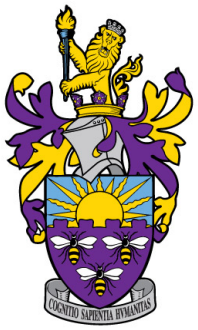

ADAPTIVELY STREAMING QCOW2 DISK IMAGES IN QEMU

A dissertation submitted to The University of Manchester
for the degree of Master of Science
in the Faculty of Science and Engineering

2024

Student ID: 11415150

Department of Computer Science

Contents

Abstract **6**

Declaration **7**

Copyright **8**

Acknowledgements **9**

1 Introduction **11**
1.1 Context . . . 11
1.2 Problem Tackled . . . 11
1.3 Motivation . . . 13
1.4 Scope and Objectives . . . 14
1.5 Dissertation Overview . . . 14
1.6 Chapter Summary . . . 15

2 Background and Literature Review **16**
2.1 Background . . . 16
2.1.1 QEMU . . . 16
2.1.2 I/O Virtualisation . . . 17
2.1.3 QCoW2 Image Format . . . 18
2.1.4 Virtual Disk Snapshots . . . 19
2.1.5 Snapshot Streaming . . . 20
2.2 Related Work . . . 21
2.3 Chapter Summary . . . 25

3 Design and Implementation **26**
3.1 Problem Characterisation . . . 26
3.2 High Level Design . . . 26
3.3 Control Flow of Adaptive Streaming . . . 27
3.4 Development . . . 29
3.4.1 Development Environment . . . 29
3.4.2 System Design Components . . . 29
3.5 Implementation Issues and Challenges . . . 33
3.6 Chapter Summary . . . 35

4 Experimental Evaluation **36**
4.1 Testing Environment . . . 36
4.2 Macro-benchmarks: YCSB . . . 37
4.2.1 Workload Configuration . . . 37
4.2.2 Results . . . 38

4.3 Micro-benchmarks: FIO 40
4.3.1 Workload Configuration 40
4.3.2 Results 40
4.4 Chapter Summary 44

5 Conclusion **45**
5.1 Summary 45
5.2 Limitations 46
5.3 Future Development and Open Research Areas 46

Bibliography **48**

Word Count: 13683

List of Tables

3.1 Development environment specifications. 29

4.1 Configuration of the guest VM used for evaluation. 36
4.2 Configuration of the FIO job file used for evaluation. 40
4.3 Useful streaming runtime ratio (USRR) of adaptive streaming, normal streaming and speed-limited streaming. 44

List of Figures

1.1 Evolution of I/O throughput during snapshot creation and streaming. . . . 12
1.2 Block I/O traces of YCSB RocksDB SSD workload with burst periods. . . 13

2.1 KVM/QEMU virtualisation architecture overview. 16
2.2 The QCoW2 logical format. 18
2.3 Creating a new snapshot of a QCoW2 virtual disk. 19
2.4 Streaming of snapshot layer C to the active layer D. 20

3.1 Execution flow of the adaptive streaming feature. 27
3.2 Snippet of the flame graph of QEMU code execution during streaming. . . 34

4.1 Comparison of average throughput, insert latency and scan latency between adaptive streaming, normal streaming, and no streaming, with YCSB-RocksDB workload. 38
4.2 Comparison of the total runtime and total active time between adaptive streaming and normal Streaming. 39
4.3 Comparison of average bandwidth between adaptive streaming, normal streaming and speed-limited streaming. 41
4.4 Comparison of streaming runtime between adaptive streaming, normal streaming and speed-limited streaming. 41
4.5 Evolution of I/O bandwidth with adaptive streaming after 30 snapshots with bursty FIO workload. 42
4.6 Evolution of I/O bandwidth with normal streaming after 30 snapshots with bursty FIO workload. 42
4.7 Evolution of I/O bandwidth with speed-limited normal streaming after 30 snapshots with bursty FIO workload. 43

Abstract

Adaptively Streaming QCoW2 Disk Images in QEMU
Trilok Bhattacharya
A dissertation submitted to The University of Manchester
for the degree of Master of Science, 2024

The development of virtual disk snapshots have enabled cloud providers and users to accomplish backup and recovery tasks, implement geographical redundancy and high speed live-migration of virtual machines (VMs). Incremental snapshots are built on top of existing image layers, with write operations redirected to the active image and reads redirected to the appropriate image. There is no limit to the number of snapshots that can be created, so the snapshot chain tends to grow quickly and becomes an overhead to the performance of a VM.

QEMU/KVM provides a feature called *snapshot streaming*, which allows users to merge the data across multiple snapshots into a single target image, thus reducing the chain length. However, the current implementation of snapshot streaming is suboptimal and lacking flexibility, causing non-negligible drop in the I/O performance of the VM during the streaming process.

The aim of this dissertation is to present a novel streaming method, called *adaptive streaming*, which is designed to track the application I/O during the streaming process, and adaptively pause and resume streaming by utilising the periods of low I/O activity between I/O bursts.

The proposed method is implemented in the QEMU/KVM for the QCoW2 disk image format, and is evaluated using the YCSB-RocksDB macro-benchmark and compared with the existing streaming implementation, improving the I/O throughput by 8.1%, and average scan latency by 7.14%. It increases the total runtime of streaming job but reduces the time streaming actively affects the performance of the VM by 37%. It also improves the I/O bandwidth on a bursty FIO workload by 31% compared to normal and speed-limited streaming. The usefulness and superiority of this functionality is realised by the improvement in the I/O performance of guest VMs.

Keywords: QEMU, QCoW2, Virtual Disk Snapshots, Snapshot Streaming

Declaration

No portion of the work referred to in this dissertation has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

ii. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

iii. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.library.manchester.ac.uk/about/regulations/`) and in The University's policy on presentation of Theses

Acknowledgements

First and foremost, I would like to express my sincere gratitude for my mentor and supervisor, Dr. Pierre Olivier, for his unwavering guidance, support inspiration, and most importantly, his engagement throughout this project, without which this work would not have been possible.

I would also like to thank my parents, for enabling me to dream this big. Their unwavering love and support is the reason I am where I am today. Special mentions to my friends in Manchester, and back in India, for their encouragement and constant input of humour and joy that kept me going.

Lastly, I thank my dearest friend Anuska, who was there throughout the year during my highs and lows, and has been a constant source of motivation and inspiration.

In the loving memory of my Dida and Thamma,

1. *Introduction*

1.1 Context

In the past few decades, there has been an exponential increase in the generation of data [1], influencing the development of powerful computing units for the purpose of data collection, storage, processing and dissemination, and with it came the overhead of configuring and maintaining such systems. This led to the emergence of cloud technologies, allowing users to create and access managed computing services for data lifecycle purposes. At the core of cloud computing is the concept of virtualisation which enables the subdivision of resources of a modern computer [2].

Numerous virtualisation technologies exist at present, such as QEMU, Xen, VMware Workstation, Microsoft HyperV, VirtualBox and so on [2], [3]. These software programs, termed as **hypervisors** or **virtual machine monitors (VMMs)**, virtualise the underlying hardware (CPU, memory, storage and other devices) to create virtual guests, providing the illusion of smaller **virtual machines (VMs)**. The VMs are encapsulated and isolated from one another, and can run applications independently as on any physical system. **Quick Emulator (QEMU)** is one such open-source virtual machine emulator, which, when used in conjunction with a virtualiser (KVM), becomes a hypervisor and achieves near native performance. It supports a number of image formats for storage disks, such as RAW, QCoW, QCoW2, VDI, VMDK, etc., with the native image format being **QEMU Copy On Write 2 (QCoW2)**. It is managed by the QCoW2 driver, that maps I/O requests from the guest blocks to the host offsets in the virtual disk files.

QEMU packs numerous features for convenience of the users, with one of the key features being the ability to take snapshots of the virtual disks. It allows saving point-in-time states of the virtual machine disk images that can be rolled back at any time, consequently providing backup and recovery capabilities. The snapshots in QEMU are live incremental snapshots, that are built on top of an existing snapshot checkpoints, with new writes redirected to the active snapshot image and reads redirected to the corresponding snapshot layer. Theoretically, there is no limit on the number of snapshot checkpoints that can be created for any VM, and as a result, snapshot chains are often very long and can span thousands in number [4], which quickly becomes an overhead to track and store. It calls for a need to synthesise unwanted snapshots for which QEMU provides a snapshot merging feature called **streaming** (or **flattening**) [5]. This feature merges the unwanted snapshot layers into existing useful layers, thereby reducing the length of the snapshot chain.

1.2 Problem Tackled

Snapshot streaming is a time-consuming, yet important feature of QEMU that synthesises unwanted snapshots in a snapshot chains. A previous study has highlighted that long snapshot chains cause persistent non-negligible memory footprint and I/O performance

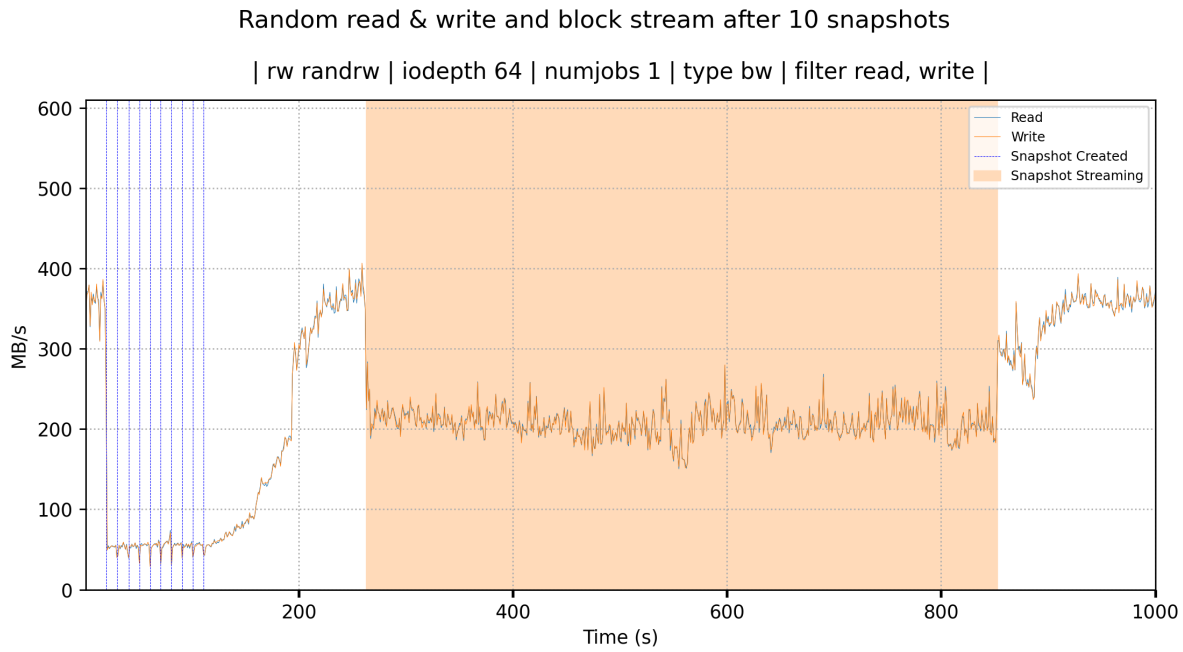

Figure 1.1: Evolution of I/O throughput during snapshot creation and streaming.

scalability issues to the guest VM [4], stressing the importance of streaming as a performance restoring mechanism. Initial experiments depicted in Figure 1.1 have highlighted that the streaming process takes a long time to complete in I/O intensive guest machines, and causes a drop in I/O performance during peak guest activity. It has been observed that the streaming time with even less number of snapshots on I/O intensive systems can reach almost 600 seconds, which is a significant amount of time during which a QEMU internal processes directly impacts the I/O performance on the guest. This has severe implications to the cloud providers, who impose *Service-Level Agreements (SLAs)* on different parameters of the guest, such as IOPS, memory and CPU. Long periods of drop in IOPS or bandwidth can result in financial penalties and loss of reputation, becoming a significant issue to tackle for businesses.

Initial exploratory experiments on FIO benchmarking tool were performed on a test environment with 10 snapshots, and snapshot streaming was initiated. The average I/O bandwidth during the streaming period, as visualised in Figure 1.1, was measured to be 218.63 MB/s, while the average bandwidth after the completion of streaming was measured as 361 MB/s. This is a significant drop of 39.44% during the streaming period, which is a motivation for this study to develop a feature that bridges this performance gap. The analysis of this exploration reveals two present issues with QEMU:

1. The snapshot streaming period, which can span for several minutes, sees a significant drop in I/O bandwidth until completion.

2. The presence of snapshots in a VM seems to cause non-negligible degradation to

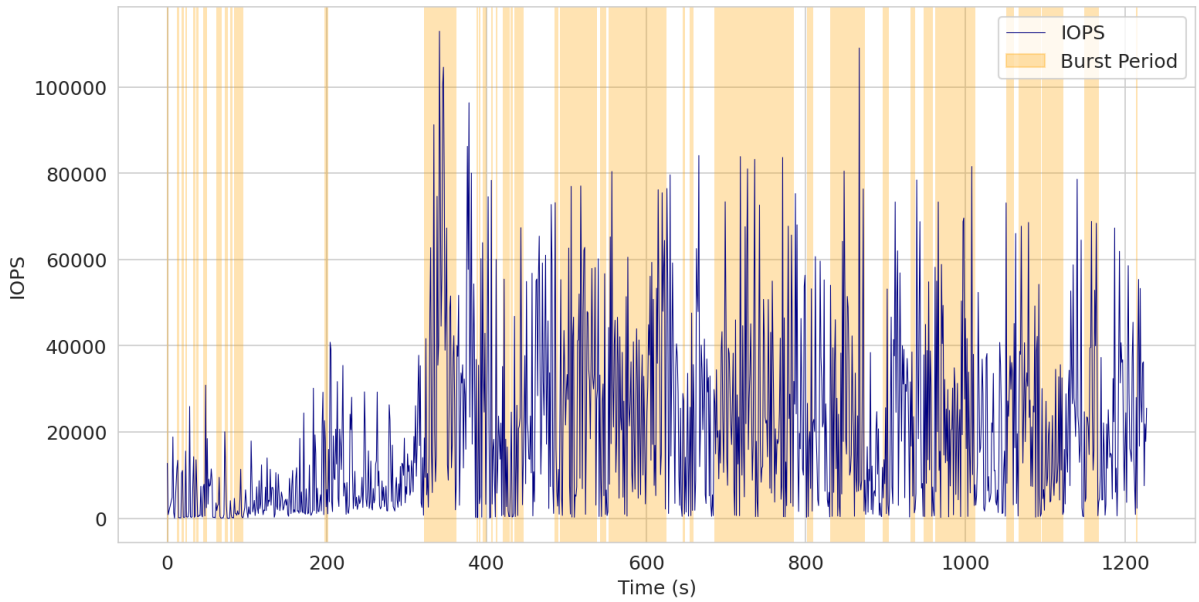

Figure 1.2: Block I/O traces of YCSB RocksDB SSD workload with burst periods.

the read and write I/O bandwidth for an extended period of time.

This study focuses on addressing the first issue of performance degradation during the snapshot streaming process.

1.3 Motivation

It is essential to study the cause and impact of these performance issues due to the wide availability and adoption of the QCoW2 disk format. A previous study [4] has highlighted that most VMs on cloud platforms have a chain size of 30-35, with some more having chains of length 1000 or more. Cloud providers often turn towards snapshot streaming to reduce the length of this chain. Many cloud providers have automatic policies in place to start snapshot streaming after a certain number of snapshots have been created [4]. This would not take into account the I/O activity of the guest VM, and can appear as a sudden drop in performance to the guest VM from the point of view of the user.

In the current implementation of QEMU, snapshot streaming is a slow process, and causes an extended period of intrusion to the I/O (Figure 1.1). Due to the open-source nature and wide-adoption of QEMU in the industry, an improvement to the streaming process will bring forth a better user experience for a wide customer base. Moreover, it will potentially improve the performance of several VMs deployed on cloud platforms that host important business components. The probability of missing SLAs for promised IOPS will reduce for cloud providers and potentially save significant financial penalties.

Storage Networking Industry Association (SNIA) maintains a repository of I/O traces captured from production environments, called the IOTTA repository [6]. These traces are

used to validate the performance of storage systems and benchmarking tools. The traces captured at the block level, and are used to replay the I/O activity on the storage systems to measure the performance of the system under different workloads. To understand the I/O patterns of a production system, a sample of the Block I/O Traces of YCSB RocksDB SSD workload from the study undertaken by Gala Yadgar et al. [7] has been analysed. It can be observed in Figure 1.2 that I/O activity is not uniform and passes through periods of high and low activity. It can be established that the I/O pattern is *bursty* in nature, where an activity burst has been defined as the period of I/O activity significantly higher than the previous I/O period. In this analysis, the burst periods are 20 times higher that the non-burst periods. Due to the established fact that snapshot streaming impacts the I/O on the guest VM, the periods of low I/O activity can be utilised to run streaming with minimal impact to the applications running on the guest. The average burst duration was found to be around 7.86 seconds, with the maximum duration of 48 seconds. The total duration of the trace was 1200 seconds, out of which the total bursty period was 574 seconds. The bursty period accounts for 47.83% of the total duration of the trace, which allows sufficient time for snapshot streaming to finish the streaming process during lower I/O activity periods.

1.4 Scope and Objectives

The research carried out in this dissertation focuses on the issue of drop in I/O performance during the streaming process, and presents a detailed study of I/O virtualisation, the QEMU QCoW2 disk format, and the snapshot streaming capability on these disks. It outlines and implements a novel approach, called **adaptive streaming**, to minimise the intrusiveness of the streaming process by intelligently streaming during periods of low I/O activity to cause minimal impact on the guest during busy periods. The proposed method has been evaluated against the vanilla[1] QEMU implementation to measure the improvement in I/O performance during streaming, using a set of micro- and macro-benchmarks. It compares the performance of adaptive streaming against the traditional methods of normal streaming in terms of metrics like I/O bandwidth, latency, and streaming time, and provides a detailed analysis of the results.

1.5 Dissertation Overview

The remainder of this dissertation is structured as follows:

Chapter 2: Background This chapter provides an in-depth overview of the background terms used in this research, such as the QEMU virtualisation stack, the QCoW2 disk format, and the snapshot streaming feature in QEMU, with a detailed study of the related work in the field of virtual disk formats and disk snapshots.

[1]Vanilla QEMU describes a default version of QEMU without experimental features.

Chapter 3: Design and Implementation This chapter establishes the characterisation of the problem, and discusses the development of the adaptive streaming approach, the system design of the method, and the implementation issues and challenges faced during the development.

Chapter 4: Evaluation This chapter evaluates the adaptive streaming method developed, on a set of production representative micro- and macro-benchmarks of FIO and YCSB-RocksDB respectively, against normal streaming and speed-limited streaming, providing a detailed analysis of the results obtained.

Chapter 5: Conclusion This chapter summarises the work done in this project, with the advantages and disadvantages of the adaptive streaming approach, and outlines the potential future work.

1.6 Chapter Summary

This chapter sets context of the research performed in this dissertation, and the problem tackled, with the motivation behind the study, and the scope and objectives of the research. It describes the need for improving the snapshot streaming process in QEMU due to the performance degradation observed during streaming with the initial experiments, and the observance of bursty I/O patterns in production systems.This chapter has also provided an overview of the dissertation structure, and the chapters that follow.

2. *Background and Literature Review*

2.1 Background

The KVM/QEMU virtualisation stack is depicted in the Figure 2.1, with the guest VM consisting of two snapshot chain of QCoW2 disk format. The focus of this research will be on the live snapshots feature supported by the QCoW2 disk format, and the snapshot streaming operation performed on it.

This section presents a detailed study of the QEMU virtualisation stack, the QCoW2 disk format, and the snapshot streaming feature in QEMU, followed by a review of related work in the domain of virtual disk images and disk snapshotting.

2.1.1 QEMU

QEMU is an open-source application widely used as an emulator, and as a hypervisor when used with a virtualiser like KVM. It's wide adoption in the industry as well as academia can be attributed to 670+ citations of the study done in 2005 by Fabrice Bellard [3], and more than 6000 forks of the project on GitLab and GitHub [8], [9].

QEMU is a user-space application that runs on top of the host operating system providing the necessary tools to create and manage virtual machines (VMs). It can emulate a wide range of hardware devices, such as CPU, memory, storage, network, and other peripherals, to provide a complete virtual environment for the guest VM. As described in Figure 2.1, KVM interacts with the hardware and provides a mapping of virtual CPUs to physical CPUs, sitting at the lowest level of the stack, while as a hypervisor, QEMU complements the capability of KVM by managing the guest processes and emulating the devices associated with the guests. On its own, KVM is not fully functional as a hypervisor, as it lacks the ability to create and manage VMs and the disk images associated with them. QEMU and KVM together allow creating a virtual environment on top of a host system, to provide a near-native performance for the guest VMs, and are widely used in cloud platforms like OpenStack, Proxmox, and oVirt.

As an emulator, QEMU provides user mode emulation for Linux, which can be used to test CPU emulators without starting separate VMs. It can emulate a complete computer

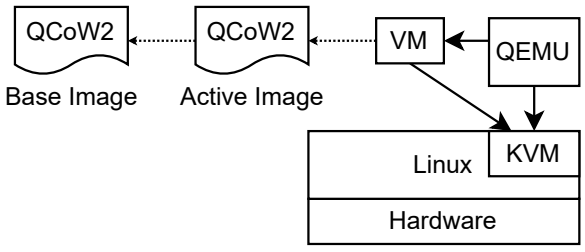

Figure 2.1: KVM/QEMU virtualisation architecture overview.

from the BIOS up to the sound card. It translates the target CPU instructions into the host system's instruction set during runtime, storing the translated binary code in a cache for future reuse. Each instruction from the target CPU is broken down into micro-operations, which are implemented in C code and compiled into object files. A compile-time tool called Dygen generates a dynamic code generator (DCG), which is called at runtime to create host functions that link together multiple micro-operations.

A common use for QEMU as an emulator is for debugging simulated embedded systems. Other alternatives to QEMU exist, such as Oracle VirtualBox, Microsoft Hyper-V and VMware ESXi, but the open-source nature of QEMU, and its simplicity finds its usage in many platforms [10]. This study will focus on QEMU with KVM as a hypervisor, to create virtual machines and manage their disk images.

QEMU provides numerous APIs for creating and launching VMs, which often require complex knowledge of the underlying hardware and software. Paired with the *libvirt* framework, QEMU can be used to manage VMs in a more user-friendly manner. The *libvirt* framework provides a high-level API for managing VMs, storage, and networks, keeping the underlying complexity of QEMU hidden from the user.

While QEMU supports a wide range of disk image formats, such as RAW, QCoW2, VHD, VDI, VMDK, etc., the QCoW2 is the default and the preferred format. Creating a disk image in QEMU is done using the *qemu-img* tool by specifying the name, format, size, and other parameters. Depending on the format, the disk images can be converted to other formats using the same tool.

QEMU is equipped with a complex suite of features catered to disk images, such as virtual disk snapshots, live migration, and block streaming, that are essential for managing VMs in a cloud environment. To perform these operations on the guest, it provides several ways to interact with the VM, such as the QEMU Machine Protocol (QMP), Human Monitor Protocol (HMP), and command line interfaces such as the *qemu-img* tool. In the scope of this research, these tools have been used extensively to create snapshots, and stream them to the active image layer.

2.1.2 I/O Virtualisation

Classic I/O virtualisation is implemented by the hypervisor intercepting I/O requests directed to devices and sending back the response to the appropriate device[11]. The OS interacts with the attached devices using Memory-Mapped I/O (MMIO) or Port-Mapped I/O (PIO) operations, and the I/O devices interact by triggering inputs and reading or writing data from the memory via Direct Memory Access (DMA). The hypervisor intercepts these MMIO and PIO operations and responds to them accordingly by emulating the behaviour of the device. This gives an illusion to that the guest OS is interacting with the physical device. Emulating DMA operations is straightforward, as the hypervisor can read/write from this memory region directly. In context of QEMU/KVM, each VM is started as an isolated process, and separate threads are spawned that represent each I/O device attached to the VM. These threads handle asynchronous activities associated with these devices, such as handling network packets. Most physical devices currently virtualised were not designed with virtualisation in mind, and hence pose significant overheads

in terms of performance. To address this, the hypervisor can provide paravirtualised drivers to the guest OS, which are aware of the virtualised environment and can interact with the hypervisor directly, bypassing the emulation layer. QEMU supports the *virtio* framework, which offers a set of paravirtualised drivers for network, block, and other devices, that can be used to improve the I/O performance of the VM. In this study, the virtio block device (*virtio-blk*) is used to create and launch the VMs, and is also the device framework to implement I/O tracking for adaptive streaming.

2.1.3 QCoW2 Image Format

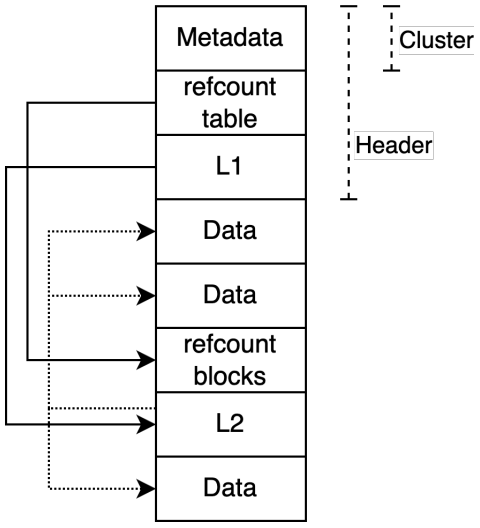

Figure 2.2: The QCoW2 logical format.

The **QEMU Copy-on-write 2 (QCoW2)** image format is one of the representation formats of a block device of fixed length, that is supported by QEMU. It is organized into smaller units of fixed size, called **clusters**, consisting of metadata headers and data payload sections (Figure 2.2). The cluster size of a QCoW2 image is same as the data allocation size of the guest VM. The QCoW2 device driver uses a two-level table structure to manage the allocation of host clusters, by maintaining reference counts for every host cluster. The first-level table (called the **refcount** table) is stored in the header and contains pointers to the second-level table (called the **refcount blocks**), which stores the refcount values for the cluster. A refcount of 0 indicate that the cluster index is free, ≥ 1 indicate that it is used. Additionally, a value ≥ 2 indicate that the writes performed on this cluster index must perform a **copy-on-write** operation.

Similar to the refcount table structure, the mapping of the guest cluster (logical block) and the host cluster (physical block) is also achieved using a two-level table structure, called the L1 and L2 tables. The L1 table is stored in the header and can span multiple contiguous clusters, containing the offset values to L2 tables. The L2 tables on the other hand, occupy one cluster unit of space. They contain offsets to the host cluster locations,

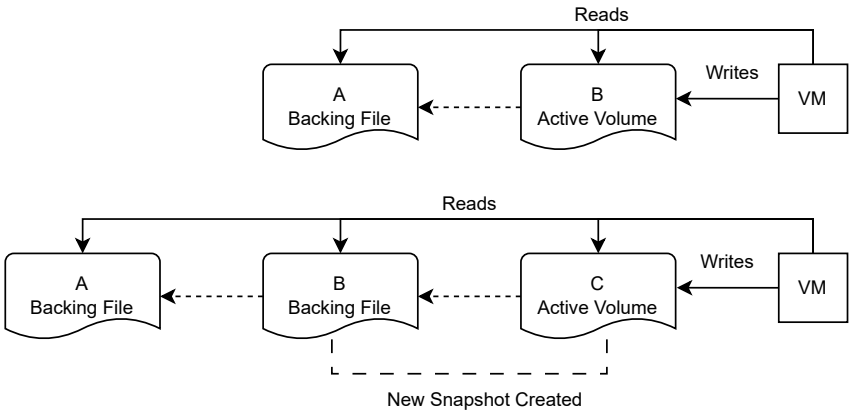

Figure 2.3: Creating a new snapshot of a QCoW2 virtual disk.

contained in the first 62 bits of each 64-bit L2-table entry. The last two bits are used to store the flags, such as the *COW* flag, which indicates that the cluster is copy-on-write. A cluster is unallocated if the cluster offset, represented by bits 9 through 55, is 0. Any read requests to an unallocated cluster is redirected to the backing file, if it exists. Apart from the refcount/refcount block tables and the L1/L2 tables, the header contains other essential metadata, such as the size of the virtual disk, and snapshot related information, like the number of snapshots and the offset in the image table to the snapshot table.

For any virtual disk address, the corresponding cluster in the image file can be located looking two levels into the L1 and L2 tables. Each entry in a L2 table is of size 8 bytes, and since each L2 table is of fixed cluster size, for a default 512-byte cluster size, there are 64 entries in the L2 table. Using the virtual disk offset provided, the indices into the L1 and L2 tables can be calculated using the formula:

$$L1_{index} = \frac{offset}{cluster_{size} \times L2_{entries}} = \frac{offset}{cluster_{size} \times 64} \tag{2.1}$$

$$L2_{index} = \frac{offset}{cluster_{size}} \ \% \ L2_{entries} = \frac{offset}{cluster_{size}} \ \% \ 64 \tag{2.2}$$

For example, if the offset is 1MB, then the L1 index is 32, and the L2 index is 0. The L1 table entry at index 32 will contain the offset to the L2 table, and the L2 table entry at index 0 will contain the offset to the cluster in the image file.

2.1.4 Virtual Disk Snapshots

QCoW2 supports live snapshots, that is, the snapshot of the disk image is taken while the VM is running without a noticeable downtime to the I/O. When a snapshot is taken of a QCoW2 disk on a running VM, a new image file is created with the L1 table copied and incrementing the refcounts of all the L2 tables and the clusters referenced by the L1 table. The new image file is called as the **active volume** (or **image**), while the previous image

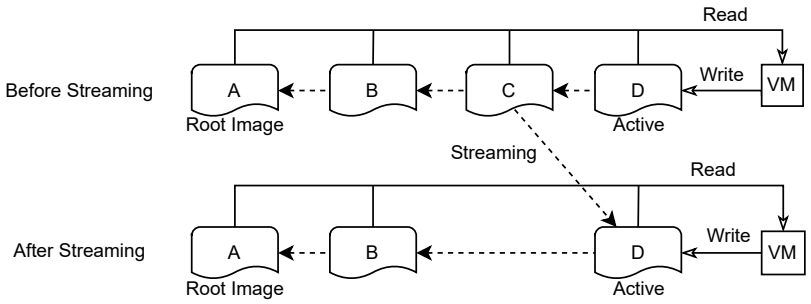

Figure 2.4: Streaming of snapshot layer *C* to the active layer *D*.

file becomes read-only and is called as the **backing file**. During guest I/O, any write request is directed on the active volume. If the cluster to be written exists in a backing file, then the cluster block is copied to the active image, and subsequent read and writes of the cluster are made on the active layer (copy-on-write). Read requests are directed to the backing file which contains the cluster, without copying it to the active image (Figure 2.3). Live snapshot of a QCoW2 disk can be taken using the *snapshot_blkdev* QMP command, by specifying the block device name and the snapshot name. For example, to take a snapshot with name *qcow2-snapshot* of a disk image called *drive0*, the command to the QEMU monitor socket would be:

```
$ echo "snapshot_blkdev drive0 qcow2-snapshot" \
    | sudo socat - UNIX-CONNECT:qemu-monitor-socket
```

Figure 2.3 shows the creation of a snapshot of the active image *B*. A new image file *C* is created, and image *B* becomes the backing file for the new image *C*.

2.1.5 Snapshot Streaming

As the number of snapshots keep increasing, the chain length of the backing image files can grow rapidly. The **block streaming** or **flattening** functionality of QEMU, visualised in Figure 2.4, helps in managing the snapshot chain length by merging unwanted layers. When a streaming process is initiated, a block job is created by QEMU that is responsible for copying the clusters between a specified base image layer and the current active layer while the VM is running with minimal impact to the guest I/O. This makes the active layer self-contained, and the intermediate backing files can be discarded. Once all the intermediate layers are streamed to the active layer, the intermediate image layers can be discarded by the user. However, the streamed intermediate layers are still valid snapshots, and can still be used to revert the VM to a previous state.

Snapshot streaming can be initiated using the *block-stream* QMP command, or *block_stream* HMP command, by specifying the block device name, the target image name, and other optional parameters. In the context of this research, QMP commands are used to start streaming. For example, to stream all the backing images of the device *drive0* to the active layer, the command to the QMP monitor socket would be:

```
$ echo '{"execute": "block-stream", "arguments": {"device": "drive0"}}' \
    | sudo socat - UNIX-CONNECT:qemu-qmp-socket
```

Once block streaming is initiated, a block job is created by QEMU, which is responsible for copying the sectors of the intermediate backing images to the target image. The QMP command for streaming takes an optional parameter *speed*, which limits the rate of streaming. In this research, new parameters for *adaptive-streaming* is introduced, which allows the user to enable the adaptive streaming feature. A detailed explanation of the flow of execution of streaming is provided in the next chapter, along with the implementation details of adaptive streaming.

Other popular hypervisors like VMware ESXi and Microsoft Hyper-V also support an equivalent feature to snapshot streaming. VMware provides the option termed *Consolidation* [12] that can be initiated from the vSphere client, and Microsoft Hyper-V provides the *Merge* option in the Hyper-V Manager, as well as the *Merge-VHD* PowerShell command to merge the snapshots programmatically [13]. Notably, Oracle VM VirtualBox do not provide a VM snapshot merge feature, but their snapshots can be deleted without affecting the VM, as the snapshots are independent to the active volume of the VM [14]. RedHat OpenStack provides snapshot management options as well, but do not provide a merge feature as of the latest documentation [15].

2.2 Related Work

Despite the widespread use of QEMU and the different disk image format, there has been limited research in the area. The study performed by Joshi et al. [16] provides an insightful, although outdated, I/O micro-benchmark comparison of different virtual disk formats, such as RAW, QCoW2, VHD, VDI, VMDK and HDD on the FIO benchmarking tool. The study compares different parameters, such as bandwidth, latency, and I/O operations per second (IOPS), with different block sizes. The authors claim that RAW and QCoW2 fare better than other disk formats with different block sizes in terms of all three parameters, with sequential and random reads. The VHD format has the highest latency with higher block sizes, but perform better with smaller block sizes. However, with sequential and random writes, QCoW2 and RAW fall short in terms of latency and bandwidth, but provide higher IOPS with random writes. These results are essential in understanding the performance of different disk formats, and indicate that QCoW2 provides a good balance between bandwidth, latency, and IOPS, making it the preferred choice for virtual disk images. This study, however, does not discuss the performance of these disk formats with snapshot chains, which is the main focus of this research.

Some researchers [4], [17]–[19] have tackled the I/O bottlenecks in QCoW2 by implementing novel disk formats, that perform optimally in specific circumstances. Researchers at IBM have developed a format called **Fast Virtual Disk (FVD)** [17], which adopts a combination of copy-on-write, copy-on-read and adaptive pre-fetching to improve the performance of the disk, achieving throughput comparable to the RAW disk format. FVD eliminates the overhead of undesirable metadata fetches by maintaining a linear mapping of the virtual block address (VBA) and image block address (IBA), delegating storage

allocations to the operating system. This approach provides a 249% increase in I/O throughput compared to the QCoW2. The research highlights instant VM creation and migration as key features which distinguishes FVD from other disk formats and noticeably outperforming QCoW2 format in terms of performance. However, the implications of FVD on exiting snapshot related operations, such as streaming are not discussed. Although the optimisations of FVD posed it to be a promising replacement of QCoW2, this implementation has now been rendered obsolete due to backward compatibility and adoption issues [4].

In their study, Meyer et al. [18] implement a distributed architecture, called **Parallax**, to facilitate the storage and provision of virtual disk images, by running a storage VM appliance on each physical host of the hypervisor to serve as storage backends. It implements volume snapshots as a software implementation running on a VM, in contrast to using expensive storage hardware, minimising infrastructure cost. Parallax allow crash-consistent snapshots asynchronously into the I/O request stream in a way similar to write barriers, and without pausing I/O operations by allowing I/O issued before and after the snapshot to be completed on their respective views of the disk after the snapshot has been written to storage. This approach allows taking snapshots in quick successions without any delay in the I/O requests resulting in little overhead. The entire snapshot operation in Parallax internally requires three write operations, with two of them being concurrent, making the snapshot process very quick, as has been evaluated in their research. More than 90% of the snapshots created in their test were completed in under a millisecond, but the average snapshot latency was observed to be 10ms. Similar to FVD, the Parallax architecture also involves significant changes to one's storage virtualisation's architecture in migrating existing environments [4], and it further does not support sharing of virtual disk images [4], [18]. The authors do not discuss the possibility of merging unwanted snapshots in the Parallax architecture, which could be a cause of concern in the long run.

Similar research exists with the development of **Petal** [19], that facilitate a set of scalable network-connected servers that act as a block device to provide blocks of virtual disk, attempting to address issues of accessibility, scalability and availability. Features of Petal include fault tolerance and recovery of a single component upon failure, dynamic load-balancing, and performance scalability, which reduce the management efforts of complex storage systems. Evaluation of this proposed solution boasts of a maximum I/O rate of 3150 requests per second and a maximum bandwidth of 43.1 MB/s. The Petal architecture supports rapid snapshot operations, that take 650ms to create, depending on the physical disk size. However, the existence of snapshot merging feature is not discussed in the research.

More recently, the relevant work done by Nguetchouang et al. in the development of **Scalable Virtual Disk (SVD)** [4] address the issue of scalability due to long snapshot chains of QCoW2 files. The authors discuss the existence of long snapshot chains in public cloud, consisting of snapshots created by the users for recovery points, and cloud provider created snapshots for sharing OS base images and distributing snapshot chains across different data centres for load-balancing and capacity reasons. They have experimentally established that long snapshot chains lead to significant memory and throughput scalability issues, observing a 91% decrease in I/O throughput and a 180-times increase

in memory consumption on the *dd* based micro-benchmark with 1000 snapshots. Their proposed solution of SVD attempts to address these issues by extending the snapshot creation process and the QCoW2 driver to access L2-table clusters directly upon an I/O request, irrespective of their position in the snapshot chain, and using a unified L2-table for all backing image files in the chain by copying the backing image's L1 and L2 tables to the active volume. This approach increases the time to create a snapshot by over 10 times, compared to vanilla QEMU, but this trade-off is diminished by improving the I/O throughput and the memory footprint to reach near-peak performance. Their evaluation with both micro- and macro benchmarks favour the SVD implementation, as there is little to no degradation of I/O throughput with micro-benchmarks of dd and fio, and sees a 48% increase in the I/O throughput and 15x decrease in memory consumption with YCSB-RocksDB macro-benchmark with 500 snapshots compared to vanilla QEMU. Their research emphasise the importance of performance improvement during snapshot streaming in the context of cloud environments, and the need for a scalable solution to manage snapshot chains. However, the authors do not discuss the impact of snapshot streaming on the performance of the VM, which is the main focus of attention in this research.

Researchers at VMware Inc. [20] venture into the under-explored area of VMs and storage on low-cost commodity devices such as mobile devices. They present a novel disk image format **Logging Block Store (LBS)** to address the requirements of an enterprise-level VM on low-cost mobile storage devices. Their study highlight a mismatch between the characteristics of the SD Card, the Android guest VM, and the non-sequential I/O mixture of the snapshot checkpoint. LBS format implements a log-based storage structure that helps in bridging this gap, and provide features of security and reliability. The format consists of two separate files, the *.lbsd* file which contains the data part and is stored in the SD card due to its larger size, and the *.lbsm* file which contains the metadata part and is stored in the internal storage of the device. This image format benefits from transforming the random I/O pattern of a VM to large sequential I/O patterns better suited for the SD card, where it sees a 17x improvement over the flat virtual disk format. Although the LBS format supports snapshots through VMware's Mobile Virtual Platform, it has a niche use-case and is not supported by QEMU and other widely used hypervisors.

Jin and Miller in their study [21] discuss the effectiveness of deduplication in virtual disk images, which is a strategy to store duplicate data blocks once, and referencing the identity of the block in the *inode* of each file which requests to store the block. They address the issue of requiring large storage spaces for storing virtual disk image files by using deduplication, instead of using base backing image overlays for building VMs. Their research evaluates the effect that deduplication would have on disk images, and showcased 80% savings in required space on OS and application contained disks. It is important to note that their study does not attempt to integrate deduplication on an existing disk image format on any VMM, rather they merely propose the improvement in storage efficiency that deduplication can bring to virtual disk images, which is left to be implemented by the VMM developers. The authors do not talk about using deduplication within different backing files of a snapshot chain, which could be a potential area of research in the future.

The issue of improving disk I/O performance has been approached in the study performed by Li et al. [22], where they attempt to address it by developing optimisations to reduce extra overheads between the guest and the host storage drivers in the Xen split driver mode. The performance of I/O operations are impacted due to the hypervisor intercepting and validating every interaction between guest and device domains. Their first optimisation that addresses the issue is to promote the Block Frontend Driver higher in the Xen architecture, bypassing generic block and I/O scheduler layer. Their second optimisation reduces the overhead of exchanging I/O data between the guest and the driver domains by decoupling some block pages from the grant table to exclusively address disk I/O operations. Evaluation of these optimisations show an improvement of around 21.5% in read and write operations compared to the default Xen split driver mode. It performs 15.9% better with realistic *http_load* workload and 23.7% better with *AS3AP-IR* workload with 5 VMs running simultaneously. The results obtained from their study are promising, but their research is limited to the Xen hypervisor, and does not compare it with other hypervisors, like QEMU/KVM.

Live snapshots methods are intrusive and can cause long downtime and I/O degradation on guest VMs. This issue is addressed in the studies done by Li et al. [23], [24], where they propose and extend a novel snapshot-efficient image format called **Improved Redirect-on-Write (iROW)**, based on the existing block drivers on QEMU/KVM. They implement a bitmap based indexing method to tackle the expensive query cost of looking up a cluster block from L1/L2 lookup tables in QCoW2, with features such as index-free VM state data, which is stored separately from the disk image data in their approach, and a *free page detection (FPD)* technique to identify and skip saving free pages during snapshot creation. Their approach is evaluated extensively against the QCoW2 format, showcasing a reduction in the time to create snapshots by 17x, time to restore by 35x, and the time to delete by 47x. The VM downtime during snapshot creation can be reduced by 45%, which is a significant improvement to the QCoW2. With a cluster size of 64KB, their method has 10% less performance loss in I/O throughput, while still consuming similar storage space as QCoW2 with sparse file support. Their work significantly improves the I/O performance during snapshot operations, but they do not discuss the impact of this approach on snapshot streaming, which may be explained by the fact that snapshot streaming was introduced in QEMU v1.1.0, and their research was done prior to its release, on QEMU v0.12.5 [25].

More recently, Hao et al. [26] attempt to address the issues around snapshot operations by developing **iConSnap**, which uses the copy-on-write (COW) mechanism to save on-demand memory pages and reduce the VM downtime during snapshot operations. Extending the COW method, they propose a lazy-incremental approach of copying the changed blocks between two successive snapshots only once, which reduces the size of the snapshot images significantly. It tracks the memory pages that are modified since the last snapshot, and only writes it when they are written again during the current snapshot. The dirty pages between two snapshots are written only once, with this method, saving the amount of data stored between snapshots. They also propose a scheduling mechanism to reduce the snapshot duration, and minimise the loss in performance, while parallelising the snapshot process and normal VM execution. Employing an improved compression

algorithm, they were able to reduce the snapshot data size significantly. With these optimisations, iConSnap reduces the snapshot creation time by 73.9%, while also reducing the performance loss by 46.7%, compared to without these optimisations. Notably, they implement an interesting approach of excluding older snapshots for space reclamation, by merging the snapshot data from older snapshots to relatively newer ones, which reduces the storage cost by 89.1%. Although their research is fairly recent, and they employ a snapshot merging technique, they do not present any results on the impact on I/O during the merging process, or compare the proposed technique with existing block streaming methods.

Despite the evident popularity of QEMU and QCoW2, and the extensive research undertaken to improve disk I/O performance, no study discusses the issue of drop in I/O throughput posed by snapshot streaming, which is the main objective of this research, and the breakthrough with **adaptive streaming** will benefit cloud providers and users alike.

2.3 Chapter Summary

This chapter provides a detailed overview of the relevant background information required to understand the terms used in this research problem, the presents a survey of the related work in the domain of virtual disk images and disk snapshotting. It categorises the lack of research in the area of I/O performance during snapshot streaming, which serves as the motivation for this research.

3. *Design and Implementation*

3.1 Problem Characterisation

As described in Section 1.2, initial experiments have highlighted that streaming is a lengthy process, and causes a significant drop in the I/O throughput of the guest during a busy I/O activity period. The disk image was benchmarked using the fio tool, which is a popular benchmarking tool for measuring I/O performance. The tool was configured to run a random read-write workload on the disk image, while 10 snapshots were taken at a regular interval of 10 seconds. The throughput was measured during this period, and once the I/O throughput restored to the normal, snapshot streaming was initiated to merge all 10 snapshots and the throughput measured during streaming. From the results visualised (Figure 1.1), it is clear that the I/O throughput drops significantly during the streaming process. The average I/O bandwidth during streaming was 218.63 MB/s, which is significantly lower than the average I/O throughput of 361 MB/s after the streaming was completed. Due to the fact that streaming is a lengthy process, the I/O throughput remains impacted for a significant period of time, enough to cause a potential violation of the SLA imposed by the cloud provider.

It solidifies the need to develop a solution that can reduce the intrusiveness of the streaming process on the guest I/O, and improve the overall performance of the guest during the streaming process. In this chapter, the *adaptive streaming* feature is presented, to address this performance drop. The solution adapts to the I/O activity of the guest, and streams the snapshots during periods of low I/O activity to cause minimal impact on the guest during busy periods. It is easy for a user to configure and use, and does not require any additional user intervention once initiated.

3.2 High Level Design

Figure 3.1 visualises a high level overview of adaptive streaming, from initiation to completion, with the different components involved in the process. When a user wants to initiate streaming of QCoW2 disks, they can use the *block-stream* QMP command to instruct QEMU to start the streaming process from an optionally specified base backing image to a target image. Optionally, they can also specify the speed at which streaming should be run, image layer filters and the developed adaptive streaming feature. The streaming process initially starts a threshold calculation period for 15 seconds before writing cluster blocks to the target image, and calculates the optimal threshold to consider for adaptively pausing the streaming process by tracking the application I/O of the guest. While the cluster blocks are streamed to the target image layer, an I/O tracker continuously monitors the I/O activity of the active image and pauses streaming for a short period of time if the I/O throughput exceeds the calculated adaptive threshold. I/O tracking is stopped when all the clusters are streamed to the target image, and streaming is completed.

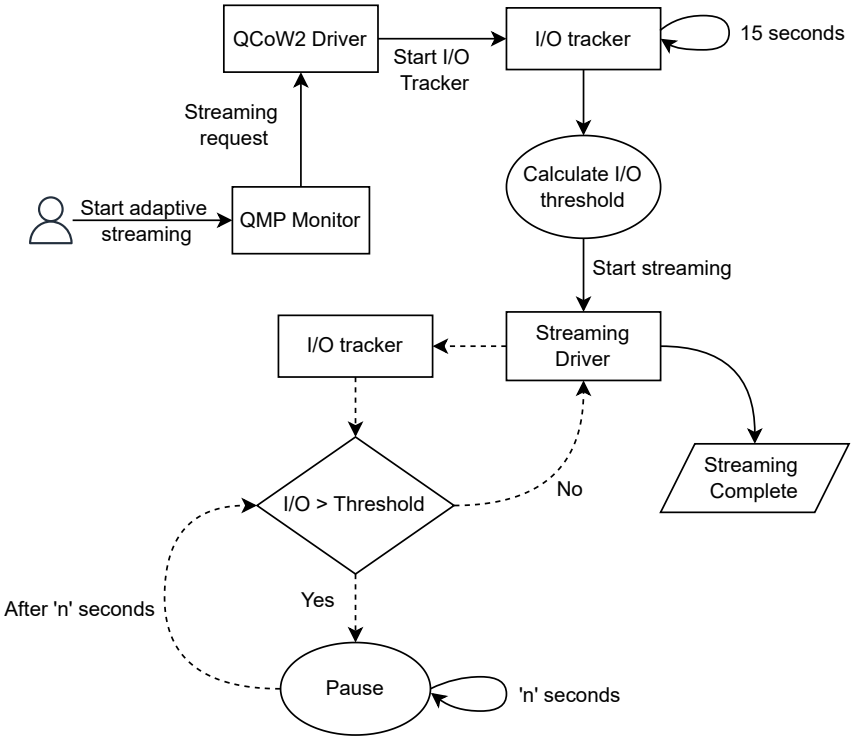

Figure 3.1: Execution flow of the adaptive streaming feature.

3.3 Control Flow of Adaptive Streaming

Initiation

Adaptive streaming can be initiated for a guest VM, using special flags to *block-stream* QMP command, with optional parameters to control the pause time and set explicit threshold values. The streaming request is received by the QMP monitor, which parses the command line parameters received, and accordingly sanitises the input parameters and sets required options for a block job driver. It prepares the request to initiate the streaming job, by initialising the required block driver objects and passing control to the block job driver.

Preparation

The block job driver is responsible for initiating and managing different block device jobs, one of them being block stream. The different phases of the streaming job which are managed by the block job driver are namely, the *prepare* phase, *start* phase, *run* phase and the *clean-up* phase. The *prepare* phase, handled by the *stream_prepare* API in the QEMU stream module, sets the contexts and resources, such as the *BlockDriverState* representing the block device on which streaming will be performed, and ensures that the backing file exists, for the streaming operation. This function also determines the start

and end point for streaming, by calculating the range of clusters that need to be streamed.

After preparing the driver for streaming, a block job is initiated in QEMU, which runs asynchronously and can be managed by a user using the QMP monitor. The *start* phase of streaming, handled by the *stream_start* API, is responsible for creating a new block job for the streaming operation. It creates an object of the *StreamBlockJob*, which is a QEMU internal structure that holds the necessary information about the image overlay, and the adaptive streaming threshold.

Stream

The *run* phase of streaming is handled by the *stream_run* API, called by the job driver when the streaming starts. When adaptive streaming is enabled by the user without an explicit threshold value, the I/O tracking is started in this phase and the active I/O of the block device is monitored for 15 seconds. The average I/O throughput is calculated during this period, and by default 30% of the average throughput is set as the adaptive threshold. This percentage is configurable from the QMP monitor command and can be set by the user.

Once the prerequisites for adaptive streaming are set, the streaming process is started while the I/O activity on the block device is monitored asynchronously. Every time a block of clusters from an intermediate backing file is written to the target layer, the average I/O threshold reported by the I/O tracker is compared with the adaptive threshold, and streaming is paused for a few seconds if the average throughput exceeds the adaptive threshold. The pause time default to 1 second and can be set by the user.

After the pause time has elapsed, the streaming process is resumed and this process is repeated until the entire range of clusters are streamed to the target image layer. In case of persistently high I/O activity on the block device, if streaming is paused for an accumulated time of more than 10 minutes, the streaming process is resumed regardless of the adaptive threshold until completion. This fallback mechanism is developed to ensure that streaming does not get stuck in a paused state indefinitely.

Cleanup

Once the streaming process is complete, the *cleanup* phase is initiated by the job driver, which is handled by the *stream_clean* API. This function is responsible for cleaning up the resources that were allocated for the streaming process and ensures that the target image does not contain references to unnecessary intermediate layers.

The streaming job is then marked as completed, and the job driver exits with a success. The intermediate image layers have been merged into the target and the user can now discard the intermediate layers, and continue using the target image for their VM. The intermediate layers, however, are still valid and can be used to restore the virtual disk to a previous state, if required.

3.4 Development

The development of this adaptive streaming approach followed the iterative software development process, where the problem was triaged, designed, implemented and evaluated in an incremental manner. It established a clear understanding of the problem and the solution, keeping the project structured so that it could be completed within the planned time. The solution was tested after the development was complete, which prevented unnecessary testing cycles of an unfinished solution. However, it was time-consuming to understand the QEMU codebase and to implement and test the solution, as the tests would be lengthy and resource-intensive.

The development effort resulted in more than 600 lines of code added, and around 300 lines modified to the QEMU source code to implement the adaptive streaming feature, and more than 5000 lines of Python and Bash scripts were written for automating the routine tasks, and evaluation tests of the feature.

The following sections describe the development environment, and a detailed architectural schema of the different software components in this feature which interacts with the QEMU block job stack.

3.4.1 Development Environment

The project was developed on a cloud representative Linux server with specifications listed in Table 3.4.1. The QEMU development version 9.0 was compiled from source and run, and the guest VMs created and tested on this system. The development was done using the QEMU source code, which is written in C, and the code was compiled using Makefile through the GCC compiler.

Specification	**Value**
Machine Architecture	x86_64
Operating System	Ubuntu 22.04.3 LTS
Processor	Intel Xeon Gold 5320 CPU @ 2.20GHz
Cores	104
Memory	64 GB
QEMU Version	9.0

Table 3.1: Development environment specifications.

3.4.2 System Design Components

The adaptive streaming feature developed in this project incorporates three independent core modules, which serves different purposes towards realisation of the solution. Addition of new parameters to the *block-stream* QMP monitor command allows the propagation of adaptive streaming settings to the block job drivers. The I/O Tracker module provides

continuous tracking of the guest application I/O. The adaptive pause module helps in determining the optimal I/O threshold to pause, and facilitates the pausing of the streaming job under the suitable conditions.

The functionality of each of these modules are described in detail in the following sections.

QMP Monitor Commands

QEMU facilitates the use of QMP monitor commands, which are JSON-based protocol commands, to interact with a QEMU instance and perform operations on the VMs. It has a rich and comprehensive API that makes it easy to perform administrative and management tasks on the VMs. The QMP monitor was chosen over other interactive protocols to implement the new parameters for adaptive streaming, such as the HMP commands, due to the development simplicity and detailed documentation for QMP monitor.

The **block-stream** command is used to stream the snapshots of a disk image to a target image. The command takes several parameters, such as the base image, target image and the streaming speed. This project adds the following new parameters to the *block-stream* command:

- **adaptive-stream**: A boolean flag to enable adaptive streaming. The default value is false.
- **adaptive-threshold**: It accepts a positive floating point integer. If the value passed is between 0 and 1, then it is assumed to be the ratio for calculating the adaptive I/O threshold at which the streaming process should be paused. If it is set to a value greater than 1, then it is assumed as the explicit threshold to consider for adaptive pause, and threshold calculation is skipped. The default value is 0.3 to set the threshold at 30% of the average I/O throughput.
- **pause-time**: It accepts a positive integer value to set the time in seconds for which the streaming process should be paused when exceeding the adaptive threshold. The default value is 1 to pause for 1 second.

As an example, the following JSON input to the *block-stream* command initiates adaptive streaming with a threshold of 40% and a pause time of 2 seconds, for the block device *drive0* to stream all the image layers between the base image *snapshot0112* and the currently active image layer:

```
{
    "execute": "block-stream",
    "arguments":
    {
        "device": "drive0",
        "base": "snapshot0112",
        "adaptive-stream": true,
        "adaptive-threshold": 0.4,
        "pause-time": 2
    }
}
```

Upon receiving the command, the QMP monitor parses the parameters and sets defaults for the optional parameters not provided by the user. It then delegates the control to the block job driver, which manages the different phases of the streaming process, as detailed in Section 3.2.

The streaming progress can be monitored using the **info block-jobs** QMP command, which provides the progress of the streaming process in terms of the total bytes written. The output of this command was modified during this project to include the status of the streaming process, which is either *paused* or *running*, to provide a clear indication of the current state of the streaming process. This was done to keep a track of the streaming phase and evaluate the performance of the adaptive streaming feature.

I/O Tracker

In order to determine a period of low I/O activity, it is essential to track the application I/O on the guest during streaming. It is crucial to ensure that the I/O tracked is the active I/O on the block device initiated by the guest, and not the I/O activity caused internally by the QEMU instance. This required investigation of the *virtio* block driver backend to understand the execution flow that leads up to an I/O request and find the appropriate API to track the I/O activity.

QEMU implements the structure *BlockDriverState (BDS)*, which represents the state of an image of the block device and facilitates interaction with it. Any I/O request to the block device backend is made in the context of this structure. The `stream_run` API in the streaming module maintains the *BDS* instances of the different image layers, and the I/O tracking is initiated on the active image layer, as any I/O requested on the guest is directed to the active layer first. Write operations are performed directly on the active image, while read operations make their way through 2-level lookup tables in the snapshot chain, to find the appropriate image layer to read from.

In this project, a separate structure `IOPSTracker` is developed to store the read and write I/O requests completed, and an instance of this structure is added as a member to the *BDS* structure to enable I/O tracking for an image layer.

The `IOPSTracker` structure is defined as follows:

```
struct IOPSTracker{
    int64_t rio_size;
    int64_t wio_size;
    int64_t start_time_ns;
    int64_t total_wait_time_ns;
};
```

The `start_time_ns` member stores the time at which the I/O tracking was started, and the `total_wait_time_ns` member keeps a track of the total time for which streaming was paused. The `rio_size` and `wio_size` members store the total read and write I/O size in bytes, since the start of the tracking. Each time the average throughput is calculated by the adaptive streaming module, the `start_time_ns`, `rio_size` and `wio_size` members are re-initialised with 0 to facilitate a new tracking cycle.

The *BDS* structure is modified to include a member of the I/O tracker, along with a mutex lock to ensure thread safety during I/O tracking. The modified *BDS* structure is as follows:

```
struct BlockDriverState
{
    ...
    QemuMutex iops_lock;
    IOPSTracker* iops_tracker;
    bool track_io;
    ...
};
```

The `track_io` member is a boolean flag that is set to true when the I/O tracking has to be started, and false when the tracking needs to stop. The `iops_tracker` member stores the I/O statistics for the image layer, until `track_io` is unset to stop tracking. The `iops_lock` is a QEMU mutex lock, that is used to ensure that the I/O tracker is thread-safe, and only one thread can access the I/O tracker at a time to update the statistics, avoiding race conditions.

The I/O tracker is a crucial core component of adaptive streaming, as it provides the necessary information about the I/O activity on the guest, which is used to calculate the adaptive threshold, and determine the optimal periods to pause the streaming process.

Adaptive Pause

The adaptive pause module is responsible for determining the optimal I/O threshold at which the streaming pauses to minimise the impact on the application I/O. The adaptive pause method is initiated in the streaming process if `track_io` is set in the `BDS` instance of the active image.

During the `run` phase of the streaming process, the I/O activity on the block device is monitored for 15 seconds, before clusters are streamed from the intermediate layers to the target layer. The average throughput is calculated 3 times for 5 second periods during this 15 second window, and the mean of the 3 samples is considered as the threshold for adaptive pause. This decision was made to ensure that the threshold is not calculated based on a single sample, that may not be representative of the I/O activity of the guest

and contain either completely high or low I/O activity.

Upon completion of the threshold calculation, the clusters from the intermediate layers are written to the target layer, while the I/O activity on the active image layer gets monitored through the `iops_tracker` member of the `BDS` instance. Every time a cluster block from an intermediate layer is written to the target layer, the average I/O at that instant is calculated, and streaming is paused for a specified pause time if the throughput goes beyond the adaptive threshold, and the job driver is notified of the change in state. Streaming resumes once the pause time has elapsed, and marks the job as running while updating the job progress to the job driver. This process is repeated until the entire range of clusters are streamed to the target image layer. The logic behind the adaptive pause algorithm can be visualised in the below snippet of code:

```
/* Code before pause */
if (current_throughput > threshold)
{
    job_sleep_ns(&job->job, pause_time*1e9);
    iops_update_wait_time(bs, pause_time);
}
/* Code after pause */
```

The `job_sleep_ns` API is a QEMU internal function that pauses the job for the specified time in nanoseconds and the `iops_update_wait_time` API updates the total time that streaming has been paused due to the adaptive pause algorithm, in the `iops_tracker` member of the `BDS` instance `bs`. The wait time is used to trigger a fallback mechanism to switch to normal streaming if the streaming process is paused for a cumulative time of more than 10 minutes. The streaming process gets resumed and I/O tracking is stopped until the completion of streaming normally, or the user manually stops the streaming process. The fallback mechanism is implemented to ensure that the streaming process does not get stuck in a paused state indefinitely.

3.5 Implementation Issues and Challenges

Due to the complex nature of the implementation, the development of the adaptive streaming feature faced several challenges and issues. The challenges were identified and resolved in an incremental manner, and the final implementation was tested and validated against traditional normal streaming methods on vanilla QEMU. The challenges faced during the development are discussed in the subsequent subsections.

Lack of developer documentation

The QEMU source code which is written in C is vast and complex and lacks detailed developer documentation, which makes it difficult for new developers to understand the codebase. To map the code flow, the *perf* tool was used to trace the execution of the code before and during the streaming process. The output from perf was used to create *flame graphs*, which are a graphical representations control flow stack. It helped visualise the functions that were called during the streaming process, and the time taken by each

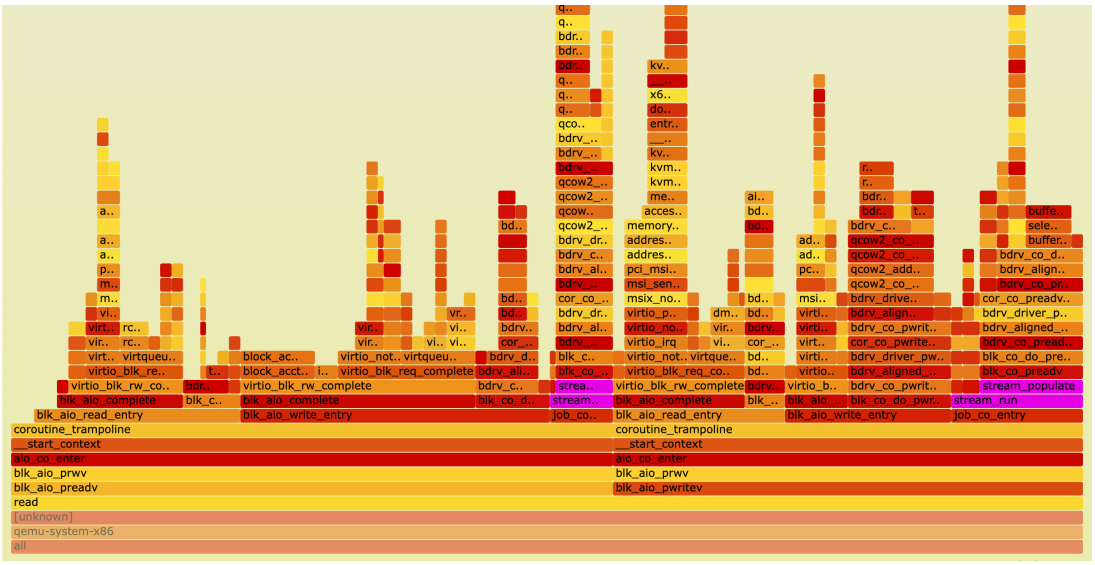

Figure 3.2: Snippet of the flame graph of QEMU code execution during streaming.

function. Initial investigations with flame graphs uncovered that QEMU compiled from source did not have debug symbols enabled, which made it difficult to identify the functions in the flame graph. Due to the lack of up-to-date documentation, it was difficult to figure out the correct parameters to enable debugging, and after several trails of different combinations of compilation flags, the QEMU source code was recompiled with debug symbols enabled, and the flame graphs were generated again to identify the functions that were called during the streaming process. The command to configure QEMU with debug symbols that proved to be useful in this research is highlighted below:

```
../configure --enable-debug-tcg --enable-debug-info --enable-debug \
    --disable-strip --target-list=x86_64-softmmu --enable-kvm \
    --extra-cflags="-g3" --extra-ldflags="-g3"
```

Lack of QEMU native APIs in libvirt

The *libvirt* library provides a set of tools, such as *virsh* and *virt-manager*, for managing virtual machines, and provides APIs to interact with the QEMU hypervisor. The command-line tool *virt-manager* was used to quickly create VMs with minimal operations, and *virsh* to interact with the VMs, such as taking snapshots and initiating streaming. However, soon it became clear that libvirt lacks support for certain QEMU APIs which were required during the development process for this project.

Discussion with a partner cloud provider highlighted the requirement of modifying the L2-cache size of the storage disk, which is not yet supported by virt-manager. Moreover, implementing a new streaming feature required adding flags to the virsh tool, which was not possible due to the risk of working on unknown software. Although the libvirt library is an open-source project, development of adaptive streaming feature was not undertaken in the scope of this project due to time and resource constraints.

These reasons led to the decision of falling back to the QEMU command-line tools

for this project, even though it required complex configurations and was less user-friendly than libvirt.

Job status output missing streaming phase

The progress of the streaming process can be monitored using the QMP monitor commands in QEMU. The output of the *info block-jobs* command provides progress of the streaming process in terms of cluster bytes written. However, it does not provide any information on the phase the streaming process is currently in. It was essential for this project to know if the streaming process is currently paused or running, for the purpose of evaluating the streaming time. This led to the inline modifications of the output of *info block-jobs* to show the status along with the completion progress.

Visualising aggregate results of FIO

Existing visualisation tools used in this research, such as *fio-plot* does not support visualising the aggregate results of multiple fio runs used for evaluation. Moreover, the tool does not support displaying the snapshot creation time, and the streaming time. This required extending fio-plot with custom features to visualise the aggregate results by accepting a text file, to display lines on the graph for snapshot creation, and shaded areas for snapshot streaming.

The fio-plot library also had a bug, which was discovered during the development phase of this project, where the tool was listing the legends twice, and the units in the y-axis were scaled incorrectly. The bug was fixed by modifying the source code, and a pull request was submitted to the fio-plot repository to include the fix in their release [27].

3.6 Chapter Summary

This chapter presented the design and implementation of the adaptive streaming feature, which has been designed to reduce the impact of snapshot streaming on the I/O performance of the VMs. The high-level design of the feature was presented, along with the control flow of adaptive streaming, and the different components involved in the process. The development environment used for the project was described, and the core modules of the adaptive streaming feature, such as the QMP monitor commands, I/O tracker, and adaptive pause, were detailed. The chapter also discussed the challenges faced during the development of the feature, and the solutions implemented to overcome them. The next chapter evaluates the adaptive streaming feature using the YCSB-RocksDB macro-benchmark, and the FIO micro-benchmark, and compares the results with the existing streaming methods in QEMU.

4. *Experimental Evaluation*

The primary goal of the project involves implementing an approach to reduce the impact of the streaming process on the I/O performance of a guest VM. The adaptive streaming feature takes advantage of the guest I/O activity, and streams the cluster blocks from backing files when the I/O activity is low between bursts. The effectiveness of this method has been measured on production representative workloads using industry standard benchmarking tools, and the performance has been compared with the existing QEMU implementations of normal vanilla streaming and speed-limited streaming.

4.1 Testing Environment

The QEMU hypervisor has been compiled and run on the internal server discussed in Section 3.4.1, with the necessary flags to enable debugging. This server is a representative of the practical data centres and cloud infrastructure. An Ubuntu 18.04 VM has been used as the guest VM to perform the experimental evaluation of the adaptive streaming feature. This VM has been configured with the specifications listed in Table 4.1, along with necessary networking components and packages to run the FIO benchmarking tool, the YCSB benchmarking suite, and other dependencies. It is launched in the background using the *daemonize* flag, and redirections to stdout and stderr are done to the file *qemu.log* for debugging purposes. QMP and HMP monitors are configured for use with UNIX sockets, to interact with the VM for capturing snapshots and initiate streaming. The VM can be accessed over SSH on the localhost port 1122.

Component	**Configuration**
vCPUs	4
RAM	4GB
Disk Image	50GB QCoW2
OS	Ubuntu 18.04

Table 4.1: Configuration of the guest VM used for evaluation.

The initial experiments discussed in Section 1.2 and visualised with Figure 1.1 has been run on a VM these specifications, which highlight the issue addressed by the adaptive streaming feature. The same setup has been used to perform evaluation of the developed feature. Subsequent sections discuss the macro- and micro-benchmarks that have been run to evaluate the effectiveness of the feature with comparison to existing QEMU streaming implementations.

4.2 Macro-benchmarks: Yahoo! Cloud Serving Benchmark

Yahoo! Cloud Serving Benchmark (YCSB) [28] is a widely used benchmarking tool to measure performance comparisons of different cloud serving systems. It supports various systems, including key-value stores that are popular in cloud environments. It consists of a workload generating client (workload executor) for a specified system, and a package of workloads that can be run on the system to simulate real-world scenarios. The workloads are defined in a configuration file, and the client generates the load on the system by running the workloads. The system under test is monitored for performance metrics like throughput, latency, and other system-specific metrics.

The YCSB benchmarking suite has been used to evaluate the adaptive streaming feature with the production representative system of the RocksDB key-value database, and the performance metrics have been compared with the existing QEMU implementation of normal streaming, with the performance baseline derived without streaming.

There are two set of operations executed by the workload executor: the *load* phase and the *transaction* phase. The *load* phase is responsible for loading the database with the specified number of records and the *transaction* phase is responsible for executing the specified number of operations on the database. The configuration for running the experiments are defined in two files which specify the workload and the database configurations. The workload configuration file contains the number of records, the number of operations, the proportion of read, write, scan, and insert operations, the distribution of the operations, and other advanced properties. The database configuration file contains the number of fields, the length of the fields, the total record count in the database, and the database directory location, among other options.

4.2.1 Workload Configuration

YCSB has been configured to run transactions on a RocksDB instance, which generates 50000 operations, with 70% of the operations being scans, and 30% being inserts. The scans are uniformly distributed between 100 and 1000 and the request region follows the Zipf[1] distribution. The database is configured to contain 150 fields of 150 bytes each with a total record count of 15000. It is a modification of the default workload E provided by the YCSB benchmarking suite, that simulates threaded conversations, where a scan operation is for posts in a given thread in a conversation.

For the purpose of validating adaptive streaming, the workload executor runs the *load* phase for 15000 records before every snapshot, for 30 snapshots of the QCoW2 disk image to generate the data for the RocksDB database. After the required number of snapshots are created, the *transaction* phase is run and snapshot streaming is initiated. YCSB runs the *transaction* phase until the completion of the streaming process.

[1] A mathematical distribution used to account for a relative popularity of a few members and relative obscurity of other members of the population.

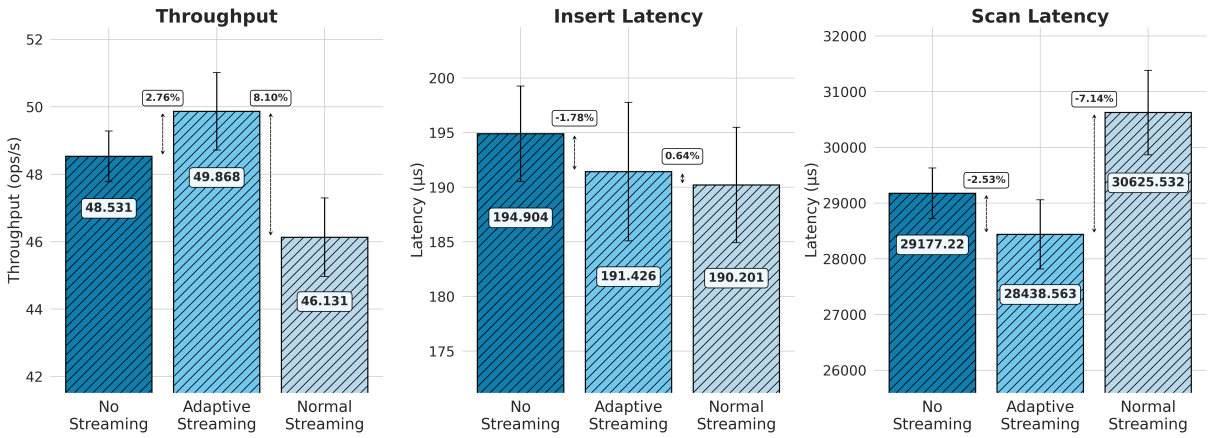

Figure 4.1: Comparison of average throughput, insert latency and scan latency between adaptive streaming, normal streaming, and no streaming, with YCSB-RocksDB workload.

4.2.2 Results

During the YCSB *transaction* phase, adaptive streaming and normal streaming were run on separate occasions and compared in terms of average throughput and average scan and insert latency of the operations. Transaction phase is run for 26500 operations for adaptive streaming and 12500 operations for normal streaming to ensure that YCSB runs long enough for snapshot to finish within the *transcation* phase, and the benchmark values are captured end-to-end for the streaming duration for both the cases. To establish a baseline for comparing the average throughput and latency, YCSB is also run without streaming for 26500 operations with 30 snapshots.

These experiments are run for 15 iterations, and the average performance metrics during the duration of the streaming is captured for the adaptive streaming case, the normal vanilla streaming case, and the no streaming case, and discussed in the following sections.

Adaptive Streaming vs Normal Streaming As depicted in Figure 4.1, adaptive streaming produced an average throughput of 49.868 operations per second (SD=1.149)[2], while normal streaming could produce an average throughput of 46.131 operations per second (SD=1.167), where higher is better. Thus, adaptive streaming produces 8.1% higher throughput compared to normal streaming. It has also improves the average scan latency by 7.14%, with the scan latency of adaptive streaming case being 28438.563 microseconds (SD=620.186), and of normal streaming being 30625.532 microseconds (SD=758.789), where lower is better for latency. The average insert latency does see a slight increase of 0.64% as compared to normal streaming, but is negligible when compared to the improvement in throughput and scan latency. At it's peak, adaptive streaming could produce a throughput of 51.02 operations per second, which is almost 5 more operations per second than normal streaming.

[2]Standard deviation for the average values are depicted in parentheses.

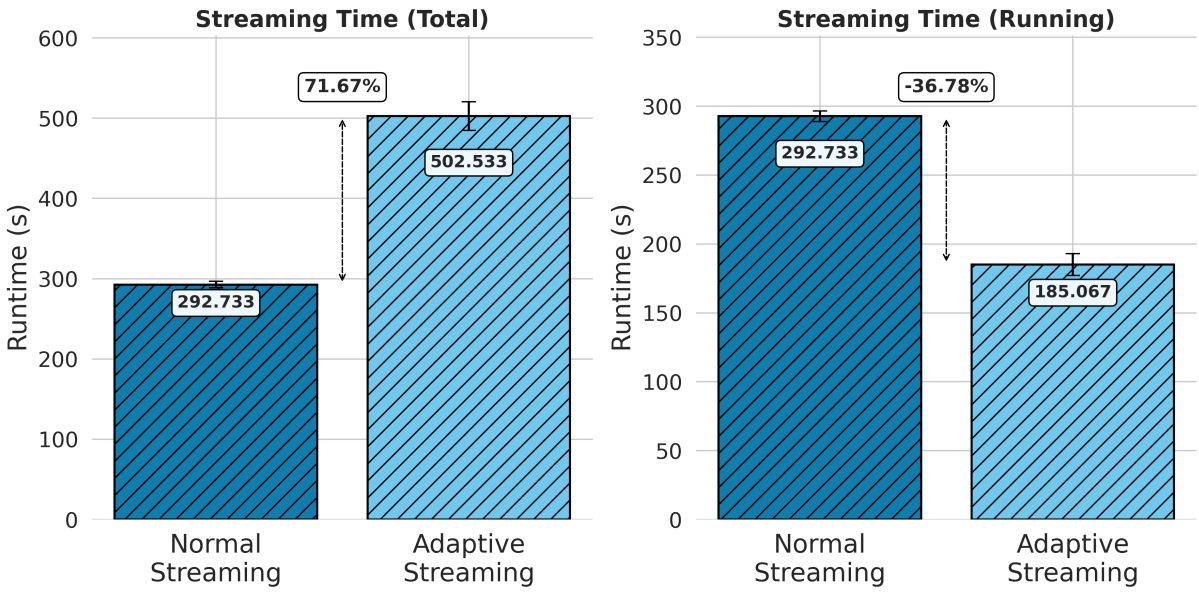

Figure 4.2: Comparison of the total runtime and total active time between adaptive streaming and normal Streaming.

Adaptive streaming does increase the total runtime of the streaming process, as shown in Figure 4.2, with the total runtime increasing by 71.67% with adaptive streaming. However, streaming is idle and paused for most of the time, in this case, and the actual process of writing the clusters to the target image layer happens when streaming is in *running* phase. Using the QMP monitor, the periods during which adaptive streaming was writing the cluster blocks were captured by measuring the streaming status. It can be visualised in Figure 4.2 that the actual time when adaptive streaming has an impact on the application I/O is much lower than that of normal streaming by 35.05%, which is an improvement of 107 seconds on average when streaming does not interfere with the normal execution of the guest. This showcases the effectiveness of the adaptive streaming feature in reducing the impact on I/O throughput of the VM by streaming during the low I/O activity periods.

Adaptive Streaming vs Without Streaming The YCSB-RocksDB benchmark *transaction* was run for the same number of operations with adaptive streaming and without streaming, to compare the performance with the baseline metrics of the guest. It can be visualised in Figure 4.1 that adaptive streaming improves the average throughput of the guest by 2.76% compared to the baseline of 48.531 operations per second (SD=0.751). Adaptive streaming has also improved the average insert and scan latency by 1.73% and 2.58%, respectively, compared to the baseline. Although these numbers are not as significant as the improvement over normal streaming, it highlights the fact that normal streaming severely impacts the application I/O during the streaming process, and adaptive streaming has been successful in bridging the throughput gap with the baseline.

Parameter	Value
Block Size	16 KB
I/O Depth	64
Workload Size	10 GB
I/O Engine	libaio

Table 4.2: Configuration of the FIO job file used for evaluation.

4.3 Micro-benchmarks: Flexible I/O Tester

Flexible I/O Tester (FIO) [29] is a micro-benchmarking tool that is used to simulate I/O workloads to measure the performance of storage systems. It provides the user with different options to configure the benchmarking suite as a job file (or command line options), such as the read/write block size, size of the workload, the type of read and write patterns, and other complex options for granular control of the workload.

4.3.1 Workload Configuration

An FIO job file is configured to run a random read-write workload of non-buffered I/O with properties listed in Table 4.3.1. The parameters signify that the I/O operations of 16 KB size are directly sent to the storage device, and the I/O operations are queued up to 64 operations at a time. The libaio I/O engine is used to perform asynchronous I/O operations. To generate snapshots with varying data, FIO keeps running to generate data between snapshot intervals. In this experiment, 30 snapshots are created at an interval of 15 seconds each, which generates a sufficient amount of data to be streamed and parallels the average number of snapshots observed in cloud environments [4].

The FIO job file is run multiple times with random runtime to mark a high I/O activity period and random sleep timers are introduced between each run to mark periods of low I/O activity, to simulate a bursty workload that represents a production environment. The high and low I/O activity periods are designed to span between 1 and 20 seconds, which is the observed behaviour in production (Figure 1.2). A bash script helps in automating this process, while streaming is started on the QEMU instance and allowed to complete. Separate FIO log files are generated for each FIO run, which is then aggregated, and benchmark numbers are computed with adaptive streaming, normal streaming and speed-limited normal streaming on the same workload.

4.3.2 Results

Different streaming methods were initiated after creating 30 snapshots, while FIO was generating I/O activity on the guest during the streaming process. Adaptive streaming is compared with normal streaming and speed-limited streaming, in terms of the average bandwidth observed, the total runtime of the streaming process, and the total active time of streaming. The comparisons are discussed in the following sections.

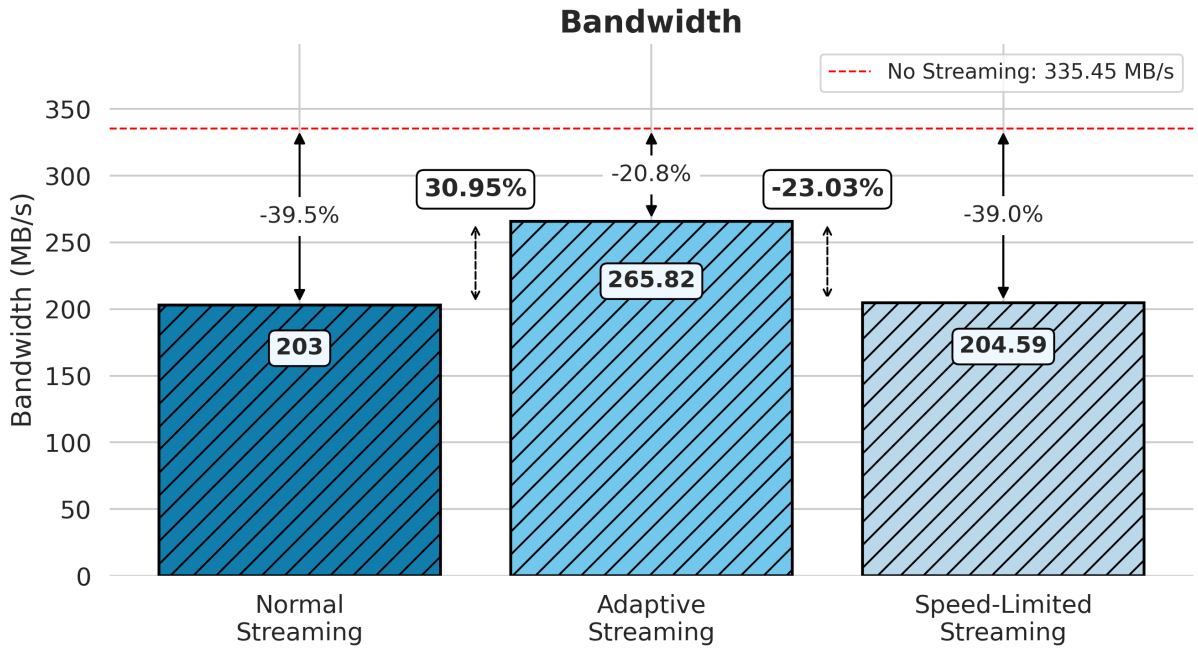

Figure 4.3: Comparison of average bandwidth between adaptive streaming, normal streaming and speed-limited streaming.

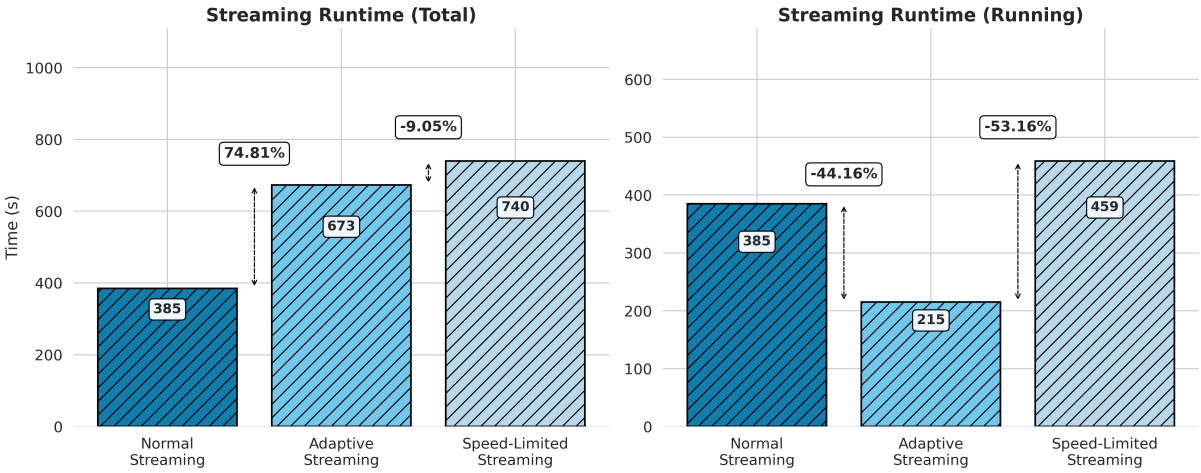

Figure 4.4: Comparison of streaming runtime between adaptive streaming, normal streaming and speed-limited streaming.

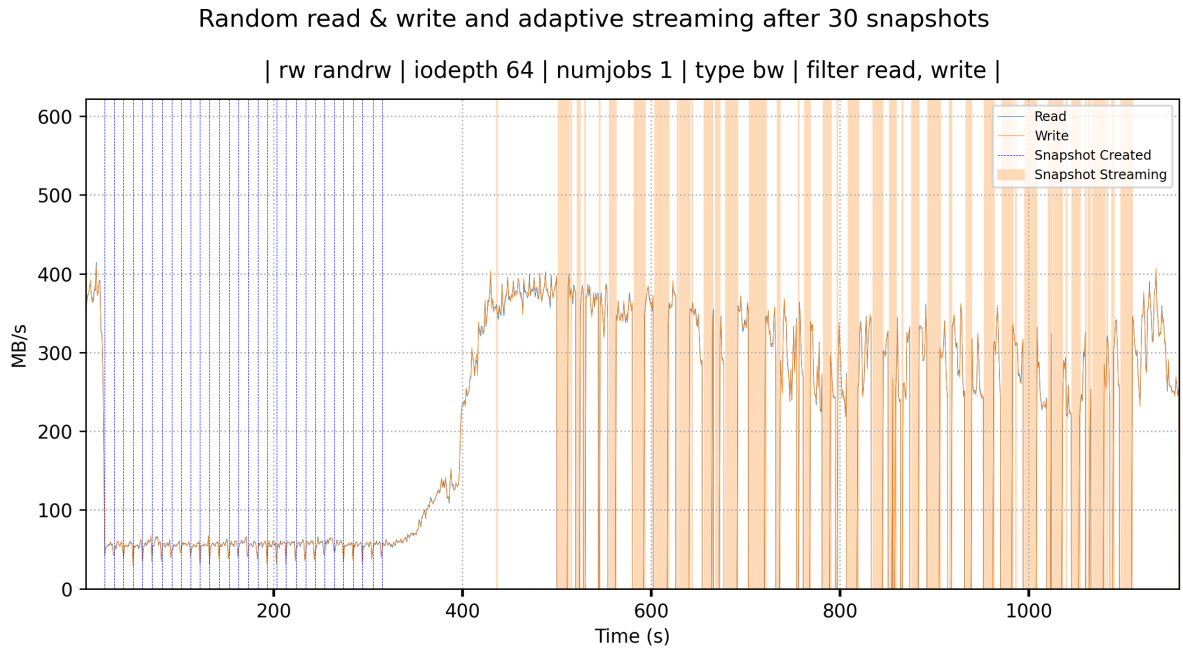

Figure 4.5: Evolution of I/O bandwidth with adaptive streaming after 30 snapshots with bursty FIO workload.

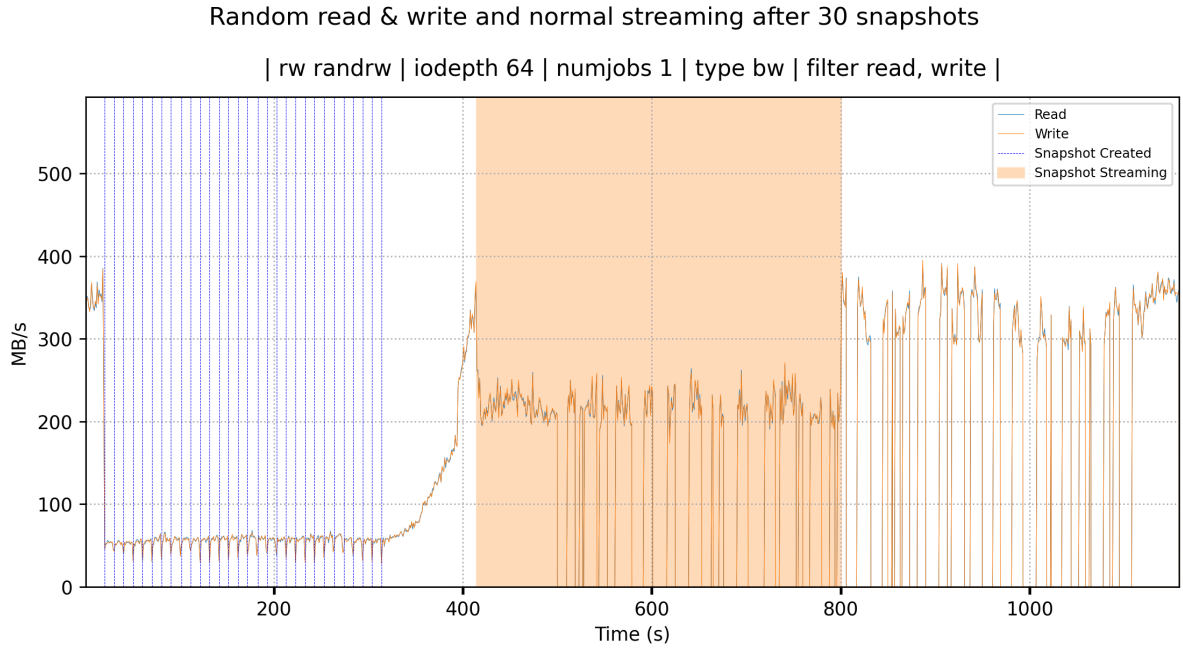

Figure 4.6: Evolution of I/O bandwidth with normal streaming after 30 snapshots with bursty FIO workload.

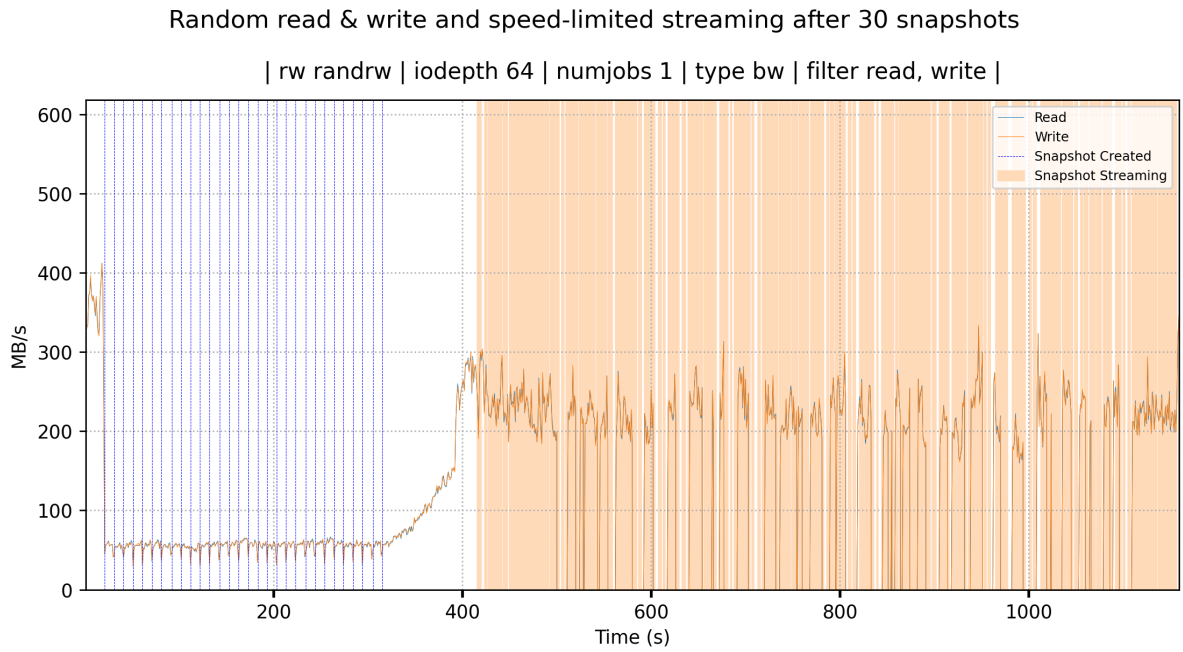

Figure 4.7: Evolution of I/O bandwidth with speed-limited normal streaming after 30 snapshots with bursty FIO workload.

Adaptive Streaming vs Normal Streaming Observing the results captured from the aggregated data in Figure 4.5, streaming is in *running* phase during low I/O activity periods and *paused* during the high I/O activity periods (orange regions denote streaming is *running*, and blue lines represent snapshot being created). The I/O bandwidth of the VM is not impacted during periods of higher I/O activity due to streaming being paused, and I/O bandwidth is maintained at a steady high during the course of the streaming process. In contrast, normal streaming as shown in Figure 4.6 runs for a shorter duration, but the I/O bandwidth of the guest is impacted while its running. Adaptive streaming outperformed normal streaming by 31% in terms of the average bandwidth during streaming, as shown in Figure 4.3. The average bandwidth of adaptive streaming is 265.82 MB/s, while the average for normal streaming was 203 MB/s. Both the streaming methods produced a bandwidth significantly less than the baseline of 335.45 MB/s produced by FIO without streaming. However, normal streaming resulted in a steeper drop of 39.5%, while adaptive streaming dropped the bandwidth by 20.8%. This positions the adaptive streaming method as a better alternative to the existing normal streaming method for systems which run intensive I/O workloads that reach peak I/O activities.

Adaptive Streaming vs Speed-Limited Streaming QEMU block streaming provides an in-built option to throttle the speed of the streaming job using the *speed* parameter in the QMP monitor command, which accepts an integer value representing the speed in bytes per second. Normal streaming with the speed limit of 60 MB/s was configured to

Method	Useful Streaming Runtime Ratio
Normal Streaming	100%
Adaptive Streaming	31.95%
Speed-limited Streaming	62.02%

Table 4.3: Useful streaming runtime ratio (USRR) of adaptive streaming, normal streaming and speed-limited streaming.

run on the same FIO workload as the previous experiment, and the run can be visualised in Figure 4.7. It can be visualised in Figure 4.4 that the speed-limited normal streaming took 67 seconds longer to finish than the total runtime of adaptive streaming, which is almost twice the total runtime of normal streaming without speed throttling. Moreover, Figure 4.3 depicts the average bandwidth of speed-limited streaming was 204.59 MB/s compared to the 265.82 MB/s of adaptive streaming, indicating that adaptive streaming provides 29.93% better bandwidth than speed-limited streaming. Speed-limited streaming performs marginally better than normal streaming, but still drops the bandwidth by 39% compared to the baseline. In this experiment, a new metric *useful streaming runtime ratio (USRR)* is defined as the ratio of the time streaming was in running phase and the total streaming runtime. The *USRR* of three streaming methods are depicted in the Table 4.3.2. It has to be emphasised that the speed-limited streaming impacts the I/O on the guest 62.02% of its entire runtime, which essentially means that it affects the guest applications for almost twice as long as adaptive streaming, while providing negligible improvement in bandwidth compared to normal streaming. On the other hand, not only does adaptive streaming run for a shorter duration of 673 seconds compared to 740 seconds of speed-limited streaming, but also affects the guest I/O for 53.16% less time due to the periods of streaming being paused, providing 23.03% higher bandwidth than speed-limited streaming.

4.4 Chapter Summary

This chapter presented the evaluation of the adaptive streaming feature with the YCSB-RocksDB macro-benchmark, and the FIO micro-benchmark, against parameters such as average throughput, average scan and insert latency, average bandwidth and the total streaming runtime. The adaptive streaming method outperformed the normal streaming methods in these benchmarks by showcasing a higher bandwidth of 8.1% on YCSB-RocksDB, and 37% on FIO. However, it increases the total streaming runtime, which was justified by the reduction in the time streaming actively affects the guest I/O, which has been measured using a metric called *useful streaming runtime ratio (USRR)*, which demonstrated that adaptive streaming affects the guest I/O for barely 32% of the total streaming runtime.

5. *Conclusion*

5.1 Summary

Motivated by the works of some notable researchers in improving the I/O performance of disk image formats in the cloud [4], [17], [23], [24], this research investigated the impact of snapshot streaming on the I/O performance of VMs by running experiments to observe the intrusion of streaming on the VM. It highlighted a sharp decrease in the I/O bandwidth while streaming is running, which is a concern for cloud providers who aim to provide uninterrupted services. The research analysed a sample of production block I/O trace data and commented on the I/O pattern in practice, which is not constant, but rather bursty.

This observation led to the development of a novel streaming method, called *adaptive streaming* on QEMU with the goal in mind to reduce the impact of snapshot streaming on the I/O performance of the VMs. It consists of three main components - QMP commands, I/O tracker, and adaptive pause algorithm. The QMP commands facilitate the initiation of the new adaptive streaming method into the QEMU block job activity, while the I/O tracker is responsible for tracking the application I/O activity on the guest during streaming. The adaptive pause algorithm calculates an *adaptive threshold* to intelligently pause and resume the streaming process. The method builds on the observation that the I/O activity of the application is not constant, and there are periods of low I/O activity between I/O bursts. The feature uses the I/O tracker to track the application I/O activity during streaming, and adaptively pauses and resumes the streaming process during these periods of low I/O activity, using the adaptive pause algorithm, with the intention of reducing the I/O intrusion due to internal processes of QEMU.

The method was implemented in QEMU/KVM for the QCoW2 disk image format, due to its wide availability and adoption, and evaluated using the YCSB-RocksDB and FIO macro- and micro-benchmarks. The evaluated results show that the adaptive streaming method performs better than normal streaming and speed-limited streaming in terms of average I/O throughput, bandwidth and latency during streaming. It reduces the intrusive time of the streaming process, by 35% on YCSB-RocksDB, and provides an average throughput improvement of 8.1% over normal streaming, and 2.76% over no streaming during the same period, which is a significant improvement in cloud environments where the VMs are expected to provide consistent performance. With the FIO benchmark, the proposed method shows an increased bandwidth of 31% during streaming time, compared to normal streaming and 29% compared to speed-limited steaming method, suitable for I/O intensive systems. It also outperforms both the method by in terms of the time streaming actively affects the VM, defined as the *USRR*, by 43.16% and 54.13% compared to normal and speed-limited streaming, respectively. Although all three methods are far from the baseline bandwidth of 335.45 MB/s without streaming, the adaptive streaming bridges the gap further than the other two methods, being 20.8% short of the baseline, compared to 39% for the other two methods.

These results position adaptive streaming as a superior alternative to the normal

streaming methods, and its adoption can help cloud providers to provide better performance guarantees to their users, and also improve the streaming experience.

The development repository for the adaptive streaming feature can be found on GitHub at - `https://github.com/triii10/qemu-adaptive-streaming`

5.2 Limitations

The adaptive streaming feature has been identified to work efficiently if the workload has a bursty I/O pattern, that is, it goes from a period of relatively higher I/O activity to similar or lower I/O activity. It may not be efficient in the following cases:

1. The I/O activity of the guest is a steady constant and does not change during the streaming process. The adaptive pause algorithm would not be able to stream the cluster blocks due to the I/O activity not dropping below the threshold.

2. The streaming process starts with a lower I/O activity period during the threshold calculation part, and increases when streaming is writing cluster blocks to the target image. The threshold calculated during the initial phase may not be optimal for the entire duration of the streaming process.

At present, the fallback mechanism to handle these cases is to use the normal streaming method, which kicks in if the cumulative paused time is 10 minutes. An approach to recalibrate the threshold during the streaming process can be implemented to address this limitation, and has been left as a future work.

5.3 Future Development and Open Research Areas

During the development course of this research, several areas of improvement were identified but could not be implemented due to time and resource constraints. The following subsections highlight some of the areas that can be explored in the future.

Adaptive Streaming for other Disk Image Formats

The adaptive streaming method was implemented with the QCoW2 disk image format in view due to its popularity and the availability of the snapshot streaming feature. This method can be extended to other readily available disk formats such as the VDI, VMDK, VHD, etc., on QEMU, as well as other hypervisors, which support snapshot streaming or equivalent features. Research performed in this dissertation indicate a clear improvement in the I/O performance of the VMs during streaming which can project similar results on other formats.

Recalibration of the Adaptive Threshold

The threshold calculation algorithm can be extended to perform a recalibration if the current threshold is not optimal, and adjusting the threshold based on the relatively recent I/O activity of the guest. This prevents the use of a fixed threshold value based on the I/O activity sampled at the start of the streaming process, which may not be optimal for the entire duration.

Research on the optimal threshold ratio

The adaptive threshold ratio was set to 30% to perform evaluation in this research. However, research can be undertaken to determine an optimal threshold ratio that can provide better performance with adaptive streaming.

Investigating drop in I/O threshold after snapshot creation

During the initial investigation phase of this research, it was discovered that the I/O throughput of the VM drops significantly after the creation of a snapshot, and remains low for a non-negligible amount of time. This can be investigated further to determine the cause of the drop in I/O throughput and suggesting methods to reduce the impact of the drop, or reduce the time to recover from the drop.

Bibliography

[1] J. Tang, T. Ma, and Q. Luo, "Trends Prediction of Big Data: A Case Study based on Fusion Data," *Procedia Computer Science*, 2019 International Conference on Identification, Information and Knowledge in the Internet of Things, vol. 174, pp. 181–190, Jan. 1, 2020, ISSN: 1877-0509. DOI: 10.1016/j.procs.2020.06.073.

[2] P. Barham, B. Dragovic, K. Fraser, *et al.*, "Xen and the art of virtualization," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, Bolton Landing NY USA: ACM, Oct. 19, 2003, pp. 164–177, ISBN: 978-1-58113-757-6. DOI: 10.1145/945445.945462.

[3] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," presented at the 2005 USENIX Annual Technical Conference (USENIX ATC 05), 2005.

[4] K. Nguetchouang, S. Bitchebe, T. Dubuc, *et al.*, "SVD: A Scalable Virtual Machine Disk Format," *IEEE Transactions on Cloud Computing*, no. 01, pp. 1–13, Apr. 1, 2024, ISSN: 2168-7161. DOI: 10.1109/TCC.2024.3391390.

[5] "QEMU Snapshots." (Jul. 1, 2024), [Online]. Available: https://wiki.qemu.org/Features/Snapshots (visited on 09/05/2024).

[6] "Snia - iotta repository." (Aug. 14, 2024), [Online]. Available: https://iotta.snia.org/traces/block-io/28568 (visited on 08/14/2024).

[7] G. Yadgar, M. O. S. H. E. Gabel, S. Jaffer, and B. Schroeder, "SSD-based Workload Characteristics and Their Performance Implications," *ACM Trans. Storage*, vol. 17, no. 1, 8:1–8:26, Jan. 8, 2021, ISSN: 1553-3077. DOI: 10.1145/3423137.

[8] "QEMU / QEMU · GitLab," GitLab. (Aug. 28, 2024), [Online]. Available: https://gitlab.com/qemu-project/qemu (visited on 08/28/2024).

[9] "Qemu/qemu," QEMU. (Aug. 28, 2024), [Online]. Available: https://github.com/qemu/qemu (visited on 08/28/2024).

[10] D. Bartholomew, "QEMU: A multihost, multitarget emulator," *Linux J.*, vol. 2006, no. 145, p. 3, May 1, 2006, ISSN: 1075-3583.

[11] E. Bugnion, J. Nieh, and D. Tsafrir, *Hardware and Software Support for Virtualization* (Synthesis Lectures on Computer Architecture). Cham: Springer International Publishing, 2017, ISBN: 978-3-031-00625-8 978-3-031-01753-7. DOI: 10.1007/978-3-031-01753-7.

[12] "Consolidate Snapshots." (Sep. 5, 2024), [Online]. Available: https://docs.vmware.com/en/VMware-vSphere/8.0/vsphere-vm-administration/GUID-2F4A6D8B-33FF-4C6B-9B02-C984D151F0D5.html (visited on 09/05/2024).

[13] Deland-Han. "How to merge checkpoints that have multiple differencing disks - Windows Server." (Dec. 26, 2023), [Online]. Available: https://learn.microsoft.com/en-us/troubleshoot/windows-server/virtualization/merge-checkpoints-with-many-differencing-disks (visited on 09/05/2024).

[14] "Oracle VM VirtualBox User Manual." (Sep. 5, 2024), [Online]. Available: `https://docs.oracle.com/en/virtualization/virtualbox/6.0/user/snapshots.html` (visited on 09/05/2024).

[15] "OpenStack Docs: Snapshot." (Sep. 5, 2024), [Online]. Available: `https://docs.openstack.org/python-openstackclient/pike/cli/command-objects/snapshot.html` (visited on 09/05/2024).

[16] G. Joshi, S. Shingade, and M. Shirole, "Empirical study of virtual disks performance with KVM on DAS," in *2014 International Conference on Advances in Engineering & Technology Research (ICAETR - 2014)*, Unnao, India: IEEE, Aug. 2014, pp. 1–8, ISBN: 978-1-4799-6393-5 978-1-4799-6392-8. DOI: `10.1109/ICAETR.2014.7012890`.

[17] T. Chunqiang, "FVD: A high-performance virtual machine image format for cloud," in *USENIXATC'11: Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX Association, Jun. 15, 2011, p. 18.

[18] D. T. Meyer, G. Aggarwal, B. Cully, *et al.*, "Parallax: Virtual disks for virtual machines," in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Glasgow Scotland UK: ACM, Apr. 2008, pp. 41–54, ISBN: 978-1-60558-013-5. DOI: `10.1145/1352592.1352598`.

[19] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," *SIGPLAN Not.*, vol. 31, no. 9, pp. 84–92, Sep. 1996, ISSN: 0362-1340, 1558-1160. DOI: `10.1145/248209.237157`.

[20] H. Tuch, C. Laplace, K. C. Barr, and B. Wu, "Block storage virtualization with commodity secure digital cards," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, London England, UK: ACM, Mar. 3, 2012, pp. 191–202, ISBN: 978-1-4503-1176-2. DOI: `10.1145/2151024.2151050`.

[21] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, ser. SYSTOR '09, New York, NY, USA: Association for Computing Machinery, May 4, 2009, pp. 1–12, ISBN: 978-1-60558-623-6. DOI: `10.1145/1534530.1534540`.

[22] D. Li, H. Jin, X. Liao, Y. Zhang, and B. Zhou, "Improving disk I/O performance in a virtualized system," *Journal of Computer and System Sciences*, vol. 79, no. 2, pp. 187–200, Mar. 2013, ISSN: 00220000. DOI: `10.1016/j.jcss.2012.05.003`.

[23] J. Li, H. Liu, L. Cui, B. Li, and T. Wo, "iROW: An Efficient Live Snapshot System for Virtual Machine Disk," in *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, Dec. 2012, pp. 376–383. DOI: `10.1109/ICPADS.2012.59`.

[24] J. Li, Y. Zhang, J. Zheng, H. Liu, B. Li, and J. Huai, "Towards an efficient snapshot approach for virtual machines in clouds," *Information Sciences*, vol. 379, pp. 3–22, Feb. 2017, ISSN: 00200255. DOI: `10.1016/j.ins.2016.08.008`.

[25] "ChangeLog/1.1 - QEMU." (Sep. 1, 2024), [Online]. Available: `https://wiki.qemu.org/ChangeLog/1.1` (visited on 09/01/2024).

[26] Z. Hao, W. Wang, L. Cui, X. Yun, and Z. Ding, "iConSnap: An Incremental Continuous Snapshots System for Virtual Machines," *IEEE Trans. Serv. Comput.*, vol. 15, no. 1, pp. 539–550, Jan. 1, 2022, ISSN: 1939-1374, 2372-0204. DOI: 10.1109/TSC.2019.2955700.

[27] "Fixing same data added twice when scaling bandwidth logs #148." (Feb. 9, 2024), [Online]. Available: https://github.com/louwrentius/fio-plot/pull/148 (visited on 09/05/2024).

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10, New York, NY, USA: Association for Computing Machinery, Jun. 10, 2010, pp. 143–154, ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152.

[29] "1. fio - Flexible I/O tester rev. 3.36 — fio 3.36 documentation." (May 9, 2024), [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html (visited on 05/09/2024).