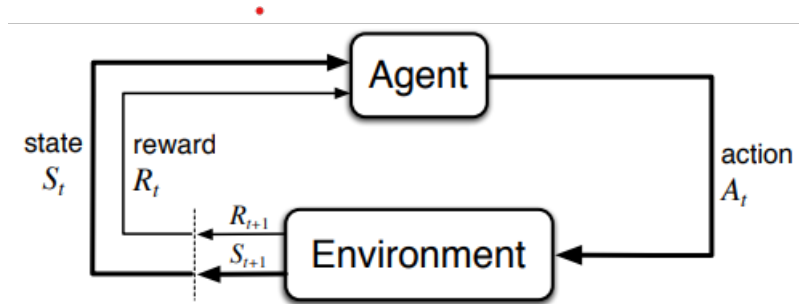


Figure 2.3: Workflow illustrating the interaction between the agent and the environment in reinforcement learning. Adapted from [38].

Figure 2.3 presents a schematic representation of a standard reinforcement learning scenario. In discrete time steps, the agent perceives the current state s_t from the set of all possible states S . It then selects an action a_t from the available actions $A(s_t)$ in the current

state. The environment transitions to a new state s_{t+1} , and the agent receives a reward r_t associated with the transition (s_t, a_t, s_{t+1}) .

The agent’s behavior is governed by its policy, which maps perceived states to actions. The ultimate aim is to learn an optimal or near-optimal policy that maximizes the cumulative reward.

2.3.2 Q-Learning

One of the foundational algorithms in RL is Q-Learning, introduced by Watkins in 1989 [39]. The algorithm belongs to the class of model-free RL algorithms, meaning it learns directly from experience without requiring a model of the environment dynamics [40].

At the core of Q-Learning is the Q-value function, denoted as $Q(s, a)$, which represents the expected cumulative reward the agent will receive by taking action a in state s and following an optimal policy thereafter. The objective of Q-Learning is to iteratively update the Q-values based on observed transitions and rewards, eventually converging to the optimal Q-values that maximize long-term rewards.

The Q-Learning algorithm proceeds as follows: the agent interacts with the environment by selecting actions based on its current estimate of the Q-values. Upon taking an action, the agent observes the resulting reward and the next state. It then updates the Q-value of the previous state-action pair using the observed reward and the estimated value of the next state.

The Q-value update rule in Q-Learning is based on the Bellman equation, which expresses the relationship between the Q-values of successive states [40]:

$$Q(s, a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'} Q(s', a') \right)$$

Here, α is the learning rate, determining the extent to which new information overrides the old one, and γ is the discount factor, representing the importance of future rewards

relative to immediate rewards. The term $r + \gamma \cdot \max_{a'} Q(s', a')$ is known as the temporal-difference (TD) target, combining the immediate reward r with the discounted maximum Q-value of the next state s' [40].

2.3.3 Function Approximation using Linear Regression Models

One of the key advantages of Q-Learning is its simplicity and ease of implementation. It requires only a table to store the Q-values, making it computationally efficient for small state and action spaces. However, Q-Learning faces challenges in environments with large state spaces, as maintaining a lookup table becomes infeasible due to memory and computational constraints.

Function approximation is a fundamental technique in reinforcement learning (RL) aimed at approximating the Q-Value function when dealing with large state or action spaces where tabular representations become impractical [40]. This approach allows RL agents to generalize from observed states to unseen states, facilitating decision-making in unexplored regions of the state space.

In the context of RL, linear regression models are commonly used for function approximation [38]. These models approximate the Q-value function by leveraging a weighted linear combination of features, with each feature capturing a distinct aspect of the state space. Employing gradient-descent methods, notably stochastic gradient descent, enables iterative refinement of the parameters governing the linear function, aimed at minimizing a predefined loss function. This iterative optimization process empowers the model to progressively enhance its predictive accuracy and capture intricate patterns within the state-action space.

Hyperparameter tuning is a critical aspect of training linear regression models in RL [41]. Hyperparameters, such as the learning rate, regularization strength, and feature scaling, significantly impact the performance and convergence of the models. A systematic approach to hyperparameter tuning involves experimenting with different combinations of hyperparameters, evaluating the performance of the trained models on a validation set, and

selecting the optimal hyperparameters based on predefined criteria, such as validation error or performance metrics [40].

2.3.4 Exploration-Exploitation Tradeoff

The exploration-exploitation tradeoff poses a significant challenge in reinforcement learning [38]. The agent must strike a balance between exploring unfamiliar actions to gather information and exploiting known actions for immediate rewards. Finding this balance is crucial for effective learning and task performance, as the agent gradually favors actions with higher expected rewards.

One classic strategy for balancing exploration and exploitation is the epsilon-greedy (e-greedy) algorithm [38]. The e-greedy policy selects the action that maximizes the estimated value with probability $1 - \epsilon$ (exploitation) and selects a random action with probability ϵ (exploration). This approach ensures that the agent continues to explore the environment while gradually exploiting more rewarding actions as it gains knowledge.

Decayed e-greedy methods aim to strike a balance between exploration and exploitation by gradually reducing the exploration rate ϵ as the agent gains more experience or as the training progresses [38]. This decay encourages the agent to explore the environment more extensively in the early stages of learning while gradually shifting towards exploitation as it becomes more knowledgeable.

2.3.5 Reward shaping

Reward shaping is a technique in reinforcement learning (RL) aimed at accelerating learning by modifying the reward signal provided to the agent. Traditional RL algorithms rely solely on sparse reward signals, which can make learning slow and inefficient, especially in complex environments. Reward shaping addresses this issue by providing additional, shaped rewards that guide the agent towards desirable behaviors. These shaped rewards are designed to provide more informative feedback to the agent, encouraging it to explore the state-action space more effectively. However, reward shaping must be carefully designed

to avoid unintended consequences such as overfitting to the shaped rewards or incentivizing undesirable behaviors [40].

Chapter 3: NVM Middleware: A control layer for persistent memory

As we have discussed, the release of Intel Optane PMM opens a major opportunity for serverless storage services. This memory technology provides a unique combination of affordable larger capacity, high-performance, and support for data persistence [42]. When configured in App-Direct mode, the Optane DIMM and DRAM DIMMs act as independent memory resources under direct load/store control of the applications. This allows the Optane PMM capacity to be used as byte-addressable persistent memory that is mapped into the system application space and directly accessible by applications. Together, these advantages enable Optane PMM to be used as persistent storage with memory-like speeds.

Unfortunately, the resource contention observed within Optane PMM can impose serious performance and contractual implications for a multi-tenant serverless storage service. Given the hallmark autoscaling features of serverless computing, the memory’s limited ability to handle accesses from multiple threads can degrade the overall system’s performance when workload spikes occur. Furthermore, these storage systems make efficient use of their infrastructure by allowing multiple users, or tenants, to share the physical resources. The performance degradation caused by Optane PMM can lead tenants to experience significant performance variations. The latter inhibits service providers from offering certain service level agreements.

To reduce the contention effect, previous studies recommend limiting the number of threads that access Optane PMM simultaneously. In [4], Yang et. al they improve the performance of an NVM-aware file system by limiting the number of writer threads that access each Optane DIMM. Similarly, Ribbon [5] controls the number of threads performing CLF and adjusts this number dynamically at runtime. While this approach provides a viable

solution, it introduces management problems for a system administrator of a multi-tenant serverless storage.

Given the complexity of serverless computing workloads, implementing efficient concurrency control mechanisms for optimizing an Optane-based serverless storage service is a challenging task. These challenges are discussed in section 3.1, but in short, service providers have three crucial tasks when implementing these control mechanisms. First, they must provide predictable performance, ensuring that all the SLAs from different workloads are met. Second, they must scale resources transparently to meet the current workload demand. Finally, they must come up with policies that allow their system to adapt quickly to sudden workload shifts. To this end, we propose a solution that takes on these responsibilities from the service providers.

In this work, we present a shim layer that addresses the shortcomings of Intel Optane PMM highlighted above, while meeting the different service level agreements from multiple tenants under shifting workloads. Our shim layer, called NVM Middleware, distinguishes between latency-critical and throughput-oriented workloads and applies different concurrency control mechanisms for each one. This enables the system to reduce the contention on the memory device, as well as the interference among workloads with different service level agreements. In addition, we propose the development of a reinforcement learning agent to adapt the NVM Middleware quickly to changing workloads. The agent takes into account the characteristics and service level agreements and learns from past experiences to scale resources accordingly.

3.1 Motivation

In this section, we discuss the pain points of controlling the number of threads to optimize Optane PMM within a serverless storage service and explain the design goals of the NVM Middleware.

3.1.1 Concurrency Control Challenges in a serverless storage service

When building an Optane PMM based serverless storage service, optimizing the memory’s performance is just the start. Early works in serverless computing have identified several tasks that a storage service must perform efficiently to meet the demands of serverless computing [1, 14, 15, 36, 43, 44]. As a result, service providers must ensure that their concurrency control policies do not interfere with these design goals. In this work, we focus on three challenges faced by service providers when designing a high-performance storage service based on Optane PMM.

Support for a wide heterogeneity of applications. In serverless computing, users typically deploy their applications as a collection of serverless functions that share data among them using remote storage. Previous studies suggest that these applications vary considerably in the way store, distribute, and process data. This diversity is reflected in multiple ways, such as data access size [14, 15], data access patterns [14], and their performance requirements [1, 36]. Therefore, service providers face the challenge of tuning the concurrency level to support many types of applications. In this work, we argue that considering the workload characteristics is key for tuning the system efficiently. The allocation of resources can vary depending on the workload type.

Compliance with Service Level Agreements. The success of a storage service relies on its ability to comply with various service level agreements (SLAs). SLAs play a critical role in governing the relationship between the storage provider and its customers. They help establish clear expectations between both parties regarding the quality of storage service. Therefore, service providers face the challenge of staying in compliance with these SLAs while they seek to optimize Optane PMM.

Automatic and transparent scaling. Serverless workloads are extremely unpredictable. These workloads can launch hundreds of functions instantaneously to meet application demands [43]. Furthermore, the data access patterns of the applications can change dramatically over time [14, 44]. Service providers face the challenge of scaling the resources,

such as number of threads, automatically to meet the demands of changing workloads. In addition, they must ensure that the system adapts quickly enough to avoid missing SLAs.

3.1.2 NVM Middleware Design Overview

We design NVM Middleware with three main design goals.

Workload-aware Contention Management. We focus our work on two main types of workloads: interactive and batch applications. Interactive applications, such as web-based platforms, enable real-time interactions between the user and the application. Low latency is critical to ensure that the user input is processed quickly, and feedback is delivered in real-time. On the other hand, batch applications, such as data analytics jobs, facilitate efficient processing of large-scale data. These workloads prioritize high throughput to process large volumes of data efficiently.

The NVM Middleware leverage insights about the workload characteristics, resource demands, and performance requirements of applications to make informed decisions about resource allocation and contention resolution. By dynamically adjusting resource allocation and contention resolution mechanisms based on the workload characteristics, the NVM Middleware mitigates contention-induced performance degradation and ensures efficient resource sharing among co-located applications. This adaptive approach enables the NVM Middleware to allocate resources judiciously to maximize overall system efficiency and meet diverse performance requirements of both interactive and batch applications. By using the content-aware contention management offered by the NVM Middleware, a storage system using Optane PMM can effectively balance the needs of different workload types, ensuring optimal performance and resources utilizing in multi-tenant environments.

SLA-driven autoscaling policies. The NVM Middleware leverages SLAs, which define the quality-of-service parameters agreed up between the service provider and their customers, to dynamically adjust contention resolution mechanisms in response to changes in service level agreement metrics. It continuously monitors SLA metrics, such as 99th

latency and throughput, and evaluates its own performance against predefined SLA targets. This real-time monitoring allows the NVM Middleware to detect deviations from SLA requirements and triggers scaling actions to dynamically adjust resource allocation. By aligning resource provisioning with SLA requirements, the NVM Middleware can ensure a consistent and reliable performance from Optane PMM, even under dynamic workload changes.

RL-driven autoscaling policies. Besides leveraging SLAs to dynamically provision resources and adjust contention resolution mechanisms, our solution proposes the use of Reinforcement Learning to learn from past experiences and predict future behaviors. These RL-driven policies enable the NVM Middleware to adapt to changing workload patterns over time and meet SLAs objectives more effectively than traditional threshold-based approaches []. Moreover, given the dynamic and unpredictable of serverless workloads, we propose a model-free algorithm, Q-Learning, to continuously learn the optimal policy based on observed experiences, allowing the NVM Middleware to adapt to new scenarios without needing to explicitly model them.

3.2 Architecture

Figure 3.1 provides an overview of the NVM Middleware architecture. Positioned as a middle layer connecting user applications with Optane PMM, its design is tailored for seamless integration within a storage service, serving as an optimization layer specifically targeting Optane PMM. It comprises a request handler, two concurrency thread pools, and a monitoring and resource management module.

The request handler serves as the primary interface for handling user I/O requests. Upon receipt, it segregates requests into two distinct non-blocking First-In-First-Out (FIFO) queues: one tailed for latency-sensitive requests and the other for throughput-centric ones. Leveraging insights into workload characteristics, the handler intelligently allocates requests to the appropriate queue. Moreover, each queue is assigned a dedicated pool of worker threads tasked with dispatching I/O requests to Optane PMM using PMEMKV. Notably,

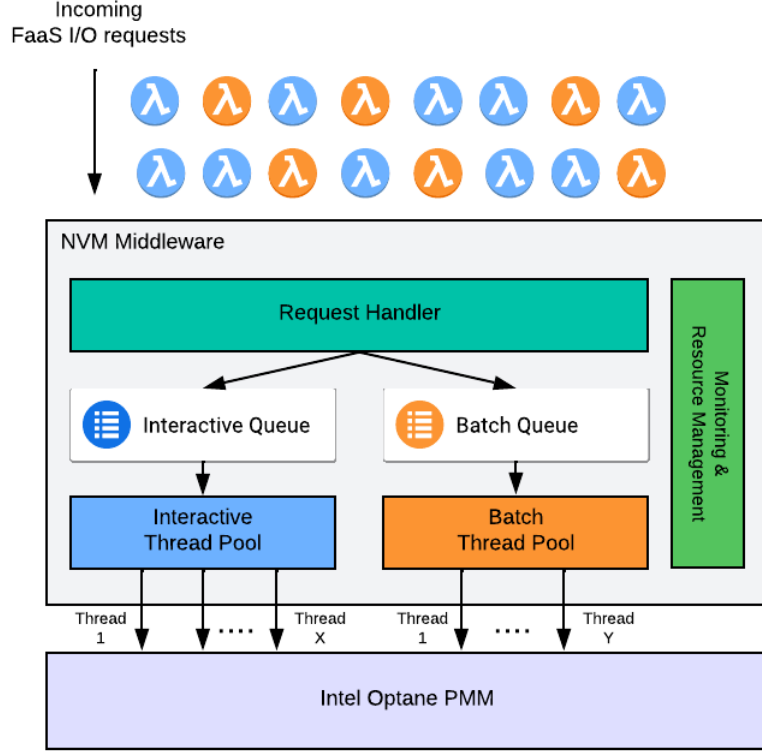


Figure 3.1: Illustration depicting the architecture of the NVM Middleware, which serves as an optimization layer for Intel Optane DC PMem. The NVM Middleware governs the concurrency level on the device, categorizing incoming I/O requests based on their type (blue for interactive and orange for batch) and assigning distinct worker threads for interactive or batch workloads. The number of threads for each workload type is dynamically configurable to adapt to the system’s requirements.

these thread pools operate independently and are dynamically managed and scaled by the Reinforcement Learning agent to meet predetermined latency and throughput goals.

The Monitoring and Resource Management module offers an interface to monitor system load and SLA performance metrics. This module initiates a separate control thread tasked with gathering data on key parameters within the NVM Middleware, such as 99th latency, throughput, and system load. Utilizing this information, the RL agent makes data-driven

decisions regarding optimal thread pool scaling. Subsequently, these decisions are communicated to the Monitoring and Resource Management module, which executes the required actions within the NVM Middleware.

3.3 Programming Interface

Table 3.1: Overview of the NVM Middleware programming interface, categorized by functionality. *System* functions facilitate resource management within the middleware, including database initialization and thread pool management. *Storage* functions provide a fundamental key-value interface for submitting requests aimed at accessing persistent memory. *RL* functions provide utilities for the reinforcement learning process, enabling monitoring of system statistics, state retrieval, and triggering scaling actions.

Category	API Name	Functionality
System	<code>start(db, interactiveThreads, batchThreads)</code>	Create the PMEMKV database, start thread pools, and initiate NVM Middleware’s monitoring.
System	<code>stop()</code>	Close PMEMKV database, stop thread pools, and stop NVM Middleware’s monitoring.
Storage	<code>get(key, mode)</code>	Submits request to retrieve a key from persistent memory.
Storage	<code>put(key, value, mode)</code>	Submits request to write a key to persistent memory.
RL	<code>get_stats()</code>	Provides the 99th percentile and throughput observed by the NVM Middleware.
RL	<code>get_state()</code>	Provides the current state within the NVM Middleware.
RL	<code>perform_action(action)</code>	Triggers a scaling action.

Table 3.3 outlines the NVM Middleware’s programming interface, presenting a set of functions designed to facilitate interaction with a storage system and the reinforcement learning agent.

The *start* function initializes the PMEMKV database, initializes the thread pools with an initial thread count, and triggers the system monitoring within the Monitoring and Resource Management Module. In contrast, the *stop* function terminates the database connection, halts all threads in the thread pools, and stops system monitoring. Furthermore,

the *get* and *put* functions facilitate key-value interactions with the persistent memory, allowing for read and write operations. These functions are designed to accommodate hints regarding the request type (e.g., latency-sensitive or throughput-oriented), aiding the request handler in queue allocation.

The *get_stats* function furnishes insights into the 99th percentile and throughput observed by the NVM Middleware at any given moment. Similarly, the *get_state* function provides real-time state information as outlined in Table 3.4.2. Finally, the *perform_action* function accepts scaling actions detailed in Table 3.4.2 and initiates the corresponding action within the NVM Middleware.

3.4 Reinforcement Learning Component

In this section, we discuss the Q-learning algorithm used by the Reinforcement Learning agent to dynamically adjust the number of threads assigned to each thread pool. The agent’s goal is to find the best combination of threads that meets predetermined latency and throughput SLAs while minimizing contention and ensuring efficient utilization of Intel Optane PMM.

3.4.1 Integration with the NVM Middleware

Figure 3.2 offers a visual representation of the interaction between the reinforcement learning (RL) agent and the NVM Middleware. At each time step, the NVM Middleware receives a diverse influx of requests, spanning both latency-sensitive and throughput-oriented tasks. These requests necessitate translation into actionable I/O commands directed towards the Intel Optane Persistent Memory Module (PMM).

Concurrently, the RL agent adeptly captures the environment’s current state, leveraging real-time workloads’ characteristics and performance metrics provided by the monitoring module. Utilizing this information, the agent orchestrates the selection of an optimal action, guiding the dynamic adjustment of threads within the interactive and batch thread pools.

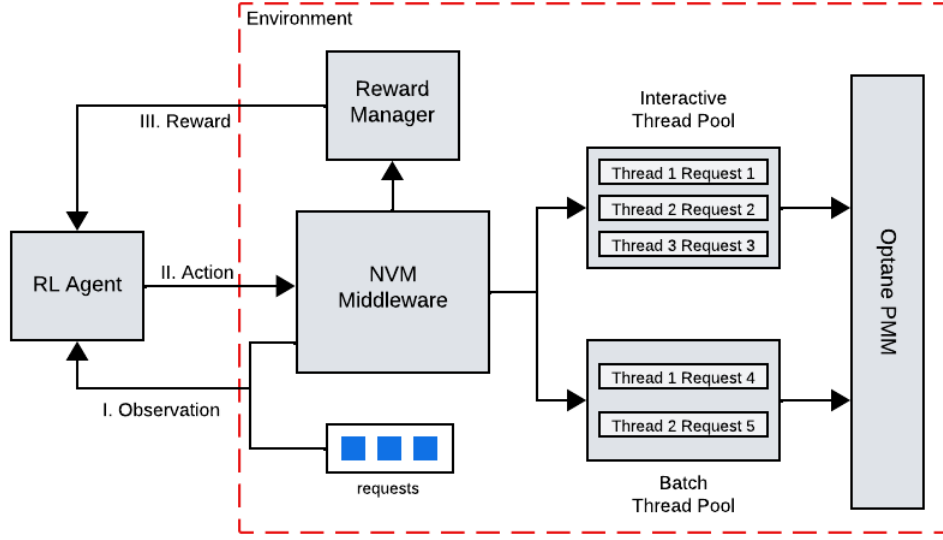


Figure 3.2: Interaction between the RL agent and the NVM Middleware. At each time step, the RL agent observes (I) the state of the environment and takes a scaling action (II) on the NVM Middleware based on its learned policy. Subsequently, the agent receives a reward (III), which is utilized to iteratively refine its policy.

This adaptive decision-making process is exemplified by actions like augmenting the count of interactive threads to address evolving workload demands.

Following action selection, the NVM Middleware’s resource management module implements the chosen course of action, fine-tuning the NVM Middleware’s interactive and batch threads to efficiently handle incoming user requests. Upon the completion of each time step, the action’s effectiveness is rigorously assessed against predefined service level agreement (SLA) targets, yielding a reward signal generated by a reward manager.

This reward serves as invaluable feedback for the RL agent, empowering iterative policy updates aimed at refining decision-making strategies in subsequent time steps. Thus, the presented framework embodies a recursive learning cycle, wherein the RL agent continuously hones its behavior through real-world interactions, ensuring adaptive responsiveness to evolving workload dynamics.

3.4.2 Reinforcement Learning Model

State Space

Table 3.2: State representation for the reinforcement learning model. The number of interactive and batch threads is capped at 32, as measurements showed performance degradation with more than 32 threads. The interactiveQueueSize and batchQueueSize represent the number of requests in the respective queues within the NVM Middleware. The block sizes and write ratios are determined by analyzing the I/O requests received from client workloads.

Name	Description	Values
interactiveThreads	Number of (interactive) threads assigned to the interactive thread pool.	$1 \leq \text{interactiveThreads} \leq 32$
batchThreads	Number of (batch) threads assigned to the batch thread pool.	$1 \leq \text{batchThreads} \leq 32$
interactiveQueueSize	Number of requests in the interactive queue.	$\text{interactiveQueueSize} \in \mathbb{R}^+$
batchQueueSize	Number of requests in the batch queue.	$\text{batchQueueSize} \in \mathbb{R}^+$
interactiveBlockSize	Average block size of interactive workload.	$\text{interactiveBlockSize} \in \mathbb{R}^+$
batchBlockSize	Average block size of batch workload.	$\text{batchBlockSize} \in \mathbb{R}^+$
interactiveWriteRatio	Proportion of write requests compared to read requests in the interactive workload.	$\text{interactiveRWRatio} \in \mathbb{R}^+$
batchWriteRatio	Proportion of write requests compared to read requests in the batch workload.	$\text{batchRWRatio} \in \mathbb{R}^+$

Table 3.4.2 presents the features of our state representation. At each time step t , we define the state s_t as a tuple:

$$s_t = (\text{interactiveThreads}_t, \text{batchThreads}_t, \text{InteractiveQueueSize}_t, \text{batchQueueSize}_t, \\ \text{interactiveBlockSize}_t, \text{batchBlockSize}_t, \text{interactiveRWRatio}_t, \text{batchRWRatio}_t)$$

where $s_t \in S$ represents the state space. The tuple encapsulates the key features characterizing the system's current state, including the number of interactive and batch threads, number of pending requests in the queues, individual workload block sizes, and write ratio for both interactive and batch workloads.

Action Space

Table 3.3: Possible actions in the action space of reinforcement learning and their effects on the NVM Middleware’s concurrency control mechanism.

Action	Effect on Interactive Threads	Effect on Batch Threads
0	No change	No change
1	Increase by 1	No change
2	Decrease by 1	No change
3	No change	Increase by 1
4	No change	Decrease by 1
5	Increase by 1	Increase by 1
6	Increase by 1	Decrease by 1
7	Decrease by 1	Increase by 1
8	Decrease by 1	Decrease by 1

Table 3.4.2 illustrates the feasible actions within the action space. Each action corresponds to a potential adjustment in the number of interactive and batch threads. The table enumerates nine distinct actions, each with its respective effect on the number of interactive threads and batch threads.

Mathematically, the set of actions A is defined as $A = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ for a given state $s_t \in S$.

Reward

To guide the optimization process of the reinforcement learning agent, we establish an algorithm (Algorithm 1) to calculate a reward value based on observed and target latency and throughput metrics. This algorithm, outlined below, serves as a crucial component in training the RL agent to make informed decisions.

1. Lines 1-5 define goals, scaling factors, and penalties. The observed and target latency (lat , lat_goal) and throughput (tp , tp_goal) metrics are scaled to a normalized range using scaling factors (max_scale_lat , max_scale_tp) and minimum scale (min_scale).

This normalization process ensures that both metrics contribute proportionally to the reward calculation.

2. Lines 6-7 compare the scaled latency (*lat*) and throughput (*tp*) metrics against the scaled target values for latency (*lat_goal*) and throughput (*tp_goal*). The absolute differences between observed and target values are computed to quantify the error in latency (*error_lat*) and throughput (*error_tp*).
3. Lines 8-12 determine the reward based on three distinct scenarios. Firstly, if both latency and throughput goals are achieved, a high positive reward is assigned. Secondly, if both goals are not met, a low negative reward is assigned, taking into account both latency and throughput errors. The disparity in penalties, represented by *lat_penalty* and *tp_penalty*, ensures that both types of errors contribute proportionately to the overall reward. Thirdly, if only the latency goal remains unmet, a low negative reward is assigned, incorporating the latency penalty and error. Finally, if only the throughput goal is unmet, a similar low negative reward is assigned, encompassing the throughput penalty and error.

3.4.3 Training Methodology

Environment Design

The environment architecture designed for training and evaluating the RL agent is depicted in Figure 3.3. This architecture comprises several key components, including an interactive multi-threaded application, a batch multi-threaded application, the NVM Middleware, and Intel Optane PMM.

To simulate a multi-tenant serverless scenario, both applications are executed concurrently. Workload patterns for each application are derived from collected serverless traces. To emulate high concurrency levels typical in serverless environments, multiple threads within each application are employed to dispatch requests to the NVM Middleware via the

API described in Section 3.3. Meanwhile, the NVM Middleware processes these requests in accordance with the workflow outlined in Section 3.2.

In order to model the time steps inherent in an RL process, the environment organizes the applications' requests into 1-second windows, processing one window per time step. Figure 3.4 illustrates the interactions between the RL agent and the environment at each time step. Beginning with a state observation from the preceding step, the agent communicates the intended action to the environment. Subsequently, the environment relays this action to the NVM Middleware, which then allocates resources accordingly. Upon successful execution of the action, the environment initiates processing for the next window of requests. Once all requests within the window are handled, the environment gathers metrics from the NVM Middleware and furnishes a new state observation along with a reward signal to the agent. The agent utilizes this reward to update its policy, perpetuating the iterative learning process.

Function Approximation

To address the challenge posed by the continuous state space in our environment, traditional Q-learning approaches become impractical due to the vast number of states that cannot be feasibly mapped into a Q-table. Consequently, we employ function approximation techniques to estimate the value of each action based on the state.

Specifically, we train nine separate linear regression models, each corresponding to one of the available actions, using stochastic gradient descent. This approach allows us to capture the underlying patterns in the data and generalize across states not encountered during training, enabling our agent to make informed decisions even in novel situations.

However, selecting appropriate hyperparameters for our regression models presents a significant challenge. Online training alone is insufficient for accurately assessing model performance, as it can be time-consuming and computationally intensive. To overcome this limitation, we adopt a batch learning approach with offline historical data.

By leveraging historical data collected from the environment, we can tune our models' hyperparameters and incorporate prior knowledge into our RL agent. This approach accelerates the learning process by bootstrapping our models with valuable insights gained from past experiences [45, 46].

To construct our dataset, we deploy a non-optimal agent that performs random actions in the environment, capturing state-action-reward transitions. Following established machine learning practices, we split the dataset into training and testing sets and employ 5-fold cross-validation on the training set to evaluate model performance rigorously.

Additionally, we preprocess the features by standardizing them using the standard scaler and apply polynomial preprocessing to enhance the model's ability to capture nonlinear relationships within the data.

Proposed Q-Learning Algorithm

Algorithm 2 outlines the Q-Learning process for training an agent to make optimal decisions in an environment. It takes the bootstrapped Q-value models M_a for all actions a and outputs the new learned models after training.

The algorithm initializes the training parameters and then iterates over a specified number of episodes. Within each episode, the environment is reset, and the agent interacts with it until the episode is complete. At each step, the agent observes the current state s_t , selects an action a_t based on an ϵ -greedy policy, takes the action, and observes the resulting reward r and next state s_{t+1} .

The Q-value models are updated based on the observed reward and next state. If the episode is not done, the target Q-value is calculated using the reward and the maximum Q-value for the next state. If the episode is done, the target Q-value is simply set to the reward.

The model for the selected action a_t is updated using the target Q-value, and the state is updated to the next state. Additionally, the exploration rate ϵ is decreased according to an exploration schedule.

3.5 Implementation

The NVM Middleware, detailed in Section 3.3, is implemented using C++. We leverage PMEMKV from the Persistent Memory Development Kit [6] to facilitate reading and writing data into Intel Optane PMM. To manage concurrent operations efficiently, we utilize the non-locking, concurrent queue provided by the Intel Threading Building Blocks [47] library for both the interactive and batch queues.

For the RL Environment, as described in Section 3.4.3, we adopt a hybrid approach employing C++ and Python. The environment itself is constructed in C++, aligning with the specifications outlined in Section 3.4.3. Conversely, the RL agent and the Q-Learning algorithm, also discussed in the same section, are developed using Python. We leverage the SGDRegressor model from the Scikit-learn[48] library to facilitate the representation of our linear regression models for function approximation. Additionally, we employ Scikit-learn for hyperparameter tuning. To seamlessly integrate the C++ and Python components, we utilize pybind11[49].

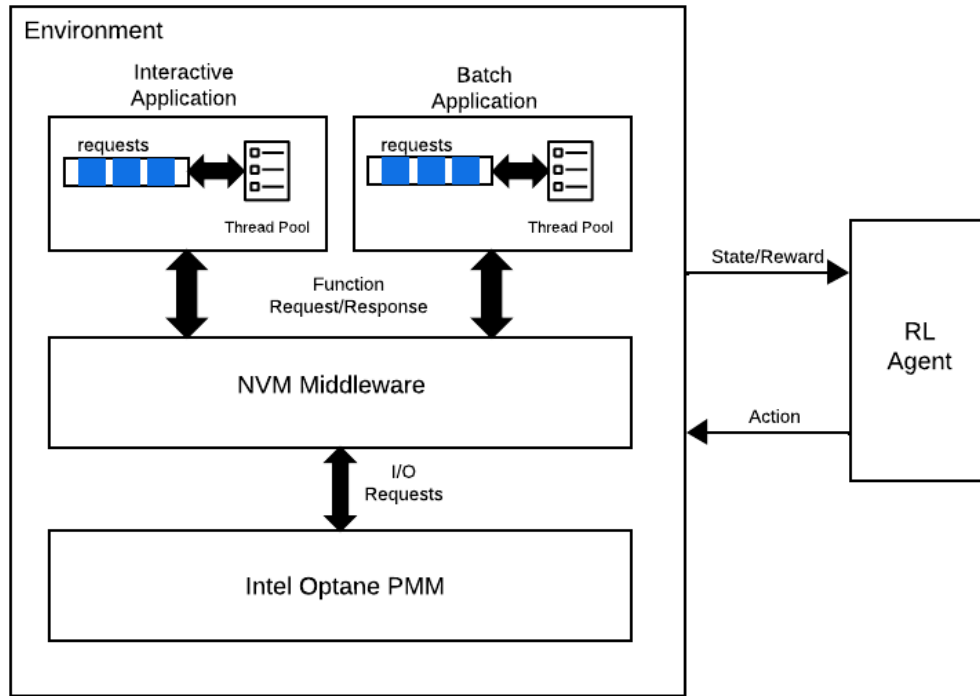


Figure 3.3: Overview of the RL environment constructed for training the RL agent. The environment employs multi-threaded interactive and batch applications to replicate workloads generated by multiple serverless functions with shared access to persistent memory. These applications interact with the NVM Middleware through function calls using its programming interface. Meanwhile, in the background, the agent continuously iterates through a closed feedback loop, observing the environment’s state and executing scaling actions based on its policy.

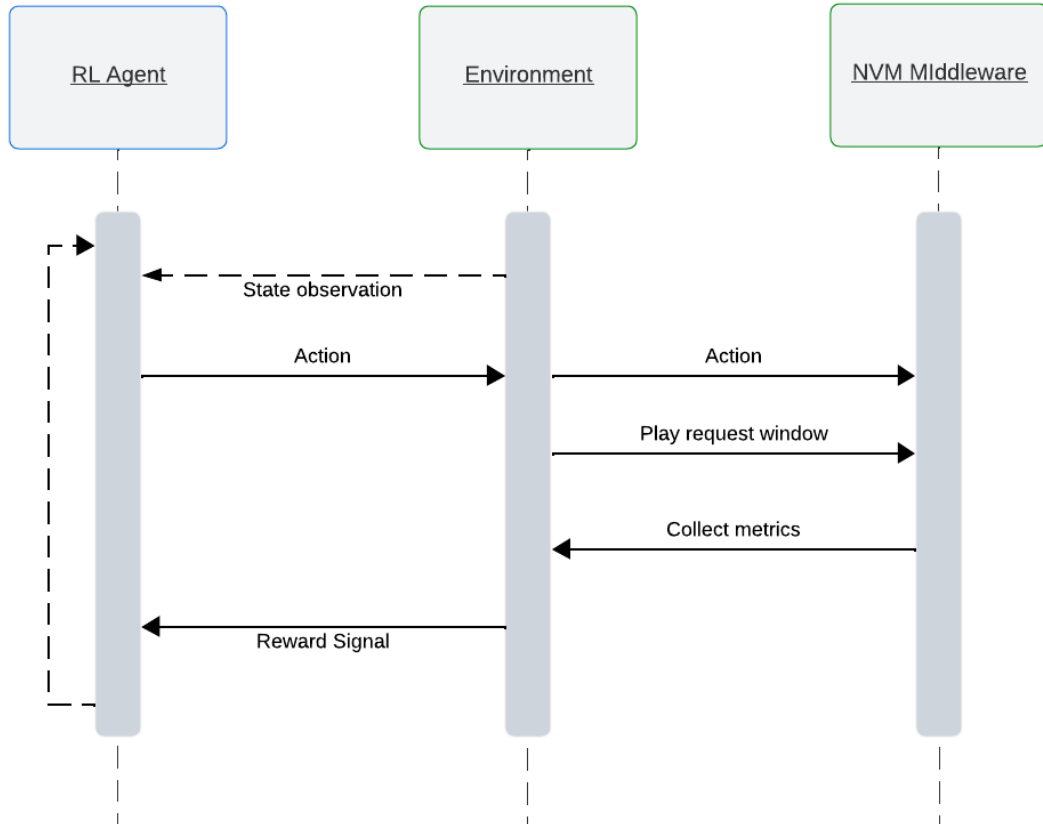


Figure 3.4: Illustration depicting the interaction between the RL agent and its environment at each time step. During each time step, the agent executes a scaling action, followed by the processing of the subsequent window of I/O requests by the environment. Subsequently, the environment gathers latency and throughput metrics to assess the effectiveness of the action, providing a reward signal that the agent utilizes to refine its policy.

Chapter 4: Evaluation

In this chapter, the NVM Middleware and the Q-Learning Model. We utilize the environment delineated in Section 3.4.3 to run a set of experiments.

4.1 Experimental Setup

4.1.1 Platform

Table 4.1: Experimental Platform Specifications

Processor	Intel [®] Xeon [®] Gold 6252
Sockets	2
Cores per socket	24
Threads per core	2
Numa nodes	2
CPU Frequency	2.7 GHz (3.7 GHz Turbo frequency)
L1d cache	1.5 MiB
L1i cache	1.5 MiB
L2 Cache	48 MiB
L3 Cache	71.5 MiB
DRAM	16 GB DDR4 DIMM x 6 per socket
Persistent Memory	128 GB Optane PMM x 6 per socket
Operating System	Ubuntu 20.04.4 LTS (Focal Fossa)

The experimental platform utilized in this study is detailed in Table 4.1.1. It features an Intel[®] Xeon[®] Gold 6252 processor with 2 sockets, each hosting 24 cores and 2 threads per core, totaling 2 NUMA nodes. Each socket is equipped with three memory channels, housing 16 GB DDR4 DIMMs and 128 GB Optane PMMs. In aggregate, the system comprises 192 GB of DRAM and 1.5 TB of Optane persistent memory. To mitigate the NUMA effect, one

socket is designated for running the NVM Middleware threads, while the other handles the interactive and batch applications, as described in Section 3.4.3.

4.1.2 Optane DC PMem Configuration

As outlined earlier, this thesis concentrates on exploring the persistent capabilities of Optane DC PMem. Consequently, Optane DC PMem is employed in the App Direct Mode throughout our experiments. To facilitate the utilization of persistent memory, we expose it via an xfs filesystem configured in dax mode, thereby bypassing the page cache. Additionally, we enhance memory management and performance by configuring the persistent memory with huge pages (2MiB) [11]. Lastly, we deploy a PMEMKV database with a capacity of 600GB, configured with its persistent concurrent engine.

4.1.3 Workload Generators

We deploy the interactive and batch applications outlined in Section 3.4.3 utilizing two distinct workload generators.

Yahoo! Cloud Serving Benchmark (YCSB)

YCSB [50] is a multi-threaded benchmark specifically tailored for assessing cloud-based databases and storage systems. Utilizing YCSB, we emulate interactive applications by generating small byte requests. To vary the workload characteristics, we modify parameters such as the read-to-write ratio, request distribution, and client threads. Leveraging the C++ version of YCSB, we extend its functionality to facilitate API calls to the NVM Middleware.

Serverless Trace Replay

We develop the Serverless Trace Replay tool in-house to replicate workloads typically encountered in real-world serverless environments. This tool operates by reading a file containing workload traces and executing them accordingly. To simulate multiple serverless functions, the tool spawns multiple threads to handle incoming requests.

For emulating an interactive application, we utilize traces collected from Azure Functions, sourced from a dataset available in [51]. This dataset (described in more detail in [14]) offers a comprehensive log of Azure Function blob accesses over HTTPS recorded between November and December 2020. Specifically, our experiments utilize requests recorded on December 6, 2020, with a specific emphasis on those involving small data access sizes (less than 1 KB), which typically signify interactive application behavior.

For modeling a batch application, we collect traces from Wukong, a serverless parallel computing framework [18]. The traces are acquired by executing a Single Value Decomposition job for a 128kx128k matrix on Wukong and capturing the resulting I/O requests generated by the framework. This dataset offers a precise representation of throughput-oriented serverless data-analytics applications, characterized by significant parallelism and substantial data access sizes spanning from 4KB to 200MB.

4.2 Efficiency of the Workload-Aware Concurrency Control Mechanism

We evaluate the effectiveness of the workload-aware concurrency control mechanism embedded within the NVM Middleware compared to a baseline scenario devoid of concurrency control. In the baseline setting, concurrency control is inactive, permitting a maximum of 200 concurrent data accesses on Optane DC PMem. This assessment involves deactivating the reinforcement learning agent and system monitoring, with a primary focus on the 99th percentile latency and throughput observed by client applications.

Six experiments are conducted, running YCSB and SVD Trace Replay concurrently. Each YCSB experiment varies parameters such as data access size (64B, 128B), read-to-write ratio (50-50, 100-0), and data request distribution (zipfian, uniform), while SVD Trace Replay maintains consistent settings across experiments. Both applications are executed with 100 client threads.

In each experiment, the baseline scenario is executed without concurrency control, followed by 42 additional tests exploring various combinations of interactive and batch threads within the NVM Middleware. Each run maintains a fixed combination of interactive (I) and batch (B) threads. Subsequently, the 99th percentile latency observed by YCSB requests and the overall throughput reported by SVD Trace Replay are recorded. Results are presented in Appendix A.

Our observations indicate substantial benefits from the concurrency control implemented by the NVM Middleware across most scenarios. Relative to the baseline, the NVM Middleware demonstrates potential enhancements of up to 98% in 99th percentile latency and 86% in throughput. Notably, certain thread combinations achieve sub-millisecond access latencies with a more predictable behavior, significantly improving application performance. However, improper thread configuration within the NVM Middleware, either insufficient or excessive, results in performance degradation exceeding that of the baseline. This emphasizes the criticality of meticulously selecting the optimal thread combination, as an incorrect choice may yield similar or inferior results compared to operating without concurrency control.

A key query arising from these findings concerns determining the optimal thread combination. We observe that prioritizing interactive threads yields sub-millisecond access latencies but compromises peak throughput. Conversely, increasing batch threads to enhance throughput metrics leads to higher access latencies. Addressing this dilemma entails selecting a combination of interactive and batch threads that satisfies both latency and throughput SLA metrics, a topic elaborated upon in the subsequent section.

4.3 Meeting SLA performance using RL

We present an evaluation of RL-driven policies aimed at balancing the number of interactive and batch threads to meet latency and throughput SLAs within the NVM Middleware. This experiment assesses the NVM Middleware’s ability to achieve predefined SLA objectives amidst varying workloads. To this end, we construct four distinct phases, each

comprising a latency-sensitive and throughput-oriented application executed concurrently in the environment (see Appendix C for phase details). Utilizing the Q-Learning algorithm (outlined in Algorithm 2), we train the RL agent to determine the optimal combination of interactive and batch threads that maximizes performance while meeting predefined latency and throughput SLA objectives for each phase. We then evaluate the agent’s ability to predict and adapt to workload changes in an unknown environment where phases are randomly alternated.

Convergence of the RL Agent

We commence the learning process (see Appendix D for details) by conducting model selection and hyperparameter tuning on the nine linear regression models employed by the RL agent (refer to Table D for tuning options). This entails generating a dataset of transitions in the environment by executing a non-optimal random agent on the environment for 150 episodes of each workload. Subsequently, we utilize these transitions to select the appropriate model and tune hyperparameters for each action. The resulting linear regression models are summarized in Table D.

With the tuned linear regression models, we proceed to execute the Q-learning algorithm for each workload using the parameters detailed in Table D. To address the exploration-exploitation dilemma, we initialize the epsilon value to 1 and decay it after each episode. This strategy facilitates comprehensive exploration of the state space early in training, followed by exploitation of acquired knowledge. We note that different phases require varying numbers of training episodes to converge to an optimal pattern, likely due to the initial dataset’s generation with a non-optimal policy. Initially, we conduct empirical tests by fixing the NVM Middleware threads to manually ascertain the optimal combination. Subsequently, we execute Q-learning and analyze the last three episodes of each phase to determine the convergent combination of threads.

Our observations indicate that the RL agent is capable of learning the optimal combination of interactive and batch threads for each workload. The learned combination of threads

matches our empirical results across all workloads. For Workload A, the agent converges to utilizing approximately 10 interactive and batch threads each. Similarly, for Workload B, the agent converges to approximately 10 interactive and batch threads. For Workload C, the agent converges to approximately 7 interactive and 3 batch threads, while for Workload D, it converges to approximately 15 interactive threads and 5 batch threads.

RL Agent Evaluation

We evaluate the trained RL agent’s performance against two baseline scenarios: one with disabled concurrency control in the NVM Middleware and another with a fixed policy of 15 interactive and batch threads. Each test comprises a long-run simulation spanning 4,000 steps, with the workload changing every 200 steps. We capture the minimum, 25th, 50th, 75th, and maximum throughput and tail latency observed by the NVM Middleware, as well as the environment rewards for each test (results presented in Appendix E). Figure E.1 illustrates the dynamic behavior of the RL agent as it adapts the combination of interactive and batch threads in response to the current workload patterns learned during training.

Our observations demonstrate the added performance benefits of RL-driven policies under varying workloads. While incorporating the NVM Middleware with a fixed policy yields performance improvements, dynamic control implemented by the RL agent further enhances performance. Increased rewards (Table E.1) observed by the RL agent indicate superior maximization of performance and adherence to target SLAs. This is reflected in the achieved throughput (Table E.2), with the NVM Middleware meeting the throughput target for 50% of steps, compared to less than 25% for the fixed policy scenario. Furthermore, although not achieving the target throughput in all steps, the NVM Middleware’s overall throughput is closer to the target compared to the other scenarios.

However, in this experiment, the 99th percentile latency (Table E.3) exhibited by both scenarios using the NVM Middleware did not meet expectations. We attribute this issue to the chosen interactive workloads not sufficiently stressing the system, leading to better

performance in the scenario with no concurrency control. Further discussion on this topic is provided in the subsequent section.

Chapter 5: Related Work

Optane DC PMem Resource Contention: Prior research has highlighted resource contention issues within Intel Optane DC PMem [4, 5]. Strategies to mitigate these challenges often involve limiting the number of concurrent threads accessing persistent memory. Yang et al. [4] enhance the performance of an NVM-aware file system by limiting the number of writer threads per Optane PMM. Similarly, Li et al. [5] address the overhead incurred by multiple threads flushing data from CPU caches to persistent memory by implementing a runtime system that regulates thread concurrency.

Our work draws inspiration from the approaches mentioned above. We design the NVM Middleware as an integration layer that applies model-guided optimizations to leverage Intel Optane DC PMem effectively. Similar to Ribbon, our work controls concurrency levels on persistent memory and dynamically adjusts thread counts as necessary. However, the NVM Middleware introduces a novel approach beyond conventional concurrency controls. It implements a workload-aware concurrency mechanism that allocates distinct worker threads for different workload types, enabling performance segregation among workloads with diverse requirements. Furthermore, unlike Ribbon, the NVM Middleware incorporates latency and bandwidth service level agreements to proactively adjust worker thread allocations for each workload type, leveraging reinforcement learning to adapt to workload changes.

Limitations of Serverless Storage: Research on serverless platforms has underscored inefficiencies stemming from limitations in existing cloud storage offerings. Several systems have emerged to alleviate communication bottlenecks in serverless functions.

Some systems minimize storage overhead by reducing traffic to remote storage and utilizing local communication mechanisms among related serverless functions [14, 52]. SAND [52] facilitates direct communication between related serverless functions running on the same

server via a local message bus. FAA\$T [14] introduces a transparent and auto-scalable in-memory caching layer to prioritize data reuse and reduce calls to remote storage. Similarly, Cloudburst [53] enhances data locality for serverless functions by implementing a key-value cache on each machine hosting function invocations.

Others focus on tailored storage solutions for data-intensive serverless workloads, bridging performance and capacity gaps between DRAM and persistent storage through multi-tier storage approaches [15, 44]. Pocket [15] utilizes application hints to determine data tier storage, while Anna [44] employs frequency analysis of key invocations to dynamically move data between tiers. These systems employ reactive policies to autoscale resources based on predefined thresholds.

In contrast, the NVM Middleware capitalizes on Intel Optane DC PMem’s unique blend of persistent data and memory-like speeds to bridge performance and capacity gaps without necessitating data tiering policies. Moreover, it adopts a proactive approach to resource autoscaling, leveraging Reinforcement Learning to predict workload changes and preemptively adjust resources.

Additionally, studies like Tariq et al.’s work on Sequoia [37] shed light on cloud storage service quality issues and propose frameworks for better scheduling of serverless function chains. While not storage-centric, this work underscores the importance of improving service quality in serverless environments. Similarly, Pisces [36] extends performance isolation policies by achieving per-tenant performance isolation and fairness in shared storage services, suggesting a potential research direction for enhancing the NVM Middleware.

Reinforcement Learning-based Optimization Policies: Cano et al. [45] introduced a novel application of Reinforcement Learning to enhance task scheduling policies in their cluster environments. Their approach involved employing linear regression models to approximate the Q-function, a fundamental component in RL algorithms. Additionally, they conducted offline pre-training of these linear models, guided by model selection and hyper-parameter tuning processes. By initializing the RL agent with pre-trained models,

they facilitated a faster learning process by providing initial knowledge about the environment. Inspired by their methodology, we adopted similar strategies in implementing our RL agent and Q-learning process.

Chapter 6: Discussions and Future Research

Storage Service Integration: Originally, this thesis aimed to design a simple storage server facilitating communication with applications via Remote Procedure Calls (RPC), leveraging the NVM Middleware to optimize Intel Optane DC PMem performance. However, our measurements were affected by network overhead in the infrastructure housing the server with Optane DC PMem. Thus, evaluating the NVM Middleware on a real-world serverless storage service remains a future research endeavor. Anna key-value database [54] proposes a promising option for assessing the NVM Middleware by extending it to incorporate a storage tier based on Optane DC PMem, thereby utilizing the NVM Middleware as the optimization layer for persistent memory.

Reducing Autoscaling Overhead: Our current approach to scaling resources involves adding or removing up to two threads per one-second window, which may delay the system’s adaptation to the optimal combination of interactive and batch threads. This delay could contribute to performance spikes observed in the results. An intriguing avenue for exploration is enabling the NVM Middleware to add or remove more than two threads per time step to expedite adaptation to workload changes, potentially enhancing SLA fulfillment efficiency. However, this approach may introduce additional actions in the action space, increasing the RL model’s complexity.

Choice of Workloads: While YCSB offers versatility in testing various workloads, its simulations may not accurately reflect real-world serverless workload characteristics. Thus, in this thesis, we rely on I/O traces from actual serverless platforms to evaluate our RL approach. However, crafting experimental workloads based on real-world serverless traces, with modified characteristics such as block size and read-to-write ratio, was necessary to test the RL agent’s autoscaling capabilities under shifting workloads. Yet, our chosen workloads may have hindered the RL agent’s ability to meet latency SLAs. Obtaining

more real-world serverless traces could shed light on the NVM Middleware’s limitations and capabilities. However, acquiring such traces is challenging and time-consuming, as the RL agent must undergo the entire learning process for each stage, from dataset generation for model selection to running Q-learning episodes.

Exploring Other Function Approximation Techniques: Feature extraction is pivotal but complex. While we select relevant features for our RL model, covering more generic use cases may necessitate additional features to enhance learning capabilities. These features could encompass workload characteristics, NVM Middleware metrics, Optane DC PMem internal metrics, and general server metrics like CPU and memory usage. Limitations in feature representation may impede linear function approximators’ efficacy [40]. Alternatively, employing deep neural networks for function approximation [55], known as deep reinforcement learning, could address these challenges. Deep reinforcement learning has demonstrated success in complex scenarios by autonomously discovering useful features, as evidenced in gaming environments [56, 57].

Learning Workload Characteristics: The current strategy employed by the NVM Middleware involves utilizing predefined hints regarding workload characteristics to allocate requests to specific queues. However, this approach may encounter scalability challenges, prompting the exploration of autonomous learning methods for workload characteristics. Several approaches can be considered, ranging from straightforward categorization based on predefined criteria to more sophisticated methods leveraging machine learning models.

Chapter 7: Conclusions

Appendix A: Reinforcement Learning Algorithms

Algorithm 1: Reward Calculation Algorithm

```

Input: System statistics: stat
Output: Reward value: reward
/* Initialize variables */
1 max_scale_lat  $\leftarrow$  1000, max_scale_tp  $\leftarrow$  10, min_scale  $\leftarrow$  1, lat_goal  $\leftarrow$  200,
  tp_goal  $\leftarrow$  250000, lat_penalty  $\leftarrow$  500.0, tp_penalty  $\leftarrow$  5000.0;
/* Scale observed and target latency and throughput */
2 lat  $\leftarrow$  ((max_scale_lat - min_scale)  $\times$  (stat.tailLatency - min_value)/(max_latency -
  min_value)) + min_scale;
3 tp  $\leftarrow$  ((max_scale_tp - min_scale)  $\times$  (stat.throughput - min_value)/(max_throughput -
  min_value)) + min_scale;
4 lat_goal  $\leftarrow$ 
  ((max_scale_lat - min_scale)  $\times$  (lat_goal - min_value)/(max_latency - min_value)) + min_scale;
5 tp_goal  $\leftarrow$  ((max_scale_tp - min_scale)  $\times$  (tp_goal - min_value)/(max_throughput -
  min_value)) + min_scale;
/* Calculate errors */
6 error_lat  $\leftarrow$  |lat - lat_goal|;
7 error_tp  $\leftarrow$  |tp - tp_goal|;
/* Calculate reward */
8 if lat  $\leq$  lat_goal and tp  $\geq$  tp_goal then
9   | reward  $\leftarrow$  10  $\times$  (error_lat + error_tp); // High reward for meeting both latency
  | and throughput goals
10 else
11   | if lat > lat_goal and tp < tp_goal then
12   | | reward  $\leftarrow$  -1  $\times$  (lat_penalty  $\times$  error_lat + tp_penalty  $\times$  error_tp); // Penalize for
  | | high latency and low throughput
13   | else
14   | | if lat > lat_goal then
15   | | | reward  $\leftarrow$  -1  $\times$  lat_penalty  $\times$  error_lat; // Penalize for high latency
16   | | else
17   | | | reward  $\leftarrow$  -1  $\times$  tp_penalty  $\times$  error_tp; // Penalize for low throughput
18   | | end
19   | end
20 end

```

Algorithm 2: Q-Learning Algorithm

Input: Pre-trained Q-value models M_a for all actions a
Output: Learned Q-value models M_a for all actions a

```
1 Initialize the training parameters  $\alpha, \gamma, \epsilon$ ;  
2 for  $episode \leftarrow 1$  to  $E$  do  
3   Reset the environment;  
4   repeat  
5     Observe the state  $s_t$ ;  
6     // Choose action  $a_t$  using the  $\epsilon$ -greedy policy  
7     Generate random number  $r$  from uniform distribution in  $[0, 1]$ ;  
8     if  $r < \epsilon$  then  
9       | Select a random action  $a_t$  from the action space ;  
10    end  
11    else  
12      | for each action  $a$  do  
13        | Predict Q-value  $Q_a(s_t)$  using model  $M_a$ :  $Q_a(s_t) \leftarrow M_a.predict(s_t)$  ;  
14      | end  
15      | Select action  $a_t \leftarrow \arg \max_a Q_a(s_t)$  ;  
16    end  
17    Take action  $a_t$ , observe reward  $r$  and next state  $s_{t+1}$ ;  
18    // Update the Q-value model using reward and next state  
19    if not done then  
20      | for each action  $a$  do  
21        | Predict Q-value  $Q_a(s_{t+1})$  using model  $M_a$ :  $Q_a(s_{t+1}) \leftarrow M_a.predict(s_{t+1})$  ;  
22      | end  
23      | Calculate target Q-value:  $target \leftarrow r + \gamma \cdot \max_a Q_a(s_{t+1})$  ;  
24    end  
25    else  
26      | Set target Q-value to the reward:  $target \leftarrow r$  ;  
27    end  
28    Update the model for action  $a_t$  with the target Q-value:  $M_{a_t}.partial\_fit(s_t, target)$   
29    ;  
30    Update state:  $s_t \leftarrow s_{t+1}$ ;  
31  until  $episode$  is done;  
32  Decrease  $\epsilon$  according to exploration schedule;  
33 end
```

Appendix B: Concurrency Control Results

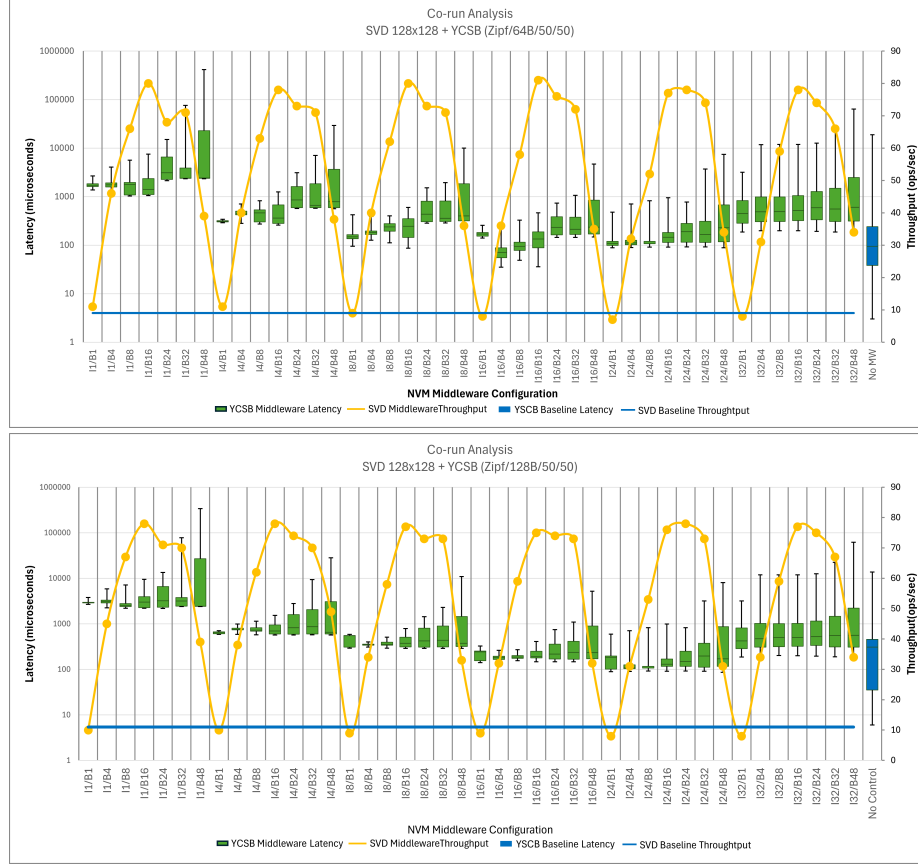


Figure B.1: Evaluation of the NVM Middleware’s concurrency control mechanism when co-running SVD I/O traces (batch) and a YCSB workload (interactive) configured with zipfian request distribution and an even distribution of read and write requests. Two runs are performed configuring YCSB with 64B and 128B data access sizes. Both applications utilize 100 client threads to send requests to the NVM Middleware. The illustration presents statistics on throughput (yellow line) and 99th percentile latency (green box plot) obtained under different configurations of the NVM Middleware, denoted as I/B (Interactive threads/Batch threads). These statistics are compared against a baseline scenario with no concurrency control (blue line and blue box plot). The latencies observed by YCSB are depicted as a distribution, ranging from the minimum to the maximum observed, along with quartiles and median values.

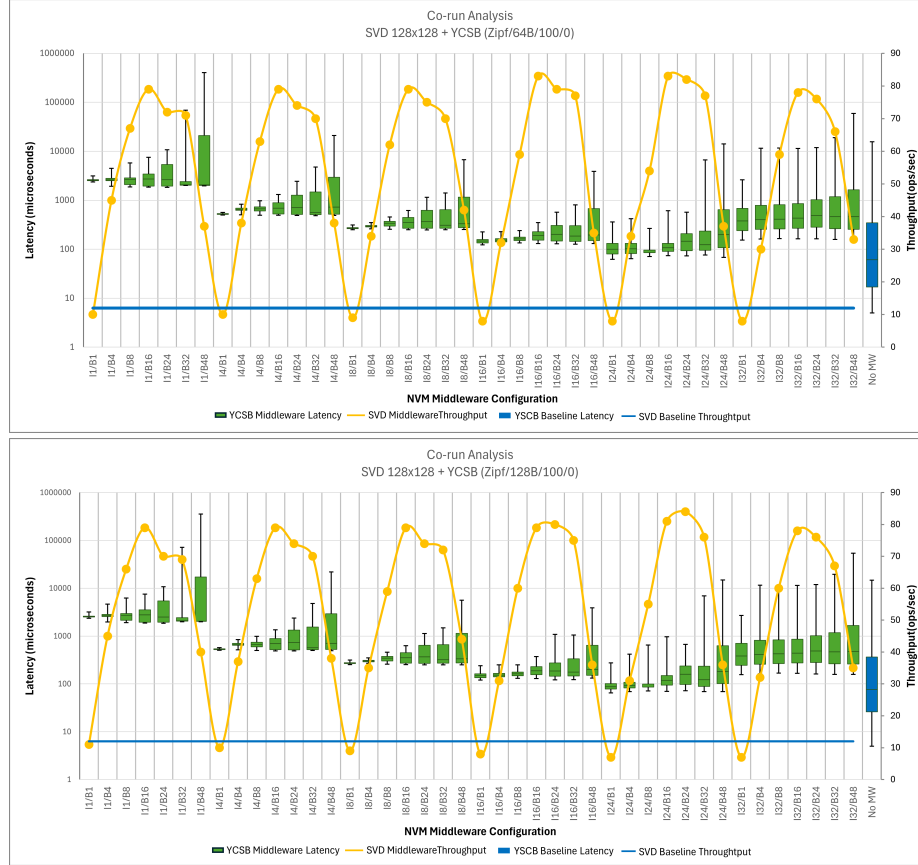
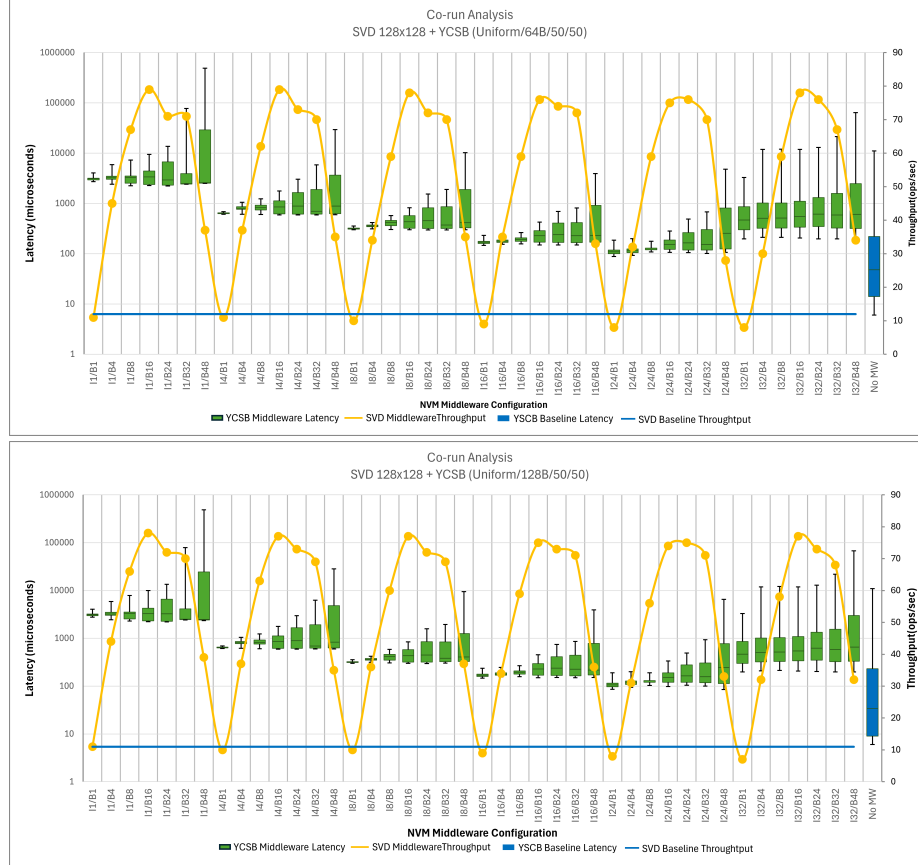


Figure B.2: Evaluation of the NVM Middleware’s concurrency control mechanism when co-running SVD I/O traces and a pure read YCSB workload with Zipfian request distribution. Two runs are performed configuring YCSB with 64B and 128B data access sizes. Both applications utilize 100 client threads to send requests to the NVM Middleware. The illustration presents statistics on throughput (yellow line) and 99th percentile latency (green box plot) obtained under different configurations of the NVM Middleware, denoted as I/B (Interactive threads/Batch threads). These statistics are compared against a baseline scenario with no concurrency control (blue line and blue box plot). The latencies observed by YCSB are depicted as a distribution, ranging from the minimum to the maximum observed, along with quartiles and median values.



Appendix C: RL Phases

To evaluate the learning capabilities of the RL agent in adapting thread combinations for optimal performance and meeting predefined SLA metrics, we design four distinct phases. Leveraging traces from Azure Functions and Wukong, we construct interactive and batch workloads by adjusting parameters such as data access size, read-to-write ratio, and client threads for each application. We intensify the workload by accelerating the pace of the traces and loop the execution to ensure sufficient simulation steps.

The phases are structured as follows:

Phase 1: We utilize 2039 requests from Azure Function blob access traces captured on December 6, 2020, between 8:20 PM and 8:22 PM. These requests are primarily reads, with a read-to-write ratio of 80-20 and data access sizes ranging between 30B and 50B. For the batch workload, we employ a subset of SVD traces comprising 3847 requests, maintaining a 50-50 read-to-write ratio and a fixed data access size of 800KB. Both applications are configured with 200 client threads for concurrent request handling.

Phase 2: The interactive workload remains unchanged from Phase 1, while the number of client threads is reduced to 150. Similarly, the batch workload maintains its configuration but with an increased number of concurrent client threads set to 320.

Phase 3: The interactive workload is consistent with Phase 1, but with an escalation in client threads to 400. For the batch workload, we modify the SVD traces to employ a fixed data access size of 4k and transition to a read-only workload. Concurrently, we assign 200 client threads for the batch workload.

Phase 4: Drawing from Azure Function blob access traces, we incorporate 7814 requests occurring on December 6, 2020, between 1:21 PM and 1:27 PM for the interactive workload. These requests are predominantly writes, with a read-to-write ratio of 10-90 and larger data access sizes averaging around 500B. The batch workload configuration remains consistent with Phase 3. Each application is allocated 200 client threads for concurrent request processing.

Appendix D: RL Agent Training

Table D.1: Overview of hyperparameters used in the tuning process for polynomial regression models employed in function approximation. The *degree* parameter denotes the degree of the polynomial function utilized. The *loss* parameter specifies the loss function employed during stochastic gradient descent. The *penalty* parameter represents the regularization technique applied to mitigate overfitting. The *alpha* parameter indicates the strength of the regularization penalty. The *learning_rate* parameter determines the scheduling of the model's learning rate. Finally, the *max_iterations* parameter sets the maximum number of passes over the training data.

Parameter	Values
degree	1,2,3
loss	squared, huber, epsilon_insensitive
penalty	l1, l2, elasticnet
alpha	0.1, 0.01, 0.001, 0.0001
learning rate	constant, optimal, invscaling
max_iterations	100, 1000, 10000, 100000

Table D.2: Hyperparameters of the polynomial regression models after hyperparameter tuning.

Model	Parameters
Model_1	degree: 2, learning_rate: 'invscaling', loss: 'squared_loss', alpha: 0.1, max_iter: 1000, penalty: elasticnet
Model_2	degree: 2, learning_rate: 'invscaling', loss: 'squared_loss', alpha: 0.01, max_iter: 10000, penalty: l1
Model_3	degree: 2, learning_rate: 'invscaling', loss: 'squared_loss', alpha: 0.0001, max_iter: 100, penalty: elasticnet
Model_4	degree: 2, learning_rate: 'invscaling', loss: 'squared_loss', alpha: 0.001, max_iter: 10000, penalty: l1
Model_5	degree: 2, learning_rate: 'invscaling', loss: 'squared_loss', alpha: 0.01, max_iter: 10000, penalty: elasticnet
Model_6	degree: 2, learning_rate: 'invscaling', loss: 'squared_loss', alpha: 0.01, max_iter: 1000, penalty: elasticnet
Model_7	degree: 2, learning_rate: 'invscaling', loss: 'squared_loss', alpha: 0.01, max_iter: 1000, penalty: elasticnet
Model_8	degree: 2, learning_rate: 'invscaling', loss: 'squared_loss', alpha: 0.1, max_iter: 10000, penalty: l1
Model_9	degree: 2, learning_rate: 'invscaling', loss: 'squared_loss', alpha: 0.01, max_iter: 100, penalty: elasticnet

Table D.3: Parameters utilized for Q-Learning by the RL agent. The target 99th percentile latency and throughput represent predefined SLAs guiding the reward calculation. The aim is to maintain the observed 99th percentile latency below 250 microseconds while ensuring throughput remains above 250,000 operations per second. The exploration rate (*epsilon*) diminishes gradually between episodes, following a decay rate (*epsilon_decay*), to strike a balance between exploration and exploitation of knowledge. Furthermore, additional episodes beyond the initially specified count are introduced to enable the agent to reach the optimal policy for phases 1, 3, and 4.

Parameter	Value
episodes	Phases 1, 3, 4: 1,000, Phase 2: 700
steps per episode	200
gamma	0.95
learning rate	0.7
epsilon	0.9
epsilon_decay	0.1
Target 99th Latency	$\leq 250\text{microseconds}$
Target throughput	$\geq 250,000\text{operations/second}$

Table D.4: Experimental reward analysis conducted on Phase 1 using the NVM Middleware with various fixed combinations of threads. The table displays statistics on rewards obtained under different configurations, denoted as I5/B5, I10/B10, I15/B15, I15/B5, and I5/B15. The values represent the distribution of rewards, ranging from the minimum to the maximum observed, along with quartiles and median scores. Based on these preliminary findings, the configuration with 10 interactive threads and 10 batch threads appears to yield the most favorable results for Phase 1.

	I5/B5	I10/B10	I15/B15	I15/B5	I5/B15
Min	-5716	-8312	-41660	-5682	-9436
Q1	-1383	2.7	-4797	-2971	-1
Median	-171	3.92	-2	-1850	0
Q3	0	4.3	3	-1035	1
Max	1	5	5	3	2

Table D.5: Experimental reward analysis conducted on Phase 2 using the NVM Middleware with various fixed combinations of threads. The table displays statistics on rewards obtained under different configurations, denoted as I5/B5, I10/B10, I15/B15, I15/B5, and I5/B15. The values represent the distribution of rewards, ranging from the minimum to the maximum observed, along with quartiles and median scores. Based on these preliminary findings, the configuration with 10 interactive threads and 10 batch threads appears to yield the most favorable results for Phase 2.

	I5/B5	I10/B10	I15/B15	I15/B5	I5/B15
Min	-4320	-7763	-20455	-5328	-10060
Q1	-463	-1.9	-1169	-368	1
Median	1.3	5.2	4.2	3.1	3
Q3	1.8	5.8	4.9	3.3	4
Max	2.8	6	6	3.8	6

Table D.6: Experimental reward analysis conducted on Phase 3 using the NVM Middleware with various fixed combinations of threads. The table displays statistics on rewards obtained under different configurations, denoted as I5/B5, I7/B3, I7/B7, I10/B10, I15/B15, I15/B5, and I5/B15. The values represent the distribution of rewards, ranging from the minimum to the maximum observed, along with quartiles and median scores. Based on these preliminary findings, the configuration with 7 interactive threads and 3 batch threads appears to yield the most favorable results for Phase 3.

	I5/B5	I7/B3	I7/B7	I10/B10	I15/B15	I15/B5	I5/B15
Min	-12490	-6582	-174852	-141354	-149647	-96900	-13211
Q1	-1230	-1727	-2567	-18768	-42951	-58414	-8583
Median	-272	-672	-4	-566	-14008	-2954	-4940
Q3	-5	-3	-2	-1	-4607	0	-2709
Max	-3	-1	0	3	-1	4	-943

Table D.7: Experimental reward analysis conducted on Phase 4 using the NVM Middleware with various fixed combinations of threads. The table displays statistics on rewards obtained under different configurations, denoted as I5/B5, I10/B10, I15/B15, and I15/B5. The values represent the distribution of rewards, ranging from the minimum to the maximum observed, along with quartiles and median scores. Based on these preliminary findings, the configuration with 15 interactive threads and 5 batch threads appears to yield the most favorable results for Phase 4.

	I5/B5	I10/B10	I15/B15	I15/B5
Min	-11341	-14699	-21259	-3912
Q1	-153	-2.5	-6908	-1
Median	-2	2.3	-3117	3
Q3	0	3.7	-4	4
Max	3	5	3	5

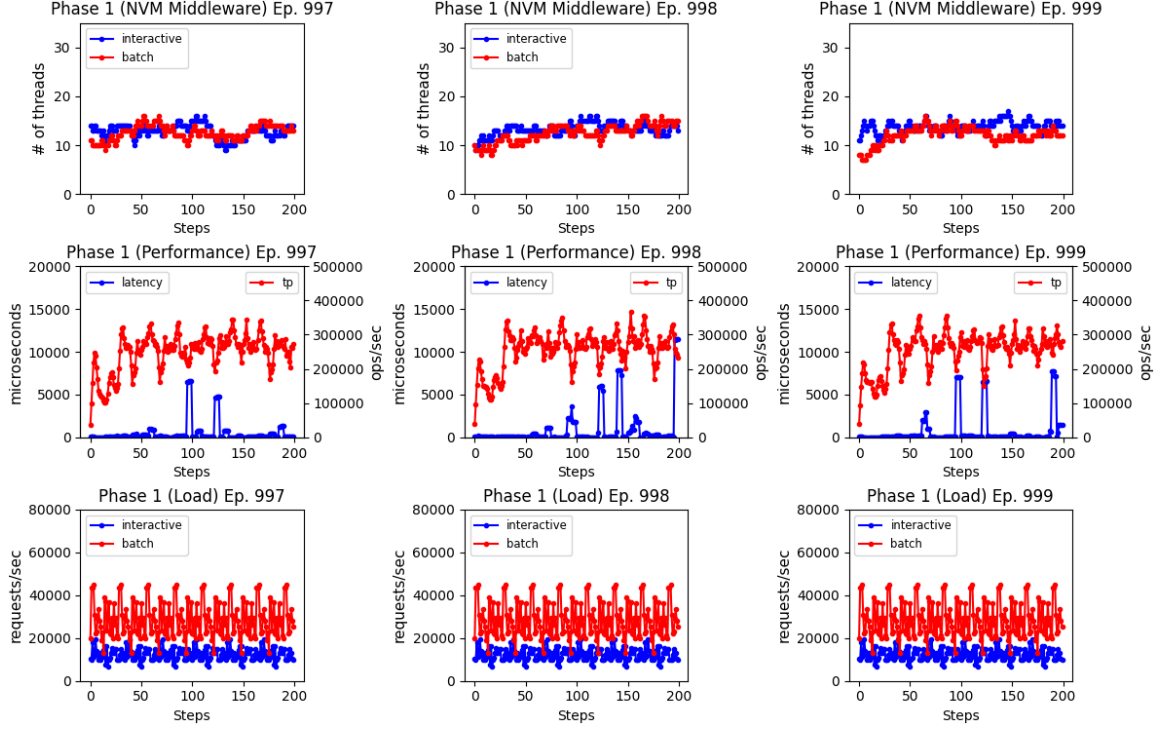


Figure D.1: Visualization depicting the learned pattern of the agent during Phase 1 of the training process. The analysis focuses on the behavior observed in the final three episodes of the Q-Learning process, which spanned 1,000 episodes. During these episodes, the exploration rate is so low that the agent predominantly exploits its accumulated knowledge. The first row illustrates the agent’s configuration of the NVM Middleware with approximately 10 interactive and 10 batch threads, aligning with preliminary results for Phase 1. In the middle row, the throughput and 99th percentile latency reported by the NVM Middleware at each time step are depicted. By employing the optimal combination of threads, the agent consistently maintains low latency (less than 250 microseconds) and a throughput exceeding 250,000 operations/second across most steps. Finally, the bottom row illustrates the operations per second sent to the NVM Middleware by Phase 1.

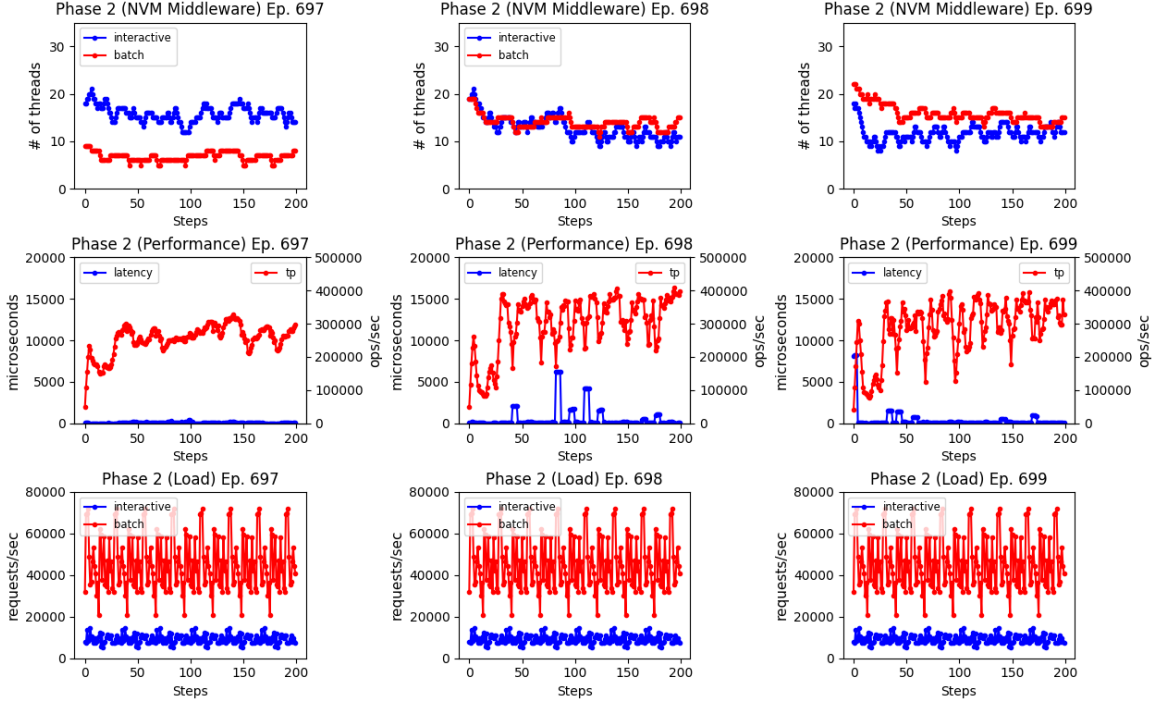


Figure D.2: Visualization depicting the learned pattern of the agent during Phase 2 of the training process. The analysis focuses on the behavior observed in the final three episodes of the Q-Learning process, which spanned 700 episodes. During these episodes, the exploration rate is so low that the agent predominantly exploits its accumulated knowledge. The first row illustrates the agent’s configuration of the NVM Middleware with high number of interactive and batch threads (between 10-15), aligning with preliminary results for Phase 2. In the middle row, the throughput and 99th percentile latency reported by the NVM Middleware at each time step are depicted. By employing the optimal combination of threads, the agent consistently maintains low latency (less than 250 microseconds) and a throughput exceeding 250,000 operations/second across most steps. Finally, the bottom row illustrates the operations per second sent to the NVM Middleware by Phase 2.

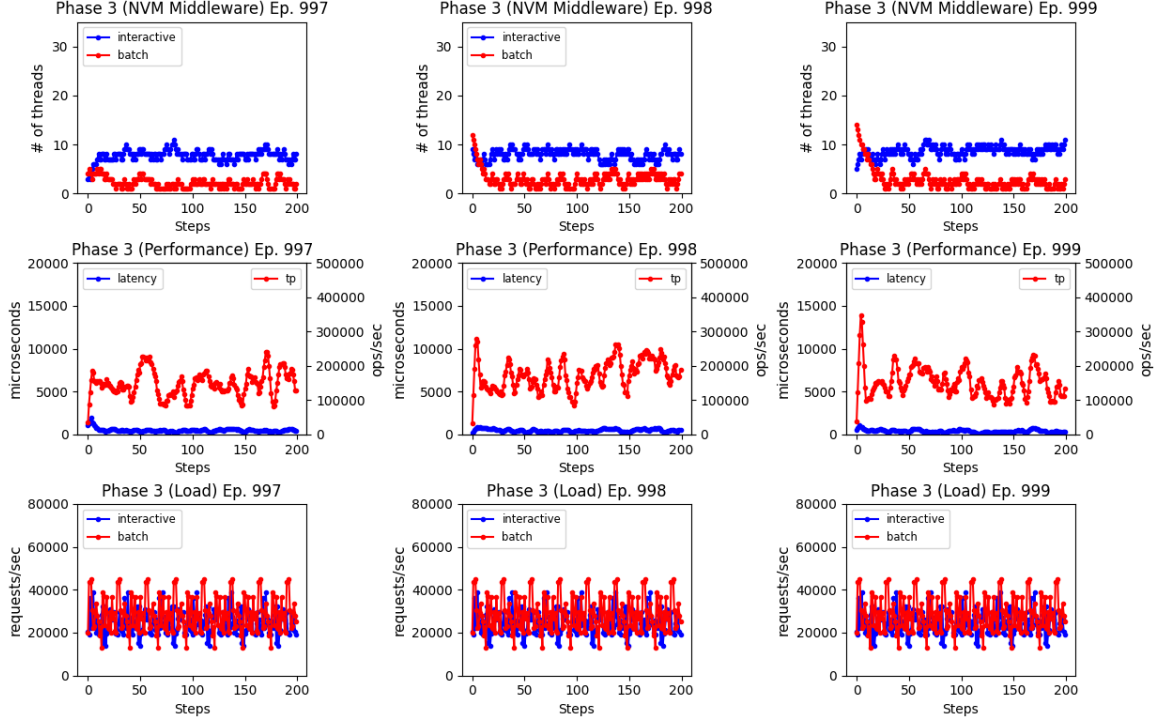


Figure D.3: Visualization depicting the learned pattern of the agent during Phase 1 of the training process. The analysis focuses on the behavior observed in the final three episodes of the Q-Learning process, which spanned 1,000 episodes. During these episodes, the exploration rate is so low that the agent predominantly exploits its accumulated knowledge. The first row illustrates the agent’s configuration of the NVM Middleware with approximately 8 interactive and low number of batch threads (approximately less than 5), aligning with preliminary results for Phase 3. In the middle row, the throughput and 99th percentile latency reported by the NVM Middleware at each time step are depicted. By employing the optimal combination of threads, the agent consistently maintains low latency (less than 250 microseconds) and a throughput exceeding 250,000 operations/second across most steps. Finally, the bottom row illustrates the operations per second sent to the NVM Middleware by Phase 3.

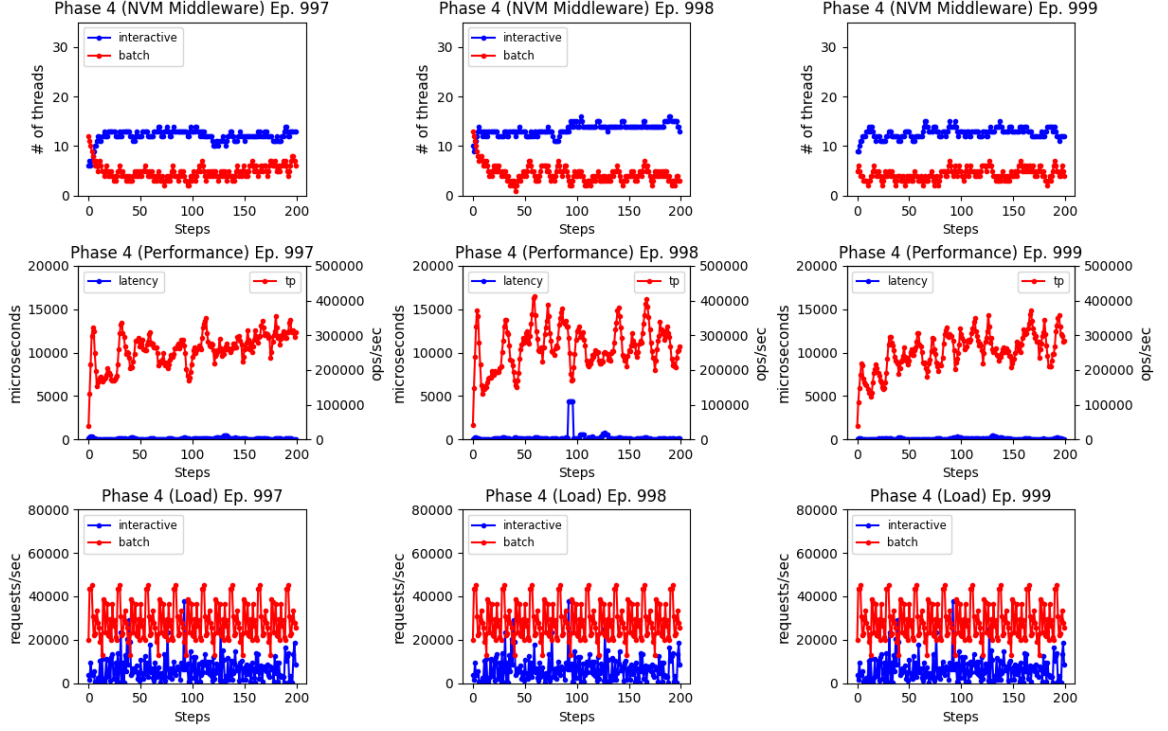


Figure D.4: Visualization depicting the learned pattern of the agent during Phase 1 of the training process. The analysis focuses on the behavior observed in the final three episodes of the Q-Learning process, which spanned 1,000 episodes. During these episodes, the exploration rate is so low that the agent predominantly exploits its accumulated knowledge. The first row illustrates the agent’s configuration of the NVM Middleware with high number of interactive threads (approximately 15) and low number of batch threads (approximately 5), aligning with preliminary results for Phase 4. In the middle row, the throughput and 99th percentile latency reported by the NVM Middleware at each time step are depicted. By employing the optimal combination of threads, the agent consistently maintains low latency (less than 250 microseconds) and a throughput exceeding 250,000 operations/second across most steps. Finally, the bottom row illustrates the operations per second sent to the NVM Middleware by Phase 4.

Appendix E: RL Agent Evaluation

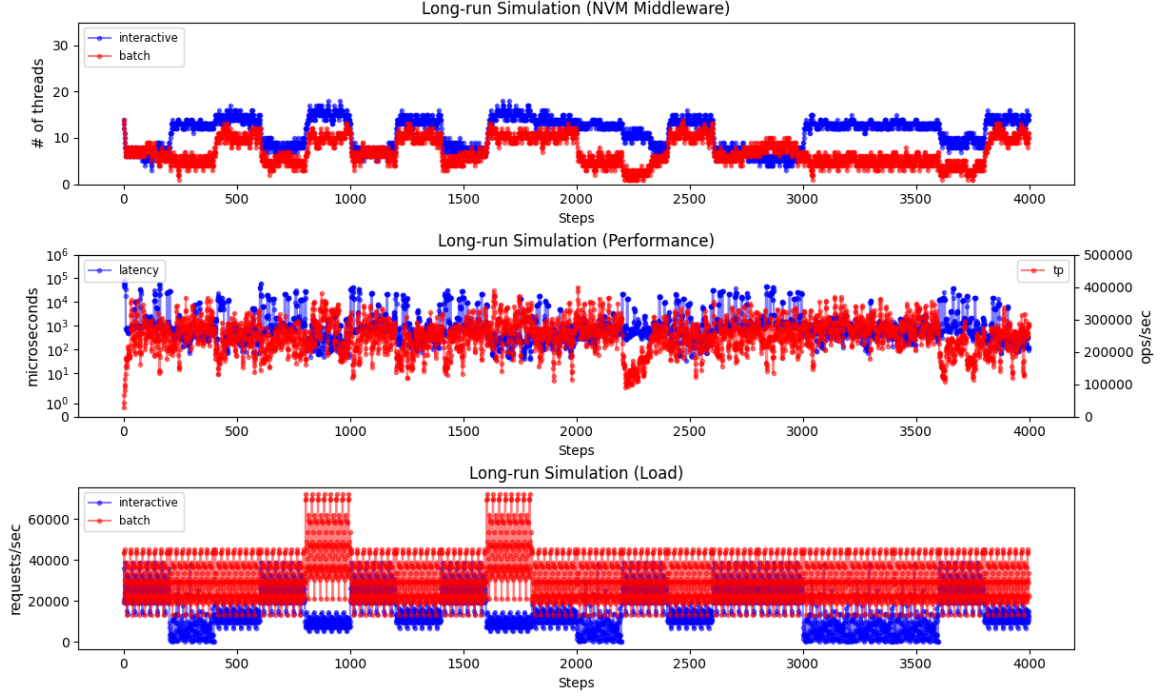


Figure E.1: Illustration demonstrating the dynamic adjustment of thread counts by the RL agent during a test comprising 4,000 steps with randomly alternating phases. The top row showcases the RL agent’s utilization of learned knowledge to optimize the number of interactive and batch threads for each phase. In the middle row, the throughput and 99th percentile latency reported by the NVM Middleware at each time step are depicted. By dynamically modifying the NVM Middleware threads, the throughput aligns with the predefined throughput SLA. However, unexpected spikes in the 99th percentile latency are observed, contrary to the training phase. The bottom row illustrates the operations per second sent by each phase, highlighting the shift in patterns every 200 steps.

Table E.1: Analysis of the rewards achieved by the RL agent under shifting workloads compared to two baseline scenarios: one without concurrency control and the other keeping the NVM Middleware threads fixed. The values represent the distribution of reward signals reported by the environment at each time step, ranging from the minimum to the maximum observed, along with quartiles and median scores. Overall, the NVM Middleware with the RL agent achieves the highest reward.

	No NVM Middleware	NVM Middleware Fixed	NVM Middleware + RL
Min	-94755.386	-65157.089	-87907.434
Q1	-10790.1	-8605.4625	-3647.9454
Median	-6983.235	-3208.995	-4.005294
Q3	-4042.9125	-2.353015	0.31523
Max	3.174046	4.89667	4.582787

Table E.2: Analysis of the throughput achieved by the RL agent under shifting workloads compared to two baseline scenarios: one without concurrency control and the other keeping the NVM Middleware threads fixed. The values represent the distribution of throughput measurements reported by the NVM Middleware at each time step, ranging from the minimum to the maximum observed, along with quartiles and median scores. Overall, the NVM Middleware with the RL agent achieves the highest throughput.

	No NVM Middleware	NVM Middleware Fixed	NVM Middleware + RL
Min	72,218.9	104,719.9	159,168.8
Q1	112,440	148,792.5	218,816.25
Median	135,508.5	186,877	252,078.5
Q3	166,340.25	231,597.75	278,205
Max	242,104.4	357,445.6	352,800.27

Table E.3: Analysis of the throughput achieved by the RL agent under shifting workloads compared to two baseline scenarios: one without concurrency control and the other keeping the NVM Middleware threads fixed. The values represent the distribution of latency measurements reported by the NVM Middleware at each time step, ranging from the minimum to the maximum observed, along with quartiles and median scores. Surprisingly, in this particular test, both configurations involving the NVM Middleware exhibited higher latency compared to the baseline without concurrency control. This unexpected outcome suggests that the interactive workloads may not have adequately stressed the system, resulting in increased latency when utilizing the NVM Middleware. Further investigation is warranted to ascertain the underlying factors contributing to this behavior.

	No NVM Middleware	NVM Middleware Fixed	NVM Middleware + RL
Min	19	89	130.95
Q1	81	239	396.75
Median	19	674.5	754
Q3	271.5	1888.75	1427
Max	82824	34090.93	34712.04

Bibliography

- [1] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383*, 2019.
- [2] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX symposium on networked systems design and implementation (NSDI 19)*, pages 193–206, 2019.
- [3] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dullloor, et al. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.
- [4] 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, 2020.
- [5] Kai Wu, Ivy Peng, Jie Ren, and Dong Li. Ribbon: High performance cache line flushing for persistent memory. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*, pages 427–439, 2020.
- [6] Steve Scargall. *Programming Persistent Memory: A Comprehensive Guide for Developers*. Apress, USA, 1st edition, 2020.
- [7] Ivy Peng, Kai Wu, Jie Ren, Dong Li, and Maya Gokhale. Demystifying the performance of hpc scientific applications on nvm-based memory systems. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 916–925. IEEE, 2020.
- [8] Intel. Solve Modern Data Challenges with Memory Tiering. <https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/memory-tiering-improving-data-management-paper.html>. 2024.
- [9] Persistent Memory Documentation. <https://docs.pmem.io/persistent-memory/getting-started-guide/introduction>. 2024.
- [10] Andy Rudoff. Persistent memory programming. *Login: The Usenix Magazine*, 42(2):34–40, 2017.

- [11] Intel. Speeding Up I/O Workloads with Intel Optane Persistent Memory. <https://www.intel.com/content/www/us/en/developer/articles/technical/speeding-up-io-workloads-with-intel-optane-dc-persistent-memory-modules.html>. 2024.
- [12] Storage Networking Industry Association. NVM Programming Model. <https://www.snia.org/sites/default/files/technical-work/npm/release/SNIA-NVM-Programming-Model-v1.2.pdf>. 2024.
- [13] Intel. pmem/pmemkv: Key/Value Datastore for Persistent Memory. <https://github.com/pmem/pmemkv>. 2024.
- [14] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. FaaS\$T: A Transparent Auto-Scaling Cache for Serverless Applications, 2021.
- [15] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, 2018.
- [16] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: {Low-Latency} video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 363–376, 2017.
- [17] Vicent Giménez-Alventosa, Germán Moltó, and Miguel Caballer. A framework and a performance assessment for serverless MapReduce on AWS Lambda. *Future Generation Computer Systems*, 97:259–274, 2019.
- [18] Benjamin Carver, Jingyuan Zhang, Ao Wang, Ali Anwar, Panruo Wu, and Yue Cheng. Wukong: A scalable and locality-enhanced framework for serverless parallel computing. In *Proceedings of the 11th ACM symposium on cloud computing*, pages 1–15, 2020.
- [19] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 13–24, 2019.
- [20] Lang Feng, Prabhakar Kudva, Dilma Da Silva, and Jiang Hu. Exploring serverless computing for neural network training. In *2018 IEEE 11th international conference on cloud computing (CLOUD)*, pages 334–341. IEEE, 2018.
- [21] Jingyuan Zhang, Ao Wang, Xiaolong Ma, Benjamin Carver, Nicholas John Newman, Ali Anwar, Lukas Rupprecht, Vasily Tarasov, Dimitrios Skourtis, Feng Yan, and Yue Cheng. Infinistore: Elastic serverless cloud storage. *Proc. VLDB Endow.*, 16(7):1629–1642, mar 2023.
- [22] Amazon. AWS Lambda. <https://aws.amazon.com/lambda/>. 2024.

- [23] Microsoft. Azure Functions. <https://azure.microsoft.com/en-us/products/functions/>. 2024.
- [24] Google. Cloud Functions. <https://cloud.google.com/functions>. 2024.
- [25] Apache Software Foundation. Apache OpenWhisk. <https://openwhisk.apache.org/>. 2024.
- [26] Mohammad Shahradd, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.
- [27] Amazon. AWS S3. <https://aws.amazon.com/s3/>. 2024.
- [28] Amazon. Amazon DynamoDB. <https://aws.amazon.com/pm/dynamodb/>. 2024.
- [29] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Voshall, and Werner Vogels. Dynamo: Amazon’s highly available key-value store. In *ACM Symposium on Operating System Principles*, 2007.
- [30] Microsoft. Azure Blob Storage. <https://azure.microsoft.com/en-us/products/storage/blobs>. 2024.
- [31] Google. Cloud Storage. <https://cloud.google.com/storage>. 2024.
- [32] Google. Datastore. <https://cloud.google.com/datastore>. 2024.
- [33] Microsoft. Azure Cosmos DB. <https://azure.microsoft.com/en-us/products/cosmos-db>. 2024.
- [34] Public Cloud Object-store Performance is Very Unequal across AWS S3, Google Cloud Storage, and Azure Blob Storage. <https://dev.to/sachinkagarwal/public-cloud-object-store-performance-is-very-unequal-across-aws-s3-google-cloud-st> 2024.
- [35] Amazon. Amazon ElastiCache. <https://aws.amazon.com/pm/elasticache/>. 2024.
- [36] David Shue, Michael J. Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 349–362, Hollywood, CA, October 2012. USENIX Association.
- [37] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM symposium on cloud computing*, pages 311–327, 2020.
- [38] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

- [39] Christopher John Cornish Hellaby Watkins. *Learning from delayed rewards*. PhD thesis, King’s College, Cambridge United Kingdom, 1989.
- [40] Stuart S Russell and Petter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 2020.
- [41] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *Journal of machine learning research*, 13(2), 2012.
- [42] Intel. Intel Optane Persistent Memory. <https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html>. 2024.
- [43] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. Understanding ephemeral storage for serverless analytics. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 789–794, 2018.
- [44] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Autoscaling tiered cloud storage in Anna. *Proceedings of the VLDB Endowment*, 12(6):624–638, 2019.
- [45] Ignacio Cano, Srinivas Aiyar, Varun Arora, Manosiz Bhattacharyya, Akhilesh Chaganti, Chern Cheah, Brent Chun, Karan Gupta, Vinayak Khot, and Arvind Krishnamurthy. Curator:{Self-Managing} Storage for Enterprise Clusters. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 51–66, 2017.
- [46] N Marivate Vukosi. *Improved Empirical Methods in Reinforcement Learning Evaluation*. PhD thesis, PhD thesis, Rutgers, New Brunswick, New Jersey, 2015.
- [47] Intel. Intel oneAPI Threading Building Blocks. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html#gs.4oc8fg>.
- [48] scikit-learn: machine learning in Python. <https://scikit-learn.org/stable/>. 2024.
- [49] Wenzel Jakob. pybind11 documentation. <https://pybind11.readthedocs.io/en/stable/index.html>. (Accessed on 02/18/2024).
- [50] Yahoo! Cloud Serving Benchmark in C++. <https://github.com/basicthinker/YCSB-C>. 2024.
- [51] Microsoft. Azure/AzurePublicDataset: Microsoft Azure Traces. <https://github.com/Azure/AzurePublicDataset>. 2024.
- [52] 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 *2018 Usenix Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.
- [53] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: stateful functions-as-a-service. *Proceedings of the VLDB Endowment*, 13(12):2438–2452, August 2020.

- [54] 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, 2019.
- [55] Q-learning with Neural Networks. <http://outlace.com/rlpart3.html>. 2024.
- [56] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.
- [57] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

Biography

Include your *biography* here detailing your background, education, and professional experience.