Efficient Storage Virtualization in Cloud Environments

Kevin Nguetchouang Ngongang

THESE
en vue de l'obtention du grade de Docteur, délivré par
L'ECOLE NORMALE SUPERIEURE DE LYON

Ecole Doctorale N° 512
Informatique et Mathématique de Lyon (Infomaths)

Discipline: Informatique

Soutenue publiquement le 12/09/2024, par:

Kevin Nguetchouang Ngongang

Efficient Storage Virtualization in Cloud Environments

Virtualisation efficace du Stockage dans les environnements Cloud

Devant le jury composé de:

David BROMBERG	Professeur des universités	Université de Rennes 1	Rapporteur
Jean-Marc MENAUD	Professeur des universités	IMT Atlantique	Rapporteur
Vania MARANGOZOVA	Maîtresse de conférences – HDR	Université de Grenoble Alpes	Examinatrice
Pierre OLIVIER	Maître de conférences	University of Manchester	Examinateur
Boris TEABE DJOMGWE	Maître de conférences	INP de Toulouse	Examinateur
Pascale VICAT-BLANC	Directrice de recherche	Ecole normale supérieure de Lyon	Directrice de thèse
Alain TCHANA	Professeur des universités	Université de Grenoble Alpes	Codirecteur de thèse

Efficient Storage Virtualization in Cloud Environments

Author:
Kevin Nguetchouang Ngongang (Laboratoire d'Informatique et du Parallélisme, ENS de Lyon)

Supervisors:
Professor Alain Tchana (Laboratoire d'Informatique de Grenoble, Université Grenoble Alpes)
Pierre Olivier, Senior Lecturer Fellow (The University of Manchester)
Pascale Vicat-Blanc, Senior Research Fellow (Laboratoire d'Informatique et du Parallélisme, ENS Lyon)

Résumé

L'avènement du Cloud computing a révolutionné le paysage de la technologie moderne, offrant aux organisations une flexibilité, une capacité de passage à l'échelle et une accessibilité sans précédent dans l'exploitation des ressources informatiques à distance via Internet. Des innovations clés telles que le serverless, les conteneurs et les machines virtuelles ont joué des rôles cruciaux dans le remodelage de la manière dont les entreprises opèrent et fournissent des services. Le serverless, incarné par le "Function as a Service" (FaaS) et mise en œuvre à l'aide de conteneurs, permet aux développeurs d'exécuter du code sans gérer de serveurs, rationalisant les processus de développement et améliorant l'agilité.

À notre époque de prise de décision basée sur les données, des solutions de stockage efficaces, parmi lesquelles la virtualisation du stockage, sont primordiales. La virtualisation du stockage émerge comme un facilitateur de la gestion efficace du stockage dans les environnements cloud, abstrayant l'infrastructure de stockage sous-jacente et permettant une évolutivité et une optimisation des ressources sans faille. Cependant, la croissance rapide des données et l'évolution des besoins des utilisateurs ont amené de nouveaux défis tels que la gestion des sauvegardes, la réduction de la latence d'accès au stockage et la tolérance aux pannes dans les environnements de stockage distribué.

Cette thèse présente trois contributions significatives pour relever ces défis:

- un format de disque virtuel évolutif pour résoudre le problème de passage à l'échelle de disques virtuels composés de longues chaines de snapshots. Ce nouveau format adapte la gestion du disque virtuel lorsque la taille de la chaîne de sauvegarde augmente.
- un système de mise en cache opportuniste des données pour les plateformes FaaS. Ce système exploite l'existence de mémoire surréservée et l'état de maintien actif des environnements d'exécution pour offrir un environnement d'exécution rentable pour les fonctions FaaS.
- un système de stockage distribué qui tire parti de l'existence de répliques secondaires de données pour garantir l'équilibre de la charge des demandes et l'équité de la gestion des ressources.

Nous avons construit un prototype de chacune de nos contributions et validé leur efficacité en effectuant plusieurs évaluations avec une série de benchmarks.

Cette thèse met en évidence la nécessité d'adapter notre compréhension du stockage à un nouveau monde basé sur les données.

Mots-clés: Stockage virtuel, Stockage distribué, Format de disque virtuel, Stockage à distance

Abstract

The advent of cloud computing has revolutionized the landscape of modern technology, offering organizations unprecedented flexibility, scalability, and accessibility in leveraging computing resources remotely over the Internet. Key innovations such as serverless computing, containers, and virtual machines have played pivotal roles in reshaping how businesses operate and deliver services. Serverless computing, incarnated by Function as a Service (FaaS) and implemented using containers, enables developers to execute code without managing servers, streamlining development processes and enhancing agility.

In our era of data-driven decision-making, efficient storage solutions are primordial. Storage virtualization emerges as an enabler of efficient storage management in cloud environments, abstracting underlying storage infrastructure and allowing scalability and resource optimization. However, the rapid data growth and evolving user needs pose challenges such as backup management, latency reduction, and fault tolerance in distributed storage environments.

This dissertation presents three significant contributions addressing these challenges:

- a scalable virtual disk format to address the problem of scaling up virtual disks composed of long chains of snapshots. this disk format adapts the management of the virtual disk when the size of the backup chain increases.
- an opportunistic caching system of data functions for FaaS platforms. this system leverages overbooked memory and the keep-alive state of sandboxes to give a cost-effective running environment for FaaS functions.
- a distributed storage system that leverages the existence of secondary replicas of data to ensure load balancing of requests and resource management equity.

We build a prototype of each of our contributions and validate their effectiveness by conducting several evaluations with a series of benchmarks.

This dissertation highlights the need to adapt our storage comprehension to a new data-based world.

Keywords: Virtual Storage, Distributed storage, virtual disk format, Remote Storage

Acknowledgements

Arriving at the end of this Ph.D. journey, I acknowledge that many people contributed near or far to this dissertation. I want to thank all of them and those who are forgotten, you remain in my heart.

First, I would like to thank my principal advisors Alain Tchana and Pierre Olivier. Through their regular expressions like “Does it work?” they instilled in me the will to do good research and research that matters. They spent much of their time teaching us with other students how to be rigorous and upright. My objective is to be an inspiration as you are and to **never give up**.

I want to thank Jean-Marc Menaud, David Bromberg, Boris Teabe, Vania Marangozova, and Bernabé Batchakui for accepting being on my jury. It is an honor to be reviewed by such known experts. I'm sure I will learn a lot from your criticism.

I want to thank all the members of the ERODS team in the LIG laboratory and, the AVALON team in the LIP laboratory which welcomed me when I arrived in France and tried to put me in a Ph.D. state as fast as possible. A special thank goes to Chiraz Benamor, Marie Narducci, and Marie Bozo, the LIP's administrator assistants for withstanding my numerous requests; you eased my laboratory life.

I want to thank all the members of the "Alain" team as it is often said in his back with whom I worked, discussed, and brainstormed during my Ph.D. as well as my friends, Djob Mvondo, Yves Kone, Stella Bitchebe, Théophile Dubuc, Lucien Ngale, Raphaël Colin, Calvin Abou Haïdar, Carole Djeumo, Papa Assan Fall, Kouam Josiane, Mathieu Bacou, Mohamed Karaoui, Brice Teguia, Augusta Mukam, Patrick Bruel. It was a pleasure and a real-life experience to exchange with all of you. I wish you a good continuation in your respective works.

I want to thank my family, especially my mother Victorine Kemmogne, my aunt Eugenie Tchamou, my cousin Yvan Tchamou, and my little brother Frank Steve Nguenang for their active support and caring during this journey.

As Michael Onyebuchi Eze, an African philosopher, said: *A person is a person through other people*; I am what I am today thanks to all of you.

Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Contents

1.1 Context and Problem Statement

In the rapidly evolving landscape of modern technology, cloud computing has emerged as a cornerstone, reshaping how businesses operate and deliver services [48]. This has been more visible in recent years with for example the venue of the COVID pandemic crisis [28]; thus increasing the flexibility of working options cloud computing provides for employees, notably remote workers, and different levels of data security offered. At its core, cloud computing offers flexibility, scalability, and accessibility, enabling organizations to leverage computing resources remotely over the Internet rather than relying solely on local infrastructure. Cloud computing then offers different types of service denoted as XaaS (X as a Service [33]) where we can find traditional ones such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS) or Function as a Service (FaaS), with any of them used depending on the level of control we want to keep on the infrastructure provided by the cloud provider.

FaaS, sometimes confused as serverless computing, represents a significant departure from traditional server-based models; it has grown in popularity in recent years and its usage is expected

to skyrocket in the years to come [58]. In a serverless architecture, users can execute code without provisioning or managing servers, they write their functions, and the cloud provider executes them on demand following certain triggers, e.g. an event occurs or a click on a website's like. This allows more efficient resource utilization and streamlined development processes. FaaS abstracts away infrastructure concerns, enabling developers to focus solely on writing code, thereby accelerating development cycles and enhancing agility[103, 106].

Virtualization is the main technology that makes cloud computing possible. Indeed, it permits the separation of a physical resource into one or more virtual devices isolated and managed independently, thus ensuring flexibility and scalability as said before. With virtualization, we can create isolated environments such as virtual machines and containers to execute cloud applications. Virtual machines (VMs) provide a virtualized abstraction of physical hardware [10], enabling multiple operating systems and applications to coexist on a single physical server. Moreover, VMs enable rapid provisioning and deployment of computing resources, empowering businesses to scale dynamically in response to fluctuating demand. Another technology, more recently used in cloud computing, is containerization. Containers, like virtual machines, isolate the resources of a physical machine [99]. They are more lightweight in the sense that they don't have to take into account hardware isolation when deployed. They are used as the main unit of execution for FaaS functions. Containerization enables developers to package applications and their dependencies into standardized units, facilitating seamless deployment across diverse environments. By encapsulating applications within containers, organizations can achieve consistency in development, testing, and deployment workflows, leading to improved efficiency and reliability.

In this era of data-driven decision-making and digital transformation (Big Data, AI, IoT), the importance of storage in cloud computing cannot be overstated. As organizations increasingly rely on cloud-based solutions to store, process, and analyze vast amounts of data, efficient and reliable storage solutions are essential. Storage technologies in cloud environments must be scalable, resilient, and cost-effective to effectively accommodate diverse workloads and data types.

To ensure the same flexibility and accessibility of storage as cloud technologies do, we need to take advantage of **Storage Virtualization**. Storage virtualization is *"the process of presenting a logical view of the physical storage resources"* [105]. Storage virtualization emerges as an enabler of efficient storage management in cloud environments. By abstracting underlying storage infrastructure from higher-level storage services, storage virtualization decouples storage resources from physical hardware, allowing seamless scalability and resource optimization [130]. This abstraction layer enables organizations to uniformly aggregate and manage disparate storage resources, simplifying storage provisioning, management, and maintenance tasks.

There are several ways to implement storage virtualization. As depicted in Figure 1.1, when it comes to VMs, hypervisors offer a storage component responsible for managing the virtual disk of the virtual machine. This storage component saves the virtual disk as a file with a specific format (such as Qcow2 [96], VMDK [119], or VHD [76]). These files are generally stored on a dedicated distant distributed storage [46] where features like snapshotting/backup, distribution, and replication [132] are often presented to ensure flexibility and reliability for end-user applications. Concerning FaaS storage, since containers are stateless, functions usually store or retrieve the data needed for their execution from a distant storage (e.g. get a dataset on AWS S3 to apply data analytics processes) [10, 114].

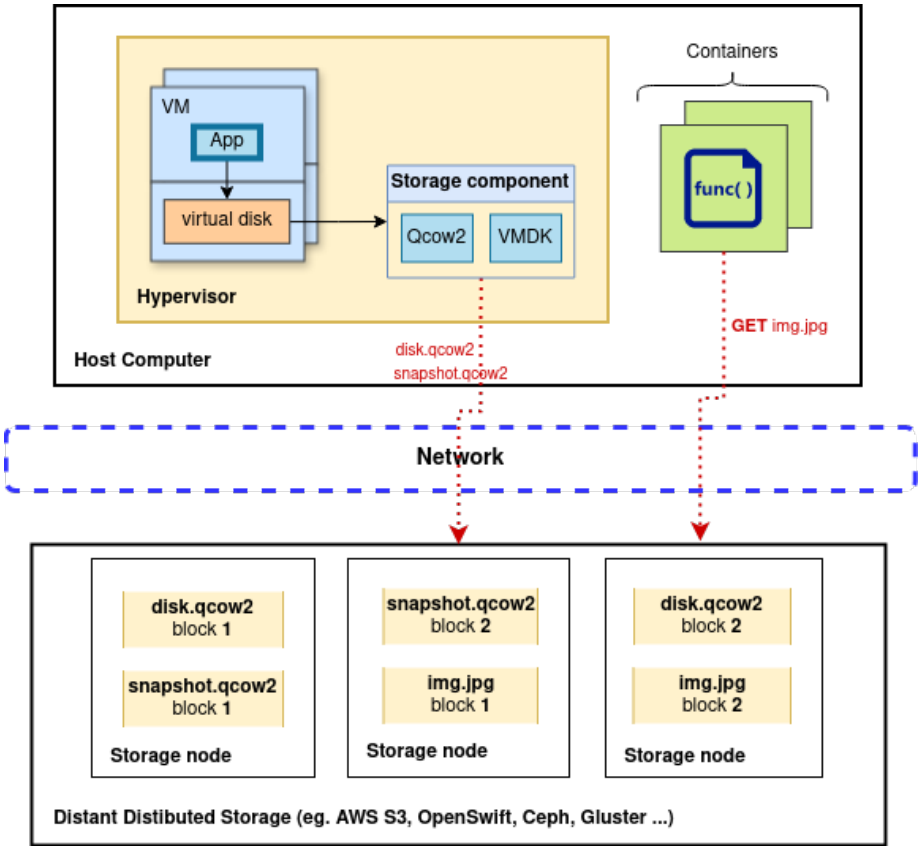


Figure 1.1: Global overview of Virtual Storage in Cloud environments

However, with the rapid growth of data [111], new issues appeared and influenced Cloud environments impacting cloud providers and users.

With the will to maintain trust backup and resilient VMs, users store all the states of their VMs' disk files are used for performing many backups. To ensure optimal backup time, the concept of incremental backup using Copy-On-Write was created. When a backup must be done, there is no copy of the VM disk file; instead, a new empty file is created and becomes the VM's actual disk, while a link with the old file is maintained, which becomes the backup. In case the VM needs to read old data, it can follow the link to read from the backup file. if the number of backups increases, the length of the backup chain also increases, leading to a longer chain. Though the existing virtual disk formats were not designed for this kind of utilization, we need to go through the entire chain following each link to fetch specific data, resulting in performance degradation. On the other hand, while CPU and memory have undergone more research than storage lately, execution of functions is becoming faster and faster. This leads to FaaS functions' bottleneck no longer being function execution time but rather data extraction and storing time since data are often stored on distant storage [37]. Another issue stemming from inefficient storage virtualization concerns the management of the distributed storage cluster itself. Due to possible storage server

faults, we need to efficiently manage replication on the distributed storage to maintain constant high performance of data-intensive end-user applications. Additionally, because storage nodes are often unequally utilized [72], we must ensure the best usage of each storage node while managing replication.

Overall, current cloud storage virtualization offerings are suboptimal as they require too much input/expertise from non-expert users and they fail to scale to modern environments and to conditions of operations that have become common in today's data centers (e.g. high snapshot frequency, data access latency, variability in the latency/performance of the various nodes composing a distributed storage system)

To caricature briefly, too much data can slow down cloud systems. Making copies for safety can make things slower, and even though computers are fast, getting data from storage can be slow.

This dissertation details three significant contributions that propose potential solutions to the problems mentioned above.

1.2 Contributions

1.2.1 OFC: Opportunistic FaaS Cache

The first contribution of this dissertation tackles the problem of FaaS functions' latency. Existing works aim to provide a local storage for the functions when executed. Past works [62, 22] proposed to dedicate resources (providing improved storage performance) that must be booked, configured, and tuned by the cloud tenants, which is at odds with the benefits (don't take care of the hardware deployed) expected from the serverless. We tackle this problem by proposing a more intelligent system **OFC**. **OFC** is an opportunistic caching system that leverages memory that would otherwise be wasted due to memory over-provisioning and sandbox keeping alive and uses this memory to serve as a cache for the distributed storage of the FaaS platform. The relevant contributions are as follows:

- OFC leverages Machine Learning especially decision trees to accurately predict function memory needs in a FaaS system;
- OFC leverages overbooked memory as well as sandbox keep-alive to design a cost-effective, elastic, and fault-tolerant caching system;
- We implement OFC in Apache OpenWhisk [88] an open-source well-known FaaS system.
- We present the evaluation of OFC, demonstrating that it improves by up to 82% and 60% respectively the execution time of single- and multi-stage FaaS applications.

1.2.2 SVD: Scalable Virtual Disk format

In the second contribution of this dissertation, we address the problem of performance degradation in long snapshot chains generated by both cloud providers and users and try to resolve it by proposing a new virtual disk format denoted as **SVD**, Scalable Virtual Disk.

The relevant contributions are as follows:

- We characterize snapshot length in a large-scale cloud provider, Outscale [91]. We assess for the first time performance and memory footprint scalability issues due to a long snapshot chain and explain the origin of the problems;

- We design and implement **SVD**, a scalable virtual disk format, a retro-compatible extension of the Qcow2 format addressing these problems;

- We evaluate a prototype of our solution in various situations, demonstrating the effectiveness of our approach by improving up to 30% of performance on RocksDB benchmarks and by reducing to 15× the memory footprint.

1.2.3 Nami, a constraints-aware replica selection system

Finally, this dissertation's third contribution tackles the NUDA problem (Non-Uniform Disk Access). Due to a non-optimal implementation in known large distributed storage systems (Ceph [24], Gluster [52] ...), sometimes storage nodes are not equally used leading to a subset of storage nodes to be overused affecting the application's performances. **Nami**, the proposed system, takes advantage of the replica to ensure an equal load balancing in distributed storage systems. The relevant contributions are as follows:

- We present our system **Nami** based on metrics chosen by system administrators which leverage the existence of secondary replicas to ensure load balancing and equal usage of resources;

- We evaluate Nami and demonstrate an improvement in I/O throughput up to 30% on various benchmarks and a reduction of the variance in resource utilization to 14% for a 10-storage nodes cluster.

1.3 Scientific Publications

International Conferences and Journals

- **Nami: Navigating between replicas in distributed storage systems**
  Kevin Nguetchouang, Pierre Olivier, Alain Tchana
  IEEE Transactions on Cloud Computing — *TCC 2024 (under review)*

- **SVD: A Scalable Virtual machine Disk Format**
  Kevin Nguetchouang, Stella Bitchebe, Theophile Dubuc, Mar Callau-Zori, Christophe Hubert, Pierre Olivier, Alain Tchana
  IEEE Transactions on Cloud Computing — *TCC 2024*

- **An Opportunistic Caching System for FaaS platforms**
  Djob Mvondo, Mathieu Bacou, **Kevin Nguetchouang**, Lucien Ngale, Stéphane Pouget, Josiane Kouam, Renaud Lachaize, Jinho Hwang, Tim Wood, Daniel Hagimont, Noël De Palma, Bernabé Batchakui, Alain Tchana
  European Conference on Computer Systems — *Eurosys 2021*

National Conferences

- **sQemu : vers un stockage virtuel scalable**
  Kevin Nguetchouang, Stella Bitchebe, Théophile Dubuc, Pierre Olivier, Alain Tchana, Mar Callau-Zori
  COMPAS 2022

- **FaaSCache : Système de cache mémoire opportuniste et sans surcoûts pour le FaaS**
  Kevin Nguetchouang, Lucien Ngale, Stephane Pouget, Djob Mvondo, Mathieu Bacou, Renaud Lachaize
  Journées Cloud 2020

1.4 Outline

The rest of the document is structured as follows.

- Chapter 2 details our contribution related to `OFC`, a caching system for FaaS platforms. It presents the Machine learning process, cache design, and evaluation.
- Chapter 3 presents a characterization of snapshots/backups chain in a cloud environment, our scalable virtual disk format `SVD`, and its evaluation.
- Chapter 4 assesses the problem of non-equally used disks in distributed storage, our solution `Nami` leveraging secondary replica and its evaluation.

Ending this dissertation, a conclusion that resumes our findings and shares some interesting perspectives for the future.

CHAPTER 2

Efficient Storage for FaaS using an Opportunistic Caching system

In this chapter, we introduce OFC, a transparent, vertically and horizontally elastic in-memory caching system for FaaS platforms, distributed over the worker nodes. We build our OFC prototype based on enhancements to the OpenWhisk FaaS platform, the Swift persistent object store, and the RAMCloud in-memory store.

Contents

2.1 Introduction

Over the past few years, the *Function as a Service* (FaaS) paradigm has deeply reshaped the way to design, operate and bill cloud-native applications. The user just uploads the functions, which will be automatically triggered by events (e.g., timer, HTTP request). Because of their stateless nature, most functions follow the *Extract-Transform-Load* (ETL) pattern [37], meaning that the function first (E) reads data (e.g., an image) from a remote persistent storage service (e.g., an object store such as AWS S3), then (T) performs some computation (e.g., image blurring), and finally (L) writes the result to the remote storage.

The two main performance limitations of current FaaS platforms are well known [58, 53], namely their scheduling latency and function execution latency. The former limitation has received a lot of attention in recent years [73, 5, 85, 77, 37, 3, 32, 19]. Here, we focus on the latter. An acute issue here is the performance bottleneck introduced by the lack of data locality. Indeed, current FaaS platforms are typically made from two very distinct layers: a compute infrastructure and remote storage backends. This decoupling is a double edged sword: FaaS applications benefit from unmatched elasticity but are hurt by the latency and throughput limitations of the backends for any access (*E&L* phases) to persistent or shared transient state. This problem is exacerbated in the case of function pipelines (e.g., for analytics) [62, 22] where the output of a function is the input of another, and intermediate outputs/inputs are destroyed once used, because most of the current FaaS platforms do not support direct and efficient communication between function instances [58, 53]. Prior works [62, 22] proposed to dedicate resources (providing improved storage performance) that must be booked, configured and tuned by the cloud tenants, which is at odds with the benefits expected from the serverless paradigm. Other works focusing on function pipelines have enabled direct and efficient communications between function instances, by leveraging platform-specific assumptions and features [5, 108, 125]. The Cloudburst platform [110] improves locality via per-worker data caches, which interact with a specialized storage backend using specific, coordination-free consistency protocols; to the best of our knowledge, Cloudburst requires manual/static provisioning of dedicated cache resources (esp. memory) on each worker.

In this chapter, we present OFC (Opportunistic FaaS Cache), an *opportunistic* RAM-based caching system to improve function execution time by reducing *E&L* latency, both for single-stage functions and function pipelines. OFC achieves this in a cost-effective manner for the cloud provider, no additional efforts (administration, code modifications) for cloud tenants, and no degradation of the data consistency and persistence guarantees. To achieve this, *OFC repurposes memory which would otherwise be wasted due to memory over-provisioning and sandbox keep alive*. Indeed, the analysis of FaaS traces (including the AWS Lambda repository) [109] reveals that users tend to over-provision the memory capacity guaranteed to their functions. Also, to accelerate function instance setup, FaaS platforms typically keep function sandboxes alive for several minutes (e.g., 10 in OpenWhisk and 20 in Azure) [106, 122] for serving future invocations related to the same function code. OFC opportunistically aggregates these idle memory chunks from all worker nodes to provide a distributed caching system for *E&L* phases.

This idea raises three main questions: *(Challenge #1)* How to accurately predict, at the function invocation granularity, the amount of memory needed by a sandbox (warm or cold)? *(Challenge #2)* How to build a vertically scalable caching system, with a capacity (up/down) scaling latency that is adequate even for short function executions (i.e., in the range of seconds or milliseconds) [109]? *(Challenge #3)* How to manage the caching system transparently (unmodified application code), efficiently (consistency and performance guarantees), and reliably (persistence and fault tolerance)?

For Challenge #1, we leverage machine learning (ML). Contrary to IaaS, in FaaS the provider has access to a wealth of information (such as function code, actual input parameters and runtime library), which makes ML appropriate in this context. In addition, the very high rate of invocation of a (performance sensitive) cloud function makes learning a model fast, because a training dataset is quickly accumulated. Moreover, most functions use a well-known set of common data types (e.g. multimedia), and their resource needs (especially memory) are correlated (although non-trivially) with some input parameters. OFC goes further and, for each function, extracts the main features that characterize its memory requirements and builds its prediction model. When a function is invoked, the actual memory size assigned to its sandbox is calculated using its associated ML model. Therefore, the difference between the booked memory (specified by the cloud tenant) and the predicted size is used for increasing the size of the cache on the worker node that hosts the sandbox.

For Challenge #2, the difficult aspect is the scaling down of the caching system when a worker node is lacking memory. To address this challenge, we use two strategies. First, we reduce the pressure on the caching system by evicting as early as possible data that are unlikely to be reused in the future. Second, we implement an optimized algorithm for object migration, allowing to keep hot objects in another worker node of the distributed cache.

For Challenge #3, we leverage ML, as well as several systems and data caching policies. First, a data item is cached only if it is likely to significantly improve the overall function execution time. To this end, OFC builds another model that outputs an estimation indicating whether or not the cache would yield improvements. Second, intermediate input/output data generated by pipelined functions are removed (but not persisted to the remote storage) from the caching system once the pipeline execution ends. Finally, to achieve the remaining goals, OFC implements several other techniques, among which: (i) asynchronous data persistence on the remote storage implemented using helper FaaS functions, (ii) adaptation of the FaaS scheduler for locality (functions are preferably run on a worker node hosting a copy of the cached data), and (iii) consistency management on the remote storage backend using "shadow objects" (i.e., placeholders for new object versions). We prototype OFC using popular software stacks: Apache OpenWhisk (OWK) [88] as the FaaS platform and OpenStack Swift [114] as the remote storage. We use RAMCloud [89] as the distributed in-memory caching system.

In summary, we make the following contributions: (i) leveraging ML to accurately predict function memory needs in a FaaS system; (ii) leveraging overbooked memory as well as sandbox keep-alive to design a cost-effective, elastic and fault-tolerant caching system; (iii) the evaluation of OFC, demonstrating that it improves by up to 82 % and 60 % respectively the execution time of single- and multi-stage FaaS applications.

2.2 Background and motivations

2.2.1 Background

We now provide background details on Apache OpenWhisk (OWK), the FaaS framework that we leverage in our prototype implementation. Like most FaaS platforms, OWK supports polyglot users to pick a programming language of their choice (e.g., Python, NodeJS, etc.) and ways to configure trigger rules (e.g., HTTP requests to a given URL, updates within a given object storage bucket, etc.). Figure 2.1 highlights the basic steps of FaaS platforms' functioning.

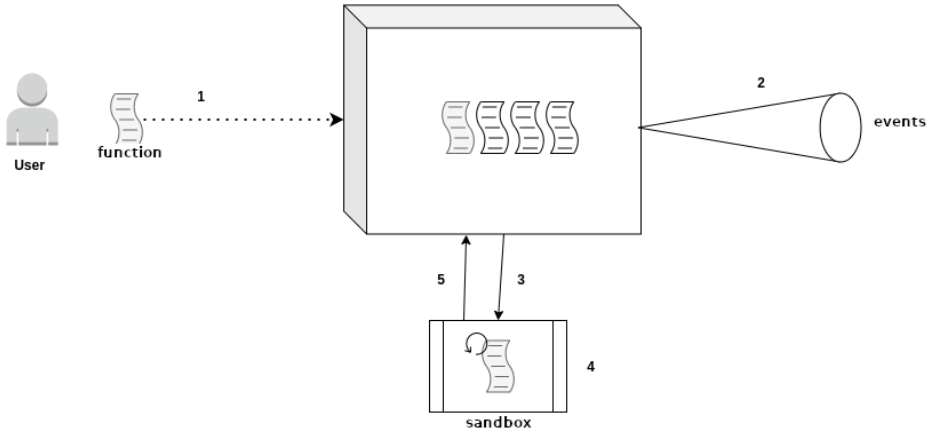


Figure 2.1: General FaaS functioning

1. The user submits the source code of his function on the FaaS platform and thus creates a function.
2. This function is associated by the user with one or more event source(s) using rules.
3. When an event from a source occurs, the platform automatically triggers the execution of the function in a sandbox, which is most often a container.
4. The function runs in the sandbox and performs the specific task that the user gave it
5. At the end of its execution, it notifies the platform and sends it the results of its execution.

In addition to single-stage functions, OWK provides support for function *pipelines* (a.k.a. "workflows" or "sequences"), which consist of a parallel and/or sequential composition of functions as you can see in the first function invocation is triggered by an external event and the next ones are driven by the platform, based on the completion of the function invocations and their dependency graph. Function pipelines are becoming increasingly popular for implementing massively parallel tasks (e.g., data analytics) in a simple and cost-effective manner [57, 62, 92, 58, 22].

Sandbox management When a function invocation is triggered, the corresponding request is forwarded to the OWK *Loadbalancer*, which is responsible for choosing a worker node that will run the function. To do so, the *Loadbalancer* maintains the current status (e.g., available resources) of all worker nodes and, using a hash of the function ID and tenant computes the identifier (index) of the *home* worker, which is the preferred worker for handling the request. This affinity is aimed at improving code locality on the workers. Each worker node hosts a component named *Invoker*, in charge of informing the *Loadbalancer* with the current status of the node and creating and starting function *"sandboxes"*. The latter are implemented with Docker containers in OWK.

Finally, we highlight three important aspects of sandbox management, which are also common to most FaaS platforms. First, for security, a given sandbox is never shared or reused between distinct functions or tenants. Second, a sandbox processes only a single invocation at a time. Third, to

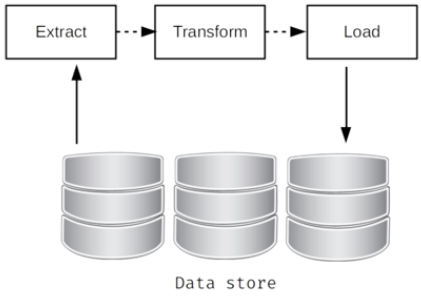


Figure 2.2: Extract-Transform-Load

amortize start-up costs and mitigate cold-start effects, a sandbox is generally kept alive for some time (600 s in OWK) in anticipation of future invocations of the same function.

FaaS storage management Since users writing their functions have no control over the underlying server resources, they cannot use the local storage to store their data. They are mandatory to use distant storage to fetch data needed by their functions and then store their results on that distant storage; leading them to an Extract- Transform-Load model executing function (see Figure 2.2).

This means that functions can be severely impacted by the time of **E**xtraction and **L**oad of data depending on the access times to the chosen distant storage.

2.2.2 Motivation

This section shows that worker nodes in FaaS platforms have an abundance of wasted memory capacity that can be used to build a distributed caching system cost-effectively, i.e., without the additional infrastructure required in prior works [62, 22]. We explain why machine learning is needed to reach this goal, and we also show that ETL-based functions could be a potential beneficiary of such a caching system.

2.2.2.1 Memory waste

Figure 2.3 illustrates how a sandbox uses memory during its lifetime on a worker node. In this example, the sandbox handles three events (`E1`-`E3`), which trigger three sequential invocations of the same function. We summarize in this figure the two main sources of memory waste in FaaS platforms.

The first cause of waste is that cloud tenants often overdimension the memory resources configured for their function sandboxes. For example, a survey of AWS Lambda usage reports that 54 % of sandboxes are configured with 512 MB or more but that the average and median amounts of used memory are actually 65 MB and 29 MB [101]. We believe that this trend is mainly due to workload variations: the same function code can be triggered with different arguments and input data, which may lead to different memory needs. For illustration, Figure 2.4 plots the memory usage of a sample function that blurs an image, as a function of the input size and as a function of its processing-specific argument (the blurring radius).

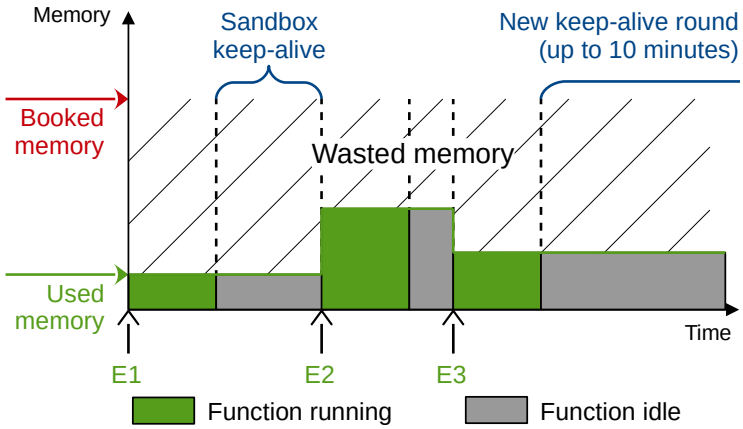


Figure 2.3: Timeline of a function sandbox with memory waste due to over-dimensioning and the keep-alive policy.

It shows that for the same function, memory usage varies widely depending on the arguments and input data, justifying resource over-dimensioning practiced by cloud tenants. Hence, we believe that this trend is likely to remain and that the resulting waste may even grow as the available range of memory configurations for sandboxes keeps increasing [13].

The second cause of waste is the sandbox keep alive policy. Indeed, to mitigate the long latency of sandbox cold starts, FaaS platforms typically keep a sandbox running for several minutes (e.g., 10 in OWK and 20 in Azure) [106, 122] after handling the first event (`E1` in Figure 2.3) that triggered its startup. Consequently, the resources assigned to a sandbox may remain unused for long time intervals. Shahrad et al. [106] observed in Microsoft Azure Functions that "45 % *of the applications are invoked once per hour or less on average, and* 81 % *of the applications are invoked once per minute or less on average. This suggests that the cost of keeping these applications warm, relative to their total execution (billable) time, can be prohibitively high.*". The authors introduced a histogram-based solution to predict invocation frequencies and patterns for each function; this way, a sandbox can be started just before the next function invocation and shut down upon completion. Such an approach works well for some workloads but must fall back on sandbox keep-alive in various circumstances (e.g., phase changes, burstiness), which are likely to become more prevalent as FaaS is increasingly used for more diverse and intensive applications, and also used as a means to absorb unpredictable load spikes. Besides, despite a number of recent optimizations, the overhead of cold starts is still significant: with general purpose, production-grade sandboxing technologies (e.g., containers or microVMs), a cold start latency under high load is in the range of hundreds of milliseconds [73, 77, 3], which is sensitive for very short functions and interactive/parallel applications. Consequently, some FaaS providers offer the possibility to book pre-provisioned, long-lasting sandboxes [11]. Overall, sandbox keep-alive remains an important technique for FaaS platforms for the time being, and we propose to leverage the (physical) memory waste that stems from it.

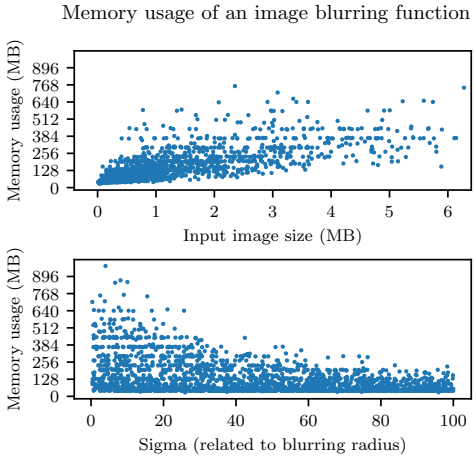


Figure 2.4: Illustration of the relation between a function's memory usage and two parameters: byte size of input data (top), and a function-specific argument (bottom).

2.2.2.2 Memory prediction using machine learning

In order to use the wasted memory described above, we need to estimate how much is available. In this section, we justify why we turn to machine learning for this task.

Figure 2.4 illustrates the complex relation between a function's arguments and input data, and its memory usage. In particular, the top figure plots memory usage against the byte size of the input data: we see that no precise correlation can be established. In other words, accurately predicting memory waste only from the byte size of the input data is not possible. Additionally, the bottom figure plots memory usage against the function's specific argument (the blurring radius): again, no precise correlation can be established from this feature alone. We have observed the same kind of trend with many other multimedia processing functions (e.g., image resizing and format conversion, speech recognition, video grayscale conversion, text summary). Furthermore, the FaaS platform has no information about a function's specific arguments: it only knows about their list and names, not about their nature, range, behavior, etc., thus adding complexity to the task of predicting the memory usage of a function.

In summary, predicting memory waste for a function invocation requires to take into account uncharacterized function-specific arguments in addition to features of the input data. As shown below, machine learning (ML) can manage this complexity without prior knowledge.

2.2.2.3 Remote shared data store latency

We measure the impact of the systematic utilization of a remote shared data storage (hereafter RSDS) imposed by the FaaS model to ETL-based functions. To this end, we run in AWS example single-stage functions (image processing) and example multi-stage functions (analytics). We use AWS S3 as the RSDS and we experiment with various input sizes. Figure 2.5a, first bar series, presents the contribution of each ETL phase for one image processing function (`sharp_resize`). For instance, *E&L* represents up to 97 % of the total execution time for a 128 kB image size. Figure 2.5b,

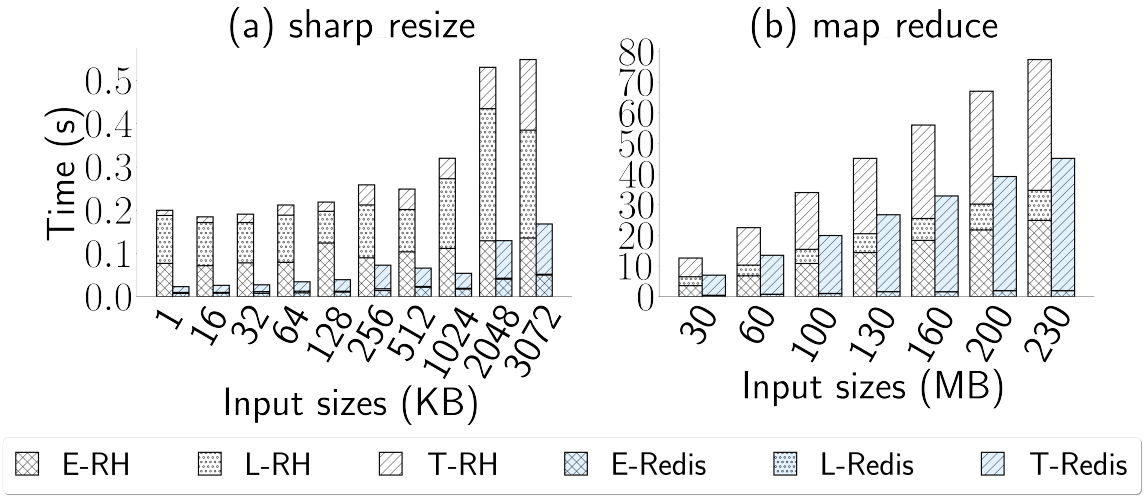


Figure 2.5: Duration of the ETL phases for a common image processing function (resizing) and a pipeline of data analytics functions (word count). The functions run on an OWK environment deployed on AWS EC2 using 3 compute nodes provided by `t2.medium` instances. We measure the execution times using AWS S3 as our RSDS (first bar series) and ElastiCache Redis as our IMOC (second bar series).

first bar series, presents results for MapReduce word count. *E&L* represents up to 52 % of total execution time for a 30 MB input text file.

A typical way [22] to address this bottleneck is to use an in-memory object cache (IMOC) such as Redis between the cloud functions and the RSDS. The second bar series in Figures 2.5a and 2.5b show the results of the same experiments as above when S3 is replaced with Redis. We can see that the contribution of *E&L* becomes negligible. However, the utilization of such an IMOC-based solution is not without downsides for FaaS tenants. They must explicitly allocate and manage IMOC resources. In addition, they must modify their function's code (or introduce a library) to interact with the IMOC layer and also to address potential consistency issues. All these constraints and extra burden are at odds with FaaS principles. OFC, the system that we propose, achieves performance benefits comparable to an IMOC-based solution but without the aforementioned limitations.

2.3 Design assumptions

Our design principles are not tied to the specific components used in our prototype (OWK and Swift) but they do rely on a small number of assumptions, which we mention below. Overall, we assume a fail-stop model (with details similar to RAMCloud [89]).

For the FaaS infrastructure, we only assume two characteristics, which are very common in current platforms: (i) the use of a sandbox keep-alive strategy (either via a simple idle timeout like in OWK and AWS Lambda, or through a control-loop approach like in Kubernetes-based platforms) and (ii) the use of a (per-function) central component to dispatch invocation requests to

the sandboxes (like the controller component of OWK or the scheduler component of Kubernetes-based platforms).

For the remote storage system, we do not make any major assumption. First of all, we are agnostic to the data abstraction level (e.g., file-based, object-based or key-value interface). We simply assume the possibility to register handlers, to be triggered upon the invocation of certain operations. In addition, we aim at supporting storage systems with various consistency guarantees including strong ones, such as linearizability. We believe that this is important because this simplifies the work of applications developers (a number of object storage systems are now evolving towards stronger consistency semantics [47, 12]) and a strongly consistent storage backend also simplifies the design of hybrid applications combining FaaS and other services (e.g., Infrastructure as a Service). For caching, we assume that the remote storage supports transparent interposition. We store full copies of the object data in the cache, and we focus on small objects (10 MB or less) because they benefit the most from caching (as shown in §2.2.2.3). Note that some of the workloads that we study use large data sets (hundreds of megabytes) but that the corresponding input, intermediate and output data are actually split into many small objects.

Finally, we assume that the ML infrastructure has access in clear text to the function invocation arguments and also to the object's metadata. We leave the support of black-box/encrypted inputs to future work.

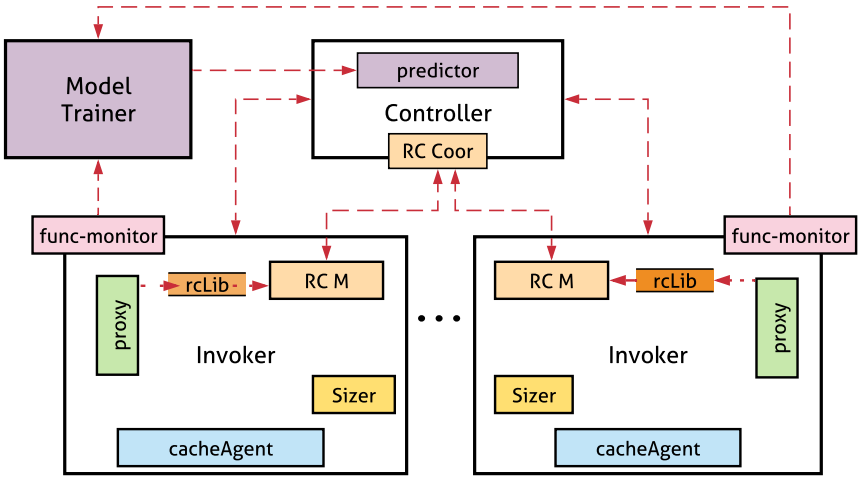


Figure 2.6: OFC architecture overview. All color-filled boxes are new components that we add to OWK.

2.4 OFC overview

Figure 2.6 presents the architecture of OFC. Components that we add to OWK in order to provide OFC are indicated by color-filled boxes in the diagram and "*new*" in our description. When OWK's *Controller* receives a function invocation request, it first asks the (*new*) *Predictor* to predict the

amount of memory (noted M_p) that should be assigned to the sandbox (details in §2.5.1). The *Predictor* also returns a boolean (noted *shouldBeCached*) indicating whether it is beneficial (i.e., will lead to a significant decrease in execution time) to cache the data objects read/written during the function invocation (§2.5.2). Based on M_p, the *Controller* then selects the invoker (worker) node that will handle the function invocation request (§2.6.5). OFC modifies the native selection algorithm, by taking into account (i) the amount of memory currently provisioned in the existing and idle sandboxes for the same function (if any) and (ii) data locality (an invoker node holding the master/in-memory copy of an object in its cache instance is prioritized). In addition to the default information sent by the *Controller* to the selected invoker node, the request also includes M_p and *shouldBeCached*. Upon receiving this information, the invoker node, via the (*new*) *Sizer* component, (re)sizes the sandbox.[1] Finally, the (*new*) *CacheAgent* resizes (down/up) the caching storage (§2.6.4) depending on the new contribution of the function to the cache.

Read and write operations performed by a function to the RSDS are transparently captured at execution time by the (*new*) *Proxy* component, which uses our (*new*) `rclib` library to redirect them to the caching system. To provide the basic functionalities of the latter, OFC relies on RAMCloud [89] (components with *RC* prefix in the figure), a RAM-based durable key-value store (§2.6.1). For write operations, the caching storage is only solicited when *shouldBeCached* is true (§2.6.3 for the caching policies). In this case, *Proxy* injects in the FaaS platform a *Persistor* function to asynchronously persist the cached data to the RSDS (§2.6.2).

To take into account false predictions of M_p (lower than the actually needed memory), the execution of each function is locally monitored by a (*new*) *Monitor* component (§2.5.3). The latter has two goals. First, it can ask the *Sizer* to quickly increase the memory capacity of a sandbox when the latter lacks memory. Second, after the completion of every function invocation, it sends the (maximum) amount of memory consumed to the (*new*) *ModelTrainer* component. The latter periodically retrains all memory prediction models using these numbers and updates the *Predictor*.

Overall, our prototype implementation requires the following volumes of new or modified lines of code: 5 kLoC for OWK (7.5 % of the original code base) including the ML part, 10 kLoC for RAMCloud (6 %), 15 LoC for Swift (0.3 %).

2.5 ML modules

The *Predictor* and the *ModelTrainer* work together to give predictions on two topics: (i) the memory requirements of a function invocation, and (ii) the performance improvement that the latter can achieve with the cache.

2.5.1 Prediction of physical memory requirements

The *Predictor* performs on a per-invocation basis: using the function's memory model, learned by the *ModelTrainer* through the function's lifetime. It takes as input the parameters of the function invocation request, and outputs the predicted memory requirement of the invocation. We store all the function models in OWK's database (CouchDB), so when a function is invoked and OWK fetches its metadata, it also gets its model to be used by the *Predictor*.

[1] On a cold start, the invoker will create the new container with this memory constraint; on a warm start, it updates the memory constraint of the existing container selected to run the invocation.

2.5.1.1 ML algorithm

Our choice of the ML algorithm, and the features considered for modeling, is guided by the following constraints, on which we expand below: model update, model output, model inputs, and prediction speed.

Model update We must be able to update the model throughout a function's lifetime for two reasons: first, the model is blank when the function is uploaded to the FaaS platform, and second, once the model is in use, it must be corrected in case of bad predictions. This warrants either for an algorithm that accepts incremental updates, or for maintaining a training dataset that is updated with more exhaustive data before retraining the model.

Model output We loosely defined the model's goal as "predicting the amount of memory", but actually we settle with predicting a *range*. Indeed, OWK defines a range of permitted memory allocations, ($[0, 2]$GB by default). We divide this into intervals in order to formulate the model as a *classifier*, making it easier to do predictions, as commonly practiced [26]. Hence, the amount of memory to allocate is the upper bound of the predicted interval. We discuss the size and number of classification intervals in §2.7.1.

Model inputs and specificity The input to the model can only include features that are readily available from an incoming request to the FaaS platform. For instance, a function that blurs images does not expect the same arguments as a function that compresses audio; because the nature of function inputs (image, audio, video, etc.) vary, so do the features that can be extracted from a request as input to the model. Thus, we must learn one model per function, that is capable of handling function-specific arguments.

Prediction speed The memory prediction intervenes on the critical path of a function invocation, which must guarantee low latencies. Wang et al. [122] measured median cold start latencies for various FaaS providers in the order of 100 ms, and median warm start latencies in the order of 10 ms. Thus, we set the target prediction time at 1 ms.

Based on these criteria we choose *decision trees*: they can be seen as a cache lookup operation, where the *Predictor* looks up the amount of memory used previously for a given set of features [117]. Moreover, decision trees are fast at classifying. We elect to use the J48 decision tree algorithm (a Java implementation of C4.5 [98]) with 16 MB intervals from 0 to 2 GB. Section 2.7.1 shows the results validating our choices.

2.5.1.2 Feature selection and processing

We select features depending on the function's input type: image, audio, and video. When applying ML to such inputs, the usual goal is to *characterize the content*, e.g., identifying shapes in an image. However, this is not the case here: our models do not learn the content of the processed media, but rather the descriptive features available in the function request parameters or media metadata, which may affect memory usage.

J48 does not require pre-processing the features, but we avoid extracting them on the critical (invocation) path whenever possible. We leverage the fact that in our situation, all data objects

reside in the RSDS: we extract the features when an object is created, and store them alongside it, as a background task. The only case in which we perform the extraction synchronously is when a function invocation stems from a storage trigger (object creation or update). We define a set of common input features, e.g., input file size is used for all functions, while image processing functions include pixel dimensions, and audio/video functions include duration, etc. Further, function-specific arguments are also used as input features (for instance, an image blurring function would receive, along with the image itself, a blurring radius). The evaluation results in §2.7.1 demonstrate that these feature sets are sufficient to make accurate predictions.

A FaaS platform such as OWK has the knowledge of the list and values of arguments sent to a function, so it is easy to extract them to be used as features. However, no semantic information is known about the arguments themselves. This raises two challenges. First, how to identify the function arguments that correspond to object identifiers (needed for feature extraction, as mentioned above)? In the case of functions triggered by object creations, the target objects are determined automatically; otherwise, we rely on manual annotation of the function arguments. Second, regarding the remaining arguments, how to feed the ML algorithm with their opaque values? (Are they floats? What is their range? Are they nominal/discrete values?) A benefit of decision trees such as J48 is that this information is mostly not required; only for nominal values do we need to learn their ensemble. This is easily done because we actually keep a training set of invocations in order to update the model when necessary (see §2.5.3), so we can have an exhaustive view of all the nominal values that the function ever received.

2.5.2 Caching benefit prediction

We state that caching is beneficial for a given invocation when the (wall clock) time taken to extract and load the data dominates the total execution time. This is expressed as the ratio $\frac{T_e+T_l}{T_e+T_t+T_l}$ being greater than 0.5; with T_e, T_t, and T_l corresponding respectively to the time taken by the extract, transform, and load phases. In this case, the *Predictor* is a binary classifier that outputs a boolean telling if caching is useful. Learning this information is similar to predicting memory intervals: we also use J48, with the same features, learning one model per function.

2.5.3 Managing prediction errors

2.5.3.1 Memory prediction

Underprediction has negative effects on the invocation: it may experience swapping activity, resulting in degraded performance, or even abrupt termination by the *Out-Of-Memory (OOM) killer* daemon of the Linux host.

As a first step to avoid this situation, the memory predictions are not actually used until the ML model is accurate enough, which we define as a *maturation criterion*.

Definition 1. *Let C_k be the k-th classification interval; a greater k corresponds to a higher amount of predicted memory, and k^* is the index of the true interval.*

The maturation criterion is:

- 90 % exact-or-overpredictions (*EO-predictions*): the model predicts C_k with $k \geq k^*$ for 90 % of the cases;

- 50 % of underpredictions are within one interval of C_{k^*}: when the model predicts C_k with $k < k^*$, we have $k = k^* - 1$ for at least 50 % of the cases.

Once the predictor meets these requirements, we further mitigate the underprediction problem in three ways. First, we conservatively use the next greater interval as the predicted memory amount, which ties into the criterion of "50 % of underpredictions are within one interval of the correct prediction" described above. By doing so, 50 % of underpredictions become exact predictions, and we ensure that we have $(0.9 + 0.1 \times 0.5) = 95\,\%$ EO-predictions. Second, if an invocation fails because of the OOM killer, it is immediately retried with the memory limit raised to the amount set by the tenant. Third, OFC also monitors invocations during their execution to measure the actual memory usage (by periodically reading statistics from `cgroup`, the facility used by Docker). Whenever a problem of memory exhaustion is detected, the model is corrected quickly to take into account this error for future invocations under the same conditions.

In addition, OFC also attempts to dynamically detect sandboxes with high memory pressure and dynamically raise their memory cap. We enable this approach only for invocations that have run for at least 3 s. Indeed, shorter invocations are frequent (50 % of the invocations in the study of Shahrad et al. [106]) and unlikely to be affected by under-predictions for memory sizing. Hence, we avoid the monitoring overheads in the case of short invocations.

2.5.3.2 Caching benefit prediction

An error from this model will not degrade performance compared to a setup without a cache. If the cache is predicted useless but could have been useful (false negative), there is no performance degradation, only a lost opportunity; and in the event that the cache is predicted useful but ends up useless (false positive), it only puts a slight overhead on the *CacheAgent* component.

2.5.3.3 Retraining

For both models, prediction errors are corrected after the fact by periodically updating them. Given that J48 is not an incremental model, the *ModelTrainer* needs to fully re-train the models when new data is available. We make this practical by maintaining a small, but valuable training dataset: after the Predictor maturation criterion described above is reached, we only add data about invocations for which the memory model predicted an interval that was too low, or extremely too high (the model predicted C_k with $k - k^* > 6$). We also give a higher weight to the training data about underprediction cases in order to better avoid them.

2.6 Cache design

This section details how OFC implements its caching system in OWK, and how it is managed and used.

2.6.1 Cache storage

Our infrastructure leverages the RAMCloud [89] distributed key-value store (with data partitioning and replication) for the management of the cached data. More precisely, in our design, each machine running an OWK Invoker also hosts an instance of a RAMCloud *storage server* (which comprises two components: a *master* and a *backup*; the former manages the in-memory storage of the primary

copy for some of the objects and the latter handles the on-disk storage for the backup copies of other objects). The storage capacity of RAMCloud is dynamically adjusted, both horizontally and vertically. Unlike in a vanilla RAMCloud setup, OFC allocates only a fraction of an Invoker machine's resources to a storage server; this fraction depends on the memory booked but left unused by functions. Section 2.6.4 describes the scaling process of each server instance.

We chose to use RAMCloud for four main reasons: (i) it is specifically aimed at aggregating the (main memory and disk) capacity of the cluster nodes, (ii) it achieves very low latency, (iii) it provides strong consistency and fault tolerance guarantees, and (iv) it ensures durability and efficient RAM usage (backup copies are stored on disk rather than RAM). Besides, RAMCloud is optimized for storing small data objects, which is in line with the object sizes that benefit the most from the cache,[2] for the workloads that we consider (see §2.2.2.3). We leave for future work the (efficient) support for arbitrary object sizes.

Regarding fault tolerance, the cache mainly relies on the support provided by RAMCloud (replication and fast recovery) and OWK (retries of failed/timed-out invocations). The cache is transparent regarding the fault tolerance model to be considered by application developers (functions are expected to have idempotent side effects).

2.6.2 Persistence and consistency

Given our objective of transparency, the caching layer introduced by OFC must not degrade the consistency and persistence guarantees offered by the RSDS.[3] This section describes how we achieve this goal. In a second part, we then explain when and how these constraints can be relaxed in order to improve performance.

To keep the RSDS up to date, OFC must synchronously forward write requests (i.e., regarding a `create`, `update` or `delete` operation for an object) to the RSDS. The rcLib uses the following approach in order to achieve better performance: the synchronous request issued to the object store contains an empty payload and is used to create a placeholder (hereafter named *"shadow"*) for the newly created/updated object *Obj*. It is associated with a set of metadata tags (both in the cache and in the RSDS): two version numbers, respectively for the latest version of *Obj* and the latest version available in the RSDS (a discrepancy between the two indicates that the RSDS does not store *Obj*'s current data payload). Once the synchronous RSDS request has completed and the write has been persistently stored in RAMCloud, the rcLib acknowledges the request to the client application (function) and schedules the *persistor*, a background task running as a (FaaS) function. The *persistor* code consists in (i) pushing *Obj*'s payload from the cache (RAMCloud) to the object store and (ii) update its metadata. The version numbers are also used by *persistor* tasks to enforce that successive updates to the same object are (asynchronously) propagated in the correct order to the RSDS. Our experiments show that this mechanism, akin to write-back, is always beneficial even for small payloads, and thus is always used for cached objects.

The notion of shadow object is also useful to provide strong consistency guarantees when a client application directly issues a request to the RSDS (e.g., typically, a non-FaaS application). Here, we leverage the support for webhooks provided by Swift: a callback function is registered and triggered upon each read request. The webhook checks if the RSDS holds the latest version

[2] By default, the maximum object size in RAMCloud is 1 MB. We extended it to 10 MB based on our observations.

[3] Some object storage systems (like Swift and AWS S3) do not provide very strong consistency guarantees such as linearizability. In such a case, client applications must typically avoid concurrent accesses to mutable objects or rely on an external synchronization facility. In our work, we assume that applications are designed according to these guidelines if needed.

of the object (by comparing the values of the two above-described version numbers). If this is not the case, the webhook notifies the OWK controller so that the latter can boost the scheduling of the corresponding *persistor* task. The webhook only terminates (and allows the completion of the external read request) once the latest data payload is available in the RSDS. Similarly, if an external client issues a write request while the cache holds a copy of the object, a webhook is used to (synchronously) invalidate the cached copy in RAMCloud before performing the operation on the RSDS. Besides, in the case of several function invocations performing (concurrent or serial) accesses to a cached object, strong consistency is enforced by RAMCloud. RAMCloud provides linearizable semantics for failure-free scenarios and strongly-consistent "at-least-once" semantics otherwise [89], and can be extended to support full linearizability and multi-object transactions [66].

While the above-described techniques (synchronous write requests, persistors and webhooks) are useful to provide full transparency, we observe that they are not always necessary in practice. Indeed, in many FaaS use cases, most or even all of the accesses to the object store are mediated through the FaaS code. Therefore, our system allows tenants to disable the above-mentioned facilities (via metadata tags and settings, on the scale of each bucket/object/account) in order to improve performance. In such a case, the consistency between the cache and the object store is relaxed (writes are only propagated lazily to the object store, upon the cache eviction decisions discussed in §2.6.3) and persistence relies on the (on-disk) replication provided by RAMCloud.

2.6.3 Caching policy

To improve cache usage for the functions that will benefit the most from it, OFC relies on the following heuristics for admitting objects in the cache and evicting them.

For a given invocation of function `F`, an object is considered for caching only if it satisfies two conditions. First, it must be smaller than the maximum object size allowed in the cache; we use 10 MB in our prototype, according to our cache efficiency characterization (see §2.2.2.3). Second, as explained in §2.5.2, the predicted performance benefits of the cache for `F` and the corresponding object(s) must be significant. Furthermore, in the case of a pipeline, the output objects produced by the intermediate stages (functions) of the pipeline are removed from the cache when the last function of the pipeline has completed. In addition to the previous policies, final output objects (i.e., produced at the end of a pipeline or by a single-stage function) are discarded from the cache as soon as they have been written back to the remote storage.

In addition, to reclaim more space, the cacheAgent periodically evicts objects that have not been recently accessed. We extended RAMCloud to maintain, for each object, a read access counter n_{access} and a timestamp T_{access} that records the epoch of the last access. In our current setup (tuned empirically), this periodic eviction is triggered every 300 s, and the eviction criteria are: $n_{access} < 5$ or $T_{access} > 30$ min.

2.6.4 Autoscaling

The horizontal scaling (in/out) of OFC relies on the support provided by OWK and RAMCloud. Below, we mostly focus on how OFC supports vertical scaling. OFC opportunistically hoards the unused (but already booked) memory on each Invoker node. Within an Invoker node, workload variations introduce two main challenges regarding this aspect. First, given that the memory consumption of most functions is input-sensitive, a sandbox may have widely fluctuating memory requirements during its lifetime (recall that a sandbox may serve multiple invocations of the same

function). Second, unexpected load spikes may require to quickly release some (or even all) of the cache resources in order to accommodate more demanding requests and/or a greater number of sandboxes. Our design is impacted by three quantitative aspects. The first aspect is the end-to-end time needed to process an empty function throughout the (distributed) OWK infrastructure, which is in the range of 8 ms. The second aspect is the time required to dynamically reconfigure (i.e., scale up or down) the memory pool of a RAMCloud instance, which is in the range of dozens of milliseconds, as shown in §2.7.2.1. The third aspect is the time taken to adjust the resource limits of a sandbox (in OWK, which uses Docker, this is a syscall to the `cgroup` Linux subsystem), which is in the range of 24 ms.

To address the first challenge, we adjust the memory of a sandbox for each invocation: scaling up the memory resources of a sandbox involves scaling down the ones of OFC, and vice versa. We optimize the critical path by executing all the memory capacity adjustements asynchronously: the function invocation is processed before the completion of the memory resizing operations (`cgroup` syscall for the sandbox and RAMCloud control request). Yet, in the case of a sandbox capacity scale-up, this may introduce the risk of memory capacity violation, leading to the failure of the function invocation (which implies retrying the invocation, and leads to increased completion times and waste). This risk is exacerbated by potential memory under-predictions and bursty workloads. To mitigate the occurrences of such events, each Invoker node provisions a slack pool of memory, whose size (initially 100 MB) is adjusted every 120 s based on an estimation by sliding window, of the local memory churn (measured every 60 s).

To address the second challenge of fast reclamation of the cache resources, we use the following decentralized approach. The `cacheAgent` on an Invoker node must choose and release objects from the local cache instance. It first selects the output objects (in the case of function pipelines, final outputs) that have been persisted on the RSDS but not yet discarded locally. If more space is required, the `cacheAgent` proceeds with input objects and evicts them on an LRU basis (until enough space is available). In parallel, it also triggers the write-back of the dirty output objects and discards them upon completion. The `cacheAgent` attempts to keep the hot input objects in the cache by offloading their master (in-memory) copy to another RAMCloud storage node. To achieve this, we do not rely on the standard object migration protocol supported by RAMCloud (which systematically sends the target object to the destination node); instead, we use the following optimized approach to speed up the migration. For each object O chosen for eviction on a node M_{old}, a new master node M_{new} is elected among the backup nodes (i.e., holding an on-disk copy of O). O is then loaded in the memory of M_{new}, and M_{old} removes it from main memory (but becomes a backup and keeps an on-disk copy). This way, no inter-node transfer of O is necessary. By doing so, OFC ensures high availability of the remaining cached objects and maintains the required replication factor for fault tolerance.

2.6.5 Request routing

We aim at (i) achieving good load balancing between the invoker nodes (regarding the load incurred by the function invocations but also by the caching service), (ii) limiting the cache management overheads (e.g., memory resources adjustments and transfers of cached objects between nodes), and (iii) improving data locality.

To this end, we modify the policy used by OWK's *Loadbalancer* component (see §2.2.1) to route function invocation requests. Similar to the original design, a request for a function `F` is always routed to an idle (warm) sandbox set up for `F` if there is one (to avoid cold starts), and otherwise,

a new sandbox is immediately created (to avoid queueing latency behind long-running requests). If a new sandbox must be created, the target Invoker node is preferably the one currently hosting the master (in-memory) cached copy in its local RAMCloud storage instance (if it exists and has sufficient resources). To find such a node, the controller parses the function invocation request (to extract the object ID among the arguments) and queries the RAMCloud coordinator. If there are multiple available sandboxes, the routing algorithm uses the following criteria, by decreasing order of priority: (i) the difference between the current memory capacity of the sandbox and the predicted capacity for the new invocation (smallest difference is preferred); (ii) the available memory capacity on the Invoker node (if the capacity must grow); (iii) the locality of the data (sandboxes co-located with the requested object are preferred); (iv) the idle time of the sandbox (more recently used sandboxes are preferred, so that the older ones can eventually time out and be reclaimed if they are in surplus).

2.7 Evaluation

This section presents the evaluation results of OFC.

Evaluation goals and methodology. We evaluate the following aspects: (i) concerning the ML module, the accuracy of the prediction model, the prediction time, and the model maturation quickness; and, (ii) concerning the caching system, the overall performance gain and costs.

The testbed is composed of 6 physical machines, interconnected via a 10Gb/s Ethernet switch running Ubuntu 16.04.7 LTS. The hardware characteristics are as follows: 2 Intel Xeon E5-2698v4 CPUs (20 cores/CPU), 512GB of RAM, an Intel Ethernet 10G 2P X520 Adapter, and a 480GB SSD. We use one machine to host all the OFC controllers (*Model Trainer* and *Controller* boxes in Figure 2.6). Another machine is dedicated to the storage system. The remaining machines are FaaS worker nodes.

Benchmarks. We evaluate single- and multi-stage functions. For the former, we use 19 multimedia processing functions, available online (see our code repository in §2.1). For multi-stage functions, we study four applications: two data analytics applications as in [62] (a MapReduce-based "word count" application; Thousand Island Scanner (THIS) **[this]**, as well as (in §2.7.2), a distributed video processing benchmark), a cloud-based Illegitimate Mobile App Detector (IMAD) application [123][4], and an image thumbnail generator pipeline (Image Processing) from the ServerlessBench suite [131].

We run each experiment 5 times and report the average results.

2.7.1 ML model evaluation

2.7.1.1 Accuracy

We first evaluate the accuracy of ML predictions concerning memory requirements, with various decision tree algorithms: J48 (an implementation of C4.5 [98]), RandomForest [18], RandomTree [97] and HoeffdingTree [54]. Then, we discuss the results regarding the prediction of caching benefits.

Prediction of physical memory requirements. We use cross-validation to prevent overfitting. Moreover, we experiment with $n = \{64, 128, 256\}$ intervals, with respective interval sizes of $\{32, 16, 8\}$MB. Table 2.1 shows evaluation results. It demonstrates that *J48 and RandomForest are*

[4]We reimplemented IMAD as a sequence of serverless functions.

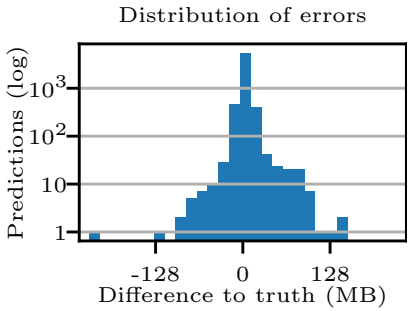


Figure 2.7: Errors in memory requirements prediction using J48, with 16 MB intervals (all functions combined).

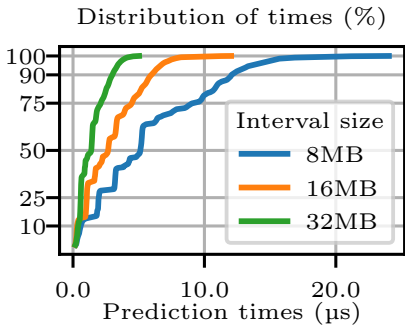


Figure 2.8: Time for memory requirements prediction, using J48 and varying interval sizes (all functions).

the most accurate algorithms: with 16 MB intervals, they achieve more than 80 % accuracy, and more than 90 % accuracy on EO-predictions; remember that we are interested in EO-predictions because it is a component of the maturation criterion, as explained in §2.5.3. An interval size of 16 MB also allows keeping prediction times very low, as shown further below. Ultimately, we elect to use J48 because its prediction time is much shorter than RandomForest's (see §2.7.1.2).

Our results also demonstrate that overpredictions are not a problem because, as shown in Figure 2.7, they remain close to the correct value: 90 % of them are within 3 intervals of the correct one, resulting in an average memory waste of only 26.8 MB with 16 MB intervals. In any case, EO-predictions are always favored over underpredictions. Indeed, as explained in §2.5.3, we use the next greater interval; Figure 2.7 does not reflect this behavior and shows raw predictions.

Table 2.1: Evaluation of ML algorithms with varying interval sizes. Results are fractions of exact, and exact-or-over predictions, averaged over all functions.

Interval size	Algorithm	Exact (%)	Exact-or-over (%)
32 MB	HoeffdingTree	81.09	87.65
	J48	91.27	95.77
	RandomForest	92.66	96.20
	RandomTree	89.84	94.23
16 MB	HoeffdingTree	72.01	84.81
	J48	83.35	92.73
	RandomForest	84.82	92.76
	RandomTree	79.23	88.69
8 MB	HoeffdingTree	63.40	79.17
	J48	75.88	87.91
	RandomForest	78.17	89.42
	RandomTree	72.27	84.12

Prediction of cache benefit. Here, we validate our choice of J48 to predict caching benefit (as defined in §2.5.2). The *precision* of the model is 98.8 % ; a higher precision means the model is more often correct when predicting that the cache is useful. Its *recall* is 98.6 % ; a higher recall means the model detects more exhaustively the cases in which the cache is useful. The F-measure — the harmonic mean of precision and recall, used as a global efficiency score — is 98.7 %. These

results outperform the other classifiers, so we find J48 is a good fit to predict cache benefit.

2.7.1.2 Prediction speed

We evaluate the classification speed, i.e., the time taken for a single prediction, for our two models. Results are shown in Figure 2.8. Over all functions, and with a memory interval size of 16 MB, the median memory requirements prediction time is 3.19 µs, and the 99th percentile is 12.54 µs. The classification speed for the cache benefit model is similar to the speed of predicting memory usage, so overall the global prediction speed remains negligible. For reference, the prediction time using RandomForest (an algorithm that produces prediction results of quality similar to J48) is 106.29 µs at the median, and the 99th percentile is 173.05 µs.

2.7.1.3 Model maturation quickness

We also checked the model maturation quickness, i.e., the number of training inputs that the model needs to learn from, in order to be accurate enough, as defined by the two criteria from §2.5.3. Only the maturation quickness of the model that predicts memory is evaluated, because using the model that predicts cache benefits is subordinated to using the former — and prediction errors of the latter are not problematic. Remember that we are in a context where the model is learned over time, so the number of training inputs that represents the quickness is actually a number of invocations of the model's function. We start checking the maturity after 100 invocations, so this is a minimum. In our evaluation, the median maturation quickness is 100 invocations; this result includes 11 of 19 functions that matured in 100 invocations or less. 75 % of the functions matured with less than 250 invocations, and 95 % did so under 450 invocations. As an illustration, Shahrad et al. [106] state that 99.6 % of the functions they study are invoked at least once per minute, so 95 % of them would mature in less than 450 min (7 h 30 min).

2.7.2 Cache performance evaluation

We perform two types of experiments. We first evaluate OFC while running a single function. We then evaluate OFC while running several sandboxes concurrently, for diverse function invocations. We compare against two alternatives based on standard OWK: OWK with all data stored in the Redis in-memory cache (noted OWK-Redis), or OWK with all data in the Swift persistent RSDS (OWK-Swift). These baselines represent the best and worst-case data access time respectively.

2.7.2.1 Micro evaluations

Benefits of OFC's cache. We first run each of our multimedia and data analytics functions alone using OFC and our two baseline configurations. We do this while varying the input data size and compare the end-to-end latency. Regarding OFC, we evaluate three scenarios for fairness: (LH — $LocalHit$) the input data is in OFC's cache, on the same worker node that runs the function, (M — $Miss$) the input data is not in the cache, and (RH — $RemoteHit$) the input data is in the cache but on a different worker node. Recall that in OFC, outputs are always buffered (i.e., stored in RAMCloud in write-back mode) regardless of the scenario, which helps multi-stage functions.

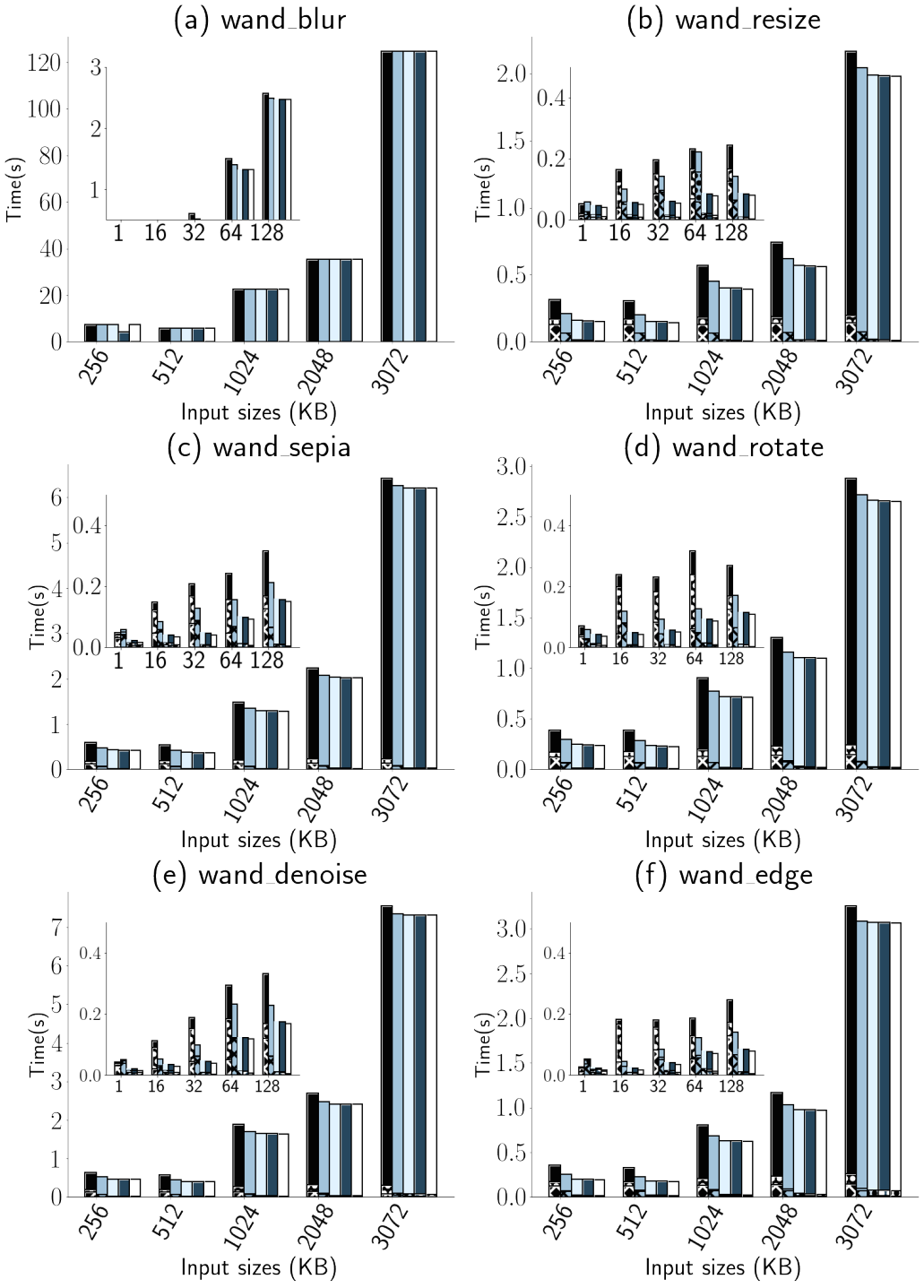
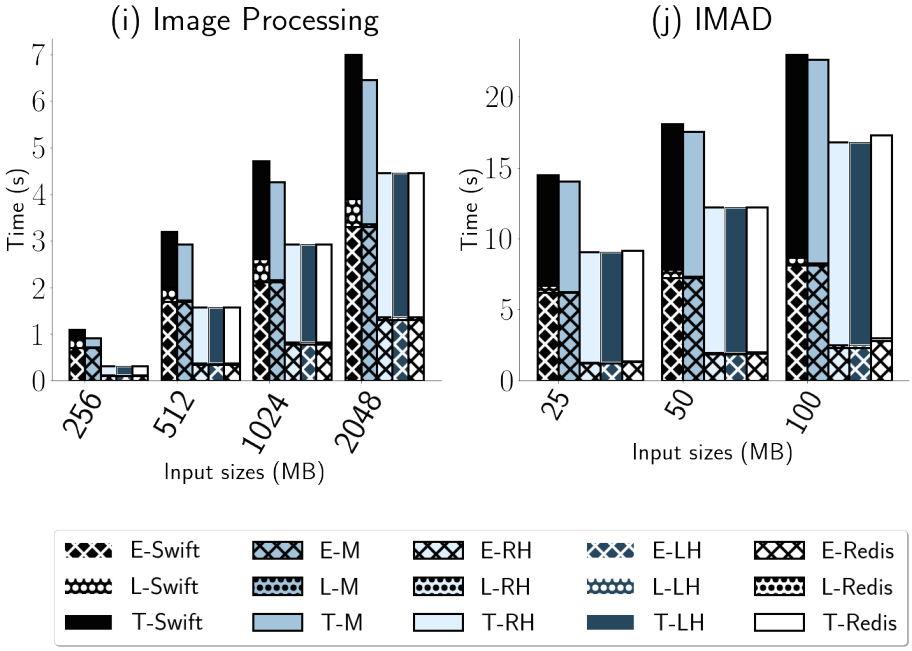


Figure 2.9: Duration of ETL phases for 6 common image processing functions and 2 multi-stage data analytics functions (IMAD, and ServerlessBench - Image Processing). We compare OWK-Swift, OWK-Redis and OFC (under scenarios LH, M, and RH).



We present results for 6 single-stage functions and the 4 multi-stage pipelines, shown in Figure 2.9. Each stacked bar in the figure shows the time for the Extract, Load, and Transform phases bottom to top. In scenario *LH*, OFC outperforms OWK-swift by up to ≈82 % for single-stage functions (180 ms down to 32 ms for the `wand_edge` function with 16 kB input) and up to ≈60 % for multi-stage functions (105 s down to 35.84 s for THIS with 125 MB input). In absolute numbers, OFC reduces the latency of `wand_edge` by a total of 150 ms (with respectively 42 and 108 ms of savings for the Extract and Load phase). Overall, OFC achieves very close results w.r.t. OWK-Redis (with maximum differences ranging from −3 % to 2 % in completion times). We attribute these minor differences to two main sources: (i) design and implementation differences between the two caches (Redis and RAMCloud) and (ii) in the case of small requests, the overhead of the *Predictor* and *Sizer* components of OFC, which adds about 6 ms of latency.

In scenario *M*, the difference between OFC and OWK-Redis is more pronounced since all initial accesses must go to Swift. OWK-Redis outperforms OFC by up to 65 % for single-stage functions and up to 46 % (336.25 s to 207.48 s for THIS with 300 MB input) for multi-stage functions. However, OFC still outperforms OWK-swift in this scenario by up to 75 % for single-stage functions and up to 24 % (see wand_edge function with 16 kB input data) for multi-stage functions (see THIS with 125 MB input data). This is explained by the fact that although the input data comes from the RSDS in OFC, outputs are always cached i.e., (the `Load` phase is improved).

The results of scenario *RH* show that retrieving cached data from a remote worker node still provides a significant speedup compared to reading from Swift. Compared to scenario *LH*, remote access increases execution time by up to 12.76 % for single-stage functions (19.6 ms up to 22.1 ms for the `wand_denoise` function with 1 kB input) and up to 0.85 % for multi-stage functions (6.52 s up to 6.57 s for `map_reduce` with 30 MB input).

In all OFC scenarios, the time needed to persist a shadow object to the RSDS in the *Load* phase

is constant (about 11 ms) since its size is independent of the output size.

Potential negative impact. OFC's cache can introduce a delay on function setup when the worker node lacks memory. We only show the results for `wand_sepia` as the observations are the same for other functions. We evaluate four scenarios regarding the status of the worker node. In the first scenario, Sc_1, shrinking the cache does not involve data migration/eviction. In the second scenario, Sc_2, shrinking the cache requires data migration. In the third scenario, Sc_3, shrinking requires eviction without migration. We compare these scenarios with the baseline, noted Sc_b, where the execution of the function does not require any cache shrinking. In the scenarios Sc_{1-3}, the current memory size of the (warm) container is 64 MB (the smallest configurable memory in OWK). We consider input data sizes ranging from 1 kB to 3072 kB, which result in function memory requirements between 84 MB and 152 MB.

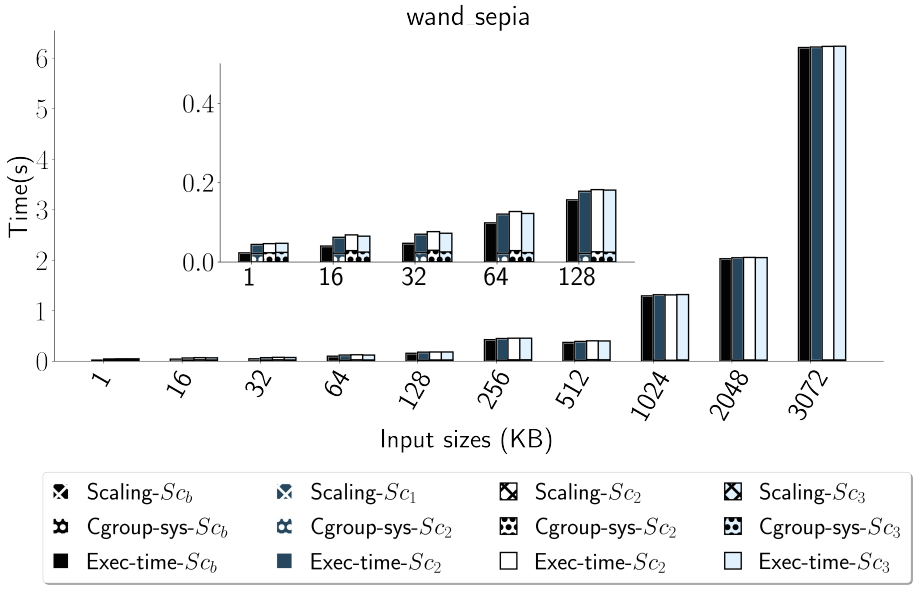


Figure 2.10: Impact of OFC's cache scaling on the `wand_sepia` function's latency. For each scenario, we plot the scaling time, the time for the container memory limit update (noted *cgroup syscall*), and the overall function execution time.

Figure 2.10 presents, for each scenario, the following metrics: the time needed to scale down OFC's cache, the time needed to increase the container's `cgroup` memory (noted *cgroup-sys*), and the overall function execution time. The time *cgroup-sys* phase is constant, in average 23.8 ms (0.8 s for the `cgroup` syscall and the remainder being the `docker update` command.). The scaling time in Sc_1 as well as in Sc_3 is also constant, in average 289 µs and 373 µs respectively. It varies in Sc_2 according to the aggregated size of the migrated objects. For this experiment, the migration times range from 401 µs (20 MB to migrate) to 2.2 ms (88 MB). More generally, we measure migration times of 0.18 ms for 8 MB, 1.2 ms for 64 MB, 3.8 ms for 256 MB, 7.5 ms for 512 MB and 13.5 ms for 1 GB. In the worst case (1 kB input size), OFC's cache scaling (*cache shrink+cgroup syscall*) takes 24.3 ms in total, which represents a 50.4 % overhead on the overall execution time (48.2 ms). We

believe that this scenario is likely to be rare w.r.t. our data caching policy, which tries to free as early as possible the unused data from the cache.

2.7.2.2 Macro-experiments

Methodology. OFC's efficiency depends on the workload characteristics. We built FaaSLoad, a load injector for OWK, which allows emulating several tenants with different loads. In this experiment, we consider one FaaS function per tenant. Overall, FaaSLoad prepares the input data (in the RSDS) for the invocations of each function, then performs the function invocations at different intervals within a given observation period. The invocation interval can be configured as periodic or based on the exponential law.

We set up 8 tenants and associate each of them with a distinct function from Figure 2.9. Our RSDS is Swift. We run FaaSLoad for 30 minutes. Functions invocation intervals follow the exponential law with $\lambda = \frac{1}{60}$, corresponding to a mean invocation interval of 1 minute. We consider three different profiles of cloud tenants, which use distinct approaches to configure the memory size of their functions: (i) *naive*, i.e., always reserving the maximum memory size allowed by OWK (2 GB); (ii) *advanced*, i.e., reserving the maximum amount of memory that has been used by a function (according to the previous runs); (iii) *normal*, i.e., reserving 1.7x the memory size chosen by an advanced tenant, a common situation in practice [109]. In a given experiment, all tenants have the same profile (naive, advanced or normal). We compare the results achieved by OFC to those of our baseline, OWK-Swift.

Results. Figure 2.11 reports the total execution time for all invocations of each function, per each tenant profile. For each scenario, OFC always outperforms OWK-Swift, with an improvement between 23.9 % and 79.8 % (54.6 % in average). For most functions, we observe slightly better results with *naive* tenants than with *advanced* ones (2–3 % of difference), because, in the *naive* case, OFC's memory capacity is larger (thus yielding more cache hits) than in *advanced* (see Figure 2.12).

Table 2.2 reports internal OFC metrics during these experiments. First, we note that there was no abrupt function termination due to memory shortage (line 9). Second, the cache hit rate is high (up to 98.9 % for *naive*, line 10). However, there is a high rate of scale-up/down operations (line 1-5) due to the variability of function inputs (hence the need to reclaim and add memory from/to the cache). However, this does not really impact the overall function execution time since cache scaling up/down takes a negligible time (line 6). For the above workload, the data is relatively easy to cache but, as noted previously, OFC provides benefits even on cache misses due to optimizing the load stage and caching intermediate objects during multi-stage requests. To show this, we also run the experiment with more tenants, 24 (3 per function) instead of 8 (1 per function). Due to space constraints, we present only a summary of the results. We observe a lower hit ratio of up to 32.3 %; yet, no failed invocation due to memory pressure is experienced (regardless of the tenant profiles). Besides, OFC's latency improvements fall from 23.9—79.8% to 4.5—44.9% due to the lower hit ratio.

2.8 Related Work

FaaS performance bottlenecks. We have previously discussed FaaS systems that use "serverful" (i.e., non-serverless) components as workarounds to mitigate performance bottlenecks stemming

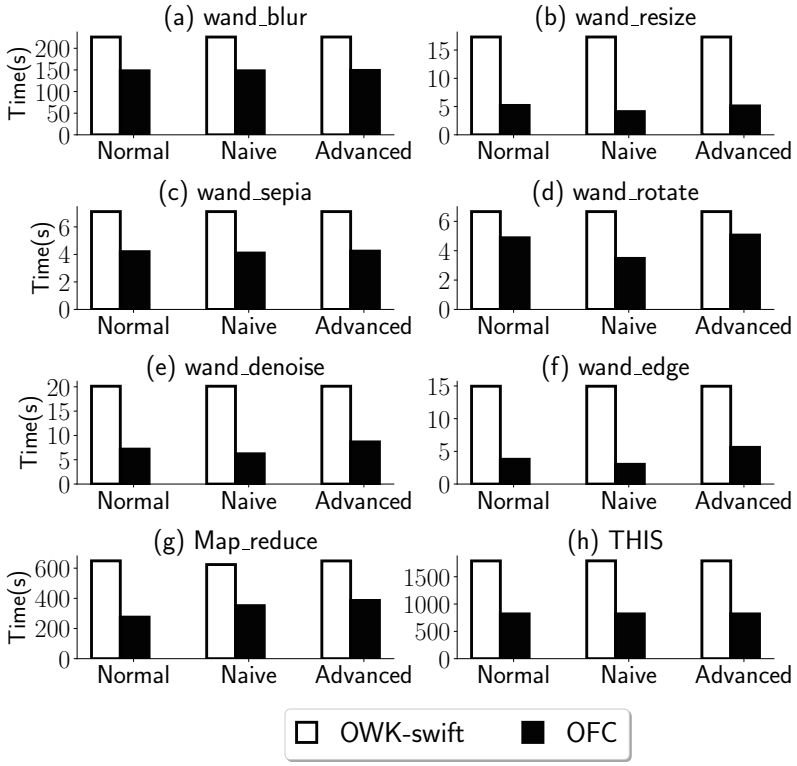


Figure 2.11: Sum of the execution times of all invocations for each function in three scenarios (distinct tenant profiles).

Table 2.2: OFC internal metrics observed for the macro workloads with 8 tenants and three different user profiles.

	Metrics	Normal	Advanced	Naive
1	# Scale up	96	94	95
2	Total scale up time (s)	28.8	28.2	28.5
3	# Scale down (no eviction)	225	224	226
4	# Scale down (migration)	7	4	4
5	# Scale down (eviction)	0	0	0
6	Total scale down times (s)	85.4	81.2	83.2
7	# Bad predictions	7	7	7
8	# Good predictions	231	230	232
9	# failed invocations	0	0	0
10	Cache hit ratio (%)	98.21	93.12	98.9
11	Ephemeral data generated (GB)	300	300	300

from shared/persistent state management [62, 92, 22]. Below, we focus on other works.

Cloudburst [110], designed concurrently to OFC, also uses caches co-located with function executors and seamlessly supports existing functions. Cloudburst relies on the Anna key-value store,

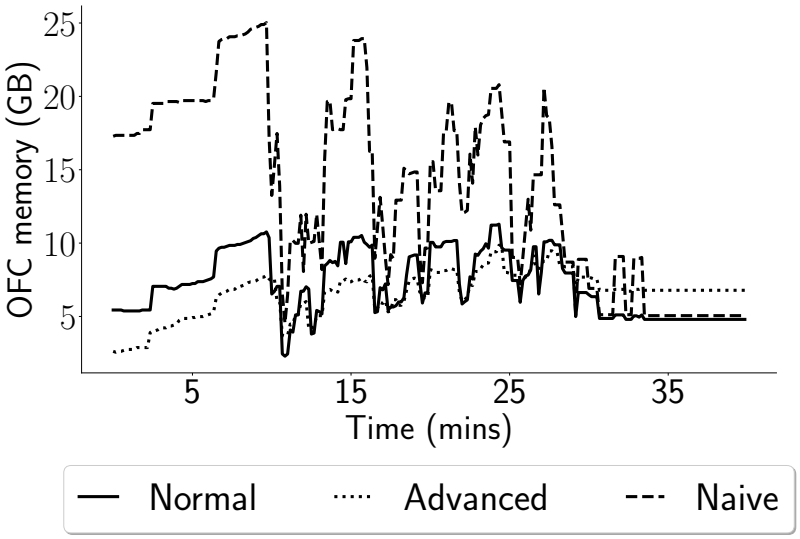


Figure 2.12: OFC's cache size evolution throughout the three experiments.

which introduces specific assumptions in terms of consistency semantics and protocols between the FaaS workers and the backend storage service. Cloudburst leverages relaxed data consistency for maximum scalability and availability, whereas OFC is geared towards stronger consistency and persistence guarantees to support a broader set of use cases (e.g., hybrid FaaS/non-FaaS workloads interacting through a shared remote storage). Moreover, Cloudburst's authors do not discuss in details how the worker caches are provisioned and sized. OFC's memory hoarding/prediction techniques could be leveraged by Cloudburst.

Faasm [108] accelerates data movement between function instances, through the use of shared memory, both within a worker node and across worker nodes, using an abstraction akin to a distributed shared memory. Faasm relies on specific assumptions regarding the sandboxes runtime (language-based isolation) and the programming interface exposed to tenants for developing their applications.

Infinicache [121] leverages FaaS sandboxes and their keep-alive policy to implement an elastic in-memory caching service that is more cost effective than traditional ones (e.g., Redis-based services like AWS ElastiCache) for large objects. Infinicache relies on dedicated FaaS sandboxes (for caching) that must be booked by Cloud tenants, who must also modify their applications to use the service.

OFC differs from the above works by offering full transparency for legacy cloud functions, no restriction on the choice of language or runtime for the functions, and only minor modifications of the FaaS and the backend storage infrastructure. Unlike the above systems, it harvests existing idle memory and does not require tenants nor cloud operators to provision and dimension dedicated resources for storage and data exchanges. Furthermore, OFC predicts memory usage and caching efficiency via ML techniques.

Lambada [82] is a specialized framework (for interactive data analytics) aimed at mitigating the performance of FaaS platforms, without any "serverful" component, thanks to domain-specific optimizations. Our work is focused on transparent and generic optimizations for FaaS applications based on the "ETL" pattern [37]. Boxer [125] has recently improved Lambada by enabling direct

network communication (and hence, direct data exchange) between function instances, which could also bring benefits to a broader range of use cases [40]. OFC's approach remains useful even when direct communications between functions are possible, because it accelerates the "E" and "L" phases of the "ETL" pattern (very common in FaaS applications, not only in function pipelines).

FaaSCache [41] helps fine tuning the keep-alive policy of a FaaS platform by leveraging insights from the well-established caching literature. OFC is complementary to this approach, which does not address data caching and exchange, nor mitigation of memory waste caused by input variability.

Machine learning for resource management. A number of works have leveraged ML to optimize server applications and cloud infrastructures [29, 26, 42, 50, 70, 4]. We focus here on the most closely related to our work.

Resource Central [26] is used within Azure to collect telemetry data of resource usage in virtual machines (VMs), learn (offline) the behavior of these VMs, and provide a service for online predictions to resource managers (e.g., VM placement decisions). The authors mention examples based on different ML algorithms for the prediction of various metrics regarding the resource usage and lifetime of VMs. Our work considers the case of function invocations, which have very small durations and "white box" inputs.

COSE [4] uses statistical learning to determine the best configuration (w.r.t. SLAs and cost) for a cloud function. In contrast, our work aims at predicting the memory requirements and I/O-sensitivity of a function, in order to transparently mitigate storage performance bottlenecks, a major source of cost and performance overheads in FaaS workloads.

The Monitorless project [50] studied several ML approaches to infer the performance degradation of non-FaaS cloud applications and opted for RandomForest despite long classification times. OFC requires fast classification since it uses the ML model on the critical path of function invocations.

Seer [42] uses deep learning and monitoring to infer the cause of QoS violations in microservices-based applications. For issues attributed to memory capacity, Seer resizes the resources of the corresponding container. Seer is not aimed at predicting memory consumption on a per-request basis.

2.9 Summary

In this chapter, we have introduced OFC and shown that such a caching layer allows significant performance improvements for the execution of diverse FaaS workloads in a cost-effective manner leading to an efficient management of storage of data used by the these workloads. Moreover, OFC's approach can be retrofitted in existing cloud infrastructures (FaaS platforms and object storage services) with limited modifications, is fully transparent for application-level code, and does not require to explicitly book or provision additional storage resources.

CHAPTER 3

Efficient Storage for VMs using a Scalable Virtual Disk Format

In this chapter, We first study virtual disk utilization in a large-scale public cloud. Then we demonstrate, through experimental measurements, that existing long chains of snapshots lead to virtualized storage performance and memory footprint scalability issues and finally we present SVD, a new virtual disk format extending Qcow2 to address these problems.

Contents

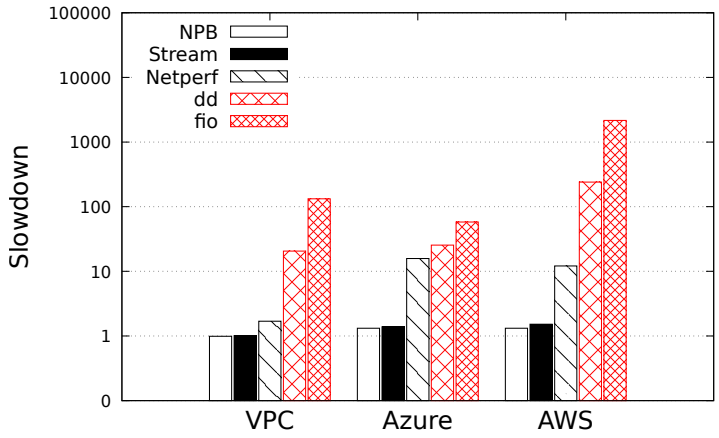


Figure 3.1: Performance slowdown incurred by virtualization for different types of applications. The results are presented on a logarithmic scale. **Lower is better.**

3.1 Introduction

As said previously a few times already, virtualization is the keystone technology that enables cloud computing. However, virtualization comes with the overhead on application performance. This overhead has been well studied [14, 55, 64, 90, 120, 79, 2, 49]. Although it concerns all types of resources (CPU, RAM, network, disk), they are not all affected with the same intensity. Figure 3.1 shows the performance degradation coming from virtualization for a wide range of benchmarks including Stream [56] (memory intensive), NPB [30] (CPU-intensive), netperf [102] (network-intensive), as well as the Linux `dd` command (disk-intensive, throughput-oriented) and fio [69] (disk-intensive, latency-oriented), when they run in AWS EC2 (`t2.medium` instance type), Microsoft Azure (`Standard_B2s` instance type), a virtualized private cloud, and on a bare metal private cloud without virtualization[1]. We use the latter as the baseline. We can observe that the two disk-intensive applications (`dd` and `fio`) experience the highest slowdown. For `fio`, it is about 1,639× the degradation experienced by NPB.

Surprisingly, and contrary to the other resource types, very little research work focuses on improving storage virtualization in the cloud. Hence, it is essential to bridge this gap, especially in the context of the exploding popularity of data-centric applications (big data, ML, and AI trends). Storage virtualization is peculiar as it is still implemented through complex multi-layered architectures [65, 134] (see §3.2). Further, disk virtualization generally uses complex virtual disk formats (Qcow2, QED, FVD, VDI, VMDK, VHD, EBS, and so on). These not only perform the task of multiplexing the physical disk into virtual ones but also need to support standard features such as snapshots/rollbacks, compression, encryption, etc. All these layers of indirection are the source of the disk virtualization overheads.

This chapter studies Linux-KVM/Qemu (hereafter LKQ), a very popular virtualization stack. LKQ supports several virtual disk formats, among which Qcow2 [96] is widely adopted in production [80]: all the principal Linux distributions offer Qcow2 cloud images [20, 25, 23, 100], and it is

[1]We chose `t2.medium` and `Standard_B2s` to match the VM size that we used in our private cloud.

also the main supported format in major cloud computing platforms including OpenStack [87] and Apache CloudStack [7]. A salient feature provided by Qcow2 is the capacity to create incremental Copy-On-Write (COW) snapshots (backing files) to save the state of the virtual disk at a given point in time and to reduce storage space usage. The virtual disk of a VM can thus be seen as a chain linking multiple backing files. We try to identify and solve scalability issues on such snapshot chains.

Our *first contribution* (§3.3) is the presentation of the characterization of chain length in the infrastructure of our cloud partner, a large-scale public cloud provider with several data centers spread over the world, using LKQ/Qcow2 for virtualized storage. We observe that snapshot operations are frequent (some VMs are subject to more than one snapshot creation per day) for three main reasons. First, cloud users leverage snapshots to create recovery points for fault tolerance reasons periodically. Second, cloud users and providers use snapshots to achieve efficient virtual disk copy operations and share elements, such as the OS/distribution base image between several virtual disks. Third, cloud providers use the snapshot feature to transparently distribute a virtual disk made of multiple chained backing files, among several storage servers, for load-balancing and capacity reasons and to avoid fragmentation. For all these reasons, we observe that the length of a chain can be very high. We identify chains composed of up to $1,000$ backing files. To our knowledge, this is the first research work performing such characterization. Prior works [27, 107, 6] mainly focused on the characterization of virtualized CPU and memory utilization in the cloud and there is also a patent submitted in 2015 but being extended without convincing solution nowadays [118].

Our *second contribution* (§3.4) is to show by experimentation that long chains lead to performance and memory footprint scalability issues. For illustration, using a synthetic benchmark based on `dd`, we measured up to 91% of IO throughput decrease and up to 180× memory footprint increase for a chain composed of $1,000$ backing files. We found that the origin of this problem lies in the fundamental design of the Qcow2 format: the Qcow2 driver in Qemu manages each backing file *individually in a recursive fashion*, without a global view of the entire chain composing the virtual disk. The evaluations of other formats (Microsoft Hyper-V's VHDX, VMWare's VMDK, and IBM's FVD) show that they use a similar approach, thus suffering from the same issue.

Our *third contribution* (§3.5), called SVD, is to address these scalability issues by evolving the virtual disk format and introducing two principles: 1) direct access upon an I/O request, regardless of their position in the chain; 2) using a single metadata cache, avoiding memory duplication by being independent of the chain length. The implementation of these principles raises three challenges. First, we should allow backward compatibility, which is essential to facilitate cloud operators' adoption of our solution. Second, we should preserve all Qcow2 features. Third, we should preserve crucial optimizations such as prefetching that come naturally with the current Qcow2 format. To cope with the above challenges, SVD extends the Qcow2[2] format to indicate the backing file which contains each cluster of the virtual disk. We rely on reserved bits in Qcow2's metadata to preserve backward compatibility. We implement these principles by extending, on the one hand, the Qemu's Qcow2 driver and, on the other hand, the snapshot operation. We thoroughly evaluate our prototype and demonstrate that it tackles Qcow2's aforementioned scalability issues. For example, on a virtual disk backed up by a chain of 500 snapshots, RocksDB's throughput is increased by 48% versus vanilla Qemu. The memory overhead on that chain is also reduced by 15×.

The rest of the chapter is organized as follows. §3.2 presents the background needed to understand how virtual disk formats work. §3.3 presents virtual disk characterization results. §3.4

[2]We consider the latest version of Qcow2 available at the time of the writing. Note that in some wiki pages [95] the term QcowX could be found. It refers to Qcow2 version X.

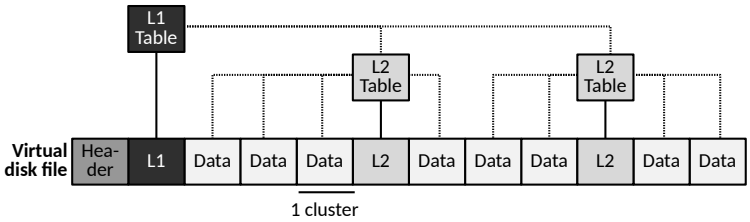


Figure 3.2: Overview of the Qcow2 format.

presents and assesses the scalability issues handled in this work. §3.5 presents our design to address the identified scalability issues. §3.6 presents the evaluation results of our design. §3.7 presents the related work. §3.8 summarize the chapter.

3.2 Background

The goal of our work is twofold: the characterization of snapshot chains in a public large-scale cloud and handling of two scalability issues happening in long snapshot chains. This section presents the necessary background to understand our contributions.

Qcow2 Overview. The Qcow2 format enables copy-on-write snapshots by using an indexing mechanism implemented in the format and managed at runtime in the Qcow2 driver, running in Qemu, to map guest IO requests addressing virtual sectors/blocks to host offsets in the Qcow2 virtual disk file(s). Figure 3.2 shows an overview of the Qcow2 format. Without any snapshot, a virtual disk is contained in a single file. The file is divided into units named clusters, containing either metadata (e.g., a header, indexing tables, etc.) or data representing ranges of consecutive sectors. The default cluster size is 64 KB. Indexing is made through a 2-level table, organized as a radix tree: the first-level table (L1) is small and contiguous in the file, while the second-level table (L2) could be spread among multiple non-contiguous clusters. The header occupies cluster 0 at offset 0 in the file, and the L1 tables come right after the header. For performance reasons, the RAM caches L1 and L2 entries (see below).

Qcow2 Snapshotting. Today, the most common way to create a live incremental snapshot of a virtual disk F for a given VM is to create a new empty Qcow2 file E and set it as the current disk (called *active volume*) for the VM while the previous virtual disk F is set as the *backing file* for E. The backing file will be queried for clusters read by the VM not present in E. All write operations made by the VM will be directed to the active volume (E), while read operations will be directed either to E if the addressed sectors are present or to backing files if not. Hence, a virtual disk snapshotted several times comprises an active volume and a chain of backing files per snapshot. With time, backing file chains can become very long (see §3.3), being sometimes composed of hundreds or even thousands of files.

Our first contribution characterizes backing file chains in our cloud partner's infrastructure. Our second contribution slightly modifies the snapshotting algorithm.

Qcow2 Cache Organization. To speed up access to L1 and L2 tables, Qemu caches them in RAM. It creates and manages one cache for the active volume and one cache per backing file. Each cache is managed independently of the others. In the following paragraph, we describe how these caches work. Qemu maintains a separate cache for the L1 and L2 table entries. With its small size, the entire content of L1 is loaded in RAM at VM boot time. The cache of L2 entries is populated on-demand, with a prefetching policy. We, therefore, focus on describing the caching of L2 entries, as they are likely to suffer from misses, thus influencing IO performance. On a cache miss, Qemu brings into the cache a set of L2 entries, a *slice* of configurable size, among which the entry at the origin of the miss. The slice is also the granularity of the cache eviction policy, which is LRU. A cache entry includes the file offset of the slice (noted `l2_slice_offset`), the number of threads that currently use the slice (noted `ref`), the actual L2 entries composing the slice, and a field indicating whether a data cluster referenced by an L2 entry has been modified (noted `dirty`); `l2_slice_offset` services as the tag when searching an entry in the cache, which is fully associative.

Qcow2 Cache Utilization. Every IO request issued by the guest OS to virtual disk *vb* traps inside Qemu. It is then handled by a thread running the para-virtualized disk driver in Qemu. One of its (driver) main goals is to translate *vb* to a data cluster offset inside the active volume or a backing file.

From *vb*, Qemu computes `l2_slice_offset`, `l2_slice_index`, and `l2_index`. It then looks if there is an entry in the cache that matches `l2_slice_offset`. If it exists, Qemu increments the corresponding `ref`. Next, thanks to `l2_slice_index`, it reads the L2 entry. If the latter describes an allocated data cluster (hereafter *cache hit*), then Qemu reads the offset of the data cluster. If the cluster is not allocated (hereafter *cache hit unallocated*), Qemu considers the cache of the following backing file in the chain. If the slice is not in that cache, Qemu will try to fetch it from the actual backing file associated with the current cache. If the slice exists on disk, it is brought into the cache. Otherwise, Qemu considers the cache of the following backing file and so forth.

Write requests need additional actions. First, the `dirty` field of the slice is set to 1. If the L2 entry is found in a backing file (not the active volume), Qemu allocates a data cluster on the active volume and performs the copy-on-write. If despite the whole chain scanning, the L2 entry is not found, Qemu creates a new data cluster in the active volume. In any case, Qemu configures L1 and L2 tables accordingly, both on disk and in the active volume's cache. A cache entry can be evicted either when the VM is terminated or when the cache is full.

IO Request Journey on a Chain. Qemu manages a chain snapshot-by-snapshot, starting from the active volume. Figure 3.3 illustrates the journey of an IO request for a chain of size 2: the base image (B) and the active volume (V). We assume that all L2 indexing caches are empty. We also assume a scenario in which virtual disk files are hosted on an NFS server mounted on a compute node running the VM. Let us assume that cluster number 2 is the target cluster, and it resides in B (meaning that it has not been modified since the creation of V). ❶ The driver starts by parsing V's indexing cache. To handle the cache miss, Qemu performs a set of function calls with some of them ❸ accessing over the network the Qcow2 file to fetch the missed entry from V's L2 table. According to its prefetching feature, Qemu fetches a slice of L2 table entries from V, and ❹ fills V indexing cache. In Figure 3.3, we assume that the size of a slice is two entries. Thus, V's cache includes two valid entries at the end of the first cache miss handling process: cluster 1 and cluster 2. After this step, ❺ PV driver hits V's cache, but the state of cluster 2 is marked unallocated

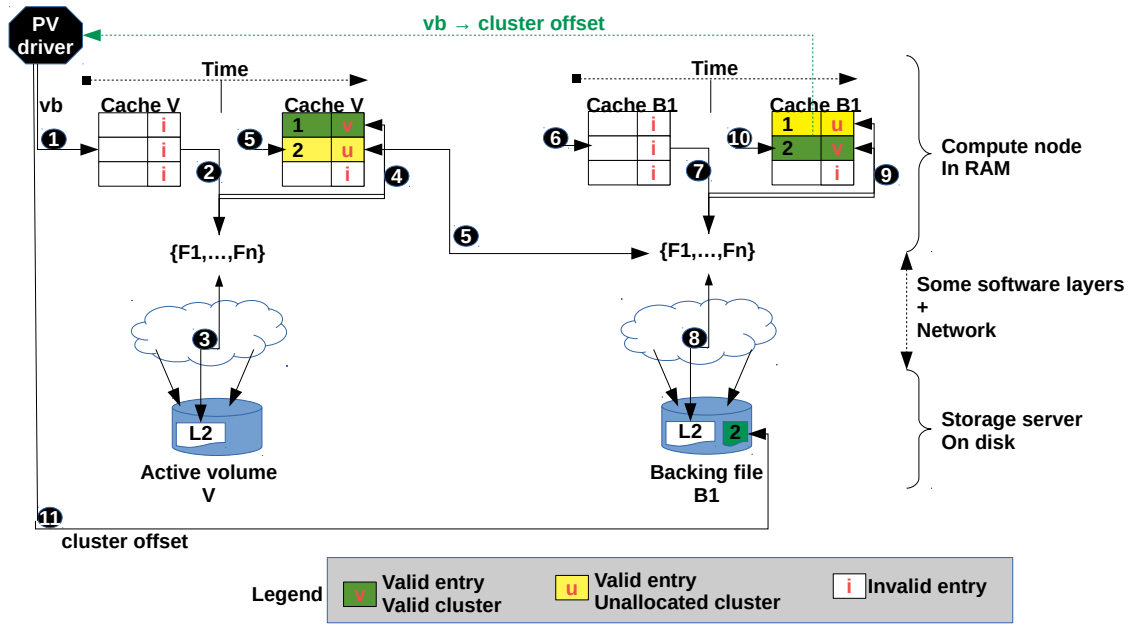


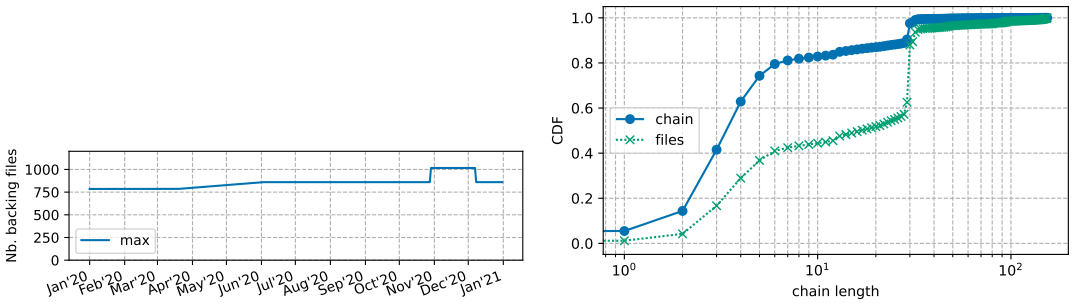
Figure 3.3: The journey of an IO request. (*vb* stands for virtual disk block)

because the referenced data cluster resides on B. ❻ This cache hit unallocated event triggers the same Qemu functions used for handling a cache miss. Qemu moves to the parent snapshot (B) for cache hit unallocated events. In fact, at VM startup, Qemu initializes a linked list corresponding to the snapshot chain of the VM's virtual disk. The caches of all the snapshots are also initialized at that time. ❻The first access to B's cache generates a miss ❼. After handling this miss (❽-❿), the offset of cluster 2 is returned to the driver. From there, ⓫ the latter can issue the IO request (consider reading here).

3.3 Chain Length Characterization

Over the entire year 2020, we performed a daily measurement of the length of each chain in our cloud industry partner's infrastructure. The study targets a datacenter located in Europe. The number of VMs in 2020 in this region is 2.8 million, corresponding to one VM booted every 12 seconds, demonstrating the large scale of our study. The software and workloads running in the VMs are highly varied. However, it is worth noting that our partner specializes in business-to-business. Hence, the VMs run enterprise workloads as opposed to private individual ones. Similar to existing cloud providers, the region runs VMs internal to our partner in addition to client VMs. Qcow2 chains back up the VMs' virtual disks in the region.

Chain Length. Our study covers a vast dataset, with the number of daily chains considered in the hundreds of thousands. Figure 3.4a presents the evolution of the longest chain's length over



(a) Evolution of the longest chain's size over the year 2020.

(b) CDF of the chain length for a daily measurement.

the period. As we can see, there is always a chain with at least a length of 800 snapshots, and the longest chain can have a length of more than 1,000.

We studied in details a daily measurement made during the period when the longest chain was of a length greater than 1000. Figure 3.4b shows the CDF for chains and files (active volumes and backing files) with respect to the chain length (for a file, to the length of the chain it belongs to). Most chains are relatively slight: chains of length ten or lower represent nearly 50% of the total number of files, and more than 80% of the chains, in the platform. We can observe a jump around size 30, with chains of size 30-35 files representing a relatively large proportion: 10% of the chains and 25% of the files. This is because, for a subset of the chains, the backing file merging operation, named streaming, is triggered around size 30. That operation merges the layers corresponding to multiple backing files into one. The files that can be merged in this way correspond to unneeded snapshots, i.e., deleted client snapshots and the ones made by the provider. Streaming helps reduce the size of some chains, however, note that valid (non-deleted) client snapshots cannot be merged. Further, although they are infrequent, there is a non-negligible number of chains of size 100 and above.

Take-away 1: *Long chains, with up to 1,000 backing files, do exist. The chain size threshold triggering streaming will cap the maximum size of many chains in the infrastructure.*

Chain Sharing. Certain backing files are shared and belong to chains corresponding to different virtual disks. The two main sources of sharing are virtual disk copy operations, as well as the use by multiple VMs of virtual disk base OS distribution images offered by the provider. A virtual disk copy is made by transforming the active volume into a backing file, and creating 2 new active volumes on top, forming 2 chains: all the backing files are thus shared between the 2 chains. Concerning base OS distribution images, they are generally composed of multiple snapshots corresponding to the different construction steps followed by the provider: the corresponding backing files are thus shared between all chains using a given base image. This is illustrated on top of Figure 3.5.

In Figure 3.6, each point corresponds to a chain of the daily measurement previously considered. The chain's length is indicated by its X value, and the number of backing files in the chain that are shared with at least another chain is indicated by the Y value. Note that in theory, a chain of length N can share from 0 up to $N-1$, files with other chains, i.e. all backing files without counting the active volume. Overall, the degree of sharing is highly variable among chains. We can observe

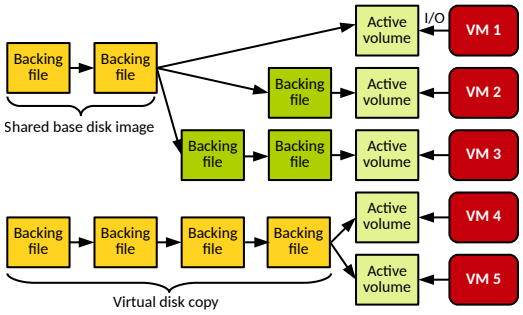


Figure 3.5: Chains can share backing files following a virtual disk copy, or when using a common base image.

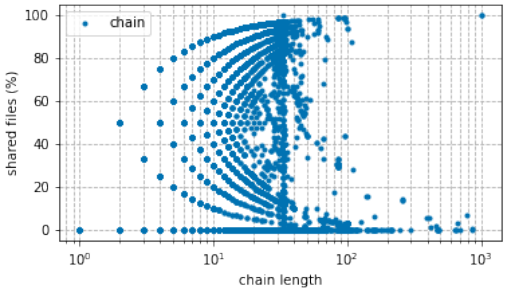


Figure 3.6: For each chain C of a daily measurement, the percentage of backing files shared with another chain according to the length of C.

a significant amount of chains of variable length with no sharing at all (Y value of 0). The high number of chains with a length $N < 30$ allows us to witness, for these chains, almost all possible degrees of sharing (from 0 to $N - 1$). The large number of points around size 30 corresponds to the high number of chains of that length, due as explained above to the streaming threshold being set to 30. Although the number of chains of size superior to 30 is smaller, one can still observe a variable degree of sharing for some of these. Note that, base OS images are generally made of around 5 chained backing files.

Take-away 2: *Backing files can be shared between several chains when multiple VMs use the same base OS image, or to achieve virtual disk copy. The degree of sharing among chains in the infrastructure is highly variable. Past a certain length (5+), most of the sharing is due to virtual disk copies.*

Snapshot Creation Frequency. We investigated the frequency of snapshot (i.e., backing files) creation. We looked in our daily measurement, for each snapshot creation operation, the time elapsed since the creation of the previous link in the chain (either a backing file or the active volume for a first snapshot).

This data is presented on Figure 3.7. Each point corresponds to a set of snapshot creation operations, placed on the Y axis into buckets corresponding to different elapsed time windows

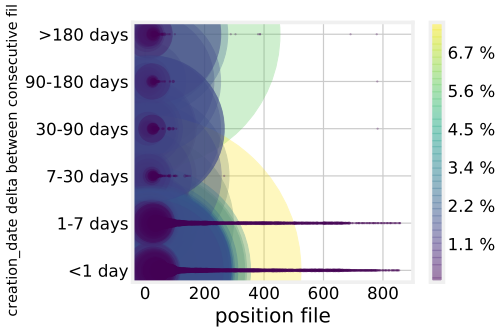


Figure 3.7: Snapshot creation frequency.

since the last link creation. Each point's X value corresponds to the position in the chain of the created backing files. Finally, the size and color of each point denotes how many snapshot creation operations are represented by the point, as a percentage of the total number of operations counted in our daily measurement.

We observed that most of the snapshots have chains of size inferior to 30. This is due to the high number of chains of these sizes, stemming from the streaming threshold set to 30. Further, although the frequency of snapshot creation is overall highly variable, many snapshots are created with a relatively high frequency (daily or more). Past work [91] noted peaks at up to 58 snapshots per hour. One can also observe that the long chains result from relatively frequent (daily/weekly) snapshotting done by clients (i.e., non-mergeable through streaming).

Take-away 3: *Although the snapshot-creating frequency varies widely among chains, a non-negligible amount of chains experience high frequency snapshotting. Long chains belong to this subset, with daily/weekly snapshots created. These snapshots are made by clients and cannot be merged with streaming.*

Origins of Long Snapshot Chains The emergence of long snapshot chains in modern virtualized environments is due to a combination of factors. First, for data backup/fault tolerance purposes, most cloud providers offer the client the possibility to create disk snapshots regularly, for example, every 24 hours or on-demand through an API. The chain length will thus grow according to the snapshot frequency. Even when the client deletes certain snapshots, they are kept by the provider as they form a necessary part of the Qcow2 chain backing the disk the VM in question is currently using. Second, snapshots may be performed by the cloud provider itself due to thin provisioning strategies; Virtual disk space being allocated on-demand, a disk may grow above the boundaries of the physical disk storing it. In addition, combined with distributed storage, a snapshot allows the virtual disk to transparently continue to grow on another physical disk without data transfer. Although they are not visible to the client, such snapshots will be placed in the chains in the same way as the client-made snapshots and will participate in the chain's size increase.

As mentioned above, reducing chain size with streaming has a limited effect because the cloud provider has no control over the client-made snapshots. Furthermore, streaming impacts guest I/O performance: we measured the disk latency from the guest with `ioping` on a standard SSD (WD Blue) and noted a 100x increase during streaming. Streaming can be quite long according to the

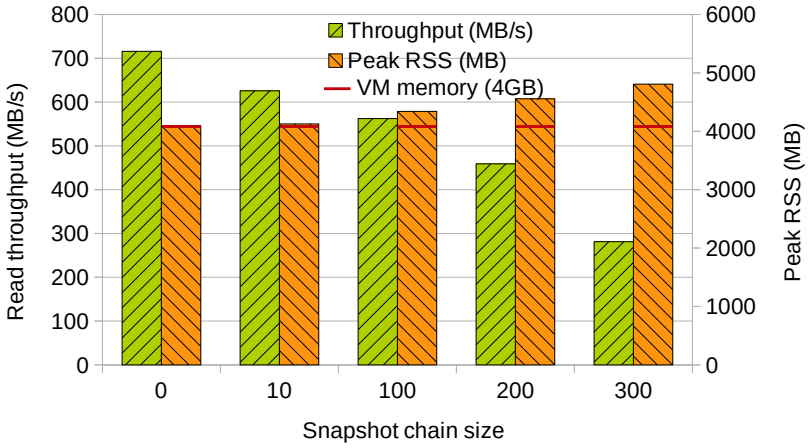


Figure 3.8: I/O performance and memory footprint evolution with snapshot chain size.

size of the merged snapshots, and a streaming operation needs to abort if the client decides to reboot/halt the VM.

Take-away 4: *Long chains are due to the client- as well as provider-made snapshots, and to the limitations of the methods (e.g., streaming) to reduce their length.*

3.4 Problem With Long Snapshot Chains

3.4.1 Problem Statement

From the illustration presented in Figure 3.3, the reader can intuitively see the two scalability issues posed by Qcow2 for long chains. The first is the memory footprint increase caused by L2 entry duplication in indexing caches. In Figure 3.3, clusters 1 and 2 are present in the two indexing caches. The second consequence is the negative impact on IO request latency. We can formalize the average cache miss cost (Y) using this equation:

$$Y = [(Hit_{\%} \times T_M) + (Miss_{\%} \times (T_D + T_L + T_F)) + (UnAl_{\%} \times T_F)] \times N \quad (3.4.1)$$

where T_M is the RAM access time (about 100ns), T_D is the disk access time (about 80μs), T_L is the time to traverse all software and network layers (about 5μs), T_F is the time to traverse only the software (about 1μs), N is the chain length, $Hit_{\%}$, $Miss_{\%}$, and $UnAl_{\%}$ are respectively the hit, miss and unallocated events ratios. Because T_D, T_L, and T_F are too high compared to T_M, even a tiny miss and unallocated ratio will lead to significant performance degradation [115]. Long snapshot chains exacerbate this degradation.

3.4.2 Assessment

A VM running on a long Qcow2 snapshot chain sees its performance and memory footprint seriously impacted. To demonstrate these points, Figure 3.8 shows the evolution of these two metrics for

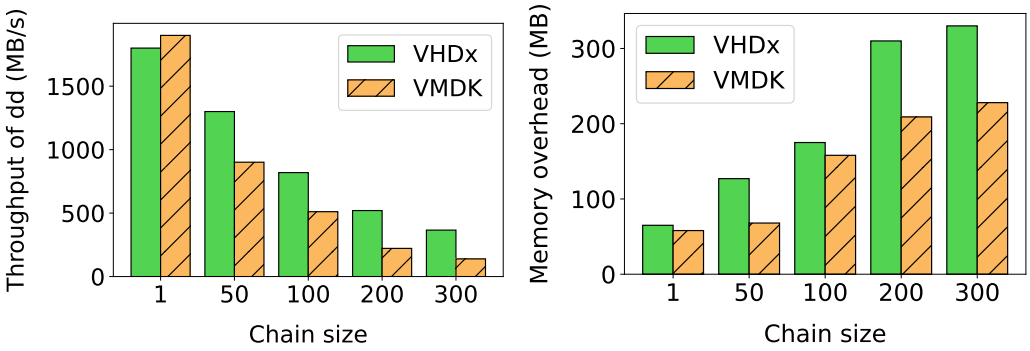
a VM running on a virtual disk with variable chain sizes, ranging from 0 to 300 snapshots. The total virtual disk size is 20 GB and each snapshot contains an incremental layer of 60 MB. All files reside locally on the host's SSD. The VM has 4 GB of allocated RAM, 4 vCPUs and runs Ubuntu 18.04. The read throughput is measured within the VM by reading the entire disk with `dd` right after 1) a first call to `dd` on the entire disk to ensure L1/L2 caches are fully populated and 2) a guest page cache drop to assure that the Qcow2 file is accessed. The memory footprint is measured from the host as the hypervisor's peak Resident Set Size (RSS) observed during the execution of the `dd` command.

As one can observe, although with small chains the read throughput is not substantially impacted, when the chain size grows performance drops significantly. On a virtual disk with a chain size of 300, the read throughput only reaches 39 % of what can be achieved on a disk with no snapshots. Regarding memory consumption, with no or a few snapshots the memory overhead that Qemu presents on top of the 4 GB used by the VM is negligible. However, with long snapshot chains that overhead becomes significant: with 300 snapshots, 711 MB of additional RAM are consumed by Qemu. We used the `massif` heap profiler of Valgrind to investigate memory consumption and discovered that the memory footprint increase is due to various data structures that are allocated on a per-snapshot basis. The main culprit for the high memory consumption with long chains is the L2 indexing cache. There is one Qcow2 driver instance running in the hypervisor for each Qcow2 snapshot in a chain. Although the maximum L2 cache size defaults to 1 MB [44], in our experiment we set it to ≈2.7 MB which is the maximum value to manage all the cache of a 20 GB disk – setting it lower seriously impacts performance. However, because there is one cache per driver instance and one instance per snapshot, one can conclude that the cache-related memory footprint increases linearly with the number of snapshots in the chain. We also profiled the Qemu hypervisor from the host during the execution of the aforementioned `dd` test on the 300 snapshots-long case and found that the guest only executes for 7 % of the time. Qemu's disk driver threads consume the remaining time.

These numbers were gathered on Qemu 4.2 but we also confirmed this behavior on a very recent (v6.0) version. We focus on 4.2 in the rest of this work as it is the version used by our cloud provider partner at the moment of the research investigation. Although this version may seem outdated, cloud providers notoriously use old software versions with backported security updates (i.e. long-term support) for obvious stability reasons.

Other formats We also evaluated other popular virtual disk formats, including Hyper-V's VHDX [76] and VMWare's VMDK [119]. As shown in Figure 3.9, these formats suffer from the same scalability problems. problems. We notice that the amount of overhead is not the same here as in the Qcow2 format, this is because virtual disk and snapshots management are different in each format but despite this fact, the scalability problem remains present.

Take-away 5: *Long chains lead to memory footprint and IO performance scalability issues.*



(a) Throughput

(b) Memory overhead

Figure 3.9: I/O performance and memory footprint for various disk formats.

3.5 Design of SVD: Scalable Virtual Disk

This section presents SVD, a solution to the two scalability issues identified in the previous sections regarding performance and memory consumption. Ideally, both metrics should be as independent as possible from the length of the backing file chain length.

3.5.1 Principles and Challenges

SVD relies on two fundamental principles, illustrated in Figure 3.10: 1) direct access to on-disk indexing/data clusters, regardless of their position in the chain, upon guest I/O requests; 2) using a single unified indexing cache, avoiding duplication of cache entries by being independent of the chain length. We apply the first principle through a backward-compatible modification of Qcow2, requiring the storage of additional metadata in virtual disk images and an update to the Qcow2 driver in the Qemu storage stack. Applying the second principle only requires a careful modification of the Qcow2 driver.

A significant challenge the implementation of SVD faces is its transparent and fast integration within the infrastructure of our cloud partner (and within cloud infrastructures in general). Our solution should be compatible with the different backends that can hold disk backing files in today's cloud infrastructure. These can be stored directly on the host disk, accessed by the host through the network, and served by centralized NFS servers or distributed file systems. Hence, we propose to modify a popular existing disk format rather than propose a new one [115]. A related challenge is also backward compatibility. Existing Qcow2 images lacking our format's metadata should still work with our updated version of Qemu (without performance/memory consumption gains on long chains). In addition, images using our format should work with Qemu that do not run our updated Qcow2 driver (once again without gains on long chains). Alternatively, vanilla disk images can be easily converted to our format to benefit from the performance/memory footprint enhancement on long chains.

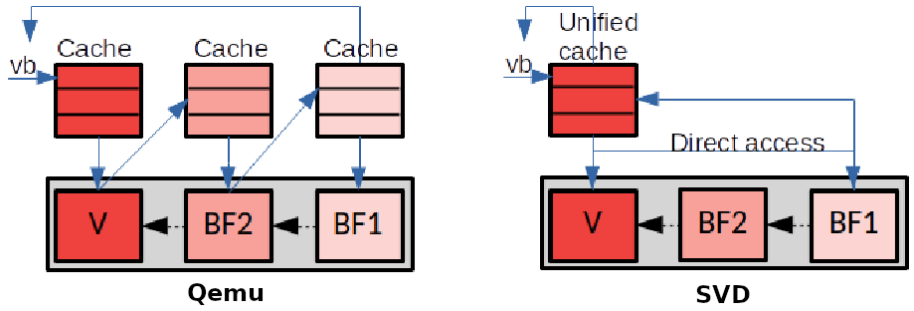


Figure 3.10: Vanilla Qemu (left) compared to our SVD, which follows two principles: direct access and unified indexing cache.

3.5.2 Format Improvement

When a guest issues an IO request, Qemu sequentially scans the active volume and all the backing files in the chain until the proper one is found, which is inefficient. Our design eliminates that chain scanning operation. We introduce new metadata in the format indicating, for each data cluster, the backing file that contains the latest (i.e., valid) version of the cluster. We call this metadata the `backing_file_index`. We leverage unused bits in L2 table entries to store that information.More precisely, we use 16 bits to encode `backing_file_index` in each L2 entry.

3.5.3 Unified Cache and Direct Access

With direct access, we maintain a single unified cache for the entire disk, independently of the length of the backing file chain. Our cache has the same organization as the vanilla Qcow2 cache presented in §3.2. As a reminder, a cache entry corresponds to a slice and contains: `l2_slice_offset` (playing the role of the tag), `ref`, `dirty`, and the L2 entries composing the slice. As noted in the previous section, in SVD an L2 entry contains `backing_file_index` in addition to the default Qcow2 values. The Figure 3.11 presents the new I/O journey with SVD contrary to the one presented in §3.2

Contrary to the vanilla version where `l2_slice_offset` was specific to each backing file, in our version, `l2_slice_offset` is related to the active volume. In addition, one can find L2 entries describing data clusters belonging to distinct backing files in the same slice. Therefore, the read and write operations are performed as follows in SVD. Let us consider vb, the offset of a virtual block that the guest wishes to read. Using the same functions a Qemu, SVD computes `l2_slice_offset`, `l2_slice_index` and `l2_index`. If both the slice and the L2 entry exist in the unified cache and that `backing_file_index` contained in the L2 entry corresponds to the active volume, there is a cache hit and the offset of the cluster data to be read is in the L2 entry. If `backing_file_index` does not correspond to the active volume, this is a cache hit unallocated. SVD locates on disk the backing file corresponding to `backing_file_index` and reads from it the slice at offset `l2_slice_offset`. Let s_b be that slice and s_v be the slice currently contained in the unified cache. SVD traverses all s_b entries and updates the L2 entries in s_v with the corresponding contents in s_b under the following condition: the value of `backing_file_index` of the L2 entry in s_v is lower or equal to that of `backing_file_index` of the L2 entry in s_b. We call "*cache correction*" these replacement operations. Then it sets `dirty` to 1 in s_v, so the slice will be written to disk when it is evicted from

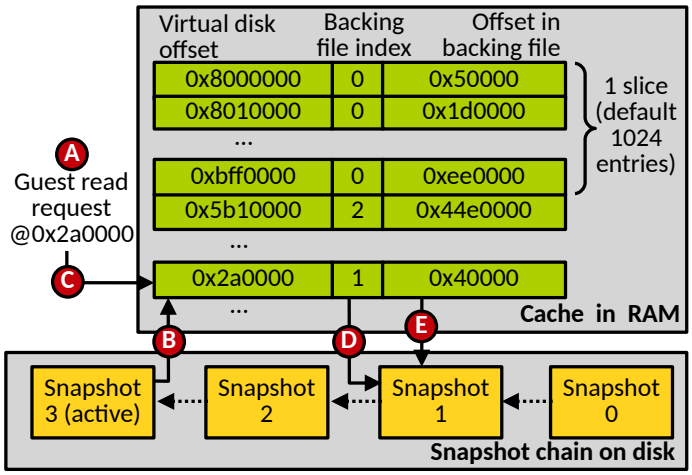


Figure 3.11: I/O journey in SVD with common cache using the new column `backing_file_index` in the cache structure

the cache. If the L2 entry does not exist in the slice, there is a cache miss, and the entry needs to be allocated as in Qemu. This means the guest is asking for a data cluster that does not yet exist on the virtual disk. If the slice is not yet present in the cache, there is a cache miss, and the slice is either fetched from the active volume if it exists, or allocated if not. These operations are similar to Qemu. It is important to note that cache correction is executed once when we resolve a cache miss for a data cluster and is no longer executed till the data cluster is evicted from the cache and needs to be prefetched from the image disk later.

3.5.4 Snapshotting

In Qemu, a new Qcow2 active volume is created on snapshot creation, with very little information (the header, the L1 table, and refcounts). We updated the snapshot creation logic to copy the entire content of both L1 and L2 tables from the previous active volume to the newly created active volume, now a backing file. The algorithm that we implement is as follows. Let `new_volume` be the file that will become the new active volume and `old_volume` the old one. We intervene in the creation of the L1 table. Recall that it is always located at the second cluster of any Qcow2 file. Let `new_l1` be the new L1 table and `old_l1` the L1 table of `old_volume`. After the allocation of `new_l1`, we parse all the `old_l1` entries. For each entry we create the corresponding L2 table in `new_volume`, then we set the current `new_l1` entry with the offset of that L2 table. Let `new_l2` be that new L2 table and `old_l2` be the L2 table pointed to by `old_l1` in `old_volume`. Then we copy the whole content of `old_l2` to `new_l2`.

Consequently, a new active volume always contains all L2 tables of the previous backing files. The copy of L2 tables may lengthen disk snapshotting time compared to the vanilla version. We could have implemented a copy-on-demand solution. However, that would mean impacting the critical path of I/O requests. This approach would increase tail latency, requiring chain scanning to find the valid backing file. The evaluation results show that the disturbance brought by the snapshot operation upon guest I/O performance is mainly acceptable, as the total size of L2 tables

is, in the worst case, in the order of MB. In addition, we believe that VM owners are likely to accept the small price of a slight increase in snapshotting time, to benefit from an essential boost in I/O performance.

3.6 Evaluation

Here we present an evaluation of SVD, aiming to answer the following three questions:

Q_1) Does SVD eliminate the memory footprint scalability issue of Qemu? (§3.6.2)

Q_2) Does SVD eliminate the IO performance scalability issue of Qemu? (§3.6.3-3.6.4)

Q_3) To what extent does SVD increases snapshotting time and disk overhead? (§3.6.5)

3.6.1 Evaluation Setup

Methodology. We systematically compare SVD with Qemu. We limited our comparison to only Qemu because its current version embeds the optimizations that prior academic works presented. We evaluate several configurations by varying three parameters: the chain length (1-1,000); the virtual disk size (50GB, 150GB); as well as the cache size (from 30 % to 100 % of the cache size needed to hold the entirety of L2 entries to index a full disk, i.e., from 1.9 MB to 6.25 MB for a 50GB disk size, and from 5.6 MB to 18.75 MB for 150 GB). For all experiments, we uniformly distribute valid clusters on the backing files of the disk's chain; meaning that all clusters are equally likely to be accessed. The virtual disk is populated at 90 % with random data for experiments with micro-benchmarks using the Linux `dd` command, and at 25 % for experiments with macro-benchmarks using the RocksDB client [36]. The release of SVD includes a highly configurable chain generation script.

Unless otherwise indicated, the size of the L2 cache is set so that it can hold all L2 entries to index the entire disk. All results presented in this section are an average value of 5 runs.

Testbed. To have a representative test environment, we employ two servers: the compute node running VMs and the storage node holding virtual disk files. Each server has 32 Intel Xeon Gold CPU cores, clocked at 2.10GHz, 192 GB of RAM, Samsung MZ7KM480HMHQ0D3 SATA SSD. They are linked with a 10Gbps Ethernet connection. The storage node serves the virtual disk files through NFS. Both servers run Debian 10 with Linux 4.19.0 as the host OS. All VMs run Ubuntu 18.04 with Linux 4.15.0 and are configured with 4GB of memory and four vCPUs. Unless otherwise indicated, the virtual disk size is 50GB.

Metrics and Benchmarks. We collect two types of metrics, *high-level* and *low-level* metrics. The former directly impact the end-user's perceived Quality of Service. We consider VM startup time, memory overhead, application execution time, and I/O disk throughput. The memory overhead is the additional memory consumed by Qemu on top of the VM's allocated pseudo-physical memory. Low-level metrics represent internal costs that help explain high-level metrics. They are: the total number of cache misses, the number of cache hit unallocated, and the cache lookup latency. The lookup latency is the time to find a data cluster's valid offset in the caching system. Storage benchmarks are run in the guests. We use microbenchmarks, including Linux `dd` (which

sequentially read the entire disk from the guest i.e. `dd if=/dev/sda of=/dev/null bs=4M`) and fio [69] (70% random access, half reads and half writes, with iodepth of 32 and with direct write), as well as macro-benchmarks, RocksDB-YCSB [36] and a measurement of the VM boot time.

For the rest of this section, when we talk about snapshots, it will be a chain of snapshots referencing each other with the data evenly distributed over all the snapshots, like the users (or the cloud provider) who would take snapshots regularly. Another difficult choice made in this section was the choice of the length of snapshots (50 and 500). It was not significant to make the experiments with all possible chain lengths. So we focus on two lengths. The first one (50 snapshots in a chain) used to represent the common length in cloud environment validated by our cloud provider partner and the former one (500 snapshots) used to represent the worst case which doesn't appear that much but when it appears, is very impacting from the user point of view.

Note It is important to note that most of the benchmarks are read-intensive because Qemu bottlenecks only appear when we have to fetch data in the whole chain, while write operations are always done in the active snapshot (the last one in the chain).Contents/svd/

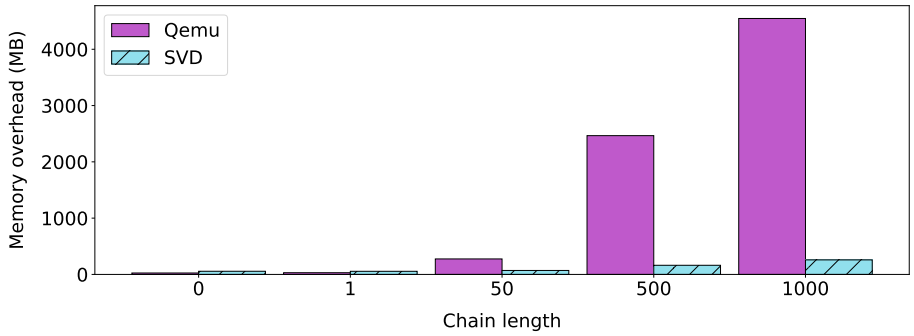


Figure 3.12: Memory overhead of SVD and Qemu after reading the entire disk from the guest with `dd`, while varying chain length. **Lower is better.**

3.6.2 (Q_1) Memory overhead

For this experiment, we measured Qemu's resident set size after having read the entire disk from the guest using `dd` and subtracted from this measurement the amount of RAM given to the VM (4GB) to compute Qemu's memory overhead. Figure 3.12 shows the results. One can observe that SVD significantly reduces the memory overhead when the chain length increases. The memory savings are as follows: 205 MB for a chain length of 50 (3.9× reduction), 2303 MB for a length of 500 (15.2×), and 4289 MB for a length of 1,000 (17.6×). Although it scales much better than vanilla Qemu, SVD's memory overhead slightly increases with the chain size. This is due to other per-snapshot data structures in Qemu that are not directly related to the caches. Finally, note that SVD comes at the cost of a slight memory footprint increase over vanilla when the disk has no or a tiny number of snapshots – a cost that is amortized by the better scalability starting from 5 snapshots.

3.6.3 (Q_2) Low-level Metrics

We use the same setup as in the previous section.

Cache Misses and Cache Hit Unallocated. We instrumented SVD and vanilla Qemu to measure the number of cache misses, the number of cache hits unallocated, and the number of cache accesses per backing file of the chain. Figure 3.13 shows the results.

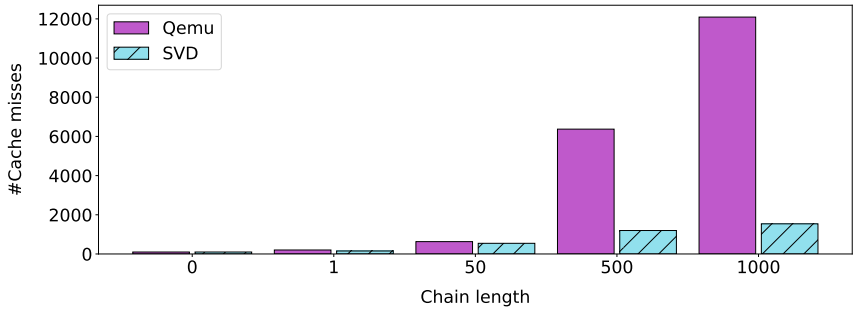
We can see that SVD leads to fewer cache misses compared to Qemu, as Figure 3.13a shows. We measure up to 10× for chain length 1,000. This difference is explained by the fact that Qemu does not implement a cache correction mechanism as we do in SVD (see §3.5). Therefore, when an L2 entry is only present in the cache of the backing file of index m in the chain, Qemu will generate $n - m + 1$ cache misses walking the chain to get it, where n is the chain length.

Concerning the number of cache hits unallocated, it is constant under SVD, see Figure 3.13b. The increase compared to a virtual disk composed of a single active volume (chain length 1) is less than 1% for the chain length 1,000. Concerning Qemu, the number of cache hit unallocated increases $10,000,000\times$ for the chain length 1,000; $4,000,000\times$ for the chain length 500 and $\approx 600,000\times$ for the regular chain of length 50. This is once again explained by the fact that Qemu looks up several caches during the chain walk.

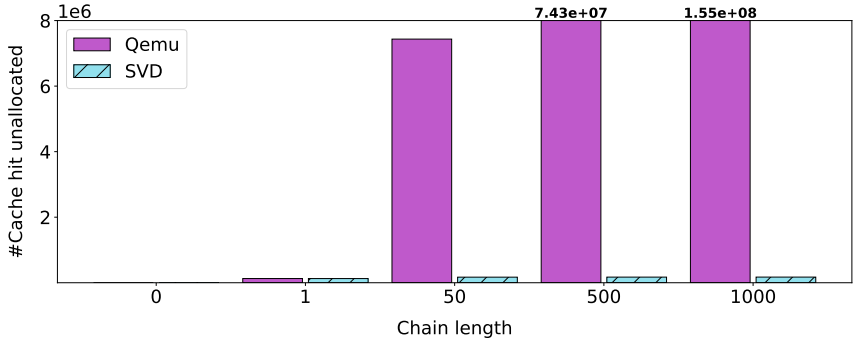
In the experiment with a chain of length 500, we count the total number of cache lookups and plot their distribution, according to which backing file in the chain holds the requested data, in Figure 3.13c. Due to the chain walks, caches are much more frequently accessed under Qemu compared to SVD. The gap is about 1,500%. The spike that appears for backing file zero, the base virtual disk image, corresponds to the boot of the VM. In fact, during that time, several IO read requests are performed on read-only files (such as vmlinuz). The spike on snapshot 500 corresponds to the accesses made on the active volume.

Cache Lookup Latency. We measured the cache lookup latency on two chain lengths: 1 and 100. Figure 3.14 presents the distribution of cache lookup latencies for all IO requests performed during the execution of the `dd` benchmark. We can observe that for both systems, the mean latency value changes according to the chain length. However, SVD leads to a better latency than Qemu when the chain length increases: the mean latency is 490 ms under Qemu and 270 ms with SVD, i.e., 1.8x faster. Contrary to Qemu, latency values under SVD are located around two mean values, 120 ms and 270 ms. 120 ms corresponds to the cache hit mean latency, while 270 ms corresponds to the cache hit unallocated mean latency. Theoretically, according to the direct access principle implemented by SVD, only one value of cache hit unallocated latency can be observed compared to Qemu. We do not observe the same kind of distribution under Qemu because, in this experiment, data clusters are uniformly distributed over all backing files. Therefore, most IO operations lead to a variable amount of cache hits unallocated, i.e., chain walks of variable length, according to the target data location in the chain. This translates into highly variable and, on average higher latencies in Qemu.

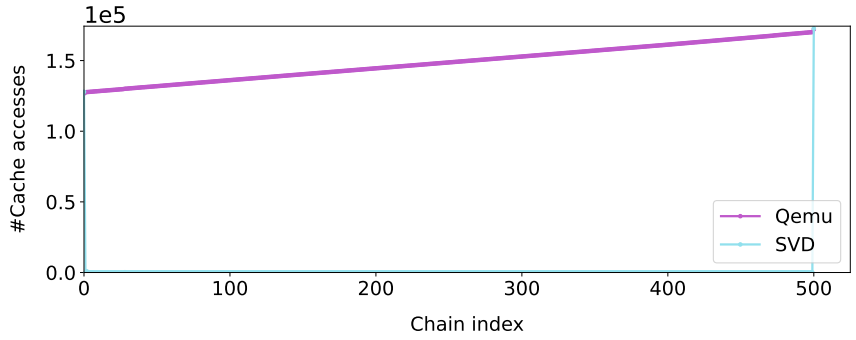
Concerning the variance of latencies value it appears that Qemu has a higher value than SVD. For the chain length of 1 snapshot, the variance is quite the same (≈ 10) because there are not many backing files and many caches to go through. But when we go up to a chain of 100 snapshots, the variance latency value of Qemu (17) is higher than SVD's value (12) while in Qemu we go through all the caches of all snapshots, we have more diverse values of latencies where we go through essential and the same caches with SVD and then we have fewer latency values to process the variance.



(a)



(b)



(c) Chain length of 500 snapshots.

Figure 3.13: During an entire disk read with `dd`, the number of (a) cache misses, (b) the number of cache hit unallocated, and (c) the distribution of cache lookups according to which backing file holds the addressed data. **Lower is better.**

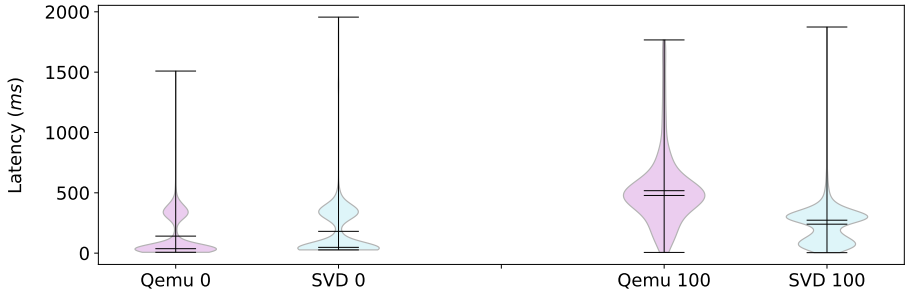


Figure 3.14: Cache lookup latency distribution. **Lower is better.**

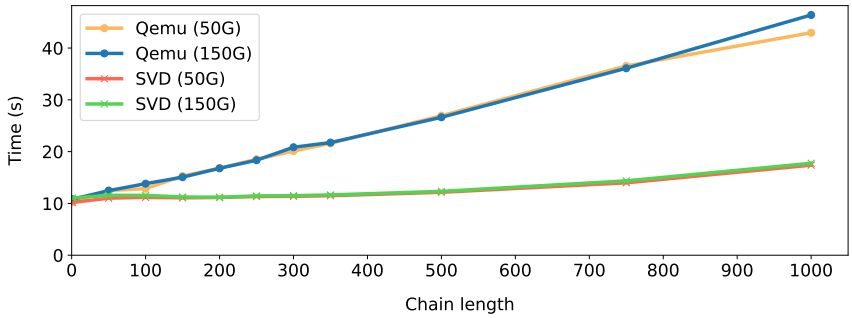


Figure 3.15: VM Boot Time. **Lower is better.**

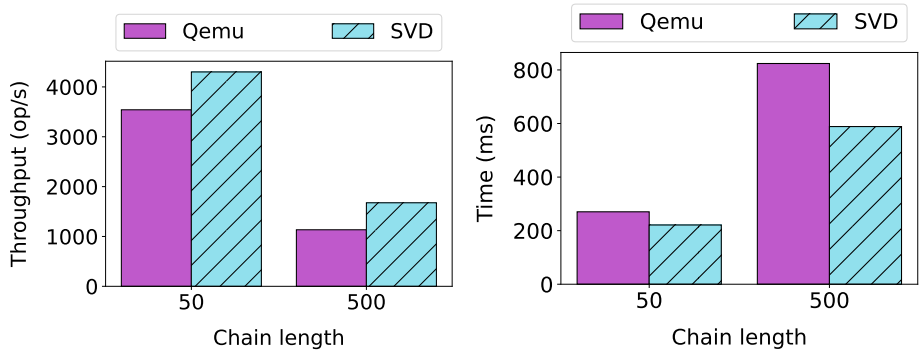
3.6.4 (Q_2) High-level Metrics

3.6.4.1 Macro-benchmarks

VM Boot Time. VM boot time is a critical metric in the cloud [74, 83]. Figure 3.15 compares the time it takes to boot a VM under SVD and Qemu while varying the chain length and the virtual disk size. The boot time increases rapidly with the chain length under Qemu: it goes from about 10 seconds on a chain of size 1 to more than 40 seconds (4×) on a chain of size 1000. On the contrary, with SVD that increase is moderate: from 10 seconds to 17 seconds (1.7×).

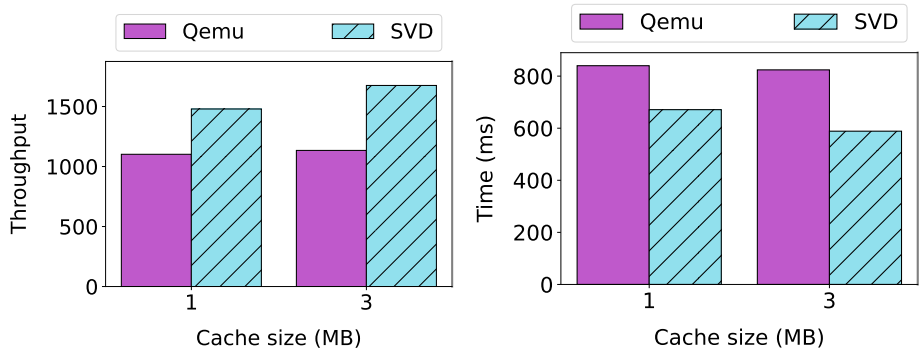
The increase in boot time for SVD can be explained by the slight increase in the number of cache misses and cache hit unallocated discussed above. We can see that the size of the virtual disk has a negligible impact on the results.

Cloud Workload: RocksDB-YCSB. We created a RocksDB database that fills 40% of the VM disk size, populated using the YCSB client, generating a uniform distribution of valid clusters on the Qcow2 chains generated. We used three YCSB workloads: YCSB-C, which simulates a user performing read-only requests; YCSB-B, which simulates a user performing a mix of write and read requests but most reads; and YCSB-D, which realizes more write vs. read requests We experimented with two L2 cache sizes (1 MB and 3 MB) and two chain lengths (50 and 500 snapshots). We measured the throughput and execution time (RocksDB's two performance metrics) of YCSB for a total of 500K requests.



(a) Throughput, with 3MB cache size. **Higher is better.**

(b) Execution time, with 3MB cache size. **Lower is better.**



(c) Throughput, with chain length 500. **Higher is better.**

(d) Execution time, with chain length 500. **Lower is better.**

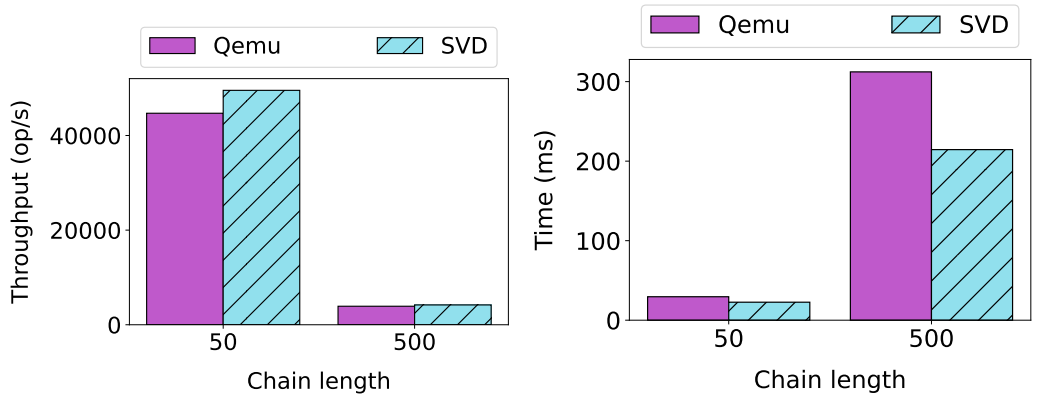
Figure 3.16: RocksDB-YCSB results for YCSB-C.

Figures 3.16a and 3.16c show the results for the throughput metric. Even if the performance of both versions decreases when the length of the chain increases, SVD still outperforms Qemu for both chain lengths (33% for length 50 and 47% for length 500). Further, with a fixed chain length at 500, the throughput of YCSB is almost constant while varying the cache size, regardless of the Qemu system.

Figures 3.16b and 3.16d present the execution time results. As for the throughput, SVD improves Qemu. Considering a chain of 50 backing files, SVD reduces the execution time of YCSB by 36% for 1MB cache size and 22% for 3MB cache size. For a chain of 500 snapshots, the improvement is about 40% with 1MB of cache size and 36% with 3MB cache size. For throughput and execution time, the improvement of SVD over Qemu is more significant when the chain length increases.

Figure 3.17 shows that even when there are more write requests (YCSB-D), as soon as the system uses read requests, performance are impacted.

Performance improvement status It's important to recall that Rocksdb is a key-value store based on the log-structured merge-tree data structure so the data are more often kept in memory so



(a) Throughput, with 3MB cache size. **Higher is better.**

(b) Execution time, with 3MB cache size. **Lower is better.**

Figure 3.17: RocksDB-YCSB results for YCSB-D.

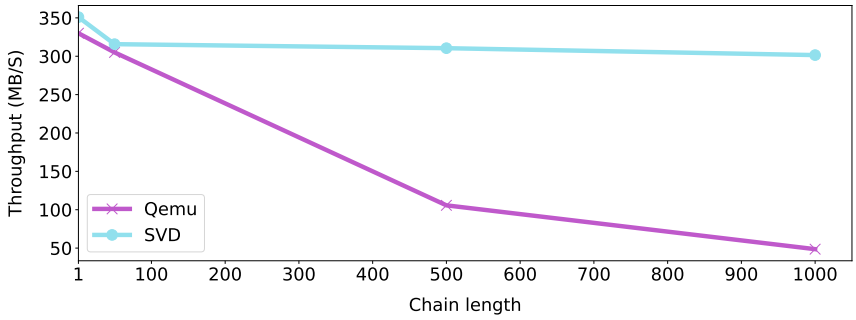


Figure 3.18: Throughput of Linux `dd` under SVD and Qemu with various chain length **Higher is better.**

it's almost normal we don't have the same gap of performance improvement compared to another storage. But despite this fact, there is still an overall non-negligible improvement from our SVD solution when dealing with heavy usage in cloud provider infrastructures.

3.6.4.2 Micro-benchmarks

Disk Throughput: Linux `dd`. The throughput of `dd` is presented in Figure 3.18 for both systems managing chains of various sizes. We can observe no degradation under SVD while Qemu severely degrades the throughput of `dd` when the number of backing files increases. Qemu incurs a slowdown of up to 84% for the chain length 1,000.

Impact of the Cache Size with fio. We studied the effect of varying the cache size for SVD and Qemu. In this experiment, we use a chain of length 500 and set the total cache size used by Qemu to equal that used by SVD. Because Qemu uses one cache per layer in the chain, when SVD

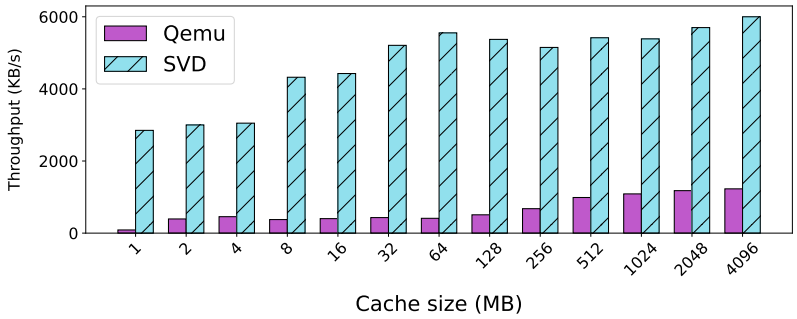


Figure 3.19: `fio` throughput while varying the cache size. (100% reads) Chain length of 500 snapshots. **Higher is better.**

is given cache size of S, Qemu would get S/L with L being the chain length. We vary the cache size given to each system from 1MB to 4GB and measure the disk read throughput with fio performing random reads of small size (4 KB) on the disk node in `/dev` on one side and on another size random reads and few write.

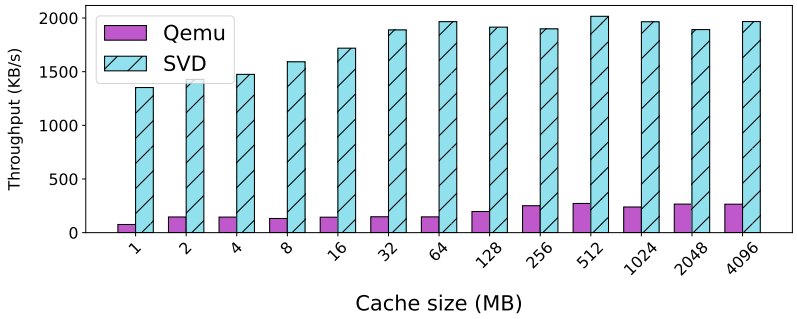
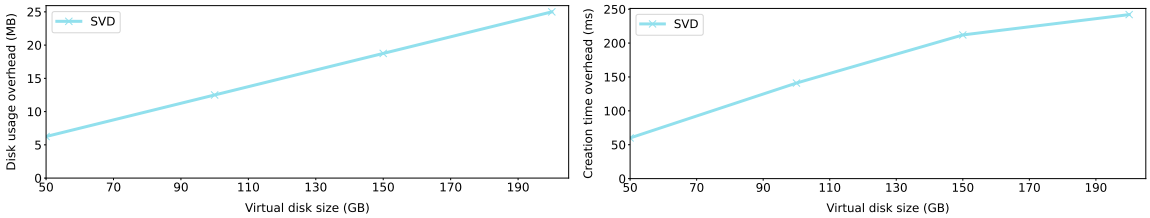


Figure 3.20: `fio` throughput while varying the cache size. (70% read - 30% write) Chain length of 500 snapshots **Higher is better.**

Figure 3.19 shows the results. We can observe that SVD significantly outperforms Qemu in all cases. With both systems, performance is sensitive to the cache size. Concerning Qemu, performance steadily increases up to 4 GB of cache. This is due to the large amount of memory required by this multi-cache solution. Regarding SVD, although peak performance is also achieved at 4 GB (6 MB/s vs. 2.5 MB/s for 1 MB of cache), from 32 MB, the payback from adding more cache size diminishes significantly. This value thus represents an excellent trade-off between near-peak performance and memory footprint. This demonstrates the high efficiency of SVD vs. Qemu.

Figure 3.20 shows similar results when operations are mixed between reads and writes with more reads. Here, performance gains from SVD are lower than Qemu's because write operations do not need to go through the chain, avoiding the performance issue.

In summary, if we want to consider Qemu optimal in term of execution time and throughput, we need to choose the maximum value of cache chose considering the minimum value of cache that can cache all clusters' data metadata which is in the case of this experiment $4GB$ for each 500



(a) Disk usage overhead per snapshot.

(b) Snapshotting time.

Figure 3.21: Impact of SVD on snapshotting. **Lower is better.**

snapshots i.e. $\approx 7MB$ per snapshot.

3.6.5 (Q_3) Overhead

As stated in §3.5.4, when creating a snapshot under SVD, L2 tables are copied to the newly created file. This may incur two overhead types: disk usage and snapshotting time.

Disk space. The disk space overhead per snapshot depends on both the VM's disk size and cluster size and the number of allocated clusters in the disk. We can model that overhead in the worst-case scenario, i.e., when every cluster is allocated (the disk is full), as follows: given S_{SQ} and S_{VQ} being respectively the size of a newly created (i.e. empty) snapshot under SVD and Qemu, we compute the disk size of S_{SQ} using the following formula:

$$S_{SQ} = S_{VQ} + \frac{VM\_disk\_size}{cluster\_size} \times L2\_entry\_size \tag{3.6.1}$$

By default, an L2 entry is 8 B, a cluster is 64 KB, and S_{VQ} is 256 KB. Using the above formula, we can compute S_{SQ} while varying the VM disk size from 50 GB to 200 GB. Figure 3.21a shows that per-snapshot overhead. It increases linearly with the size of the VM disk. To compute the total disk overhead (still in the worst case), the per-snapshot cost needs to be multiplied by the chain length. Recall that from our characterization, we observed that the dominant virtual disk size in the cloud is 50 GB, giving, according to our model, a per-snapshot overhead is about 6 MB. This gives a total overhead, in the worst case, of 60 MB for a chain of length 10 (0.1% of the virtual disk size), 600 MB for length 100 (1.2%), and 6,000 MB for length 1000 (12%).

Snapshotting Time. We measured the time spent to create a new snapshot under SVD and Qemu for different VM disk sizes. The results are presented on Figure 3.21b. Due to the copy of all L2 entries, SVD takes much more time to create a snapshot than Qemu. For a 50GB VM, we need about 70ms to create a snapshot under SVD and 7× less time under Qemu. Furthermore, this overhead increases with the VM disk size. Indeed, for a 200GB VM, the snapshot creation time under SVD is about 12× that of Qemu. Nonetheless, in absolute, the snapshot creation latency is quite low under SVD (in the order of ms). It allows for a relatively high snapshot frequency.

We indeed generate an overhead when creating a snapshot in terms of time and space. For the time overhead, we add an overhead that grows in a logarithmic order (slowly) and we assume that increasing a little bit (unperceivable to the end user) the time is an excellent tradeoff against the

Table 3.1: Snapshot creation time.

VM Disk Size(GB)	vQemu Time(ms)	SVD Time(ms)
50	10	70
100	15	156
150	21	233
200	23	265

overhead time on each request that we are resolving with SVD. For the space overhead, it depends on the virtual machine disk size. Knowing that users reserve light VMs for their tasks i.e. with small disk size, the size of the overhead space on the host disk will be negated.

It is important to notice that this overhead only occurs at snapshot creation. L1/L2 tables are not updated in backing files when the active volume is accessed. The principle of backing files is to maintain an old state so when the active volume is modified, it's only him who is modified. Then, there is no aftereffect update in the backing files behind the active volume so no additional overhead.

3.6.6 Evaluation Summary

We introduce and assess how SVD improves performance compared to the default Qemu using two new features: *direct-access* and *unified caching*. As depicted in Figure 3.12, *unified caching* helps decrease the amount of metadata (L2 entries) cached by maintaining a shared cache for all opened backing files. With *direct-access*, we can directly access the backing file containing the L2 entry we seek. Figure 3.13c illustrates that with SVD, we achieve direct access to the cache of the backing file containing the data, whereas Qemu engages in multiple recursive cache accesses of the backing files until it reaches the desired data.

3.7 Related Work

Virtual Disk Formats. Fast Virtual Disk [115] is a virtual disk format proposed by IBM in 2011, that increases I/O performance by avoiding using a host filesystem, reducing the size of on-disk metadata, and using an on-disk journal. FVD supports only internal snapshots, which means that the chain is stored in a single file. This may not be as flexible as the external snapshots offered by the format we focus on, Qcow2, for example, when subsets of a chain need to be stored on different storage nodes for load-balancing or capacity reasons. It is also unclear how FVD performs on long chains composed of hundreds or thousands of snapshots. FVD presents a set of optimizations in scenarios where the virtual disks reside on network-attached storage: copy-on-read and adaptive prefetching. They help to improve I/O access times by hiding network latency and this remains compatible with SVD since we maintain old features of Qcow2 active while extending it. These optimizations are efficient with distributed storage but are orthogonal with our problem since they do not permit resolving the long chain problem on local storage. The system we propose is an evolution of Qcow2 which is backward compatible with vanilla Qcow2 disk, something that makes adoption much easier versus proposing an entirely new format. Finally, FVD can be considered as deprecated as it was developed for Qemu 0.14, dating from 2011, and has not been ported to modern versions.

Parallax [75] is a distributed architecture storing virtual disk images and using commodity servers as storage backends, as opposed to high-end storage arrays/switches. Among other features, Parallax offers low-overhead and high-frequency snapshots and notes, similar to our work, that the performance overhead and memory consumption of traditional formats such as Qcow2 increases with snapshot chain sizes. Similar to FVD, migrating an existing cloud environment to Parallax requires significant changes to the virtualized storage system's architecture, whereas we rely on the widely used Qcow2 format, and are backward-compatible with environments that do not use our system. Further, contrary to our Qcow2 format, Parallax does not support sharing of virtual disk images, a feature heavily used in the industry to lower storage overheads of commonly used volumes e.g. base images.

Storage Performance and Availability during VM Migration. Noting that VM migration significantly disrupts guest I/O performance, A few papers [63, 133] focus on maintaining good storage performance and availability during migration. Netchannel [63] proposes various techniques to maintain local/remote virtual disk availability during migration. One is the ability to switch the physical device associated with a virtual one seamlessly. Another proposed technique is the capacity for migrated VMs initially plugged into a local disk on the host to transparently keep using that disk through a proxy once they are migrated to another host. In another study [133], the author proposes to study the storage I/O the behavior of guests to infer the most efficient data transfer schedule to reduce disruption as much as possible during VM migration.

Scheduling Impact on Virtual Storage Performance. Several studies [86, 60] noted that VM scheduling could have a non-negligible impact on guest I/O performance. The authors of a study [86] characterize the impact on processor and I/O performance of various VM scheduler configurations, for concurrently-running guest with CPU- and bandwidth-intensive, as well as latency-sensitive behaviors. In another paper [60], the authors propose a guest task-level priority-boosting technique to selectively increase the priority of I/O-bound tasks to increase storage performance while maintaining CPU fairness.

Virtualized Storage Performance and Power Consumption. Other studies focus more generally on virtualized storage performance and power consumption [128, 116]. Ye et al. [128] note that existing power consumption reduction techniques for non-virtualized HDDs do not apply in a virtualized setting. They propose to bridge the semantic gap between VM and VMM through several techniques tailored for such environments, reducing disk spin-ups and increasing disk sleep times, to save energy. Another paper [116] focuses on the particular problem of interrupt delivery to VMs, including the ones coming from block devices. The authors propose an optimized interrupt delivery system for KVM. It is mostly evaluated on network workloads but also shows moderate performance improvements on storage workloads.

Cloud Storage and File Systems. Finally, several papers [93, 16, 71] focus on cloud storage and filesystems. The Frugal Cloud File System [93] proposes integrating multiple services (AWS EBS, Azure Cache, etc.) into a single solution that aims to be flexible from the performance and cost point of view. DepSky [16] introduces a cloud-based storage system targeting security/dependability by spreading and replicating storage over multiple clouds. Depot [71] proposes a cloud storage system that can tolerate buggy clients and servers to minimize trust assumptions.

3.8 Summary

For the first time, we presented the characterization of snapshots' chain length in a large-scale public cloud. Among other results, our analysis revealed the presence of long snapshot chains, leading to scalability issues for both memory footprint and performance. We present SVD, a solution to these two issues in the form of a slight extension of the Qcow2 format while preserving backward compatibility. We built SVD following the principles of direct access and single indexing cache, regardless of the chain length. We evaluated SVD extensively and compared it with vanilla Qemu using a wide range of benchmarks, demonstrating that our solution effectively tackles the above issues. For instance, SVD improves the IO throughput of RocksDB by up to 48% compared to Qemu and reduces the memory footprint by 15x when the chain length is 500.

CHAPTER 4

Efficient Distributed Storage using Constraints-based Replica Selection

This chapter describes the NUDA (Non-Uniform Disk Access) issue and his impact on load balancing in distributed storage. It also presents **Nami**, *a novel solution we establish to address temporal load balancing in distributed storage.*

Contents

4.1 Introduction

The explosion of data, coupled with the desire to process them fast and automatically (AI/ML/IoT), puts significant pressure on storage systems. These systems are becoming one of the main bottlenecks in data centers [113, 43]. Storing data almost indefinitely, serving it in record time (milliseconds, or even microseconds for certain applications [15, 8]), and ensuring no loss in case of a crash are essential requirements.

To address these needs, storage systems have been developed in recent years based on two principles: server aggregation and replication [59]. Server aggregation federates the power (capacity and computation) of multiple machines (called storage nodes) to store as much data as possible, preventing a bottleneck at a single location. Replication involves storing the same data on multiple storage nodes, ensuring fault tolerance. Glusterfs [52] and Ceph [24] are two of the most popular open-source distributed storage systems that follow these principles for their performance and robustness.

Distributed storage systems grapple with a pivotal question: how can we ensure that all storage nodes, on average, experience similar levels of usage? This issue is commonly called the *load-balancing* problem. Unlike other resources such as CPU, which can be measured with a single-dimensional usage metric, distributed storage presents a unique challenge due to its dual components: spatial and temporal usage. Spatial usage is defined as the amount of bytes utilized on the storage device, typically expressed in gigabytes (GB). Temporal usage, on the other hand, refers to the volume of input/output (IO) requests executed within a specific time frame, denoted in I/O operations per second (IOPS). Up to now, existing distributed storage systems have predominantly focused on spatial usage load-balancing. This occurs at data block creation time, encompassing the creation of data replicas.

This work focuses on *temporal load balancing*, which is more dynamic compared to spatial load balancing. We show in Section 4.2.2 that existing systems such as Ceph [24] and Gluster [52] leave the system in what we call a *Non-Uniform Disk Access* (NUDA) state. These storage systems statically choose one replica as the primary block, that is the block accessed by all I/O requests in the direction of a particular data, while other replicas of the same data block serve as backups in case of failure. Due to workload variation, statically choosing the primary replica that satisfies I/O request load-balancing is practically impossible.

Previous works [59, 51, 61] proposed replica migration for IOPS balancing. However, using replica migration for IOPS balancing faces practical challenges. The migration time is lengthy, not suitable for real-time I/O request responses, and saturates network links, disrupting ongoing I/O requests. Applying this approach in a large-scale cloud environment exacerbates these issues. Even Ceph's authors acknowledged [78] that capacity balancing alone is insufficient for load balancing across storage nodes.

Other studies [129, 84, 9] have explored the replica selection approach. This approach involves choosing any replica, rather than a single one as nowadays' distributed storage systems [24, 52]), to handle I/O requests and balance the I/ over the storage system. However, these studies have some limitations. Ye et al. [129] introduced a simulation-based solution for a specific set of requests to a particular dataset. The challenge is that it requires knowing the incoming set of requests beforehand, which is practically impossible in today's highly dynamic cloud and IoT systems. Nwe et al. [84] focused solely on storage system latency and targeted non-scalable systems like HDFS and MongoDB, in contrast to scalable systems like Ceph and Gluster. This narrow focus limits the applicability of the findings to non-scalable storage environments. Awang et al. [9] proposed a

simulator that considers the potential affinity between data and outputs a set of replicas to use to balance IOPS in storage systems. While these works highlight the replica selection approach, they primarily rely on simulations and don't delve deeply into the possibilities offered by the approach, focusing on non-truly scalable storage systems.

We exploit the existence of secondary replicas to dynamically select the most suitable replica and storage node for each I/O request, thereby distributing accesses across as many nodes as possible. While this concept is straightforward, its implementation poses several challenges, including comprehending the storage system's dynamics through metrics and adapting the replica's usage based on heuristics implemented by the system administrators.

The proposed system denoted as Nami, achieves this goal by dynamically assigning a weight to each storage node. These weights indicate the probability of a storage node being chosen to fulfill a data block I/O request.

Our initial investigation (Section 4.2.2) highlights the odd utilization of resources and the presence of NUDA in distributed storage systems such as Ceph and Gluster. For illustration, using a synthetic benchmark based on `fio` [38], we measured up to 40% standard deviation of storage nodes' disk usage. We found that the origin of this problem lies in the fact that these systems choose for each data block a replica as the primary one to serve IOPS and leave the other replicas to be used only in case of failures. Knowing that it's impossible to assert primary replica will be distributed equally between disks (Section 4.2.2). This leads to the disparity in disk accesses depending on the access patterns of clients' applications.

In a subsequent phase (Section 4.3), we aim to address the issue of the NUDA state by introducing a dynamic replica selection approach that preserves the spatial load-balancing feature at the same time. We propose Nami, a solution to this problem that can be implemented in any distributed storage system. A key feature of Nami is to allow sysadmins to implement its weight calculation algorithm based on observed system requirements (disk usage, node's latency, disk type). Another key feature of Nami permits to dynamic change weights of storage nodes by collecting periodically a set of metrics used in the weight calculation. For illustration, we present and implement Nami in Ceph [24] with two weights computation algorithms based on the latency metric; the first algorithm ranks on order storage nodes' network latency - the less the latency is, the more the weight is. The second algorithm is more participatory, the calculation of each weight depends not only on the network latency of the storage node but also on the latencies of every node.

The final part of this chapter involves the evaluation (Section 4.4) of Nami. Using each of the two weight calculation algorithms presented, we assess the I/O throughput of popular benchmarks like `fio` [38] and `RocksDB` [36] with fixed and variable emulated latencies during workload execution. We observe a performance increase with Nami by up to 30% compared to vanilla Ceph. Furthermore, the measurement of disk utilization demonstrates that Nami balances requests more evenly across disks, resulting in a final disk utilization standard deviation of 13% compared to 40% with Ceph. At the end, we compare the benefits of the two weight computation algorithms that we introduced. We observe that the first algorithm (based on ordered latencies) has a smaller failure rate of selecting a good replica when we choose a replication factor of 3; whereas the second algorithm based on a (participatory computation) has a smaller failure rate with a replication factor of 4; meaning that the efficiency of the weight calculation heuristic also depend on the intern configurations of the distributed storage.

In summary, this chapter contributes:

- An exploration of server resource utilization (spatial and temporal) in existing distributed storage systems.

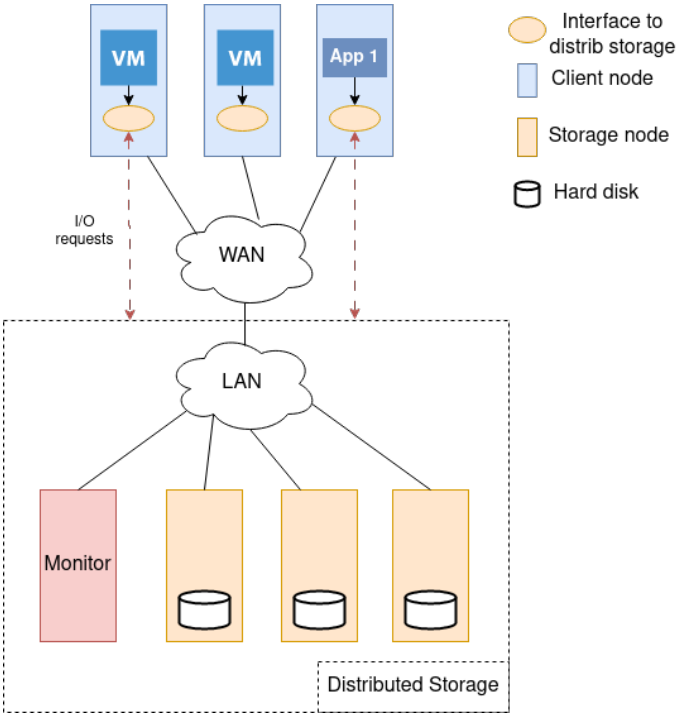


Figure 4.1: Distributed storage systems architecture.

- Presentation of our replica selection approach called **Nami** and the implementation of a prototype in the Ceph storage system.
- Evaluation of Nami, demonstrating an increase in I/O throughput and a more balanced usage of storage system resources.

4.2 Background and Motivations

4.2.1 Background

Distributed Storage Systems Distributed storage systems play a crucial role in the efficient management and availability of data in modern computing environments. Unlike centralized storage systems, distributed storage systems distribute data across multiple nodes, referred to as storage nodes, typically within a local network. This approach provides enhanced resilience, increased elasticity, and improved performance.

In our study, we have chosen the Shared-Nothing storage system (SN-SS) as our distributed storage architecture [59]. This choice is justified by its recognized scalability, robustness, and reliability, especially suitable for long-term write-intensive needs [59, 72]. The configuration of such an architecture is illustrated in Figure 4.1.

The SN-SS is a category of distributed storage system where compute nodes, and hosting user applications (such as virtual machines, containers, and bare-metal apps) are entirely independent of storage nodes. Storage nodes are connected to a local network to facilitate maintenance and data migration operations; client nodes access them through a wide area network (WAN) such as the Internet or an enterprise's intranet. It is essential to note that data is stored in fixed-size blocks, typically a few kilobytes, with the block being the smallest storage unit in operating systems [67].

Client nodes (or computation nodes) contain client applications (such as virtual machines, etc.). These applications use an interface to access the distributed storage, which can be a disk mount point or a client daemon. This interface presents the entire distributed storage cluster as a single central storage for all applications. It is responsible for determining to which storage node I/O requests must be directed to retrieve data.

Replication in Distributed Storage Data replication is an important feature of distributed storage systems, ensuring the same data availability in case of node or network failures. The implementation of replication involves creating identical copies of data on multiple nodes, providing redundancy and fault tolerance.

When a client application wants to write a data block (Figure 4.2), a storage node is chosen by an interface (using the CRUSH algorithm not relevant to expand in our study) to the distributed storage as shown in Figure 4.1 for its initial write, referred to as the primary replica. This storage node represents the primary node for that data. Other nodes are chosen to store replicas of this data in parallel in the background. However, as long as the primary replica is safely stored, the data is considered written, and the client can be informed. All of this is made by taking into account capacity balancing which ensures the equitable sharing of data in each storage node.

During data reading, operations are primarily directed to the primary replica. However, the existing capacity balancing policies implemented by mainstream distributed storage systems do not guarantee an equitable distribution of primary replicas. This uneven distribution can lead to the unequal utilization of storage nodes.

To illustrate this point, consider the data distribution shown in Figure 4.3, where two primary replicas of different data blocks are located on the same node, for instance here the primary of the red data block and the primary of the yellow data block are stored on the storage node S_1. In case of two I/O requests to the red and the yellow data blocks, the storage node S_1 will be overloaded because I/Os are directed to primary replicas, while the storage nodes S_2 and S_3, which also possess the same data blocks remain completely free and unused.

Other possible scenarios could be when the first storage node is overloaded in terms of network load, but I/O requests continue to be directed to it, while other nodes are available. There are several similar cases (disk usage of storage nodes, CPU-dependent background tasks, etc.) where utilizing secondary replicas could significantly improve the storage system's performance and, consequently, that of client applications.

4.2.2 Replica selection and Migration in Cloud Storage

In today's digital era, the exponential growth of data necessitates robust storage infrastructures capable of handling immense volumes of information while ensuring reliability, availability, and performance; this is even more true with the massive coming of Edge computing. Distributed storage networks have emerged as a fundamental framework to meet these escalating demands, distributing data across multiple nodes [24, 52, 104]. One method to ensure those escalating demands is for

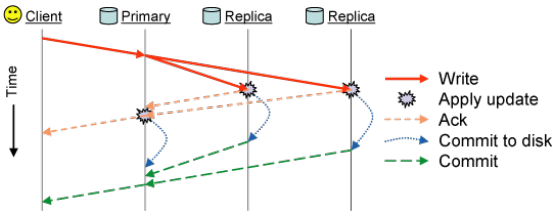


Figure 4.2: Write workflow in distributed storage systems.[126]

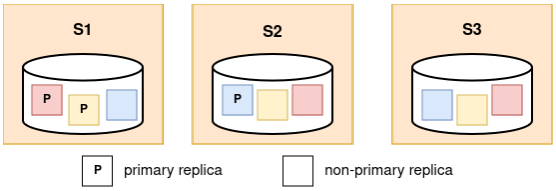


Figure 4.3: Example of data repartition where more primary replicas are presented on a single node.

example usage of data replication when they are created or data migration when faults occur on some servers. However, the efficient selection of replicas and the migration policy within these networks pose significant challenges that directly impact system performance and reliability.

In these distributed environments, selecting the optimal replicas for read and write operations is a complex task [17]. Factors such as minimizing latency, ensuring data consistency, load balancing, and fault tolerance are critical considerations. Suboptimal choices can lead to increased access times, reduced throughput, data inconsistency, or even vulnerability to system failures. Some works combine migration of replicas [81] dealing with the fact that migrations, even though costly, save more time for the request. The main challenge is when to migrate. Papers like [61] showed how it is difficult to plan a good migration and that it depends on too many factors as clients or providers.

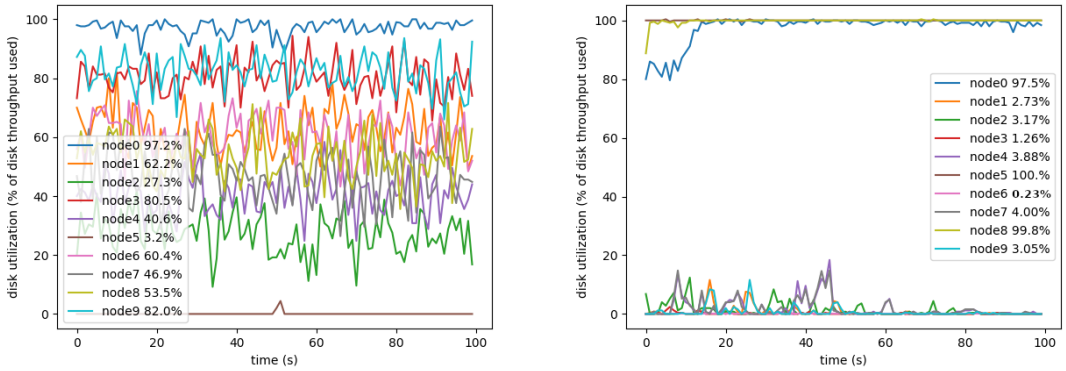
Improved selection strategies can mitigate latency, enhance data availability, optimize resource utilization, and increase disk lifetime in data centers for the sake of providers.

We conduct an experiment in a distributed storage of 12 nodes: 3 compute nodes where we execute workloads and 9 storage nodes which make up the storage cluster. The distributed storage systems used are Ceph [24], (one of the best-known open-source distributed storage solutions), and Gluster [52], (one of its great competitors over the years [31, 1]). We run a various number of VMs simultaneously from 1 to 10 distributed between compute nodes; each VM runs a *fio* workload [38]; random R/W with 70% reads and 30% an iodepth of 4 and the default libaio engine. The replication factor is 3. Section 4.4 presents in detail the experimental setup.

Figure 4.4 shows the utilization of each disk during the execution of the workload. It is the percentage of disk bandwidth used among all storage nodes on both Ceph (Figure 4.4a) and Gluster (Figure 4.4b) clusters. One striking observation is the unequal usage of nodes (then disks); while some are fully utilized, others remain significantly underused. In Ceph, we observe that the standard deviation of disk utilization is 30% with the most used node at almost 100% while the least used is at 25%. In the case of Gluster, we have a larger standard deviation larger (43%). It is less catastrophic in Ceph due to its policy of capacity balancing [21]. A similar analysis was conducted by Ceph developers [78].

This discrepancy can result in two main concerns: Firstly, it may lead to performance unpredictability due to the presence of unused disks. Secondly, the lifespan of overused disks might decrease. It is important to note that replacing a disk in the cloud or applying any maintenance operations can be costly for the provider due to data migration [127]. All this shows how the cluster state is important in the choice of an optimal replica selection algorithm.

Current approaches to replica selection often rely on heuristics or basic algorithms that may



(a) Ceph cluster.

(b) Gluster cluster.

Figure 4.4: Disks usage during workloads. The percentage indicates the average percentage of the node's disk usage over the workload

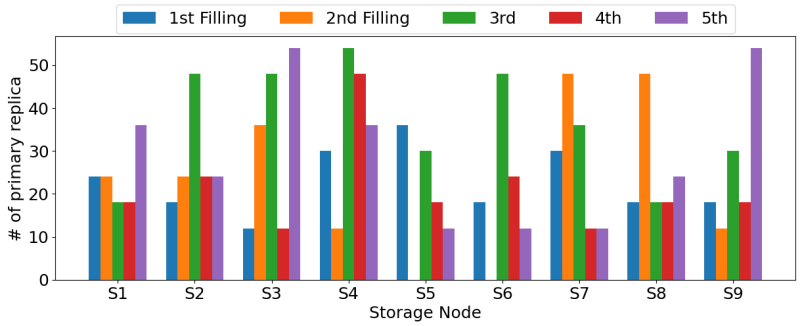
not adequately address the complexities of modern distributed systems or may rely on migration policies [112, 61]. Some approaches prioritize proximity or load distribution without considering dynamic network conditions or varying access patterns, resulting in suboptimal performance and resource inefficiency [39].

We conduct another analysis to understand why capacity balancing is not sufficient to handle query balancing. Using the same setup of cluster storage presented above, we fill 5 times the distributed storage with 13 VMs' virtual disks of size 50GB and we collect the distribution of primary replica over the storage nodes.

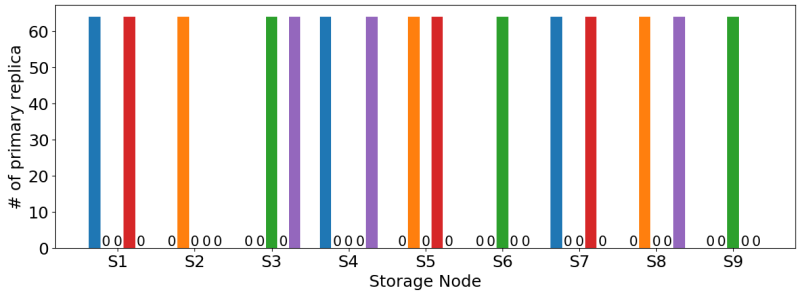
In Figure 4.5, we present the repartition of primary blocks on Ceph and Gluster. We observe that on Ceph (Figure 4.5a), storage nodes have unbalanced primary replica distribution; for example, storage node S_9 has the lowest number of replicas at the 2nd filling (12) where he has the highest in the 5th filling (54). On Gluster it is different because administrators when setting up the cluster storage, can choose the storage nodes that will contain all the primary replicas but the number of these storage nodes is always N/rep where N is the number of storage nodes and rep the replication factor; in our case, it is 9/3 i.e. 3 storage nodes who will contain all the primary replica equally distributed between them. During each filling of the storage node (Figure 4.5b) we vary these 3 storage nodes.

This raises two main concerns: the first is that primary replicas are non-equally distributed between all the storage nodes so, knowing that I/O requests use the default primary replica, it logically leads to unbalancing of I/O requests; the second concern is the fact that after each filling of the storage, the storage nodes with the high number of primary replica is never the same. All this shows that capacity balancing is not enough to achieve IOPS load balancing.

In conclusion, with modern distributed storage systems, the primary replica, from which all read requests are served, is allocated statically and does not change throughout time. However, the storage system is a dynamic environment and the performance of storage nodes, as seen from each of the clients, will evolve. Hence, a static primary replica allocation is suboptimal.



(a) Ceph cluster.



(b) Gluster cluster

Figure 4.5: Primary replica distribution among storage nodes after 5 distinct fillings of the storage.

4.3 Constraints-aware Replica Selection with NAMI

4.3.1 Overview

Figure 4.6 depicts the architecture of **NAMI**. NAMI aims to optimize the selection of replicas based on constraints predefined by the storage cluster's sysadmin. It incorporates several key components to ensure a balanced distribution of loads and optimal performance of the storage system. This section provides a more in-depth exploration of each component, its functioning, and its interaction with other elements of the system.

Metrics Collector The Metrics Collector is the initial component of our architecture responsible for retrieving metrics associated with each storage node. An instance is presented on each node of the distributed storage system and frequently collects metrics. These metrics include network latency, current disk load primarily, and other relevant parameters configurable by the system administrators. Real-time data collection provides an accurate snapshot of the current performance of each node. (step **a.**)

Controller The Controller plays a central role in the decision-making process. It aggregates metrics collected by the Metrics Collector and employs heuristics to assign a weight (**Weight adaptor**) to each storage node. These weights represent the relative probability of a node being

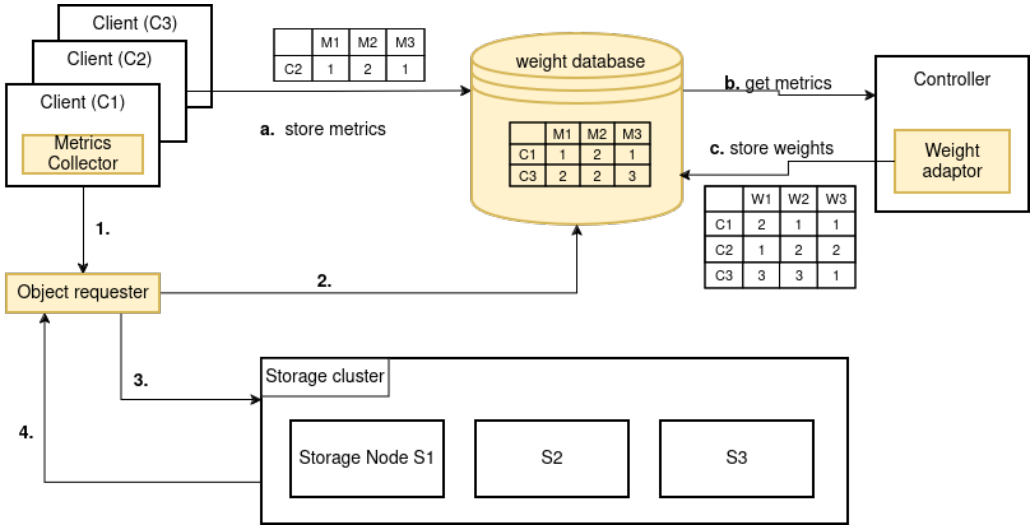


Figure 4.6: Architecture of **Nami** with 3 clients and 3 storage nodes. M_i and W_i are resp. the metric and the weight of S_i
a,b,c steps are background periodic steps.
1-4 steps are I/O request handling steps

selected during an I/O request to a replica. The Controller also ensures the dynamic updating of these weights based on changes in the environment. (steps **b.** and **c.**). This entity takes as input the heuristic to be used in defining node weights and executes it when there is a variation in metrics collected by the Metrics Collector.

Weight database The Weight Database is a key-value store that persistently retains the weights assigned to each storage node and the metrics used by the Weight adaptor. Each storage node has access to this store to retrieve weights associated with different nodes since it is distributed and replicated. This distributed approach ensures continuous availability of weights and lightweight access, even in the event of node failure or network disruption.

Object Requester Each client node in the storage system is equipped with an Object Requester component [24, 52] that sends I/O requests to the storage cluster; we modify it to consider weights during request handling. When an I/O request is initiated (step **1.**), the Object Requester intervenes using weights stored in the Weight Database (step **2.**). It applies sorting to determine optimal storage nodes, from which one is selected to serve the request (step **3.**), minimizing perceived latency for the client. This proactive approach ensures that each request is directed to a replica that optimizes the overall system performance.

4.3.2 Metrics Collector

The Metrics Collector plays a central role in the dynamic collection of metrics associated with each storage node. This operation occurs periodically with a configurable period, and all metrics

are retrieved and stored in the Distributed Store. A key characteristic of this component is the meticulous management of writes to avoid unnecessary overload on the Weight Database. As metric collections happen periodically, constant writes to the key-value store that may not be necessary can be averted.

Periodic Collection The Metrics Collector is programmed to query each storage node at defined intervals, thus regularly collecting metrics. These metrics encompass access latency, system load, node availability, network bandwidth, and other relevant parameters. A powerful tool for retrieving these metrics is the **sar** tool from the sysstat suite [45]. Since storing is done in the Weight Database, which is a key-value store, each entry takes the form of "$C_n M_m$" $\rightarrow [v_1, v_2, ..., v_n]$, with C_n being the client from which the metric originates, M_m the m^{th} activated metric, and the array v_i containing all metric values we collected on each storage node i of the cluster. For example, assuming the latency is the metric illustrated in figure 4.6. the metrics presented as a matrix will be stored in the key-value store like this:

$$\begin{array}{c} \textbf{key} \rightarrow \textbf{Value} \\ C_1\_lat \rightarrow [1,\ 2,\ 3] \\ C_2\_lat \rightarrow [1,\ 1,\ 2] \\ C_3\_lat \rightarrow [3,\ 2,\ 1] \end{array}$$

The periodicity of this collection ensures a real-time representation of the system's performance.

Weights Storing Rather than storing all metrics collected in a raw manner, the Metrics Collector considers the variation in values. It compares new metrics with previously stored values to determine the need for an update. This approach avoids unnecessary writes and reduces the load on the Distributed Store. A strategy based on the standard deviation between new and old values decides whether to modify metrics in the database. This adjustable approach allows adaptation to diverse environments where the frequency of changes may vary. It provides an efficient mechanism to keep the Weight Database updated while minimizing unnecessary data storage and processing.

4.3.3 Weight Adaptor

The Controller initiates a Weight Adaptor that performs weight calculations using the previously collected metrics. This component is relevant for assigning weights to storage nodes, thereby influencing the selection of replicas; one of the features of Nami and what makes it flexible is to possibility for sysadmin to write and include their weight compute algorithm. In this section, we present two different heuristics of the Weight Adaptor will be presented, highlighting the subtlety and adaptability of this approach.

Consider the following matrix of latency values between a client C_i and a storage node S_i in milliseconds (Table 4.1). We will use this matrix to illustrate our algorithms.

4.3.3.1 Algorithm 1: Latency-Ordered Weights

This heuristic, while relatively simple, proves to be effective in optimizing node selection based on latency (see §4.4). It assigns a higher weight to the node with lower latency (algorithm 1). The underlying idea is to prioritize nodes that provide faster response times, thereby enhancing

	C1	C2	C3
M1	1	1	3
M2	2	1	2
M3	3	2	1

Table 4.1: Matrix example of latencies

Algorithm 1: Latency-ordered weighting

Data: M_{ij} metrics gathered by the i^{th} client on each S_j
Data: n number of clients
Data: m number of storage nodes
Result: W_{ij} weights returned by the i^{th} client
initialize(W_{ij});
sort_desc(M_{ij});
for $j \leftarrow 1$ **to** m **do**
  $W[i,j] \leftarrow m - indexOf(M[i,j]) + 1$; `/* lowest latency has higher weight value */`

the overall system performance. The adaptability of this approach lies in its ability to adjust the acceptable latency threshold according to the specific requirements of the system.

For instance, considering the latency matrix in Table 4.1, the corresponding weight matrix would be the ones on Table 4.2

	C1	C2	C3
W1	3*	2*	1
W2	2	2	2
W3	1	1	3*

Table 4.2: Matrix weight algorithm 1
(*) node likely to be selected

Thus, storage node S_1 is more likely to be chosen (during the period when we collect these latencies) to serve replicas for requests from clients C_1 and C_2 because it has the highest weight for each of these clients. However, given that it will be solicited by two client nodes, it can quickly become overloaded. The sysadmin might be tempted to define a weight calculation algorithm that depends on other storage nodes. An example of such a calculation algorithm could be the following.

4.3.3.2 Algorithm 2: Participatory weighting

The second heuristic we present here introduces a participatory approach where each storage node is assigned a weight based on the average latency of the storage node seen by other clients (see algorithm 2). The higher the average latency for a particular node, the higher its weight. The goal is to prioritize the storage node that has the least influence on others when there are numerous requests. This approach encourages a balanced distribution of load and considers the relative performance compared to the entire storage network.

By employing this participatory strategy, the system not only focuses on the performance of individual nodes but also considers their impact on the overall network. The objective is to prioritize nodes that, when accessed, contribute less to the latency experienced by other clients. This can lead to a more equitable distribution of the storage load and enhance the overall efficiency of the storage system.

Algorithm 2: Average-based weighting

Data: M_{ij} metrics gathered by the i^{th} client on each S_j
Data: n number of clients
Data: m number of storage nodes
Result: W_{ij} weights returned by the i^{th} client
initialize(W_{ij});
initialize(tmp) ; `/* buffer array */`
for $j \leftarrow 1$ **to** m **do**
  $tmp[j] \leftarrow \frac{\sum_{k=1,k\neq i}^{n} M_{kj}}{m-1}$; `/* average of metrics seen by other clients */`
sort_desc(tmp);
for $j \leftarrow 1$ **to** m **do**
  $W[i,j] \leftarrow indexOf(tmp[j])$; `/* high average has high weight */`

In this manner, with our example in Table 4.1, the new weight calculation would yield the following result:

	C1	**C2**	**C3**
W1	2*	1*	1
W2	1	1*	2
W3	1	1*	3*

Table 4.3: Matrix weight algorithm 2
(*) node likely to be selected

In this scenario, the options for client C_2 are more diverse, allowing us to choose storage node S_2 to avoid overloading nodes S_1 and S_2 which are the optimal choices for C_1 and C_3 respectively.

4.3.3.3 Weight Calculation Flexibility

It is important to emphasize that we offer significant flexibility to administrators here. They have the freedom to choose the heuristic that best suits their needs, adjust the parameters of existing ones, or even add new custom ones to align with specific storage infrastructure requirements, as demonstrated with the two previous heuristics. In summary, the Controller represents the most subtle point of the architecture, enabling advanced and adaptable analysis of metrics to assign weights to storage nodes. The two algorithms presented here do not pretend to be efficient ones or concurrent with each other; they illustrate the diversity of possible approaches, thus providing a robust solution for the optimal selection of replicas based on the specific needs of the system. The flexibility offered to administrators ensures continuous adaptation to changes in the storage environment.

4.3.4 Object Requester

The Object Requester component comes into play when an I/O request is issued, utilizing the weights assigned by the Controller to guide the selection of the optimal storage node. This section will detail the request process based on weights, also emphasizing the redirection of requests to the chosen node holding the replica. As replica selection mechanisms exist in almost all contemporary distributed storage systems [24, 52, 104, 112], the significance lies in showcasing how our weight-based solution is concretely applied in a distributed storage environment.

4.3.4.1 Request Routing

When an I/O request is initiated, the Object Requester steps in to select the optimal storage node based on the assigned weights. The routing process is as follows:

Weight Retrieval Before making a decision, the Object Requester checks if a cached version of node weights are present in memory. If so, it uses these weights for routing. Otherwise, it proceeds to retrieve weights from the Distributed Weight Database.

Node Selection Based on the obtained weights, the Object Requester chooses the optimal storage node to fulfill the request. This selection is made respecting the weight distribution, where a node with a higher weight has a greater probability of being chosen.

Request Redirection If the selected node possesses the requested replica, the request is redirected directly to that node. Otherwise, the Object Requester employs the standard routing process to locate the next node with the highest weight containing the replica and redirects the request to that node.

4.3.4.2 Weight Caching

To minimize performance impact, weights are cached in RAM. They are updated only when changes are detected in the distributed storage, signaling a weight update by the Controller. This approach prevents unnecessary remote weight retrieval operations for each request, ensuring efficiency in the system. To implement this, **Nami** installs a watcher on the distributed key-value store. This watcher acts as a monitoring mechanism, keeping track of changes in the weights stored in the Weight Database. By doing so, the system can promptly respond to updates initiated by the Controller, maintaining a real-time and synchronized awareness of weight changes across the storage nodes.

The installation of a watcher not only optimizes the efficiency of weight management but also contributes to the overall responsiveness of the system. It enables a proactive adjustment of weights without the need for constant polling, ensuring that the Object Requester has access to the most up-to-date weight information when making decisions during I/O requests. This dynamic and responsive approach to weight management enhances the adaptability of the system to changing conditions, promoting stability and optimal performance in diverse operational scenarios.

The sequence diagram in Figure 4.7 summarizes all these operations. A watcher continually operates, obtaining weights from the weight database and storing them in the RAM of the Object Requester. When an I/O request happens, the cached weights are accessed. These weights help

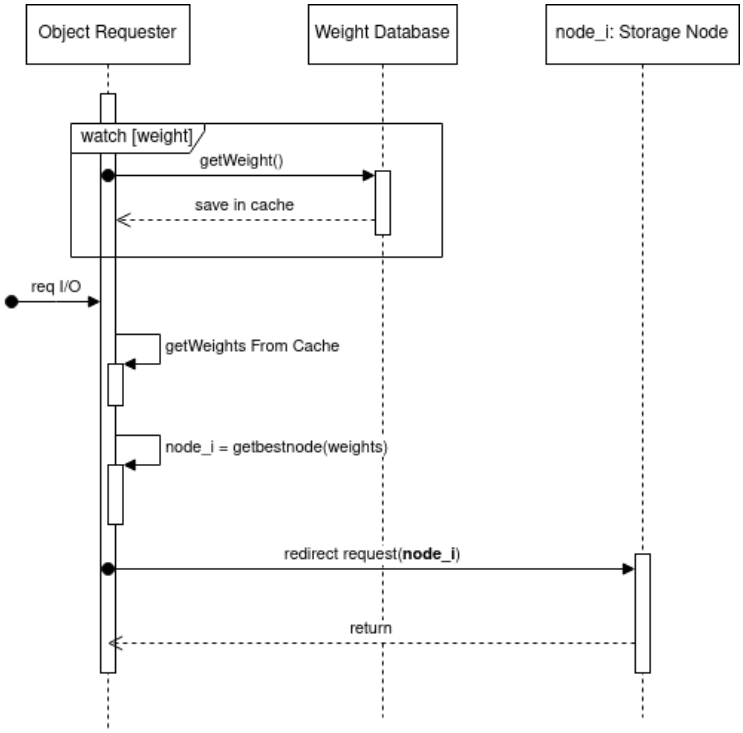


Figure 4.7: Watcher processus and request routing.

identify the optimal storage node, which has a replica of the required data, and the request is then directed to that node.

4.3.5 Ceph Integration

We prototype NAMI into Ceph [126] to demonstrate its functionality in a real-world environment. The choice of Ceph is based on its possession of all the characteristics of existing scalable and distributed storage systems, its widespread use worldwide, its proven track record (10 years of longevity), and, importantly, its open-source and community-driven nature.

Metrics Collector Despite being implemented in Python [94] as a script, this component stands out for its flexibility. For the proposed heuristics, tools such as **sar** [45] and **ping** were utilized to retrieve metrics such as latencies and disk usage rates of storage nodes.

Once decided on the metrics to use, the sysadmin implements a Python function which will return the metrics and this function will be passed as a parameter of our weight adaptor to calculate the different storage nodes' weights.

Controller The Controller serves as the configuration or administration server for Ceph clusters. We run background scripts like weight database updater when metrics change, including various heuristics of the **Weight adaptor**.

Weight database Each collected metric and computed weight is stored in **etcd** [35]. We chose etcd for its essential attributes in distributed systems. It is reliable, lightweight, distributed, and incorporates recognized consensus algorithms such as Paxos.

Object Requester This component incurred more costs as it required modifying the Ceph code. The goal was to modify this class while maintaining the integrity of the Ceph application. The Ceph blog [124] helped identify functions involved in I/O requests that we needed to modify to include our weight-based routines. Before each read/write operation, the `_calc_target` function calculates the target storage node for the request. Since our algorithm needs to retrieve the list of weights at each call to this function, an **etcd** watcher is added to the initialization of each storage system client to avoid frequent reads from the store.

Then, only two places in the code were modified. The function `_calc_target` and the constructor of Objecter where we added the watcher behavior.

The watcher activates only when there is a variation in weight values in the store above a set and configurable threshold. When activated, it retrieves weights to store in memory. Thus, the `_calc_target` function uses the values in memory to select the replica that will serve the request.

4.4 Evaluation

Here we present an evaluation of Nami and the in-depth study made around it, aiming to answer the following three questions:

Q_1) Does Nami increases the overall I/O performance ?

Q_2) Does Nami resolve the NUDA state problem i.e. the huge deviation and discrepancy in the usage of disks in cluster storage?

Q_3) How many times the replica selected by Nami is really used ?

4.4.1 Evaluation Setup

Methodology. We compare ceph [24] with and without Nami prototype included. We evaluate several configurations by varying three parameters: the number of Virtual Machines running a fio [38] benchmark (1 to 10 virtual machines); the number of replicas of the storage system (3, 4) and the weight adaptor algorithm used as presented in section 4.3. Note that we are not using erasure coding to fully take advantage of secondary replicas.

Testbed. To have a representative test environment, we employ twelve servers in cloudlab [34] organized as follows: two compute nodes running VMs that will be running a particular workload - a fio/YCSB benchmark in the VMs; nine storage nodes forming the cluster storage where all VM images (virtual disk) will be stored during experiments; and one node where the weight adaptator algorithm is executed. Each server has 32 Intel Xeon Gold CPU cores, clocked at 2.10GHz, 192 GB of RAM, Samsung
MZ7KM480HMHQ0D3 SATA SSD. They are linked with a 25Gbps Ethernet connection. Both servers run Ubuntu 20.04 with Linux 5.19.0 as the host OS. All VMs run Ubuntu 18.04 with Linux 4.15.0 and are configured with 4GB of memory and four vCPUs. Unless otherwise indicated, the virtual disk size of all VMs is 50GB.

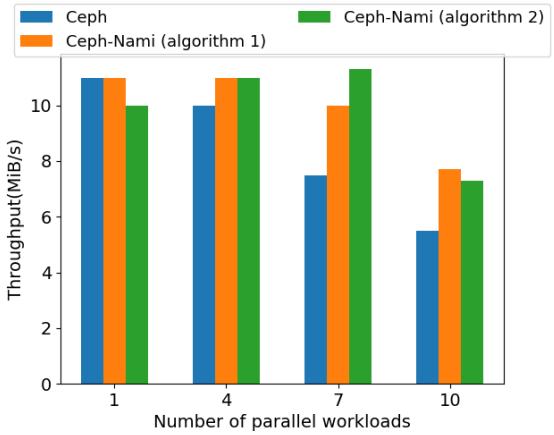


Figure 4.8: `Fio` Throughput of Ceph vanilla and Ceph-NAMI with the weight adaptor algorithm varying and the number of parallel workloads running varying

Metrics and Benchmarks. We collect two types of metrics, including *high-level* and *low-level* metrics. The former directly impacts the end user's perceived Quality of Service. We consider application execution time and I/O disk throughput. Low-level metrics represent internal costs that help explain high-level metrics. They are the disk utilization, the deviation in utilization, and the number of replica selections made during the workload execution.

We use fio [38] (microbenchmarks), as well as macro-benchmarks, RocksDB-YCSB [36]. The fio benchmark parameters are the same in all experiments: random R/W with 70% reads and 30% writes. The YCSB-C workload is the one used on the RocksDB benchmark execution.

It is important to note that most of the benchmarks are read-intensive because the concept of replica selection frequently appears when we have to fetch data in the whole storage, while write operations are always new or are preceded by reads in case of deletion or update.

4.4.2 Throughput (Q_1)

Throughput in a storage system is one of the most significant metrics as it directly reflects the system's performance for both administrators and end users. In this section, we present two types of results. The first, a more general metric, is the average throughput of our experimental benchmarks (`fio` and `RocksDB-YCSB`). The second explores the evolution of throughput during the experiments' execution.

4.4.2.1 Mean Throughput

We vary the number of VMs running on the compute nodes, and the weight calculation heuristic (algorithm) also varies between the two presented in Section 4.3. It is important to recall the described algorithms: Algorithm 1 embodies a "selfish" heuristic, where the weight assigned to each node is contingent solely upon its latency; conversely, Algorithm 2 embodies a participatory approach wherein node weight is influenced by the latencies observed across other nodes. The results are depicted in Figure 4.8.

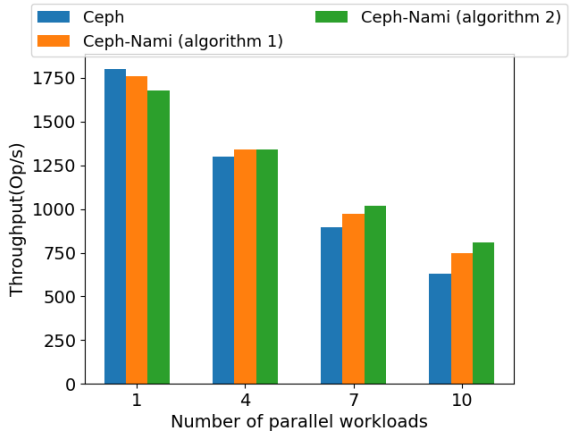


Figure 4.9: `RocksDB` Throughput using YCSB-C Workload of Ceph vanilla and Ceph-NAMI with the weight adaptor algorithm varying and the number of parallel workloads running varying

Figure 4.8 illustrates that as the number of workloads attempting to access the storage system increases, the throughput decreases. Without NAMI, the decrease is significantly more pronounced due to the lack of efficient redirection to replicas on free storage nodes. This holds for any chosen weight adaptor heuristic. We observe a performance gain ranging from 9% for an environment with 4 parallel workloads to 26% for 10 workloads, demonstrating the effectiveness and scalability of NAMI.

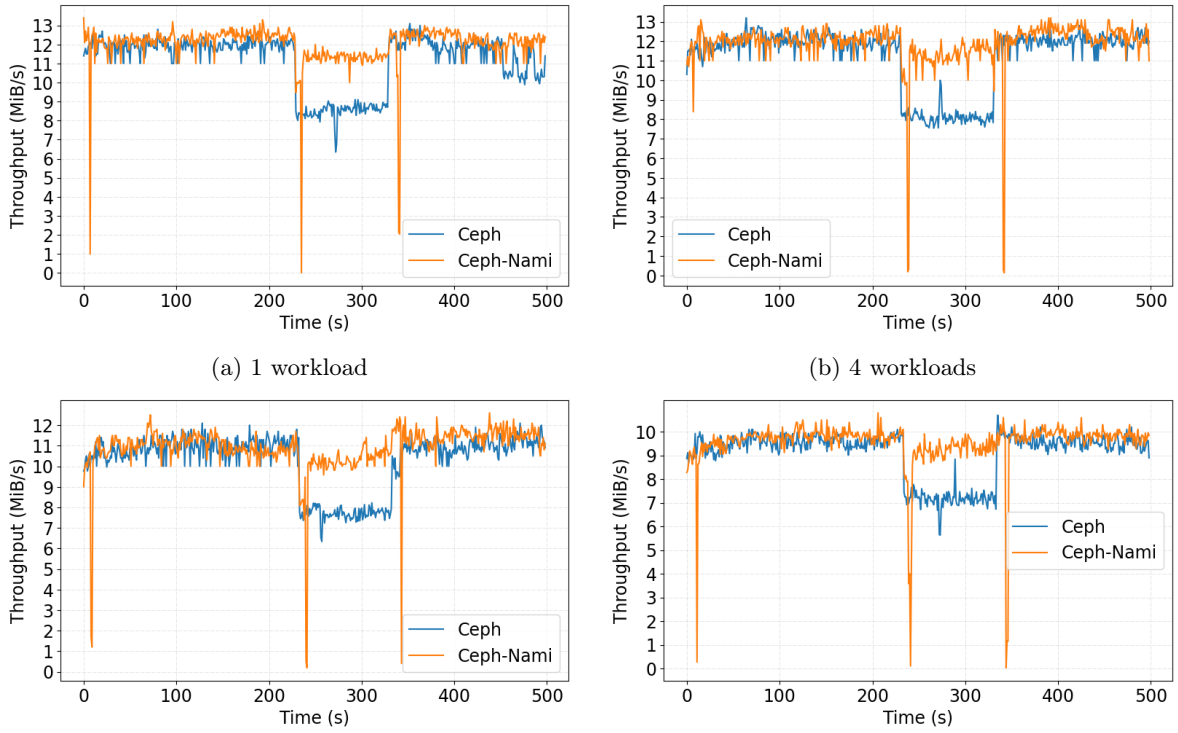
When there's only one workload in progress, algorithm 1 outperforms algorithm 2. This superiority arises because there is no need for consultations between node latencies, given the singular workload. However, when dealing with four or seven parallel workloads, algorithm 2 proves more effective. In this scenario, it becomes essential to consult the latency metrics of all storage nodes to ensure a well-balanced distribution of requests among these concurrent workloads.

In Figure 4.9, we presented the throughput results of our experiment when the workload is the YCSB-C (read-intensive) of `RocksDB`. It appears that independently of the number of parallel workloads running, NAMI seems to be a little better; an increase of 3% for 4 parallel workloads to 20% for 10 parallel workloads. But the increase in performance is less visible than on `fio` benchmark because `RocksDB` is an in-memory key-value store; then, our approach is effective only when data requested are not in RAM and we need to go for storage nodes to fetch data.

4.4.2.2 Evolution of Throughput

In this subsection, we explore the behavior of NAMI when there are dynamic changes in the system. To achieve this, we run the same experimentation as in the previous section but vary the latency of the storage nodes' network card. We use the **traffic control** (tc) tool [68] in Linux for this purpose, introducing an artificial latency of *1ms* between the 250th and 350th seconds of our experimentation. The added artificial latency aims to simulate unforeseen actions in real cloud environments, such as maintenance, VM migrations [59], and others.

Figure 4.10 presents the results. We observe that during the period of latency modification, without NAMI, there is a clear and instant performance drop of $50ms$, explained by Ceph not



(a) 1 workload

(b) 4 workloads

(c) 7 workloads

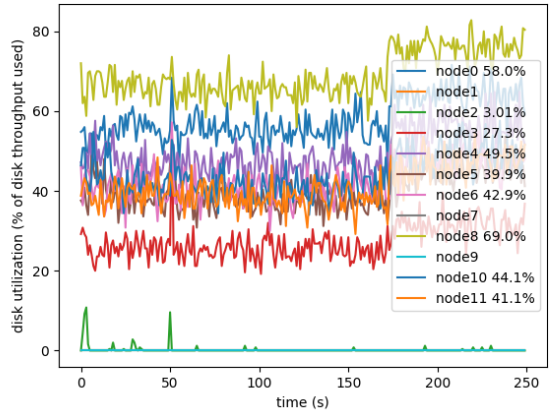
(d) 10 workloads

Figure 4.10: Evolution of Throughput during Workloads.
(*) heuristic used: Algorithm 1

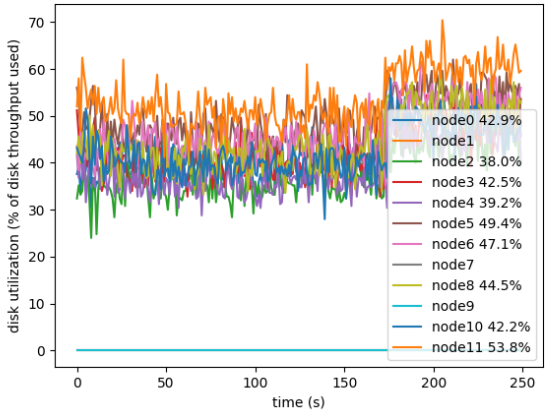
considering latency variations when choosing replica locations. With NAMI, which accounts for this, there is initially a half-second performance dip as the latency is abruptly changed, requiring the monitoring server to recalculate weights for all storage nodes. Subsequently, a clear stabilization is visible, and the throughputs during the latency change range are nearly the same as when there is no artificial latency, demonstrating that NAMI considers latencies and often selects replicas on storage servers with lower latencies. NAMI allows for a performance gain of 26% for 1 workload, increasing to 33% for 10 workloads.

4.4.3 Resource Utilization (Q_2)

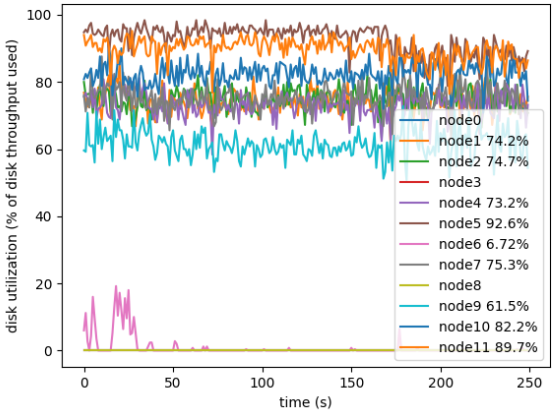
We have observed that NAMI appears to be more or less effective based on the performance illustrated in the previous section. The aim here is to present slightly more granular metrics to explain why NAMI seems so promising. For clarity and space considerations, we will limit our results to the first heuristic, and whenever we refer to *balancer*, it pertains to the first heuristic presented in Section 4.3.



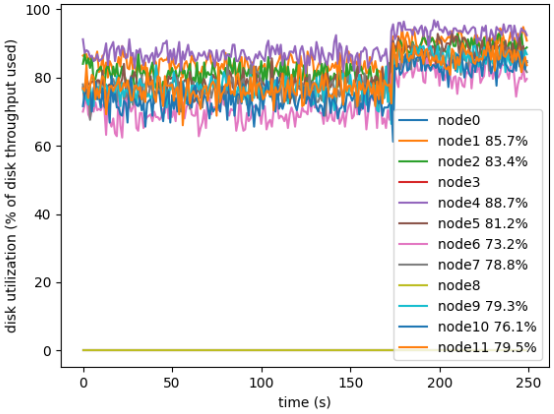
(a) 1 workload with Ceph



(b) 1 workload with Nami



(c) 4 workloads with Ceph



(d) 4 workloads with Nami

Figure 4.11: Evolution of node's disk usage during workloads. The nodes without values are compute nodes; they are not interesting in our studies. The percentage indicates the mean percentage of the node's disk usage over the workload.

4.4.3.1 Disk Utilization

We analyzed and retrieved disk utilization rates using the sar tool [45] during our experiments, and the results are depicted in Figure 4.11.

For 1 workload (Fig 4.11a), which presents disk utilization on the vanilla Ceph system, we observe that disks are inequitably used. When the most used disk is at almost 90% usage, some others are at $20-30\%$, indicating that a few storage nodes are serving most of the requests. With Nami (Fig 4.11b), all disks are more likely to be equally used, allowing them to handle more requests

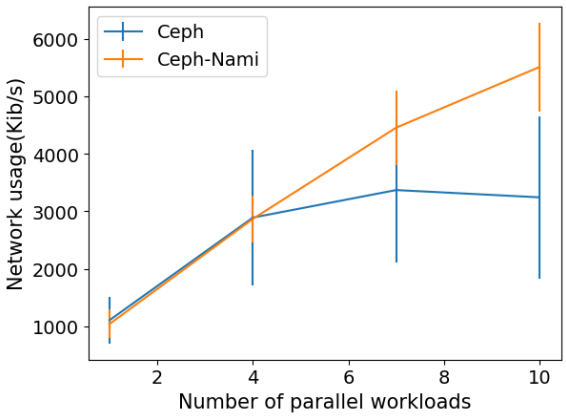


Figure 4.12: Network usage (in **Kib transferred/s**).**Higher is better**

and leading to better I/O throughput.

In summary, we observe a 40% standard deviation of disk usage for all nodes of the storage cluster without NAMI and only 13% standard deviation when using our efficient replica selection approach, demonstrating more equitable disk usage.

4.4.3.2 Network Usage

In addition to disk utilization rates, **sar** also allowed us to retrieve the network activity of our storage nodes, presented in Figure 4.12.

Figure 4.12 illustrates how the average network usage evolves based on the number of workloads. Notably, for 10 workloads, NAMI produces a higher average network usage. This can be explained by the fact that storage nodes without NAMI were not all utilized to transfer data on the network, resulting in a decreasing average with many low values and very few high values involved in the calculation. With NAMI, all nodes are utilized, preventing low values from dragging down the average.

While Figure 4.12 is an error bar, the second observation is the variation measured by the standard deviation of network usage among the storage nodes. With Ceph, the variations are too large proving once again that the network card of some storage nodes is less used because those nodes serve fewer requests, whereas, with NAMI, there are very few variations due to the equal distribution of requests achieved through replica selection.

4.4.4 Replica Selection Effectiveness (Q_3)

In this section, we aim to demonstrate the impact of our replica selection NAMI on I/O requests and the utilization of chosen replicas. We executed the same workloads as before, varying the weight calculation algorithm presented in Section 4.3, and adjusting the replication factor (*rep factor*)—the number of replicas generated by the system when writing new data (2, 3).

Algorithm 1				
	rep factor = 3		rep factor = 4	
# workloads	% replica selected	% failed requests	% replica selected	% failed requests
1	19.91	0.28	14.94	2.44
4	14.88	0.62	14.84	1.52
7	19.71	0.52	14.51	1.2
10	18.08	0.98	15.40	1.13

Algorithm 2				
	rep factor = 3		rep factor = 4	
# workloads	% replica selected	% failed requests	% replica selected	% failed requests
1	11.97	0.87	23.94	0.43
4	7.26	1.57	14.52	0.78
7	6.78	2.01	13.57	1
10	9.1	1.46	18.18	0.73

Table 4.4: Replica selection statistics

Our dynamic replica selection approach appears effective as it manages to choose a significant percentage of replicas to meet the requirements of requests. According to Table 4.4, for a *rep factor* of 3, approximately 20% of requests utilized replicas other than the default primary replicas, and 15% for a *rep factor* of 4. This is explained by the redirection of requests to other replicas, taking into account precomputed weights.

The selection rate of replicas by the second heuristic is even higher (up to 24%). Recall that this heuristic computes weights based on metrics from other client nodes (Section 4.3), offering greater diversity in replica selection.

The low request failure rate ($\approx 1\%$ in Table 4.4) indicates that the selected replicas generally successfully respond to requests. This could be attributed to the relevance of the selection criteria in the weight computing algorithm. In our case, the highlighted criterion is the access latency of storage nodes described in Section 4.3. Therefore, it is conceivable to implement adaptation algorithms allowing chosen replicas to effectively meet requirements.

One last point we wanted to highlight is that the rate of chosen replicas and the request failure rate can also be influenced by the system's *replica factor*. This could be a more or less impactful criterion in the weight computing algorithm we choose to implement. As we observe in our results, the replication factor of 3 grants less failure rate for algorithm 1 whereas it is the rep factor of 4 that grants a lower failure rate for algorithm 2.

4.5 Related Work

Numerous studies have been conducted in the field of load balancing for storage systems, here we talk about reviews on replica management with a particular focus on data migration and replica selection.

Replica Migration Several works [59, 51, 61, 81] have been undertaken to enhance replica management in distributed storage systems. Some aim to balance capacity utilization among storage nodes [59, 51], assuming it will lead to a load usage equilibrium.

Others, like [61, 81], proposed synchronous or asynchronous data migration plans to balance

workload, regardless of capacity balance among storage nodes. It is noteworthy that these works consider different constraints in making migration decisions. For instance, [81] uses replica geolocation to propose migration plans based on the origin of I/O requests, while [51] considers VM disk images and suggests image or block-oriented data migrations. On the other hand, GoSeed[61] acts as a simulator and post- or pre-workload calculator to determine an optimal migration policy based on replica states and access patterns.

Replica Selection In [129], Ye et al. presented a replica selection algorithm for distributed storage systems based on a quorum. Their algorithm aims to select a group of replicas (quorum) to which I/O requests are directed to optimize performance. The algorithm relies on a Monte Carlo simulation considering a list of read/write requests to determine the replica set to use. However, this requires a precise view of the system activity (number of requests incoming), which is not always feasible in diverse applications, such as VM or container-based environments.

Nwe et al. [84] introduced a replica selection algorithm for non-scalable storage systems like HDFS, and MongoDB ([72]). This algorithm only considers latency metrics to select the replica set for improved overall system performance. In contrast, with Nami, we consider various metrics such as resource balance (disk and network) and allow users to adapt the replica selection algorithm by adding other metrics. Moreover, Nami focuses on distributed scalable storage systems, anticipating future increases in data volumes.

In [9], the authors assumed an inherent affinity between data and sought to predict access patterns to select a particular replica set. However, this may be limiting given the diverse range of possible applications in cloud infrastructures, leading to numerous access patterns and the challenge of choosing the best replica for I/O requests.

4.6 Summary

In conclusion, we have proposed a novel architecture designed to optimize replica selection in storage clusters based on constraints defined by the sysadmin to manage efficiently distributed storage systems. Compared to existing replica selection strategies, our approach stands out by providing a more holistic view of system dynamics, accommodating diverse constraints, and offering flexibility in adapting the replica selection algorithm. Furthermore, Nami focuses on scalability, anticipating the future growth of data volumes in storage clusters. We also evaluate the impact of a good weighting of storage nodes when the system varies during workloads. These evaluations and scalability tests help to validate the effectiveness and efficiency of the Nami architecture and show an increase in the standard deviation usage of disks from 40% to 13% and an overall increasing throughput performance of $\approx 30\%$.

CHAPTER 5

Conclusions

Summary

In this dissertation, we have explored the evolving landscape of cloud computing, focusing on the storage virtualization of key technologies such as serverless computing, containers, and virtual machines. The contributions presented in this dissertation addressed critical challenges faced by storage systems in cloud computing environments, offering innovative solutions to improve performance, scalability, and reliability.

Firstly, we introduced OFC, an opportunistic caching system designed to reduce latency to access storage in Function-as-a-service (FaaS) platforms. By leveraging machine learning and overbooked memory resources often done by end-users to set up a cost-effective environment to execute functions, OFC demonstrated significant improvements in function execution times, enhancing the efficiency of FaaS applications.

Secondly, we proposed the Scalable Virtual Disk (SVD) format to address the issue of long snapshot chains generated by cloud providers and users. Through careful design and evaluation, SVD showed promising results in improving performance and memory footprint scalability, offering a practical solution to snapshot management challenges.

Lastly, we presented Nami, a system aimed at addressing the Non-Uniform Disk Access (NUDA) issue as well as resource inequity utilization in distributed storage environments. By leveraging secondary replicas and constraints-based load balancing techniques coupled with system administrators' specific chosen metrics, Nami effectively mitigates performance disparities among storage nodes, enhancing overall system reliability and resource utilization.

These contributions represent significant advancements in the field of cloud computing storage, offering practical solutions to pressing challenges faced by cloud providers and users alike. Through empirical evaluations and real-world simulation deployments, we have demonstrated the effectiveness and scalability of our proposed solutions.

Perspectives

Enhancement of Current Contributions

There are several avenues for enhancing the current contributions to ensure broader adoption by the community for real-world applications.

Improve prototype features The existing implementation of OFC presently only intercepts requests to an Apache OpenSwift distant storage [114]. Expanding its capabilities to dynamically intercept requests to various distant storage platforms, such as AWS S3 [12], could be valuable. Evaluating the feasibility and cost implications of integrating this new functionality would be beneficial. Also, one interesting study will be to look for the feasibility of a unique model for all categories of functions instead of one model for each category. This will help reduce the file size of all the models used while the FaaS platform is running.

A Converter tool Regarding SVD, while we have ensured backward compatibility in its design, enabling users to seamlessly transition from Qcow2 to SVD for existing chains based on Qcow2, remains a priority. Developing a tool capable of facilitating this conversion would significantly enhance our contribution.

Future Directions

Machine Learning to look for relevant metrics While Nami primarily adopts an approach-oriented perspective, addressing the challenge of system administrators lacking clarity on which metrics to prioritize for equitable resource allocation is essential. Developing a module that utilizes Machine Learning to classify these critical metrics could greatly assist system administrators in decision-making processes.

Exploit emerging hardware Talking.gm about SVD, a promising direction involves considering emerging storage hardware, such as NVMe (persistent memory), to further refine its efficiency. Leveraging NVMe allows for optimizing data cluster storing order and I/O request sequencing to maximize overall bandwidth utilization. Additionally, with NVRAM, the need for frequent L1 and L2 cache flushing is minimized, and persistent storage of L1/L2 tables eliminates the overhead associated with snapshot creation in SVD. These advancements promise to significantly enhance the performance of SVD by mitigating unnecessary overhead.

Bibliography

[1] Luca Acquaviva et al. "Cloud distributed file systems: A benchmark of HDFS, Ceph, GlusterFS, and XtremeFS". In: *2018 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2018, pp. 1–6.

[2] Keith Adams and Ole Agesen. "A comparison of software and hardware techniques for x86 virtualization". In: *ACM Sigplan Notices* 41.11 (2006), pp. 2–13.

[3] Alexandru Agache et al. "Firecracker: Lightweight Virtualization for Serverless Applications". In: *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020. URL: https://www.usenix.org/conference/nsdi20/presentation/agache.

[4] Nabeel Akhtar et al. "COSE: Configuring Serverless Functions using Statistical Learning". In: *Proceedings of the 2020 IEEE International Conference on Computer Communications (INFOCOM 2020)*. July 2020.

[5] Istemi Ekin Akkus et al. "SAND: Towards High-Performance Serverless Computing". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 923–935. ISBN: 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/akkus.

[6] Pradeep Ambati et al. "Providing SLOs for Resource-Harvesting VMs in Cloud Platforms". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 735–751. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/ambati.

[7] Apache Foundation. *Apache CloudStack - Storage Overview*. http://docs.cloudstack.apache.org/en/4.15.0.0/adminguide/storage.html. 2022.

[8] Todd Arnold et al. "(how much) does a private WAN improve cloud performance?" In: *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*. IEEE. 2020, pp. 79–88.

[9] Wan Suryani Wan Awang et al. "Affinity Replica Selection in Distributed Systems". In: *International Conference on Parallel Architectures and Compilation Techniques*. 2019. URL: https://api.semanticscholar.org/CorpusID:199022539.

[10] AWS. *Virtualization*. https://aws.amazon.com/what-is/virtualization/.

[11] AWS News Blog. *Provisioned Concurrency for Lambda Functions*. https://aws.amazon.com/fr/blogs/aws/new-provisioned-concurrency-for-lambda-functions/. Accessed: 2021-02-16. Dec. 2019.

[12] AWS News Blog. *Amazon S3 Update – Strong Read-After-Write Consistency*. https://aws.amazon.com/fr/blogs/aws/amazon-s3-update-strong-read-after-write-consistency/. Accessed: 2021-02-16. Dec. 2020.

[13] AWS News Blog. *New for AWS Lambda – Functions with Up to 10 GB of Memory and 6 vCPUs*. https://aws.amazon.com/fr/blogs/aws/new-for-aws-lambda-functions-with-up-to-10-gb-of-memory-and-6-vcpus/. Accessed: 2021-02-16. Dec. 2020.

[14] Paul Barham et al. "Xen and the art of virtualization". In: *ACM SIGOPS operating systems review* 37.5 (2003), pp. 164–177.

[15] Uwe Bauknecht and Tobias Enderle. "An Investigation on Core Network Latency". In: *2020 30th International Telecommunication Networks and Applications Conference (ITNAC)*. 2020, pp. 1–6. DOI: 10.1109/ITNAC50341.2020.9315007.

[16] Alysson Bessani et al. "DepSky: dependable and secure storage in a cloud-of-clouds". In: *Acm transactions on storage (tos)* 9.4 (2013), pp. 1–33.

[17] Kirill Bogdanov et al. "The Nearest Replica Can Be Farther than You Think". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. SoCC '15. Kohala Coast, Hawaii: Association for Computing Machinery, 2015, pp. 16–29. ISBN: 9781450336512. DOI: 10.1145/2806777.2806939. URL: https://doi.org/10.1145/2806777.2806939.

[18] Leo Breiman. "Random Forests". In: *Machine learning* 45.1 (2001), pp. 5–32.

[19] James Cadden et al. "SEUSS: Skip Redundant Paths to Make Serverless Fast". In: *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys'20)*. EuroSys '20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3392698. URL: https://doi.org/10.1145/3342195.3392698.

[20] Canonical. *Ubuntu Cloud Images*. https://cloud-images.ubuntu.com/. 2022.

[21] *Capacity balancer*. https://docs.ceph.com/en/latest/rados/operations/upmap/#upmap.

[22] Joao Carreira et al. "Cirrus: a Serverless Framework for End-to-end ML Workflows". In: *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*. ACM, 2019, pp. 13–24. DOI: 10.1145/3357223.3362711. URL: https://doi.org/10.1145/3357223.3362711.

[23] CentOS Contributors. *CentOS Cloud images*. https://cloud.centos.org/centos/7/images/. 2022.

[24] Ceph. *Distributed Storage*. https://ceph.com/en/.

[25] Debian Contributors. *Debian Official Cloud Images*. https://cloud.debian.org/images/cloud/. 2022.

[26] Eli Cortez et al. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 153–167. ISBN: 9781450350853. DOI: 10.1145/3132747.3132772. URL: https://doi.org/10.1145/3132747.3132772.

[27] Eli Cortez et al. "Resource Central: Understanding and Predicting Workloads for Improved Resource Management in Large Cloud Platforms". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 153–167. ISBN: 9781450350853. DOI: 10.1145/3132747.3132772. URL: https://doi.org/10.1145/3132747.3132772.

[28] Debian Contributors. *Remote work helps Zoom grow 169% in one year, posting $328.2M in Q1 revenue*. https://social.techcrunch.com/2020/06/02/remote-work-helps-zoom-grow-169-in-one-year-posting-328-2m-q1-revenue/. 2021.

[29] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: Association for Computing Machinery, 2014, pp. 127–144. ISBN: 9781450323055. DOI: 10.1145/2541940.2541941. URL: https://doi.org/10.1145/2541940.2541941.

[30] NASA Advanced Supercomputing (NAS) Division. *NAS Parallel Benchmarks*. https://www.nas.nasa.gov/software/npb.html.

[31] Giacinto Donvito, Giovanni Marzulli, and Domenico Diacono. "Testing of several distributed file systems (HDFS, Ceph and GlusterFS) for supporting the HEP experiments analysis". In: *Journal of physics: Conference series*. Vol. 513. 4. IOP Publishing. 2014, p. 042014.

[32] Dong Du et al. "Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS'20. Lausanne, Switzerland: ACM, Mar. 2020, pp. 467–481. DOI: 10.1145/3373376.3378512. URL: https://doi.org/10.1145/3373376.3378512.

[33] Yucong Duan et al. "Everything as a Service (XaaS) on the Cloud: Origins, Current and Future Trends". In: *2015 IEEE 8th International Conference on Cloud Computing*. 2015, pp. 621–628. DOI: 10.1109/CLOUD.2015.88.

[34] Dmitry Duplyakin et al. "The Design and Operation of CloudLab". In: *Proceedings of the USENIX Annual Technical Conference (ATC)*. July 2019, pp. 1–14. URL: https://www.flux.utah.edu/paper/duplyakin-atc19.

[35] etcd. *A distributed, reliable key-value store for the most critical data of a distributed system*. https://etcd.io/.

[36] Facebook. *A persistent key-value store for fast storage environments*. https://rocksdb.org/. 2012.

[37] Henrique Fingler, Amogh Akshintala, and Christopher J. Rossbach. "USETL: Unikernels for Serverless Extract Transform and Load Why Should You Settle for Less?" In: *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys '19. Hangzhou, China: Association for Computing Machinery, 2019, pp. 23–30. ISBN: 9781450368933. DOI: 10.1145/3343737.3343750. URL: https://doi.org/10.1145/3343737.3343750.

[38] *Flexible I/O tester*. https://fio.readthedocs.io/en/latest/fio_doc.html.

[39] Laura Flores. *ceph read balancer*. https://github.com/ljflores/ceph_read_balancer_2023/blob/main/CephaloconReadBalancer2023.pdf.

[40] Sadjad Fouladi et al. "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers". In: *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. Renton, WA: USENIX Association, July 2019, pp. 475–488. ISBN: 978-1-939133-03-8. URL: https://www.usenix.org/conference/atc19/presentation/fouladi.

[41] Alexander Fuerst and Prateek Sharma. "FaasCache: Keeping Serverless Computing Alive With Greedy-Dual Caching". In: *Proceedings of the Twenty-Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. ACM, Apr. 2021.

[42] Yu Gan et al. "Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 19–33. ISBN: 9781450362405. DOI: 10.1145/3297858.3304004. URL: https://doi.org/10.1145/3297858.3304004.

[43] Yixiao Gao et al. "When Cloud Storage Meets RDMA". In: *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 519–533. ISBN: 978-1-939133-21-2. URL: https://www.usenix.org/conference/nsdi21/presentation/gao.

[44] Alberto Garcia. *Qemu - QCOW2 Cache documentation*. https://github.com/qemu/qemu/blob/master/docs/qcow2-cache.txt. 2018.

[45] Sebastien Godard. *sysstat*. https://sysstat.github.io/.

[46] Google Cloud. *Cloud Storage*. https://cloud.google.com/learn/what-is-cloud-storage.

[47] Google Cloud. *Google Cloud Storage Documentation — Consistency*. https://cloud.google.com/storage/docs/consistency. Accessed: 2021-02-16.

[48] Google Cloud. *top cloud computing trends facts statistics 2023*. https://cloud.google.com/blog/transform/top-cloud-computing-trends-facts-statistics-2023?hl=en.

[49] Abel Gordon et al. "ELI: Bare-metal performance for I/O virtualization". In: *ACM SIGPLAN Notices* 47.4 (2012), pp. 411–422.

[50] Johannes Grohmann et al. "Monitorless: Predicting Performance Degradation in Cloud Applications with Machine Learning". In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019, pp. 149–162. ISBN: 9781450370097. DOI: 10.1145/3361525.3361543. URL: https://doi.org/10.1145/3361525.3361543.

[51] Ajay Gulati et al. "BASIL: automated IO load balancing across storage devices". In: *Proceedings of the 8th USENIX Conference on File and Storage Technologies*. FAST'10. San Jose, California: USENIX Association, 2010, p. 13.

[52] Red Hat. *Storage for your future*. https://gluster.org/.

[53] Joseph M. Hellerstein et al. "Serverless Computing: One Step Forward, Two Steps Back". In: *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. 2019. URL: http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

[54] Geoffrey Holmes, Richard Kirkby, and Bernhard Pfahringer. "Stress-testing Hoeffding Trees". In: *European conference on principles of data mining and knowledge discovery*. Springer. 2005, pp. 495–502.

[55] Nikolaus Huber et al. "Evaluating and Modeling Virtualization Performance Overhead for Cloud Environments." In: *CLOSER* 11 (2011), pp. 563–573.

[56] Mc Calpin John D. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. http://www.cs.virginia.edu/stream/.

[57] Eric Jonas et al. "Occupy the Cloud: Distributed Computing for the 99%". In: *Proceedings of the 2017 Symposium on Cloud Computing*. SoCC '17. Santa Clara, California: Association for Computing Machinery, 2017, pp. 445–451. ISBN: 9781450350280. DOI: 10.1145/3127479.3128601. URL: https://doi.org/10.1145/3127479.3128601.

[58] Eric Jonas et al. *Cloud Programming Simplified: A Berkeley View on Serverless Computing*. Tech. rep. UCB/EECS-2019-3. EECS Department, University of California, Berkeley, Feb. 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

[59] Awais Khan, Prince Hamandawana, and Youngjae Kim. "A Content Fingerprint-Based Cluster-Wide Inline Deduplication for Shared-Nothing Storage Systems". In: *IEEE Access* 8 (2020), pp. 209163–209180. DOI: 10.1109/ACCESS.2020.3039056.

[60] Hwanju Kim et al. "Task-aware virtual machine scheduling for I/O performance." In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 2009, pp. 101–110.

[61] Roei Kisous et al. "The what, The from, and The to: The Migration Games in Deduplicated Systems". In: *20th USENIX Conference on File and Storage Technologies (FAST 22)*. Santa Clara, CA: USENIX Association, Feb. 2022, pp. 265–280. ISBN: 978-1-939133-26-7. URL: https://www.usenix.org/conference/fast22/presentation/kisous.

[62] Ana Klimovic et al. "Pocket: Elastic Ephemeral Storage for Serverless Analytics". In: *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 427–444. ISBN: 978-1-939133-08-3. URL: https://www.usenix.org/conference/osdi18/presentation/klimovic.

[63] Sanjay Kumar and Karsten Schwan. "Netchannel: a vmm-level mechanism for continuous, transparentdevice access during vm migration". In: *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 2008, pp. 31–40.

[64] John R Lange et al. "Minimal-overhead virtualization of a large scale supercomputer". In: *ACM SIGPLAN Notices* 46.7 (2011), pp. 169–180.

[65] Duy Le, Hai Huang, and Haining Wang. "Understanding Performance Implications of Nested File Systems in a Virtualized Environment". In: *Proceedings of the 10th USENIX Conference on File and Storage Technologies*. FAST'12. San Jose, CA: USENIX Association, 2012, p. 8.

[66] Collin Lee et al. "Implementing Linearizability at Large Scale and Low Latency". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. Monterey, California: Association for Computing Machinery, 2015, pp. 71–86. ISBN: 9781450338349. DOI: 10.1145/2815400.2815416. URL: https://doi.org/10.1145/2815400.2815416.

[67] Linux. *Block driver*. https://linux-kernel-labs.github.io/refs/heads/master/labs/block_device_drivers.html.

[68] Linux. *traffic control*. https://www.man7.org/linux/man-pages/man8/tc.8.html.

[69] Jay F Lofstead et al. "Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS)". In: *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*. 2008, pp. 15–24.

[70] Martin Maas et al. "Learning-Based Memory Allocation for C++ Server Workloads". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 541–556. ISBN: 9781450371025. DOI: 10.1145/3373376.3378525. URL: https://doi-org.ins2i.bib.cnrs.fr/10.1145/3373376.3378525.

[71] Prince Mahajan et al. "Depot: Cloud storage with minimal trust". In: *ACM Transactions on Computer Systems (TOCS)* 29.4 (2011), pp. 1–38.

[72] Carlos Maltzahn et al. "Ceph as a scalable alternative to the hadoop distributed file system". In: *login: The USENIX Magazine* 35 (2010), pp. 38–49.

[73] Filipe Manco et al. "My VM is Lighter (and Safer) than Your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. ISBN: 9781450350853. DOI: 10.1145/3132747.3132763. URL: https://doi.org/10.1145/3132747.3132763.

[74] Filipe Manco et al. "My VM is Lighter (and Safer) than Your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP '17. Shanghai, China: Association for Computing Machinery, 2017, pp. 218–233. ISBN: 9781450350853. DOI: 10.1145/3132747.3132763. URL: https://doi.org/10.1145/3132747.3132763.

[75] Dutch T Meyer et al. "Parallax: virtual disks for virtual machines". In: *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*. 2008, pp. 41–54.

[76] Microsoft Corporation. *Virtual Hard Disk v2 (VHDX) File Format*. https://winprotocoldoc.blob.core.windows.net/productionwindowsarchives/MS-VHDX/[MS-VHDX].pdf. 2018.

[77] Anup Mohan et al. "Agile Cold Starts for Scalable Serverless". In: *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, July 2019. URL: https://www.usenix.org/conference/hotcloud19/presentation/mohan.

[78] Esteban Molina-Estolano and Carlos Maltzahn. "Dynamic Load Balancing in Ceph". In: 2008. URL: https://api.semanticscholar.org/CorpusID:15534475.

[79] Roberto Morabito, Jimmy Kjällman, and Miika Komu. "Hypervisors vs. lightweight virtualization: a performance comparison". In: *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, pp. 386–393.

[80] Eyal Moscovici and Amit Abir. *How to Handle Globally Distributed QCOW2 Chains*. KVM Forum. 2017.

[81] Amina Mseddi et al. "Efficient Replica Migration Scheme for Distributed Cloud Storage Systems". In: *IEEE Transactions on Cloud Computing* 9.1 (2021), pp. 155–167. DOI: 10.1109/TCC.2018.2858792.

[82] Ingo Müller, Renato Marroquín, and Gustavo Alonso. "Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure". In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. SIGMOD '20. Portland, OR, USA: Association for Computing Machinery, 2020, pp. 115–130. ISBN: 9781450367356. DOI: 10.1145/3318464.3389758. URL: https://doi.org/10.1145/3318464.3389758.

[83] Vlad Nitu et al. "Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor". In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '17. Xi'an, China: Association for Computing Machinery, 2017, pp. 1–14. ISBN: 9781450349482. DOI: 10.1145/3050748.3050758. URL: https://doi.org/10.1145/3050748.3050758.

[84] Thazin Nwe et al. "Consistent Replica Selection Mechanism for Cloud Data Storage". In: 2020. URL: https://api.semanticscholar.org/CorpusID:226862290.

[85] Edward Oakes et al. "SOCK: Rapid Task Provisioning with Serverless-Optimized Containers". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, July 2018, pp. 57–70. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/oakes.

[86] Diego Ongaro, Alan L Cox, and Scott Rixner. "Scheduling I/O in virtual machine monitors". In: *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 2008, pp. 1–10.

[87] OpenStack. *OpenStack Documentation - Get Images*. https://docs.openstack.org/image-guide/obtain-images.html. 2022.

[88] Apache OpenWhisk. *Apache OpenWhisk*. https://openwhisk.apache.org/. Accessed: 2020-06-10.

[89] John Ousterhout et al. "The RAMCloud Storage System". In: *ACM Trans. Comput. Syst.* 33.3 (Aug. 2015). ISSN: 0734-2071. DOI: 10.1145/2806887. URL: https://doi.org/10.1145/2806887.

[90] Pradeep Padala et al. "Performance evaluation of virtualization technologies for server consolidation". In: *HP Labs Tec. Report* 137 (2007).

[91] Loïc Perennou et al. "Workload Characterization for a Non-Hyperscale Public Cloud Platform". In: *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE. 2019, pp. 409–413.

[92] 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)*. Boston, MA: USENIX Association, Feb. 2019, pp. 193–206. ISBN: 978-1-931971-49-2. URL: https://www.usenix.org/conference/nsdi19/presentation/pu.

[93] Krishna PN Puttaswamy, Thyaga Nandagopal, and Murali Kodialam. "Frugal storage for cloud file systems". In: *Proceedings of the 7th ACM european conference on Computer Systems*. 2012, pp. 71–84.

[94] *Python3*. https://www.python.org/.

[95] Qemu. *Features/Qcow3*. https://wiki.qemu.org/Features/Qcow3. 2016.

[96] Qemu Contributors. *QCOW2 Documentation*. https://github.com/qemu/qemu/blob/master/docs/interop/qcow2.txt. 2020.

[97] J Ross Quinlan. "Learning Efficient Classification Procedures and their Application to Chess End Games". In: *Machine learning*. Springer, 1983, pp. 463–482.

[98] J Ross Quinlan. "C4. 5: Programs for Machine Learning". In: (1993).

[99] Allison Randal. "The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers". In: *ACM Comput. Surv.* 53.1 (Feb. 2020). ISSN: 0360-0300. DOI: 10.1145/3365199. URL: https://doi.org/10.1145/3365199.

[100] Red Hat. *Preparing and Uploading Cloud Images with Image Builder*. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/composing_a_customized_rhel_system_image/creating-cloud-images-with-composer_composing-a-customized-rhel-system-image. 2022.

[101] Ran Ribensaft. *What AWS Lambda's Performance Stats Reveal*. https://thenewstack.io/what-aws-lambdas-performance-stats-reveal/. Accessed: 2020-02-03.

[102] Jones Rick. *Hewlett-Packard Netperf Benchmark*. https://github.com/HewlettPackard/netperf.

[103] Josep Sampé et al. "Serverless Data Analytics in the IBM Cloud". In: *Proceedings of the 19th International Middleware Conference Industry*. Middleware '18. Rennes, France: Association for Computing Machinery, 2018, pp. 1–8. ISBN: 9781450360166. DOI: 10.1145/3284028.3284029. URL: https://doi.org/10.1145/3284028.3284029.

[104] SeaweedFS. *Store billions of files, and serve them fast*. https://seaweedfs.github.io/.

[105] EMC Education Services. *Information Storage and Management: Storing, Managing, and Protecting Digital Information*. Wiley, 2010. ISBN: 9780470618332. URL: https://books.google.fr/books?id=sCCfRAj3aCgC.

[106] Mohammad Shahrad et al. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/shahrad.

[107] Mohammad Shahrad et al. "Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider". In: *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*. USA: USENIX Association, 2020. ISBN: 978-1-939133-14-4.

[108] Simon Shillaker and Peter Pietzuch. "Faasm: Lightweight Isolation for Efficient Stateful Serverless Computing". In: *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, July 2020. URL: https://www.usenix.org/conference/atc20/presentation/shillaker.

[109] Arjun Singhvi et al. "Archipelago: A Scalable Low-Latency Serverless Platform". In: *CoRR* abs/1911.09849 (Nov. 2019). arXiv: 1911.09849. URL: https://arxiv.org/abs/1911.09849.

[110] Vikram Sreekanti et al. "Cloudburst: Stateful Functions-as-a-Service". In: *Proc. VLDB Endow.* 13.11 (2020), pp. 2438–2452. URL: http://www.vldb.org/pvldb/vol13/p2438-sreekanti.pdf.

[111] Statista. *Worldwide data created.* https://www.statista.com/statistics/871513/worldwide-data-created/.

[112] Yi Su et al. "An In-Network Replica Selection Framework for Latency-Critical Distributed Data Stores". In: *IEEE Transactions on Cloud Computing* 10.2 (2022), pp. 944–956. DOI: 10.1109/TCC.2020.2976008.

[113] Xiaoyi Sun et al. "Client layer becomes bottleneck: workload analysis of an ultra-large-scale cloud storage system". In: *Proceedings of the 14th IEEE/ACM International Conference on Utility and Cloud Computing Companion.* UCC '21. Leicester, United Kingdom: Association for Computing Machinery, 2022. ISBN: 9781450391634. DOI: 10.1145/3492323.3495625. URL: https://doi.org/10.1145/3492323.3495625.

[114] OpenStack Swift. *OpenStack Swift.* http://swift.openstack.org. Accessed: 2020-06-10.

[115] Chunqiang Tang. "FVD: A High-Performance Virtual Machine Image Format for Cloud." In: *USENIX Annual Technical Conference.* Vol. 2. 2011.

[116] Cheng-Chun Tu et al. "A comprehensive implementation and evaluation of direct interrupt delivery". In: *Acm Sigplan Notices* 50.7 (2015), pp. 1–15.

[117] Nedeljko Vasić et al. "DejaVu: Accelerating Resource Allocation in Virtualized Environments". In: *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems.* ASPLOS XVII. London, England, UK: Association for Computing Machinery, 2012, pp. 423–436. ISBN: 9781450307598. DOI: 10.1145/2150976.2151021. URL: https://doi.org/10.1145/2150976.2151021.

[118] VMware. *Tracking data of virtual disk snapshots using tree data structures.* https://patents.google.com/patent/US10860560B2/en. 2018.

[119] vmware. *VMDK disk format specification.* https://www.vmware.com/app/vmdk/?src=vmdk.

[120] John Paul Walters et al. "A comparison of virtualization technologies for HPC". In: *22nd International Conference on Advanced Information Networking and Applications (aina 2008).* IEEE. 2008, pp. 861–868.

[121] Ao Wang et al. "InfiniCache: Exploiting Ephemeral Serverless Functions to Build a Cost-Effective Memory Cache". In: *18th USENIX Conference on File and Storage Technologies (FAST 20).* Santa Clara, CA: USENIX Association, Feb. 2020, pp. 267–281. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/wang-ao.

[122] Liang Wang et al. "Peeking Behind the Curtains of Serverless Platforms". In: *2018 USENIX Annual Technical Conference (USENIX ATC 18).* Boston, MA: USENIX Association, July 2018, pp. 133–146. ISBN: ISBN 978-1-939133-01-4. URL: https://www.usenix.org/conference/atc18/presentation/wang-liang.

[123] Lavoisier Wapet et al. "Preventing the propagation of a new kind of illegitimate apps". In: *Future Generation Computer Systems* 94 (2019), pp. 368–380. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2018.11.051. URL: https://www.sciencedirect.com/science/article/pii/S0167739X18313505.

[124] Noah Watkins. *OpRequest flow in RADOS OSD server*. https://makedist.com/posts/2014/05/05/oprequest-flow-in-rados-osd-server/.

[125] Michal Wawrzoniak et al. "Boxer: Data Analytics on Network-enabled Serverless Platforms". In: *11th Annual Conference on Innovative Data Systems Research (CIDR 2021)*. Jan. 2021. DOI: 10.3929/ethz-b-000456492.

[126] Sage Weil et al. "Ceph: A scalable, high-performance distributed file system". In: *Proceedings of the 7th Conference on Operating Systems Design and Implementation (OSDI'06)*. 2006, pp. 307–320.

[127] Kan Wu et al. "The Storage Hierarchy is Not a Hierarchy: Optimizing Caching on Modern Storage Devices with Orthus". In: *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 307–323. ISBN: 978-1-939133-20-5. URL: https://www.usenix.org/conference/fast21/presentation/wu-kan.

[128] Lei Ye et al. "Energy-efficient storage in virtual machine environments". In: *Proceedings of the 6th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 2010, pp. 75–84.

[129] Zhen Ye and Weijian Hu. "An Adaptive Replica Selection Algorithm for Quorum based Distributed Storage System". In: *International Journal of Grid and Distributed Computing* 9 (May 2016), pp. 55–68. DOI: 10.14257/ijgdc.2016.9.5.06.

[130] Hiroshi Yoshida. "Storage Virtualization". In: *Encyclopedia of Database Systems*. Ed. by LING LIU and M. TAMER ÖZSU. Boston, MA: Springer US, 2009, pp. 2830–2830. ISBN: 978-0-387-39940-9. DOI: 10.1007/978-0-387-39940-9_1335. URL: https://doi.org/10.1007/978-0-387-39940-9_1335.

[131] Tianyi Yu et al. "Characterizing Serverless Platforms with ServerlessBench". In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '20. Association for Computing Machinery, 2020. DOI: 10.1145/3419111.3421280. URL: https://doi.org/10.1145/3419111.3421280.

[132] Zerto. *Differences between backup and replication*. https://www.zerto.com/resources/a-to-zerto/the-differences-between-backup-and-replication/.

[133] Jie Zheng, Tze Sing Eugene Ng, and Kunwadee Sripanidkulchai. "Workload-aware live storage migration for clouds". In: *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 2011, pp. 133–144.

[134] Ruijin Zhou et al. "An End-to-End Analysis of File System Features on Sparse Virtual Disks". In: *Proceedings of the 28th ACM International Conference on Supercomputing*. ICS '14. Munich, Germany: Association for Computing Machinery, 2014, pp. 231–240. ISBN: 9781450326421. DOI: 10.1145/2597652.2597667. URL: https://doi.org/10.1145/2597652.2597667.