ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

DirectDrive: Simulating IO operations in an attached storage network

Practical Work

Pasquale Jordan

January 31, 2024

Advisor: Tommaso Bonato; Professor: Torsten Hoefler
Department of Computer Science, ETH Zürich

Abstract

This report presents a comprehensive framework for simulating I/O operations within Microsoft Azure's DirectDrive using advanced simulation tools such as htSim and LogGOPSim. The study employs the UMass Trace File Dataset to create a realistic emulation environment that translates real-world I/O operations into network interactions via the GOAL file format. A significant aspect of the research is developing and refining a simulation toolchain that enhances the accuracy and efficiency of these simulations. While the project focuses on establishing and optimizing the simulation methodologies rather than comparative analysis with traditional disk storage, it sets a foundational approach for future studies to explore enhancements in storage technologies and operational efficiencies in cloud-based systems. The operations analysis and development of a robust simulation toolchain help deepen our understanding of the operational dynamics within advanced storage networks, setting the stage for ongoing research and innovation in high-performance and scalable storage solutions.

Contents

Chapter 1

Introduction

1.1 Motivation

The rapid expansion of cloud computing and data-centric services necessitates robust, scalable storage solutions capable of handling increasingly complex I/O operations. Microsoft Azure's Next-Generation Block Storage, DirectDrive, represents a pivotal advancement in this domain, providing high-performance and resilient storage options. This project aims to simulate these I/O operations within DirectDrive, focusing on the creation and analysis of a comprehensive trace collection tailored for use with the LogGOPSim simulator.

The significance of developing this trace collection lies in its potential to bridge theoretical models and practical applications. By accurately capturing and replicating the dynamics of disk I/O operations within a simulated environment, this trace collection serves several critical purposes:

- **Enhanced Simulation Accuracy:** The traces derived from real-world data enable the simulation of storage systems under various operational scenarios, reflecting true storage behaviors more accurately than hypothetical models. This is particularly useful for predicting system performance under different load conditions and configurations.
- **Toolchain Development and Validation:** With a focus on the DirectDrive operations, the trace collection aids in refining the translation toolchain that converts real-world I/O operations into simulation-ready formats. This ensures that the toolchain can handle complex, dynamic interactions typical of modern storage networks, thereby increasing the reliability and robustness of simulation tools like LGS.
- **Educational and Research Utility:** The availability of a well-documented, realistic trace dataset can be invaluable for educational purposes, allowing students and researchers to explore the nuances of networked

storage systems. It also provides a benchmark dataset for the community, facilitating further research on storage optimization, fault tolerance, and performance scaling.

- **Future Enhancements and Scalability Studies:** The trace collection not only supports current simulation needs but also sets the foundation for future work to explore enhancements in storage technology. Researchers can use the traces to test new theories, algorithms, and architectures, driving innovation in storage solutions.
- **Industry Impact:** By simulating operations that are emblematic of those used in enterprise-level cloud storage systems, this project contributes directly to improving the design and management of the storage infrastructure used by businesses across the globe. The insights gained from such simulations can lead to more efficient, cost-effective storage solutions that are robust against a variety of operational challenges.

Through this project, we aim to create a resource that not only enhances understanding and simulation of current storage systems but also facilitates the development of next-generation storage technologies that are more adaptable, efficient, and capable of meeting the burgeoning demands of data-driven applications.

1.2 Related Work

The landscape of distributed systems and storage solutions has rapidly evolved, benefiting significantly from advancements in network simulation and the development of sophisticated storage architectures. This section outlines the foundational works that inform the current project on simulating I/O operations in Microsoft Azure's DirectDrive storage system.

One of the cornerstone tools in the simulation of distributed systems is LogGOPSim, a simulator designed to model large-scale applications within the LogGOPS framework. Introduced by Hoefler, Schneider, and Lumsdaine [1], LogGOPSim enhances the understanding of parallel algorithms and their performance across extensive networks, allowing for the simulation of over a million events per second. This tool's capabilities are crucial for analyzing the complex interactions within DirectDrive, providing a realistic performance outlook based on diverse operational scenarios.

The Group Operation Assembly Language (GOAL), developed by Hoefler, Siebert, and Lumsdaine [2] complements the simulation tools. GOAL offers a flexible way to express and optimize collective communications within distributed systems, providing a structured approach to describing the operations that underpin network interactions. The language's utility in modeling intricate dependencies and communication patterns makes it indispensable

for simulating the network behaviors observed in DirectDrive and other similar technologies.

DirectDrive itself was extensively discussed in a presentation by Greg Kramer [3], which outlined its role as Azure's next-generation block storage architecture. Kramer's exposition on DirectDrive emphasizes its innovative approach to storage solutions, offering insights into its architecture, operational dynamics, and strategic importance to Azure's infrastructure. This presentation serves as a primary resource for understanding the specific architectural enhancements and performance benchmarks that DirectDrive aims to achieve.

Additionally, the impact of DirectDrive on contemporary cloud services is highlighted in "Socrates: The new SQL Server in the cloud" by Antonopoulos et al. [4]. This study utilizes DirectDrive to evaluate the performance of SQL Server deployments in Azure, showcasing the practical applications and benefits of integrating advanced block storage solutions in managing database systems in the cloud. The findings from this work provide a real-world example of how DirectDrive can be leveraged to enhance cloud computing environments, making it a relevant inclusion in the study of storage systems.

Together, these works provide a comprehensive backdrop against the current project's position. They underscore the theoretical and practical aspects of network simulation and storage system design and highlight the ongoing but limited innovations shaping DirectDrive research.

Chapter 2

Background

This research leverages foundational works by Hoefler et al. and Greg Kramer from Microsoft, necessitating an understanding of certain preliminary concepts critical to our development. To effectively read and understand this report, basic knowledge of network communications and performance analysis is required. This foundational knowledge will enable the reader to grasp the methodology behind visualizing network interactions and the significance of the results obtained.

2.1 DirectDrive: Azure's Next-Generation Block Storage

Azure Disks, integral components of the Azure Infrastructure as a Service (IaaS) platform, provide block storage solutions for Azure Virtual Machines. Within this context, DirectDrive represents an advanced block storage architecture tailored for Microsoft Azure, laying the groundwork for Azure's disk offerings, notably including the Ultra Disk, Azure's premier performance disk solution.

The public unveiling of DirectDrive occurred during a presentation by Greg Kramer at the SDC2022 [3], marking a pivotal moment in its dissemination. While subsequent literature, such as the work by Antonopoulos et al. [4], has referenced DirectDrive, *Kramer's discourse remains the primary exposition of DirectDrives architecture and functionalities.*

2.1.1 Design Objectives

The inception of DirectDrive was driven by the ambition to deliver a storage solution that was not only durable and performant but also highly available at the scale demanded by cloud services. Its design emphasizes versatility, catering to a diverse array of deployment scenarios:

1. Disaggregated or hyper-converged infrastructures,
2. Small to medium-scale implementations, including on-premises and edge computing,
3. Hyper-scale environments like Azure,
4. Fully virtualized single-server setups.

DirectDrive addresses the complexities of deploying across these varied environments by enabling shared disk configurations (supporting single-writer, multiple-reader, and multiple-writer scenarios) and ensuring crash consistency and the availability of incremental snapshots for enhanced reliability. Furthermore, it facilitates disk migrations under active I/O operations, a critical feature for hyper-scale applications.

By presenting block storage as a unified disk to users and incorporating dynamic performance scaling, DirectDrive significantly simplifies user interaction with storage resources. A deliberate effort to minimize the number of intermediary layers aims to streamline the I/O pathway and bolster consistency across operations.

2.1.2 Architecture and Supported Disk Types

DirectDrive accommodates two disk formats:

1. Native 4k disks, with both logical and physical sectors sized at 4k,
2. Simulated 512b disks featuring 512b logical sectors and 4k physical sectors.

Disk management within DirectDrive is organized around fixed-size segments termed "slices," which are coordinated through a system of replicas. This structure facilitates efficient storage and retrieval. Due to supported disks requiring 4k physical sectors, the architecture prefers 4k aligned slices.

DirectDrive's innovative approach extends to its comprehensive network components, including the Host, Virtual Disk Controller (VDC), and a VDC-Proxy, ensuring seamless integration between the operating system and DirectDrive Network. This setup enables runtime upgrades of the VDC without disrupting ongoing disk interactions, showcasing the system's adaptability and resilience.

The Change Coordinator Service (CCS), a pivotal component, efficiently sequences and replicates slice changes while maintaining a stateless architecture, facilitating rapid recovery and replication across fault domains.

The Block Storage Service (BSS) is crucial in managing slice data and ensuring data integrity through scrubs and repairs, further emphasizing DirectDrive's commitment to reliability and performance.

The MetaData Service (MDS), the "brain of the operation," dynamically assigns CCS and BSS instances to slice replica sets, managing disk control requests and error correction with precision. Its deployment across multiple fault domains and utilization of a Paxos Ring for state replication enhance system robustness against failures.

DirectDrive's architecture also includes a Gateway Service (GS) and a Software Load Balancer (SLB), optimizing the MDS's access control and load balancing, underscoring the architecture's focus on scalability and security.

This intricate network of components underscores DirectDrive's ability to offer high-performance, reliable, and scalable block storage solutions, making it an ideal foundation for cloud-based storage systems. Through strategic deployment across various environments and a focus on minimizing complexity, DirectDrive not only simplifies user interaction but also provides a robust platform for the development and execution of our toolchain.

2.2 GOAL File Format for Network Interaction Modeling

The Group Operation Assembly Language (GOAL) [2] file format represents a significant advancement in the domain of network interaction modeling, particularly within systems that necessitate intricate dependent network interactions, such as the I/O interactions of Direct Drive storage solutions. Introduced by Höfler et al., GOAL offers a comprehensive framework for expressing collective communication operations, vital for optimizing the performance of distributed systems. By abstracting group communication into a domain-specific language, GOAL facilitates the description, optimization, and execution of complex network operations, thereby enhancing the flexibility and efficiency with which complex interactions can be expressed and simulated, like computational models in cloud-based environments.

GOAL enables the detailed simulation of network behaviors critical for accurately representing interactions in Direct Drive systems through its unique approach to dynamically defining and scheduling group operations. Its utility extends beyond simple communication patterns, allowing for sophisticated operations modeling that leverages the underlying network architecture's full capabilities. This characteristic is particularly beneficial for translating disk I/O traces into network interactions, offering insights into the performance implications of various architectural choices within Microsoft Azure's Direct Drive storage framework.

2.3 UMass Trace File Dataset: A Resource for Storage System Analysis

The UMass Trace File Dataset [5] represents a pivotal resource in the study and simulation of storage system behaviors, particularly within block storage solutions like Azure's DirectDrive, which market themselves as a drop-in replacement for traditional storage solutions. Compiled with meticulous attention to representing the various dynamics of disk I/O operations, this dataset provides a rich tapestry of real-world storage access patterns, making it an invaluable asset for developing and testing storage solutions.

Central to the dataset's utility is its detailed capture of disk access traces, encompassing a wide array of read and write operations across diverse storage scenarios. Including timestamped events allows for the precise reconstruction of storage workloads, offering insights into performance bottlenecks and optimization opportunities. Moreover, the dataset's versatility in simulating various storage configurations underpins its role in enhancing the fidelity of storage system simulations.

Incorporating the UMass Trace File Dataset into our toolchain facilitates a nuanced understanding of network interactions within DirectDrive. This enables the refinement of our simulation models to reflect the complexities of real-world disk I/O patterns accurately. The dataset's comprehensive coverage of storage access behaviors, from sequential reads to random writes, empowers our simulations with a depth of realism critical for validating and optimizing our toolchain.

By leveraging the UMass Trace File Dataset, our research stands at the forefront of storage system analysis, bridging the gap between theoretical models and practical application in the evolving landscape of cloud-based block storage solutions.

2.4 Leveraging LogGOPSim for Network Interaction Simulations

LogGOPSim [1], a sophisticated simulation framework, is crucial for analyzing parallel algorithm performance across extensive networks, as detailed by Höfler et al. This simulator extends the LogGPS model to support full MPI message semantics and collective operation simulations, making it uniquely suited for large-scale application simulations. Its efficiency and ability to simulate over a million events per second facilitate exploring complex network interactions within Direct Drive systems.

The native compatibility of LogGOPSim with GOAL file formats enhances its utility, providing a seamless integration for modeling and simulating network

interactions based on disk I/O traces. This compatibility was a pivotal factor in selecting LogGOPSim to test and debug the execution of our toolchain. The simulator's design, emphasizing simplicity and performance, efficiently handles extensive process simulations, thereby offering an ideal platform for validating the network interaction models derived from Direct Drive's disk storage operations.

2.5 htSim: Enhanced Network Simulation

htSim [6] is a high-performance discrete event simulator tailored to analyze congestion control algorithms and their behavior in complex network environments. Developed initially to explore TCP stability and extended for Multipath TCP performance evaluation, htsim's capabilities have been pivotal in the research and development of advanced networking protocols like NDP, DCTCP, and DCQCN. Its design, inspired by ns2 but achieving significantly higher speeds, enables the efficient simulation of extensive networks, making it an indispensable tool for our project.

Our project leverages an advanced version of htsim. This particular iteration of htsim is a private fork enhanced by Tommasso Bonato, uniquely modified to include native support for GOAL file formats. This adaptation significantly extends htsim's utility, enabling direct and efficient simulation of network interactions from disk I/O traces. Bonato's contributions have transformed htsim into an indispensable tool for our research, allowing for precise modeling of network behaviors under the Direct Drive architecture. Our fork's capability to handle GOAL files streamlines the testing and debugging process, offering unparalleled insights into the performance dynamics of networked storage systems.

Chapter 3

DirectDrive Operations

DirectDrive, as a cornerstone of Azure's block storage solutions, natively offers high-level filesystem operations. This allows developers and users alike to not worry about implementation details and instead develop for DirectDrive like for any other disk storage device. This is done by simulating a Virtual Disk Controller (VDC), which translates the IO operations (mount, read, write) into the underlying DirectDrive operations.

Besides the VDC, which is accessed through the system's hypervisor, there is also a VDC proxy that allows debugging and upgrading of the underlying VDC during execution without downtime. This is especially useful in a production and large-scale environment like Azure's main block-storage solution.

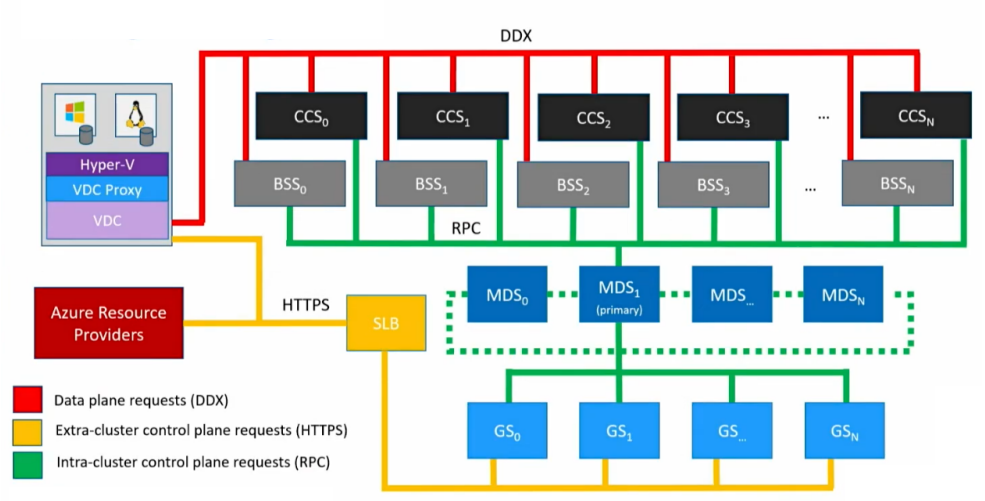


Figure 3.1: Direct Drive Topology

3.1 Trace Mapping Process Overview

Trace mapping plays a pivotal role in evaluating real-world IO interactions using the DirectDrive storage solution. This is a non-trivial step due to the complex nature of the DirectDrive storage and dependencies between IO interactions. However, once trace mapping is possible, it will be easy to compare and evaluate different underlying network types against one another and even optimize the network for storage interactions.

3.1.1 UMass Trace File Interpretation

As mentioned in 2.3, we will use the UMass Trace File Dataset for I/O operation to DirectDrive operation translation evaluation. The trace files provided by UMass are CSV files with at least five columns, of which the first five columns are always defined as the following in order:

- **Application Specific Unit (ASU)** The ASU uniquely identifies each application. Although not technically accurate, we interpret it as the host identifier. (See Section 4.4.1).
- **Logical Block Address (LBA)** The LBA is the offset into the ASUs accessible disk block. We use it for simplicity's sake as the start address of the underlying read or write operation.
- **Size** The size read or written to the disk in bytes.
- **Opcode** The opcode can be 'R' or 'W,' representing a read or write operation, respectively.
- **Timestamp** The timestamp of the operation, we ignore this field. It is, however, guaranteed that all rows are ordered increasingly by this field.
- **Optional other fields (Ignored)** As per the UMass trace file spec, an undefined number of optional other fields might follow. These fields are ignored for obvious reasons.

Since the trace file is ordered by increasing timestamps, we can interpret the file rows sequentially in order as interactions with our Direct Drive Network.

3.1.2 UMass Trace File Limitations

Only the first four fields are used when mapping a uMass trace to a goal file. The timestamp is ignored as the goal is not to replicate the exact executions of the operations but instead replicate the access patterns, which can then be used to simulate traditional I/O operations in the DirectDrive network. For further evaluation however (e.g. a direct performance analysis) these timestamps would be important, however due to the different nature and

hardware requirements of a traditional disk setup compared to a DirectDrive network, this evaluation would only help in terms of validating if the established baseline, of a traditional disk, is met or not.

How these implicit dependencies between ASUs and time in the trace files should be interpreted is not completely clear, as all these interactions could have likely occurred on either one underlying disk or multiple disks in a RAID configuration or similar. Therefore, we assume that interactions only depend on the previous interaction of the same ASU/Host. This should allow highly parallel executions of the traces but make sure that all operations for the same ASU/Host are executed in order. As this behavior introduces a lot of execution overhead and is not necessarily required in all execution environments, there will be a way to deactivate the dependencies between operations in the final toolchain.

As mentioned in the previous subsection, the trace file only has two possible Opcodes: 'R' (read) or 'W' (write). DirectDrive, on the other hand, supports a third operation: mounting. Due to our interpretation of ASUs as different hosts, we will lazily trigger a mount operation on each first read or write interaction of a new ASU/Host to emulate sensible real-world behavior.

All emulated reads and writes share the same address space. As address space will be emulated by the VDC respectively, this is a simple design decision that has to be kept in mind when choosing or adapting trace datasets. Because it is not unlikely that multiple applications write to the same file and, therefore, address region, one could conclude that our abstraction of ASUs as different hosts is broken. However, it is important to highlight that different hosts might share the same underlying 'virtual disk' and, therefore, access the same files. However, if a clear separation between host accesses is desired, a behavior similar to multiple disks can be easily simulated by choosing non-overlapping address regions for accesses.

3.2 Detailed Operations Analysis

To understand the operation execution inside the DirectDrive network, it is essential to understand the previously defined roles of the various services/components in a DirectDrive system (See Section 2.1). Additionally, the reader should understand how the virtual disk address space maps to the underlying Block Storage Services; more about that in the coming Section 3.2.1.

Due to the limited information in the main source, sometimes assumptions and simplifications had to be made; we will, however, elaborate on all of these assumptions in Section 7.2.

3.2.1 Preliminaries

Below, you will find a recap of the services at play and some detailed information on how data is laid out across the network and disks. The following chapters, although not essential to understanding the operations, highlight some aspects that help explain DirectDrive's theoretical performance compared to typical storage solutions.

Roles/Services

Here is a short overview of the components and their roles; for more detail, go back to the background section (2) :

- **Host** Initiator of all IO interactions.
- **Change Coordinator Service (CCS)** Sequences and replicates changes.
- **Block Storage Service (BSS)** Stores, retrieves and repairs slice data.
- **Metadata Service (MDS)** Assigns CCS and BBS instances to replica sets and handles errors.
- **Gateway Service (GS)** Sits between the "outside" and the MDS.
- **Software Load Balancer (SLB)** Load balances access to the GS/MDS.

Virtual Disk Layout

As mentioned in 2.1, the disk data is stored in slices, i.e., the data is striped into configurable fixed-sized chunks from the client's perspective. Each slice can then be mapped to a list of BSS and CCS or vice versa. Each BSS contains a specific list of slices. Slices are allocated to BSS so that neighboring slices are not stored in the same BSS, allowing fast reads and writes as data can be written in parallel across the BSS. The slices are managed through a replica set, where one slice always has one responsible CCS and N BSS, where N is chosen based on disk resilience requirements. The replica set mappings are stored in the MetaData Service in a centralized manner. The Change Coordinator Service sequences and replicates the slice across all responsible BSS, and because the MDS stores the mappings, the CCS can, therefore, be considered as practically stateless, as new instances can be spun up with ease in case of an error. The Block Storage Service stores, retrieves, scrubs, and repairs slice data. Multiple slices can and will be allocated to the same CCS or BSS.

SCM Writeback Cache

The Block Storage Services (BSS) heavily use a Writeback Cache. Instead of directly writing to disk, data is first written to non-volatile memory

(SCM), after which a "Promise to Write" completion will be given. This writeback cache will then be used and written until it is close to full. In contrast to a normal SSD, which traditionally uses non-volatile memory as an intermediate, this allows further control over how and when data is written to the underlying storage and therefore has higher consistency guarantees compared to directly writing to an SSD, in which a garbage collection on the controller might cause unexpected slow downs. This way, even regular updates of the same slices will additionally not incur any unnecessary IO costs, and therefore, the disks' life expectancy will also be prolonged.

Completion Side I/O Throttling

The Virtual Disk Controller (VDC) implements disk throttle in the completion path. This basically means that instead of issuing requests on the VM at the desired IO target, requests will be issued immediately, and the responses to the VM will be delayed at the Disk Controller level to meet the VM's IO target. This way, resilience to potential slow-downs can be achieved, as potential late completions might not have any impact at all, as they have been issued earlier already. If there are, however, any issues with this approach, the system will switch to traditional IO throttling.

From our perspective, however, this approach significantly simplifies our simulation. We can simply issue requests as fast as possible and do not have to worry about OS specifics, as we can assume that the host will internally ensure that the IO target is met.

I/O Overhead

Reading and Writing data from and to disks is not free. Therefore, whenever a disk interaction is supposed to happen, we will estimate it by inserting a CPU blocking operation (`calc`) scaled with the size of the data transferred from/to disk.

For this, we have introduced a per-byte overhead, which has been chosen based on approximations of an average SSD performance (6GB/s reads, 1.5GB/s writes). This parameter can be adapted by changing the value correspondingly of the correspondingly named variable `per_byte_overhead` in `calc_io_time` in the `interaction.py` source file.

3.2.2 Mount Operations

Mount operations in DirectDrive are characterized by the way data is stored. As data is stored in slices and replicated across multiple BSS, the Host requests the corresponding map of slices from the MDS. This map of slices tells the Host where he can find his corresponding virtual storage regions. How specifically this map of slices looks is not known, however it is safe to

assume that it simply is a map from address regions to a CCS and a list of BSSs.

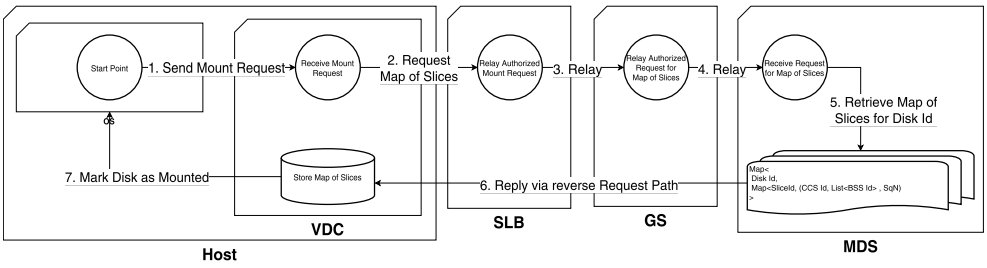


Figure 3.2: Flow diagram of a mount operation

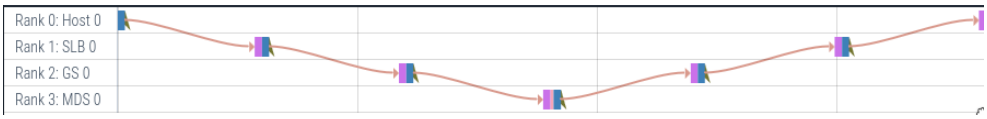


Figure 3.3: Visualization of a minimal mount operation (One instance per service)

3.2.3 Read Operations

Read operations in DirectDrive are characterized by the Host's full knowledge of the map of slices and, therefore, the responsible BSSs that contain the wanted data correspondingly. Therefore, the host can directly read from each responsible BSS. Due to the slicing of data, these reads are also quite fast as a bulk of the read operation can happen in parallel, as slices are spread across multiple unique BSS.

Each data slice also has a corresponding sequence number (SqN) that takes the function of a version number. It is used to detect that out-of-date data was read, in which case the Host would simply try reading from another BSS and notify the MDS of this issue. How these sequence numbers are communicated is not clear from the talk, however it is safe to assume that the MetaData Service also manages these and might even communicate them to the Host initially via the map of slices.

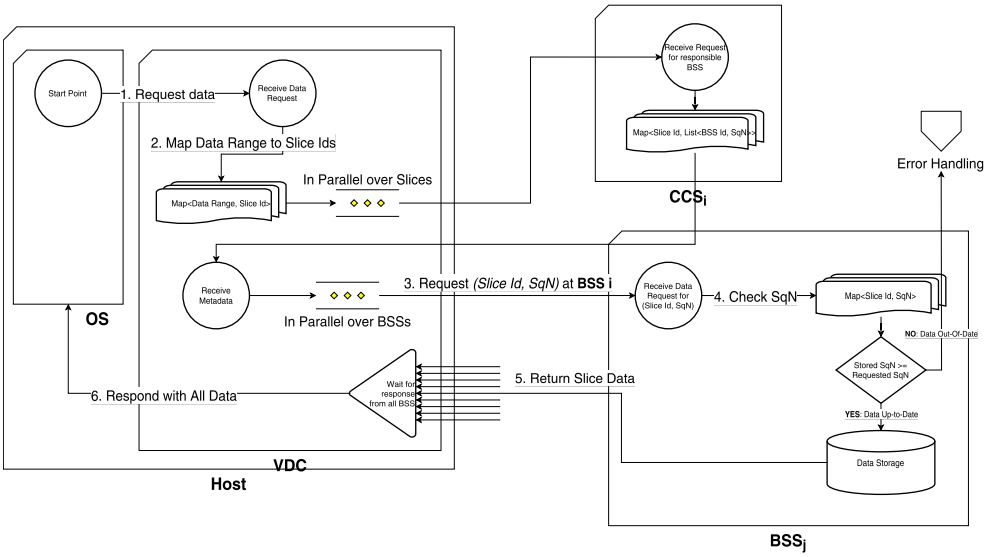


Figure 3.4: Flow diagram of a read operation

Figure 3.5: Visualization of a minimal read operation (One instance per service)

3.2.4 Write Operations

During write operations, DirectDrive ensures that all data is written to the full replica set via the CCS. Instead of the host writing all the data to all BSS in the replica set, the CCS will do so, and the host only has to write the data to the CCS once. Depending on the configuration, a write will be completed from the Host's point of view as soon as enough BSS out of the quorum has successfully received all the corresponding data. On a successful write, the host will receive the new update sequence number, which differentiates his new data from the data that has previously been at this slice.

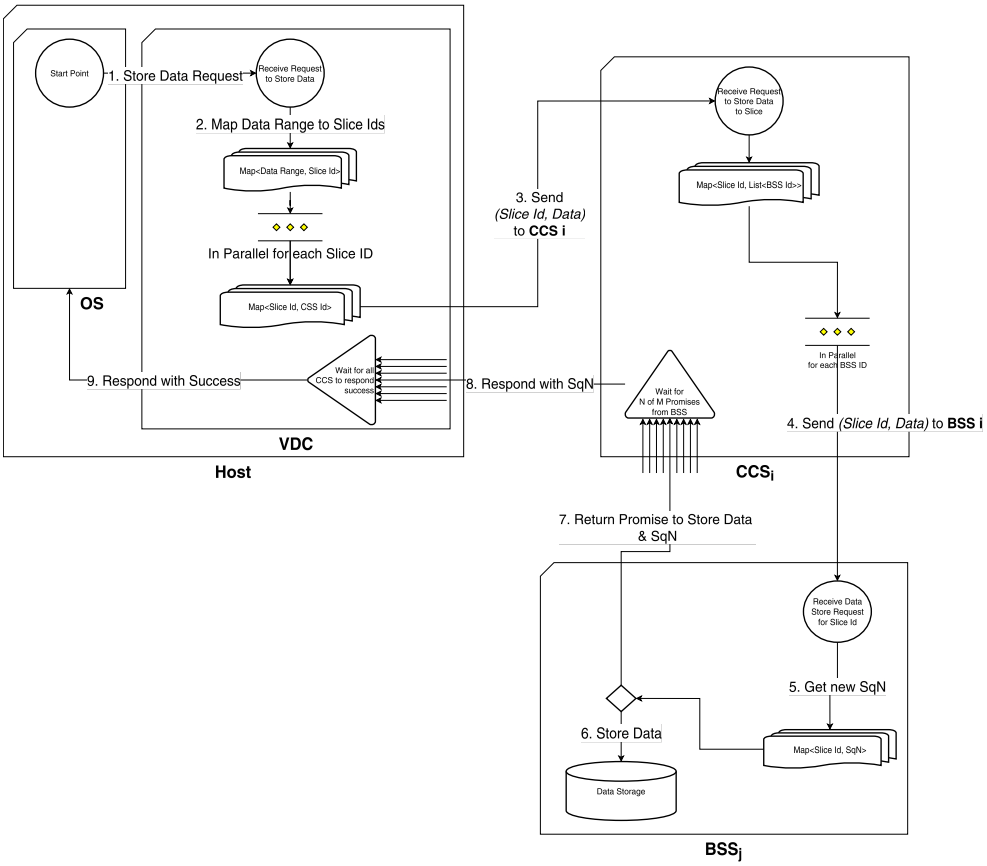


Figure 3.6: Flow diagram of a write operation

Figure 3.7: Visualization of a minimal write operation (One instance per service)

3.2.5 Data Replication and Fault Tolerance

Data replication across DirectDrive's distributed architecture is mainly handled through the interaction between CCS and BSS, as already highlighted in 3.2.4. In case there are any errors during reads or writes, they will be reported to the MDS, which then can either escalate them or, if possible, fix them accordingly directly. To ensure all replicated data is always accessible, BSSs are spread across fault domains. This way even in case a central switch breaks or is unresponsive, data is still accessible via other BSS. Similarly the MDS and its replicas are also spread across multiple fault domains to allow failovers.

3.3 Performance Insights

Analyzing the translated operations unveils critical performance insights: Many of the operations, which in a typical filesystem would need to be serialized, can be parallelized thanks to the distribution of slices across different BSS. Additionally, the use of SCM Writeback Chaches, as described in 3.2.1, ensures that writes to the BSS are faster than possible on a normal comparable disk. Through the use of Completion Side I/O Throttling, resilience to slow-downs is also achieved.

Overall, there is a significant (theoretical) improvement in IO operation performance compared to a traditional disk setup.

3.4 Conclusion

The meticulous analysis of DirectDrive operations and their translation from traces shows that although details of implementation-specific things like error handling or communication of sequence numbers are not known, we can paint a big enough picture to model a significant subset of the relevant interactions within DirectDrive.

Now that this essential information has been collected, we can continue by actually implementing the translation from our flow diagrams to corresponding goal interactions.

Chapter 4

Translation Toolchain

So far, we have mostly discussed how interactions in a DirectDrive Network take place and highlighted differences and similarities to traditional storage solutions. Now, we are equipped with all preliminary and required knowledge and discuss the translation toolchain implementation.

4.1 Toolchain Requirements

Our toolchain should make it easy to go from real-world IO operations to ready-to-use goal files. It should be usable via a simple command-line interface (CLI) but also have an advanced programming interface (API) available to allow extension and building new tools on top. The CLI should be self-documenting and descriptive enough to allow new users to have ease of entry. The toolchain should also be fully configurable with sensible default values.

Additionally, we require that our toolchains' hardware requirements be minimal; we could simply ensure that our toolchain does not require more than what the simulation or the translation from goal to binary requires. However, this way, we ensure that even in case of a sudden performance improvements of packet-based simulators this tool stays useful for the potentially growing datasets that will need to be translated.

4.2 Translation API

As discussed in 2.1, a DirectDrive network can be broken down into its topology, basic components, and some other minor configuration parameters. Therefore a DirectDrive network can be fully described through the following parameters:

- Topology (No. of Hosts, SLB, GS, MDS, CCS, BSS)

```
# Create network
topology = NetworkTopology(
    host_count=1, slb_count=1, gs_count=1,
    mds_count=1, ccs_count=1, bss_count=1,
    strategy=1
)
network = DirectDriveNetwork(
    topology=topology, slice_size=4096, disk_size=80192
)

# Add network interactions
network.add_mount(0)
network.add_read(0, 1024, 4096)
network.add_write(0, 2048, 3072)
```

Figure 4.1: A minimal example of using the API

- Slice Size (optimally 4k aligned)
- Slice to BSS Mapping (alternatively a list of available space on each BSS)
- MDS quorum ring size
- CSS redundancy
- BSS quorum size

Some of these parameters might overlap with the topology, and therefore, only the topology or the parameter is necessary (e.g., MDS quorum ring size).

For this reason, the library makes a 'DirectDriveNetwork' class available that takes exactly these parameters and some interaction configuration as input.

Therefore a most minimal Python script that uses our API could look similar to Figure 4.1.

4.3 Operation Translation

The general approach is relatively straight-forward: One takes the corresponding flow diagram and translates all arrows into corresponding `send` and `recv` operations (with a unique but shared tag for each transmission) and introduces dependencies between all operations to ensure that no illegal out-of-order execution can happen.

Below, you can find some examples that show what a minimal execution for each operation would look like and also some basic explanations. (Some lines have been swapped and indentation added for readability). We have verified for multiple examples that the generated goal files (and their corresponding

executions) do align with our expected outputs based on the flow diagrams. This verification process was also one of the reasons for a new visualization, as we will discuss later.

4.3.1 Mount

Minimal Example

```
// Host #0
rank 0 {
    s0: send 4096b to 1
    r12: recv 4096b from 1 tag 1
    r12 requires s0
}
// SLB #0
rank 1 {
    r1: recv 4096b from 0
    s2: send 4096b to 2
    r10: recv 4096b from 2 tag 1
    s11: send 4096b to 0 tag 1
    s2 requires r1
    s11 requires r10
}
// GS #0
rank 2 {
    r3: recv 4096b from 1
    s4: send 4096b to 3
    r8: recv 4096b from 3 tag 1
    s9: send 4096b to 1 tag 1
    s4 requires r3
    s9 requires r8
}
// MDS #0
rank 3 {
    r5: recv 4096b from 2
    c6: calc 683
    s7: send 4096b to 2 tag 1
    c6 requires r5
    s7 requires c6
}
```

Figure 4.2: A snippet from a goal file that does exactly one mount operation

In Figure 4.2, `rank 0` takes the role of the host, while rank 3 is the MDS. `rank 1` and `rank 2` are the SLB and GS, which, in this case, do nothing more than pass on the message to the next hop and wait for a reply, which will be sent to the original sender.

1. `s0`: `rank 0` initiates the mount request by sending data to `rank 1`
2. `r1` + `s2`: `rank 1` relays the mount request to `rank 2`

3. r3 + s4: rank 2 relays the mount request to rank 3
4. r5 + c6 + s7: rank 3 receives the mount request, creates a reply and responds
5. r8 + s9: rank 2 relays the mount response to rank 3
6. r10 + s11: rank 2 relays the mount response to rank 3
7. r12: rank 0 receives the mount response

Complex Execution

Mounting is a relatively straightforward process that doesn't change (except for rank numbers) with the DirectDrive network configuration. Therefore, a minimal example fully showcases the mounting process.

4.3.2 Read

Minimal Example

```
// Host #0
rank 0 {
    s0: send 1024b to 4
    r4: recv 4096b from 4
    r4 requires s0
    s5: send 1024b to 5 tag 1
    r9: recv 1024b from 5 tag 1
    s5 requires r4
}

// CCS #0
rank 4 {
    r1: recv 1024b from 0
    c2: calc 683
    s3: send 4096b to 0
    s3 requires r1
}

// BSS #0
rank 5 {
    r6: recv 1024b from 0 tag 1
    c7: calc 171
    s8: send 1024b to 0 tag 1
    s8 requires r6
}
```

Figure 4.3: A snippet from a goal file that does exactly one read operation

In Figure 4.3, rank 0 takes the role of the host that tries to read some data. rank 5 is the BSS where the data is actually held, and rank 4 is the for this

BSS responsible CCS.

1. s0: rank 0 initiates the read by sending a request to rank 4
2. r1 + c2 + s3: rank 4 responds to this read request with the responsible BSS (rank 5)
3. r4 + s5: rank 0 receives the responsible BSS from the CSS and finally directly sends a read request to rank 5
4. r6 + c7 + s8: rank 5 receives the read request, retrieves the data and sends it to rank 0.
5. r9: rank 0 receives the data

Complex Execution

If you recall the flow diagram of the read operation, you will remember that the actual reading of data happens in parallel across all required slice IDs. Although this is, in theory, a simple change, it causes a linear increase in file length with the number of slices read for each operation. This is the case as it is necessary to replicate the corresponding logic required for a simple slice read for each slice that is accessed, even though the data is likely not used. However, since the host does not require receiving all responses at once or in a certain order, no special or new logic is introduced.

4.3.3 Write

Minimal Example

In Figure 4.4, rank 0 takes the role of the host that tries to write some data. rank 5 is the BSS where the data is actually held, and rank 4 is the for this BSS responsible CCS.

1. s0: rank 0 initiates the write by sending a write request to rank 4 (consists of data and slice id)
2. r1 + c2: rank 4 receives the request and stores it locally
3. s3 + r7: rank 4 writes the data to all responsible BSS and waits for a successful reply
4. r4 + c5 + s6: rank 5 receives the write request, stores the data, and sends a successful reply to rank 5
5. s8 + r9: rank 4 sends success promise to rank 0

Complex Execution

If you recall the flow diagram of the write operation, you will remember that there are multiple places in which requests should be sent in parallel.

```
// Host #0
rank 0 {
    s0: send 1014b to 4
    r9: recv 4096b from 4 tag 3
    r9 requires s0
}

// CCS #0
rank 4 {
    r1: recv 1014b from 0
    c2: calc 676
    c2 requires r1
    s3: send 1014b to 5 tag 1
    r7: recv 1024b from 5 tag 2
    s3 requires c2
    r7 requires s3
    s8: send 4096b to 0 tag 3
    s8 requires r7
}

// BSS #0
rank 5 {
    r4: recv 1014b from 4 tag 1
    c5: calc 676
    s6: send 1024b to 4 tag 2
    s6 requires c5
    c5 requires r4
}
```

Figure 4.4: A snippet from a goal file that does exactly one write operation

Therefore, a write will cause a linear increase in file length with the number of slices written to for each operation. First, the data can be written to the CCSs in parallel across all slice IDs. After that the CCS can write the data in parallel to all BSS. However, as only some of the BSS need to have successfully written the data before the CCS will see it as a successful write, we would, in theory, need some non-deterministic requirements to ensure that a quorum has been met. As this cannot be modeled using the GOAL language format, we have decided to only support full quorum writes (i.e., a quorum of N=M). However, as soon as all CCSs have returned a promise, the host will also mark the data as successfully written, and the write operation is over.

4.4 Trace File Translation

As already mentioned in 2.3, we are using the publicly available trace files from the uMassTraceFileRepository [5]. For a detailed description, please refer back to the background sections 2.1, 2.2 and 2.3.

Here is a quick recap of the relevant columns of the CSV files:

- **Application Specific Unit (ASU)** (Host identifier)
- **Logical Block Address (LBA)** (Storage address)
- **Size**
- **Opcode** ('R' for read or 'W' for write)
- **Timestamp** (ignored, but guarantees ordering)

4.4.1 Local vs Network Storage Differences

We have already discussed how each IO operation would be translated into GOAL operations; there are, however, some underlying differences between traditional disk operations and network-based storage solutions. One example would be disk operations, which are inherently synchronous compared to network operations. Additionally, due to the loose definition of a DirectDrive disk from the presentation, we will have to be more concise about this.

Operation Order

As already mentioned above, disk operations traditionally happen synchronously. However, this is not a requirement for the DirectDrive network, which can, due to its decentralized approach, highly parallelize its requests without much overhead. However, to keep comparability to disk drives, an option was added to depend on each operation execution on the previous one on a host-by-host basis. This means that a single operation execution must be fully finished before the next one can start (for a specific host). This option (`--op-depens`/`--no-op-depens`) is set by default and can, however, be disabled to simulate full-load access and find potential bottlenecks.

Hosts vs ASU

The attentive reader might have already noticed the discrepancy in the usage of the ASU as the host identifier. Traditionally, disks are only used by a single host and multiple applications. However, with DirectDrive, a network can simulate multiple disks across multiple hosts and multiple applications. This means that although multiple applications technically run on the same host in the trace files, we will only use a tiny subset of the DirectDrive network's potential performance due to unused parallel potential.

Therefore we have decided to interpret each application as its own host. Due to operational dependency across hosts, this will have the most potential for safe parallel accesses compared to only using one host.

Address Space

Additionally we decided to share the disk address space across hosts. This is likely not the case in real DirectDrive production environments for obvious security concerns. However, by doing so, we ensure that the adaptation of applications to hosts and potential similar disk accesses are taken into account. Therefore, the simulation will be as close to the original disk trace as possible while still presenting a theoretically correct and comparable execution.

4.4.2 Actual Translation

Now that all the relevant differences and adaptations necessary have been discussed, we can show how the line-by-line translation takes place.

Due to all the preparation that has already happened during the DirectDrive abstraction, the actual parsing boils down to a couple of lines of code:

```
network = DirectDriveNetwork(
    topology=topology, slice_size=slice_size,
    disk_size=disk_size, op_depens=op_depens,
)

with open(trace_path, 'r') as f:
    for (asu, lba, size, opcode, *_) in csv.reader(f):
        network.add_interaction(op_code=opcode, asu=int(asu),
                                address=int(lba), size=int(size))
```

Figure 4.5: Actual snippet that is used to translate traces

Due to the simple translation process, it should be easy to extend the trace file parsing to any desired format that has similar arguments to the uMass dataset.

4.4.3 Performance

All code has been written in python (with the addition of some Bash scripts). As Python is an interpreted language, this code might not be the fastest possible version of itself. However, optimizations ensure that the pipeline can run on basically any setup (given enough time).

First of all, potential bottlenecks have been found using Scalene and optimized accordingly. Most of these are not noteworthy, however one of the optimizations has decreased RAM usage significantly.

Instead of holding all state (which mostly consists of GOAL-style strings) in memory, temporary files will be allocated for each rank and used to hold the state. This has decreased the toolchain's memory footprint to a constant,

whereas previously, it was a large factor (> 100) of the input trace file size by the end of the execution. However this feature has to be used carefully, as it is limited by the maximum allowed number of file pointers that a process is allowed to open. As we will open one file per rank, this has to be kept in mind.

Now the tool is able to translate a large trace file from the uMass dataset (file size: 148MB, over 5.3 Million IO operations,) within $<$ 20min on my test system (Intel i7-1165G7 @ 2.80GHz; Boosts up to 4 GHz). (Note that it is likely necessary to increase the `/tmp` partition size, this can be done through the following command `sudo mount -o remount,size=60G /tmp/`, which resizes the `/tmp` folder to 60GB until the next reboot) Additionally it is safe to assume that the performance scales with the single core clock speed, as the translation happens in a synchronous manner on a single thread. Although this shows the performance quite well, it also highlights future assurance of this translation toolchain: The generated goal file has a size of 17 GB and is compromised of 99 ranks, with 266 Million overall goal instructions (`send, recv, calc`) and 219 Million overall dependencies. This does not only sound exceptionally large, but I was not even able to translate the goal file into it's more efficient binary version using the `txt2bin` tool, due to signification memory requirements. To put this into reference the largest trace file I ran through the full pipeline during overall testing was smaller than a factor of > 500 compared to the full trace file (i.e. 100k I/O interactions, 2.7 MB trace file size, 224 MB goal file size).

Due to the fact that packet-based simulators are inherently slow and other goal-related tooling similarly does not fare well with such large data sizes, this should ensure the tool's future usability.

Chapter 5

Visualization Toolchain

The following section outlines the architecture and components of the visualization toolchain, highlighting the integration of data collection, processing techniques, and visualization strategies designed to offer in-depth insights into network interactions.

To validate the executions of the goal files against our expected executions, we decided to use the LogGOPSim integrated visualization tooling. However, due to unknown reasons, we were not able to successfully start the legacy visualization software. (See Figure 5)

As the previous visualization could have profited from more in-depth information anyway, we decided to develop a new visualization framework.

5.1 Tool Requirements

This subsection delves into the essential criteria and technical prerequisites that our visualization tool must meet to effectively map and analyze network interactions, ensuring clarity, precision, and ease of use for performance tracing.

Although we are, in this case, using this tool for the specific case of visualizing DirectDrive networks, the tool should, in general, be able to visualize any goal execution and, therefore, share the same minimum requirements of the DrawViz tool. A few things can be noticed from the DrawViz output from the paper:

- Each rank is represented through a timeline.
- It is easily recognizable if the rank is blocked, due to operations, or free
- Data transfer is easily recognizable through the use of arrows.

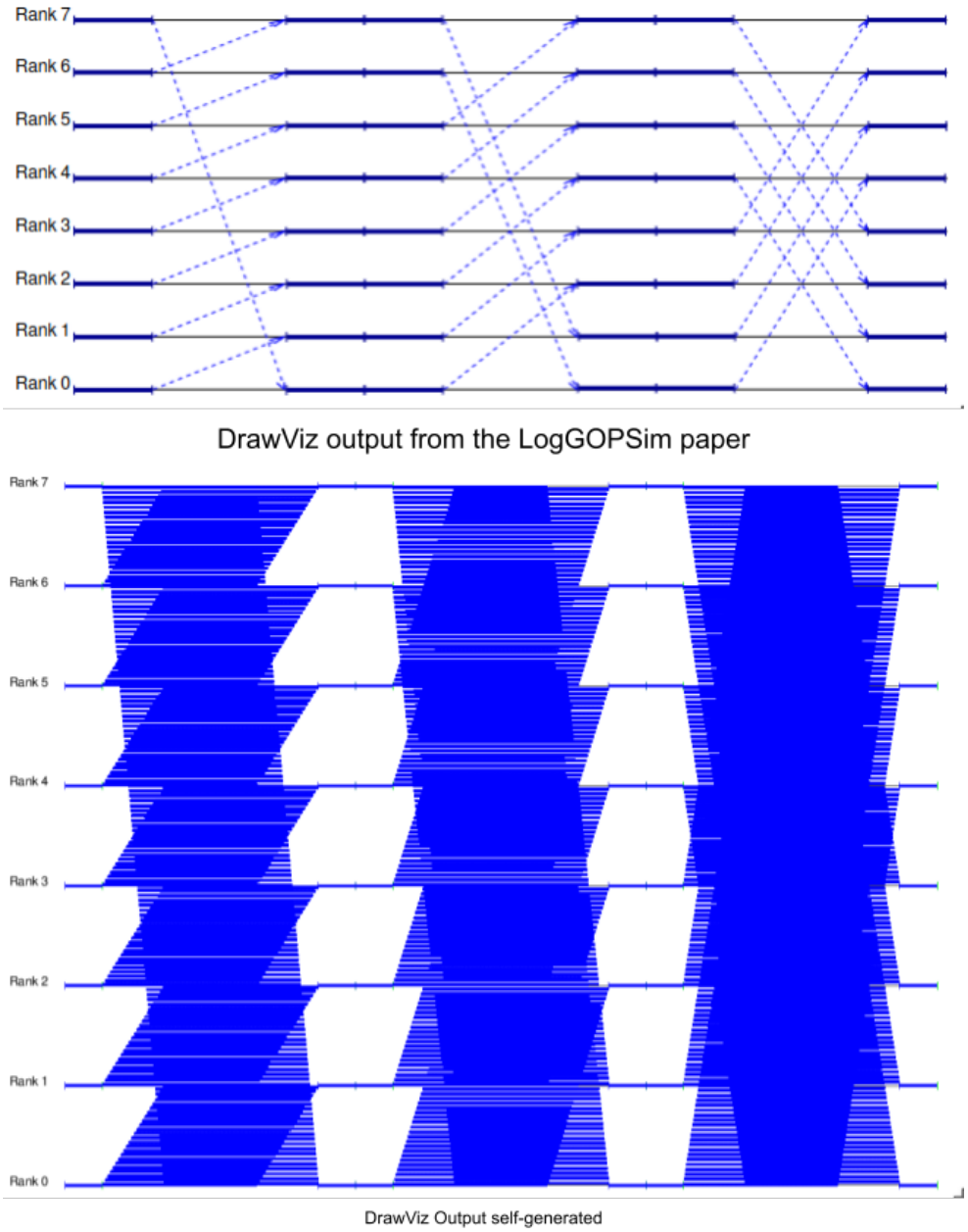


Figure 5.1: Old toolchain. Expected vs. actual generated output

As already mentioned above, a rank has to be in one of two states at any time: blocked (dark blue) or free (no color/grey line). A rank can be blocked for multiple reasons: Either it is doing calculations, and its CPU is in use, or it is transmitting data, and its NUC and/or CPU are in use. The differentia-

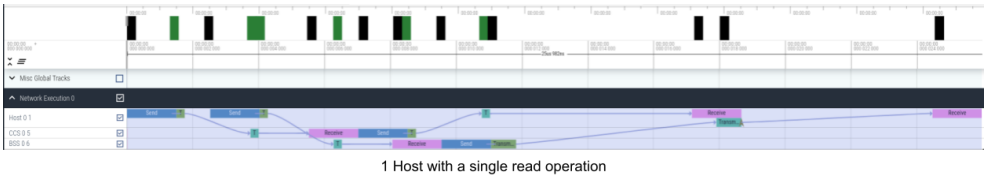


Figure 5.2: One of the first prototypes of the new visualization

tion between CPU and NUC is not visible from the DrawViz visualization, although it is part of the LogGOP model.

Similarly to the previously mentioned points, our visualization software should, therefore, be able to show for each rank whether it is currently blocking or not and also visually display sends and corresponding receives. Additionally, it should be able to show more in-depth information on the reason for the current block for detailed debugging purposes.

The visualization tool heavily relies on the underlying simulation framework (in this case, LogGOPSim), so we also decided to follow terminology from the LogGOPS framework and its architecture.

Due to possible different needs in terms of required detail, the user might want to only have a simple, abstract view of what is happening, similar to DrawViz. Therefore, there will be three final visualization modes: `basic`, `advanced`, and `expert`, all with a different detail grade of the underlying LogGOPS framework interactions.

Additionally, as DirectDrive traces quickly become especially large, we require interactivity of the visualization to be an integral part of the new toolchain.

5.2 Tooling

During research for some good potential visualization software, we came across Perfetto [7]. Perfetto excels in providing a comprehensive platform for performance tracing and analysis, making it ideal for visualizing network interactions. Although Perfetto is designed for program execution traces and similar tasks, we found that by representing ranks as threads, one could generate a legible and interactive visualization. For tasks like ours, where visualizing the flow and dependencies between network interactions is crucial, Perfetto's fine-grained data collection and versatile tracing capabilities allow for in-depth analysis. This enhances efficiency and effectively diagnoses potential issues in our translation from IO interactions to network interactions.

Figure 5.3: Interactive visualization of two reads using *Simple* mode

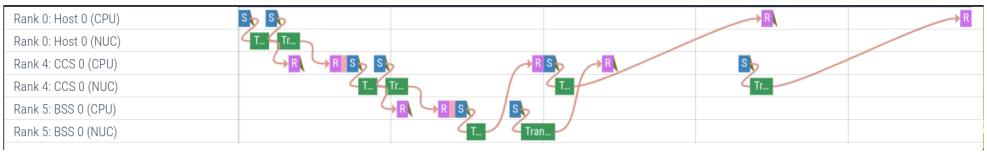


Figure 5.4: Interactive visualization of two reads using *Advanced* mode

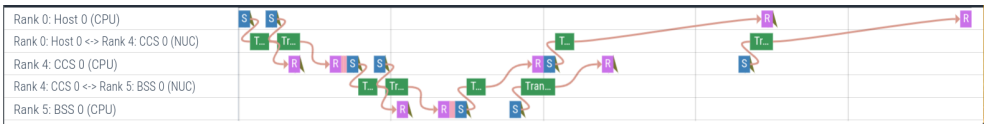


Figure 5.5: Interactive visualization of two reads using *Expert* mode

5.3 Modes

As already mentioned above, we introduced three different modes:

- `Simple` (Fig. 5.3): Most basic mode available. Shows `send`, `recv`, and `calc` blocks.
- `Advanced` (Fig. 5.4): Additionally shows transmit block and splits ranks into NUC and CPU.
- `Expert` (Fig. 5.5): Additionally, have one row for each used direct communication channel, which will be used exclusively for 'Transmit's.

Each mode shows a different level of detail, of the actual execution. This way, the user can decide, depending on their knowledge of the underlying subject, as well as their need for fine-grained details, which visualization might fit their needs best.

5.4 Examples

Below you can find a single example that is explained in depth in all different visualization modes. In case you are interested in more examples, check out the Appendix

Chapter 6

Full Pipeline Usage

This chapter will showcase how one can translate, visualize, and execute a uMass trace file using our DirectDrive translation toolchain. It should be seen as a usage manual and leaves out many technical details in favor of simplicity and conciseness.

6.1 Setup

Clone the repository[8] at DirectDriveSim - Github. Ensure that you have python3 installed (tested with version 3.9 and 3.11)

6.2 Dataset

As a dataset, I have chosen a select few IO operations from the uMass Financial dataset. You can replace this with any other file that uses the uMass trace file specification.

6.3 Generating the GOAL file

To generate the goal file, we can simply call the `trace2goal` helper script, that can be found in the repository[8]. The script will install all the necessary Python dependencies with it on the initial startup.

```
./trace2goal mytrace.trace mytrace.goal
```

6.4 Visualizing a LogGOPSim Execution

Now that we have a goal file, we can visualize what is happening using LogGOPSim.

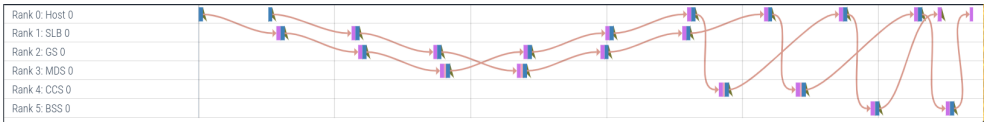


Figure 6.1: Simple Visualization

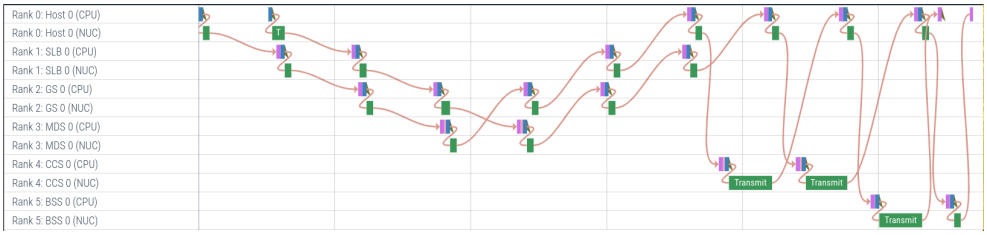


Figure 6.2: Advanced Visualization

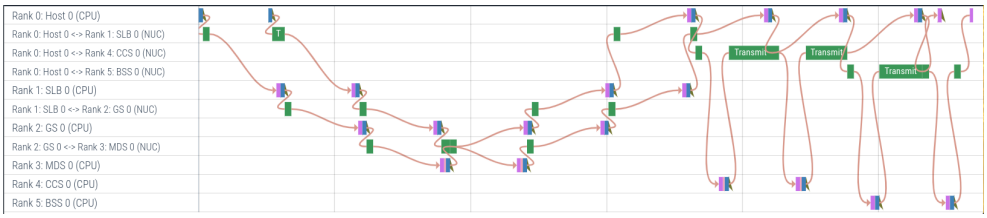


Figure 6.3: Expert Visualization

To do so, we will first need to translate the goal file from its human-readable format to a compact binary format using `txt2bin`.

```
./txt2bin -i "mytrace.goal" -o "mytrace.goal_bin"
```

After that, we can run the simulation using LogGOPSim and the `-V` to generate the intermediate visualization file provided by LogGOPSim.

```
./LogGOPSim -f "mytrace.goal_bin" -V "myvisualization.viz"
```

To generate a visualization, we can simply call the `visualize` helper script, which can be found in the repository[8]. The script will install all the necessary Python dependencies with it on the initial startup.

```
./visualize myvisualization.viz myvisualization.trace
```

By passing either of the `--simple`, `--advanced` or `--expert` flags, you can decide how detailed the visualization will be. See Figures 6.4, 6.4 and 6.4

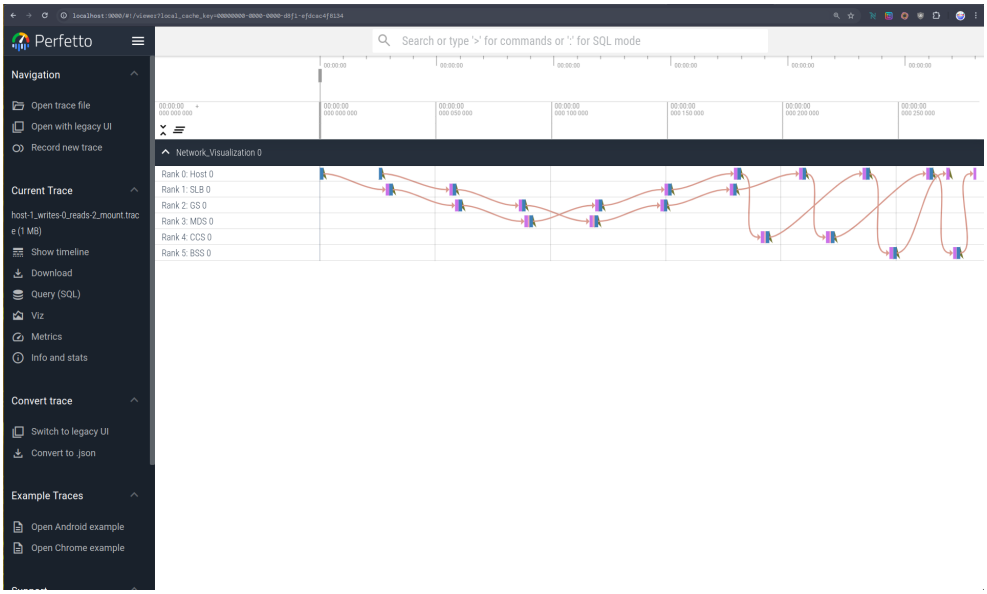


Figure 6.4: Self-hosted perfetto instance, with a simple trace example opened

6.5 Viewing a Visualization

After that, you can preview the visualization either directly on the Perfetto Web UI or using a custom build of Perfetto that allows you to highlight all flows at once! Make sure you have `wget`, `python3` and `unzip` installed to use the self-hosted instance.

```
./utils/host_customized_perfetto.sh
```

The script will host a webserver at . To use it, navigate to that URL after starting the script and waiting for the download to finish.

In the WebUI, you can then open your `myvisualization.trace` file by selecting "Open trace file" in the Navigation section of the menu.

6.6 Execution using htSim

Additionally, it is possible to execute the generated goal file using htSim. To learn more about how to do that, take a look at the custom fork of htSim [9] at SPCL htSim - Github

In Figure 6.6, you can also find a sample performance graph of the execution of the first 100 IO operations from the Financial UMass Dataset. Additionally, you can also find a minor comparison of different Congestion Control algorithms in Figure 6.6.

Figure 6.5: Performance graph of a single htSim Execution

Test	SMaRTT	MPRDMA
WebSearch100	7.56ms	7.89ms
Financial100	28.0ms	29.3ms

Figure 6.6: htSim Parameters: `Fat Tree - 128 nodes - 100Gbps - 4KB MTU`

Chapter 7

Conclusion

7.1 Summary of Key Findings

This study on the simulation of I/O operations within Microsoft Azure's DirectDrive has yielded several important insights and developments:

- **Effective Simulation of I/O Operations**: The utilization of the UMass Trace File Dataset enabled a precise mapping of real-world I/O operations to a simulated environment. This facilitated a comprehensive analysis of how I/O behaviors manifest within DirectDrive, laying a solid groundwork for subsequent evaluations and enhancements of storage systems.
- **Toolchain Development**: The creation and refinement of a simulation toolchain utilizing the GOAL file format, along with advanced simulation frameworks like LogGOPSim and htSim, has markedly improved both the accuracy and efficiency of simulations. This toolchain effectively translates real-world data into formats that are amenable to detailed, nuanced simulation activities.
- **Operational Dynamics Analysis**: Through detailed simulations, substantial insights were gained into the operational dynamics of DirectDrive. This analysis highlighted critical operational processes, including how data mounting, reading, and writing are managed across the storage system, which is vital for optimizing and managing cloud storage architectures.
- **Challenges and Limitations**: The project also identified several challenges, particularly in representing non-deterministic events within a deterministic simulation framework. Overcoming these challenges is essential for enhancing the fidelity and applicability of the simulations, allowing them to mirror the complexities encountered in real-world settings more accurately.

These findings demonstrate that the simulation toolchain developed in this project is robust and capable of significantly contributing to the advancement and optimization of cloud storage technologies, particularly within environments as complex as Microsoft Azure's DirectDrive. Continued development and expansion of this toolchain will facilitate a broader understanding and application to diverse storage technologies and scenarios.

7.2 Assumptions

Due to the limited information available on DirectDrive, there is a limited number of hard facts, and therefore, sometimes, we had to introduce assumptions in order to fully model the network. Because of this, some open questions arose during research, which will also be mentioned in this section.

7.2.1 Network Topology

From the network layout shown in the talk, it is not entirely clear how to arrange endpoints and switches. However, it was often mentioned which components should be spread across fault domains and which are fault-resistant. For example, it was mentioned that the MDS and BSS components should be spread across multiple fault domains. For other components, we were able to make some reasonable assumptions based on the other information that was available to us. For example, although not said explicitly, we came to the conclusion to spread the CSS and GS across fault domains, too.

This is only relevant for htSim, as LogOPSim assumes a complete graph as a base network. In htSim, on the other hand, links and switches are simulated independently, and therefore the topology is relevant for the simulation execution. htSim, per default, assumes a fatTree layout, where all leaves are links, while all other nodes are switches (see Figure 7.2.1). To calculate the position of each component, we used a best-effort, greedy approach that goes in order over the component kinds (Host, SLB, GS, MDS, CCS, BSS) and tries to place each component greedily as close as possible to its target position t_{pos}. id_{comp} has to be a unique number in the range $[0, count_{comp})$, describing the unique id for each component group.

$$t_{pos} = (id_{comp} + 1) * no\_ranks / (count_{comp} + 1)$$

The order of component kinds has been chosen based on the expected number of components and their fault tolerance. The reasoning behind this is quite simple: Component Kinds that are less redundant and have a smaller number of components should be placed first to increase their chance of being placed around their optimal position t_{pos}. t_{pos}, on the other hand, partitions the

leaves into $count_{comp}$ groups and places one of each component in the center of this group.

```
# Spread all components evenly across the network
no_total_ranks = self.get_total_ranks()
positions = [False] * no_total_ranks

def spread_across_network(kind, count):
    fac = no_total_ranks / (count + 1)
    for i in range(count):
        val = f'{kind}{i}'
        pos = round((i + 1) * fac)
        if positions[pos]:
            pos_l = (pos - 1) % no_total_ranks
            pos_r = (pos + 1) % no_total_ranks
            while positions[pos_l] and positions[pos_r]:
                pos_l = (pos_l - 1) % no_total_ranks
                pos_r = (pos_r + 1) % no_total_ranks

            if not positions[pos_l]:
                positions[pos_l] = val
            else:
                positions[pos_r] = val
        else:
            positions[pos] = val

spread_across_network('host', host_count)
spread_across_network('slb', slb_count)
spread_across_network('gs', gs_count)
spread_across_network('mds', mds_count)
spread_across_network('ccs', ccs_count)
spread_across_network('bss', bss_count)
```

Figure 7.1: Pseudo Code of the Greedy Component to Network Topology Mapping Process

7.2.2 Operations

Various assumptions were made about the various operations to keep this report organized. Most of these assumptions are sensible follow-ups from the information provided in the talk. Below, you can find all the operations and the various assumptions we made.

Mount Assumptions

1. All responses from the MDS will also be routed through GS and SLB (just like the request)
2. Before any read or write to a disk, a mount operation is required

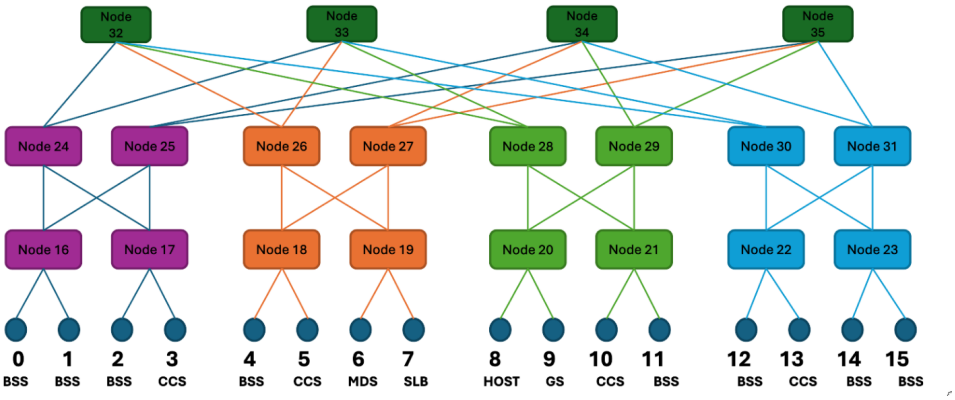


Figure 7.2: Example Mapping of ranks to a fat tree using our greedy approach (8 BSS; 4 CSS; 1 Host; 1 SLB; 1 GS; 1 MDS)

3. The MDS state is kept up to date only through error handling. All other operations do not change their state.

Read Assumptions

1. To keep the initial system simple, error handling is ignored. It was not further discussed in the talk, and many assumptions about how things interact would need to be made. SqN out-of-date errors can also only happen in case of a drive failure or in case of writing with a quorum of $N < M$.

Write Assumptions

1. In the CCS, an $N = M$ quorum is used to wait for responses.

7.3 Challenges

The project faced several challenges, particularly in terms of representing potentially non-deterministic events within the deterministic GOAL format and the performance limitations of simulation tools. Addressing these challenges required us to adopt a best-effort approach, acknowledging that achieving all our objectives was impossible given the constraints. Compromises were necessary to make progress, including prioritizing essential features and functionalities of the simulation toolchain while accepting that some goals remained beyond our reach under the current conditions. Innovative workarounds and modeling strategies were also developed to handle the

non-deterministic events within the deterministic framework. These pragmatic strategies were instrumental in refining the simulation processes and outcomes, enabling us to advance the project as effectively as possible within the existing limitations.

7.3.1 Redundancy

It is not possible to create non-deterministic dependencies in the goal format. This creates a little bit of an issue, as we can't model reactive actions like load balancing or error handling because of this. Therefore, whenever multiple options are available as recipients, we will have to follow a strategy that models a certain behavior to the best of our ability. For example, assume you are simulating a host trying to read certain data from ranks 2, 5, and 15. As you do not currently know the network's real state, you can't guarantee that one of these ranks has the least load without running a full simulation. Therefore, you need some other way to determine what rank to use. We decided to implement multiple heuristics (round-robin, random, first), that should model various behaviours we would like to expect. For example, in the case of the MDS, only one instance should be used at all times unless there are errors in this instance, of course. We would simply decide on the next main MDS in case of an error and continue. As we do not currently model errors, we do not need to worry about the edge case of an error happening, but we can model this behavior easily through the 'first' access method. We can easily simulate the expected behavior by accessing only the first of our available MDS.

7.3.2 Parallel Requests

Again, due to the fact that non-deterministic dependencies are not possible, we can not require only the first of n receives. Instead, one could pick a certain set of receives at random and only require them to terminate. (The same goes for sends.)

This is also the reason for Assumption 1 in the Write operation. Although one could decide on a random set of BSS to wait for termination, this does not really simulate real-world behavior and is a bad approximation at best. So, instead, we choose to only support the full quorum case in such cases.

7.4 Open Questions

As already highlighted various times before this, David Kramer's talk [3] was one of the only resources available for this report at the time of writing this. Therefore, although the report goes into quite some detail from time to time, a lot of open questions arose. Below, you will find a list of some

of the open questions that arose during this project; although they can't be answered at the time of writing this, they might come in handy in the future:

7.4.1 Question 1

Do we communicate the new SqN (Sequence Number) to the MDS, or are SqNs optional for the first read? Mount Assumption 3 prevents the MDS from having an accurate SqN. Either the SqN is optional for (the first) read operation, or we would have to somehow synchronize the SqN with the MDS on every write operation.

7.4.2 Question 2

How does error handling specifically work? According to the talk, the error handling in DirectDrive is quite sophisticated and typically centrally handled through the MDS. However, how errors are handled in detail has not been discussed, and it is only known that in case of an error, the MDS will be notified and ensure that concrete actions to resolve this error are taken.

7.5 Future Work

The research presented in this report opens several pathways for further exploration and enhancement of the simulation capabilities for Microsoft Azure's DirectDrive. Key areas identified for future work include:

- **Support for Other Network Storage Models**: Extending the simulation framework to include other network storage models would enable comparative analysis, providing a broader perspective on the efficiencies and performances of various cloud storage solutions.
- **Performance Evaluation**: Further studies should focus on evaluating the performance impacts of different network topologies and access patterns on DirectDrive. This could lead to optimizations that enhance system performance and user experience.
- **Incorporation of Error Handling**: Developing simulations that include error scenarios would offer insights into the robustness and reliability of DirectDrive under adverse conditions, contributing to the development of more resilient storage systems.

These efforts will not only deepen the understanding of DirectDrive's operations but also enhance the reliability and efficiency of cloud-based storage systems.

Bibliography

[1] T. Hoefler, T. Schneider, and A. Lumsdaine, "Loggopsim: Simulating large-scale applications in the loggops model," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 597–604.

[2] T. Hoefler, C. Siebert, and A. Lumsdaine, "Group operation assembly language-a flexible way to express collective communication," in *2009 International Conference on Parallel Processing*, IEEE, 2009, pp. 574–581.

[3] Greg Kramer. "Direct Drive - Azure's next-generation block storage architecture," Microsoft. (Jan. 3, 2023), [Online]. Available: https://storagedeveloper.org/events/agenda/session/347 (visited on 02/13/2024).

[4] P. Antonopoulos *et al.*, "Socrates: The new sql server in the cloud," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1743–1756.

[5] UMass Trace Repository. "Storage - umass trace repository." (2023), [Online]. Available: https://traces.cs.umass.edu/index.php/storage/storage (visited on 03/05/2024).

[6] M. Handley *et al.*, "Re-architecting datacenter networks and stacks for low latency and high performance," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17, Los Angeles, CA, USA: Association for Computing Machinery, 2017, pp. 29–42, ISBN: 9781450346535. DOI: 10.1145/3098822.3098825. [Online]. Available: https://doi.org/10.1145/3098822.3098825.

[7] Perfetto Team, *Perfetto*, https://perfetto.dev/, Accessed: 2024-04-18, 2024. [Online]. Available: https://perfetto.dev/.

[8] P. Jordan, *DirectDriveSim: Simulation Toolchain for Microsoft Azure's DirectDrive*, https://github.com/TrimVis/DirectDriveSim, Accessed: 2024-05-03, 2024.

[9] T. Bonato, *htSim Network Simulator: Fork with GOAL support*, https://github.com/spcl/HTSIM, Accessed: 2024-05-03, 2024.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

DirectDrive: Simulating IO operations in an attached storage network

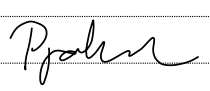
Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Jordan	Pasquale

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zürich, 05.05.2024	

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.