UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale in Ingegneria Informatica

Virtualized FPGA-acceleration of Distributed File Systems with SR-IOV and Containerization

Anno Accademico 2023/2024

Relatore
Ch.mo prof. Alessandro Cilardo
Ing. Vincenzo Maisto

Candidato
Biagio Spena
matr. M63001373

Abstract

Apache Hadoop software library is an open-source framework which takes advantage of distributed computing to process massive datasets across physical clusters. The development and the deploy of a physical cluster has high requirements like a careful configuration of each node, efficient resource allocation and maintenance costs. Reliability of data stored in the cluster, is guaranteed by replication or by Erasure Coding, the second one is accelerated by the FPGA's parallel computing capabilities. An alternative solution to a physical cluster, in order to overcome its strict limits, is the virtual cluster. A virtual cluster can be built through the exploiting of technologies like Container and Device Virtualization. This Thesis project aims to develop a Virtual Cluster based on FPGA Acceleration for a Distributed File System, using technologies such as Hadoop, Docker Container and Single Root I/O Virtualization.

Contents

Chapter 1

Introduction

In the field of processing large data sets efficiently, a topic that has gained popularity in recent years is that of **Distributed Computing**. Distributed computing[1] is a methodology where processing and data storage are distributed across **multiple nodes**, instead of being managed by a single device. These multiple nodes are enclosed under the name of **Distributed System**: a distributed system is one in which components located at networked computers communicate and coordinate their actions only by passing messages[1]. Through this mechanism, distributed computing provides high scalability, reliability and flexibility. Within a distributed system, data storage is provided by the **Distributed File System** (**DFS**)[1]. A distributed file system is a distributed system that allows computers on a network to store and access remote files in the same way as local files. Unfortunately, distributed computing brings with it great challenges both in the **de-**

velopment and **deployment fields**. For example, the communication and coordination of all the nodes, among the network, are prone to component failures. It is necessary to implement **data consistency**. Besides, a distributed system requires **careful configuration** of each node to ensure they can communicate efficiently.

In recent years, a very popular trend has been the use of **FPGAs in data-centers**. These types of accelerators are excellent for applications that require frequent adjustments, thanks to their ability to be reconfigured. One of the strengths of FPGAs is their higher performance compared to traditional CPUs, in specific workloads that require signal processing and real-time data analysis. Another aspect that should not be overlooked is their low energy consumption.

In order to maximize the performance and efficiency of a distributed system, the **Virtualization**[2] was introduced. It implies the emulation of the hardware components (such as CPU, I/O devices, accelerators, etc...) to the nodes of the distributed system. However, the virtualization of hardware accelerators brings up new challenges like **careful resources management** and **performance overhead**, introduced by latices and lowered throughput, due to the addition of virtualization layers.

The next two chapters will introduce **technological details** about case studies in distributed file systems and virtualization fields: **Hadoop Distributed File System**, **Container Virtualization Technol-**

ogy and **Physical Device Virtualization**. Consequently, the **Intel Open FPGA Stack** will be described qualitatively. In the fifth chapter, I will present the **Hardware Virtualization Architecture**, used to create a virtual cluster as an alternative solution to a physical cluster. The last two chapters are dedicated respectively to the **Test Validation**, as a prof of concept of the hardware virtualization architecture, and to che **Conclusions**.

Chapter 2

The Apache Hadoop Framework

Apache Hadoop software library[3] is an **open-source framework**, developed by the Apache Software Foundation, used for distributed processing of massive datasets across physical clusters (known as **Hadoop Clusters**) of commodity hardware. Hadoop provides both **data availability** and **reliability**, due to the data replication mechanism and due to storing data across multiple nodes of the cluster. By sending computations where the data are stored, the framework guarantees the **data locality property**, reducing the bandwidth utilization in a system.

2.1 The Hadoop Architecture

Hadoop has a **master-slave architecture design** characterized by **three main layers**[4] (Figure 2.1):

- **Hadoop Distributed File System (HDFS)** : The Storage Layer.
- **YARN** : The Resource Management Layer.
- **MapReduce** : The Application Layer.

Figure 2.1: Hadoop Architecture

2.1.1 HDFS: The Storage Layer

Hadoop Distributed File System (HDFS)[5] is a **Java-based distributed file system**, running on commodity hardware, which supports a traditional hierarchical file organization. HDFS stores user

data, in a distributed manner, within files by splitting them into one or more chunks (**data-blocks**). Each data-block has a default size of 128 MB and can be replicated through a default replication factor of three (both data-block size and replication factor are configurable). In this way, the file system is **highly fault-tolerant**. The storage layer has a **master-slave architecture** (Figure 2.2) consisting of two daemons (background processes): **NameNode** and **DataNode**.

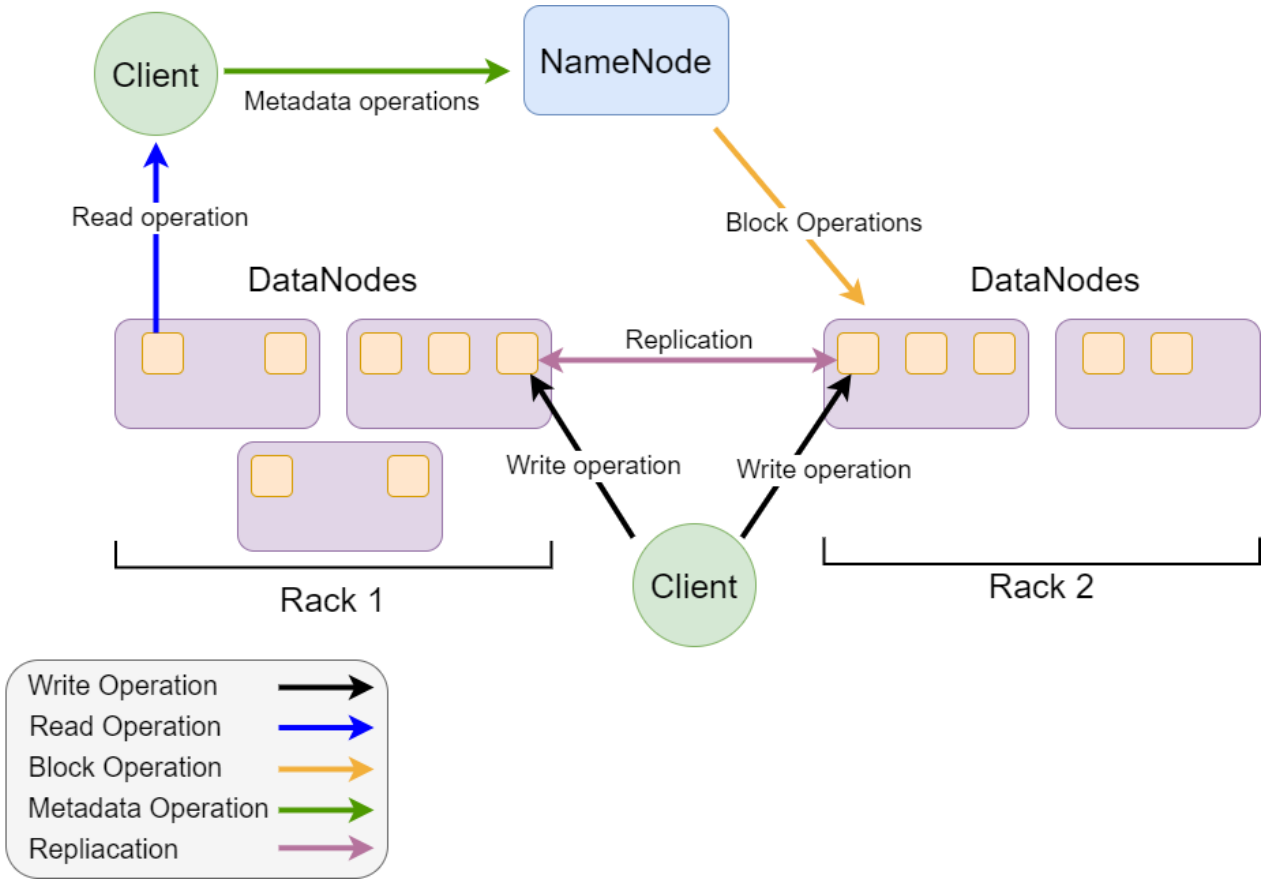


Figure 2.2: HDFS Architecture

The **NameNode** is the **master daemon**, one per hadoop cluster. It stores metadata (like the number of data-blocks, their locations, number of replicas, etc...), executes file system namespace operations

(like opening, closing, and renaming files and directories) and handles clients' access to files. Another namenode's primary function is to assign the data-blocks to DataNodes, making all decisions regarding blocks replication.

The **DataNode** is the **slave daemon**, one per node in the hadoop cluster. It stores the actual data, executing both data-blocks operations (like creation, deletion and replication) and read/write operations from the file system's client. Two or more DataNodes are gathered together in **racks**.

The namenode joins the **file system image** (also known as fsimage, which contains the complete state of the file system at a point in time) and **edits log files** (record the changes that were taken on the file system) only during start-up, in order to update the file system state. This operation causes the increase of the edits log files size over time, and consequently will increase as well the next restart time of the namenode. To avoid these side effects, the **Secondary NameNode** daemon is employed. The Secondary NameNode keeps the edits log files size low by merging the fsimage and the edits log files periodically. Besides, it performs regular checkpoints. Usually, the Secondary NameNode runs on a different machine than the namenode[6].

2.1.2 YARN: The Resource Management Layer

YARN (Yet Another Resource Negotiator)[7] is a **framework for distributed computing**, which distinguishes the resource management from processing components. The Resource Management Layer executes applications (like MapReduce jobs or DAG of jobs) by sending computations where the data are stored. As HDFS, YARN has a **master-slave architecture** (Figure 2.3) composed of two daemons: **ResourceManager** and **NodeManager**.

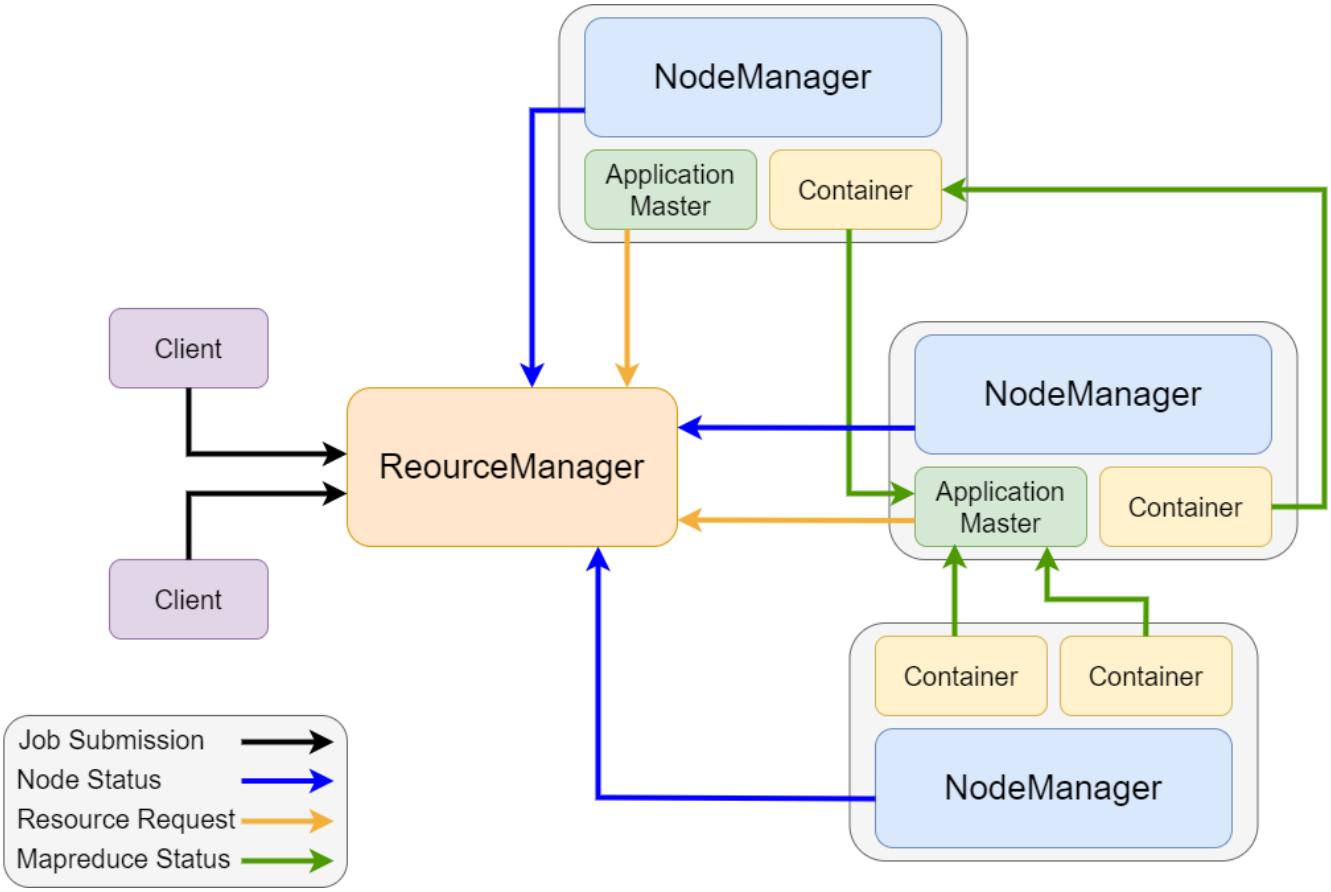


Figure 2.3: YARN Architecture

The **ResourceManager** is the **master daemon**. Its roles are to manage the resources (grouped in containers) among all the appli-

cations in the system and to assign **map tasks** and **reduce tasks** to the NodeManager. The daemon has two main components: the **Scheduler** and the **ApplicationsManager**. The former one, a **pure scheduler**, allocates resources or containers to the running applications, but it does not perform monitoring or tracking of the applications' status. The latter one accepts **job submissions** from the clients and secures resources on a node (negotiates the first container) to launch the specific **ApplicationMaster**.

The **NodeManager** is the **slave daemon**. It monitors the containers' resource usage, reporting it to the resourcemanager, and pursues the node's health in which it is running. The resourcemanager-nodemanager pair is also known as the **data-computation framework**; moreover, from an architectural perspective, each cluster contains a single resourcemanager and one nodemanager for each node in the cluster.

Apart from the master and the slave daemon, there is the **ApplicationMaster**. It is a **framework specific library** (there is one per application), which negotiates resources for the running application from the resourcemanager. Besides, it works with one or more nodemanagers to execute and monitor the tasks.

YARN supports three scheduling policies:

- **FIFO**
- **Capacity scheduler**

- **Fair scheduler**

The **FIFO policy** runs the applications in submission order (ignoring the priority value) by placing them in a queue. The **Capacity policy**[8] (Hadoop default policy) allocates resources to different queues, divided hierarchically, where the applications are scheduled using FIFO policy. Among these queues, there is one dedicated to starting small jobs as soon as they are submitted. The **Fair policy**[9], similar to the capacity policy, allocates resources fairly, taking into account the job priority.

2.1.3 MapReduce: The Application Layer

MapReduce[41011] is a **software framework** for writing applications (which do not have to be written in Java) that process large datasets in a parallel way on large clusters, by working on sets of <**key, value**> **pairs**. A **MapReduce Job**, which is a unit of work executed by the framework, takes as inputs fixed-size splits (divided by Hadoop) called **input splits or splits**. The job is divided into **map tasks** and **reduce tasks**, each task adopts a **user-defined map/reduce function** implemented through interfaces and/or abstract-classes. MapReduce creates a map task for each split, while the number of reduce tasks can be set by the user. In fact, by setting the latter as zero, no reduction will be executed.

A mapreduce job is executed in three phases 2.4 :

- **Map Phase**

- **Shuffle & Sort Phase**
- **Reduce Phase**

In the first one, the job maps the input <key, value> pairs to zero or multiple **intermediate <key, value> pairs**. In the second phase, the output values from the map phase are fetched and grouped by keys. Finally, in the third one, the set of intermediate values is reduced into a **smaller set** and represents the job output.

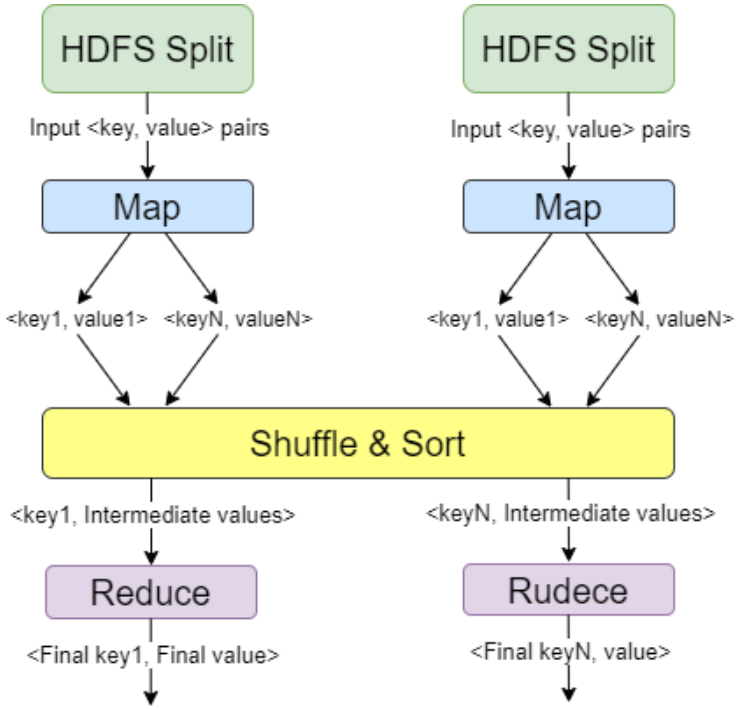


Figure 2.4: MapReduce Job Execution

2.2 The Hadoop Cluster

The **Hadoop Cluster** is a computational cluster used for storing and analyzing huge amounts of unstructured, structured and semi-

structured data in a distributed computing environment, making the cluster highly. It is highly configurable through parameter settings in the **site-specific configuration files**[12] and the supported modes. There are three different modes[13]: the **Standalone Mode** (for testing and debugging), the **Pseudo-distributed Mode** (for development and debugging) and the **Fully Distributed Mode**. In Standalone Mode, Hadoop is configured to run as a single node cluster (Figure 2.5), which is used primarily for testing and debugging. In the second mode, a **pseudo-distributed cluster** is utilized, where all the daemons run as separate processes on separate JVMs. In the Fully Distributed Mode, Hadoop is configured to run as a multiple node cluster (Figure 2.6), with data are distributed across nodes. The Multi Node Cluster operates on a **master-slave architecture**: master node assigns tasks to the slave nodes and stores metadata, managing the resources across the cluster. In contrast, slave nodes store data and perform the computing.

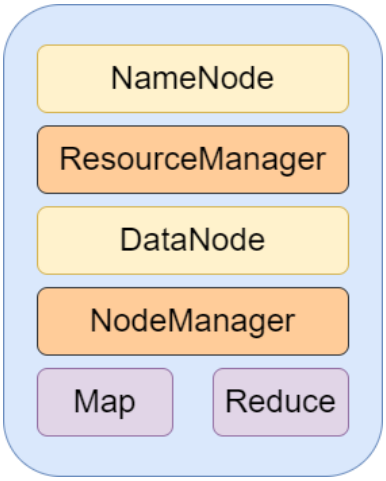


Figure 2.5: Single Node Hadoop Cluster

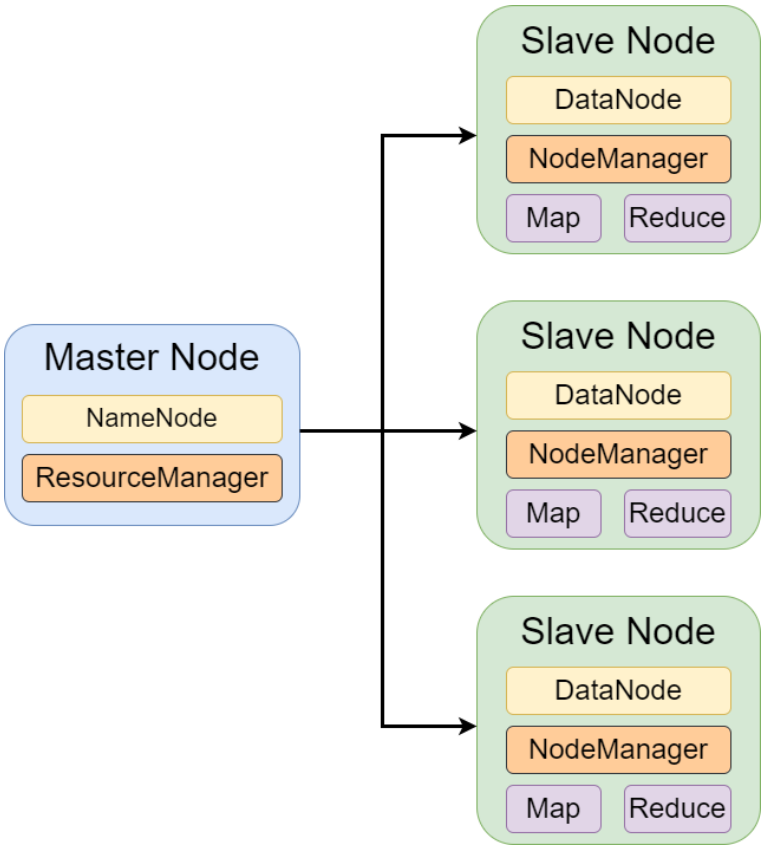


Figure 2.6: Multiple Node Hadoop Cluster

The Hadoop cluster provides high scalability[14] by supporting two method:

- **Scaling-out**: Adds more hardware to the existing cluster.
- **Scaling-up**: Increases the resources of individual machines in the Hadoop cluster.

2.3 Erasure Coding

The replication factor of three generates a **200% overhead**[15] in storage space and other resources. In order to overcome this overhead, is implemented the **Erasure Coding** (**EC**)[16] method: fault-tolerance strategy based on **redundant coding**, which provides effective recovery using the remaining data when some data are lost. EC is based on two phases: **Encoding Phase** and **Decoding Phase**. In the Encoding Phase (Figure 2.7), a specific encoding matrix is used to generate **m check blocks** from **k original data blocks**, with both check blocks and data blocks distributed across various nodes. In the Decoding Phase (Figure 2.8): when a node fails, data recovery can be achieved by reading any k blocks from the remaining nodes. An example of erasure codes is **Reed-Solomon** (**RS**)[17], an error correction scheme that can fully recover from as many errors as the amount of added coding information.

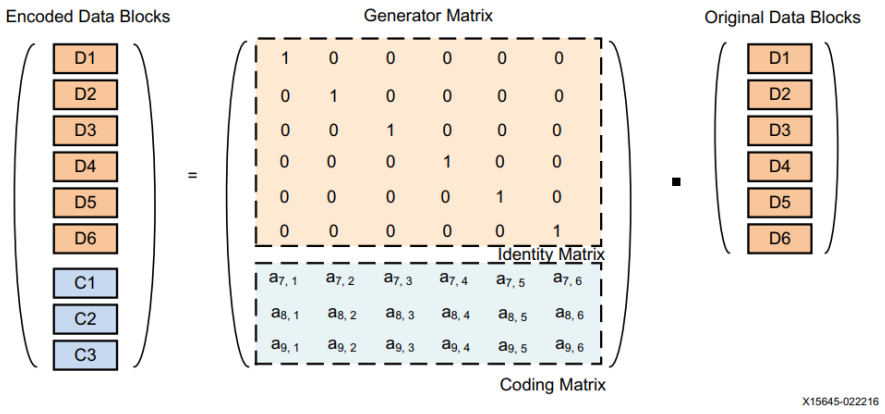


Figure 2.7: Encoding of Erasure Codes

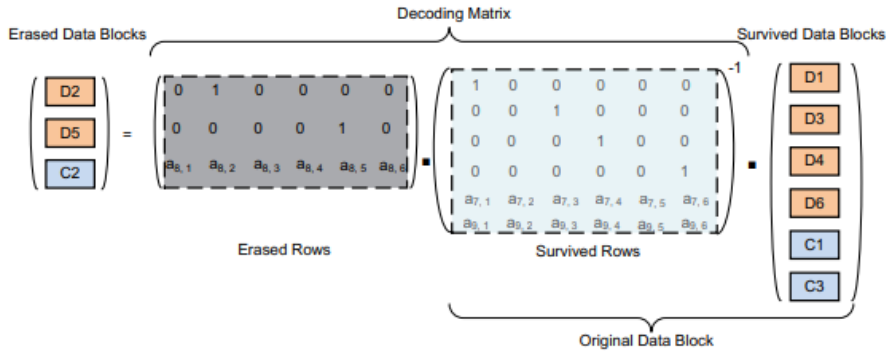


Figure 2.8: Dencoding of Erasure Codes

In the storage system, an EC implementation is a **Redundant Array of Inexpensive Disks** (**RAID**)[15]. RAID implements EC by:

- **Striping**: Divide logically sequential data (file) into smaller units, known as **striping cells**. For each cells, a specific number of **parity cells** are calculated and stored.
- **Storing**: Store consecutive striping cells on different disks.

Any error in a striped cell can be retrieved through decoding calculations based on the available data and parity cells.

2.3.1 Erasure Coding in HDFS

The introduction of EC can enhance storage efficiency while maintaining similar data durability as traditional replication-based HDFS deployments. The distributed file system supports EC through the following elements[15]:

- **NameNode Extensions**: Striped HDFS files are logically made up of block groups, each containing internal blocks. To minimize NameNode memory usage from these additional blocks, the ID of a block group can be obtained from the ID of any of its internal blocks.

- **Client Extensions**: The client read and write paths were improved to operate on multiple internal blocks within a block group simultaneously.

- **DataNode Extensions**: Each Datanode runs an **ErasureCodingWorker task** for background recovery of failed erasure coded blocks. When failes EC blocks are detected by the NameNode, it picks a DataNode to execute the recovery.

- **Erasure Coding Policies**: These define how to encode/decode a file. Each policy consists of an **EC schema** and the **size of striping cell**. The EC schema specifies the number data and parity blocks, as well as the codec algorithm. The size of striping cell defines the granularity of striped reads and writes. Example of policies supported are RS-3-2-1024k and RS-6-3-1024k.

HDFS erasure coding can use **Intel Intelligent Storage Acceleration Library** (**ISA-L**) to accelerate encoding and decoding processes. ISA-L, an open-source collection of optimized low-level functions, contains a high-performance block Reed-Solomon type erasure codes, op-

timized for Intel AVX and AVX2 instruction sets.

2.3.2 Cluster and Hardware Configuration for EC

EC demands for a minimum of as many DataNodes in the cluster as the configured EC stripe width[15]: the EC policy RS[3:2], requires a minimum of five (data blocks + parity blocks) DataNodes. To preserve rack fault-tolerance, it's crucial to have a sufficient number of racks. A formula to calculate the sufficient number of racks is: (data blocks + parity blocks) / parity blocks. For EC policy RS[3:2] are required a minimum of three racks. For clusters that do not meet the minimum number of racks, HDFS will still try to distribute a striped file across multiple nodes to preserve node-level fault-tolerance.

Chapter 3

Virtualization Technology

Virtualization grants the ability to create multiple emulated environments, known as **virtual machines (VMs)**, from a single physical hardware. VMs, running on the same machine, share the same physical resources (offering their more efficient use) and run their own operating system (**guest OS**). Virtual machines are managed by the **virtual machine monitor (VMM)**, or **hypervisor** (Figure 3.1).
The hypervisor is the **only privileged component**, which manages the physical hardware on behalf of VMs. VMs guarantee a good degree of reliability, security and isolation: the guest OS within the VM "sees" an isolated physical machine, and cannot damage the other VMs or the VMM.

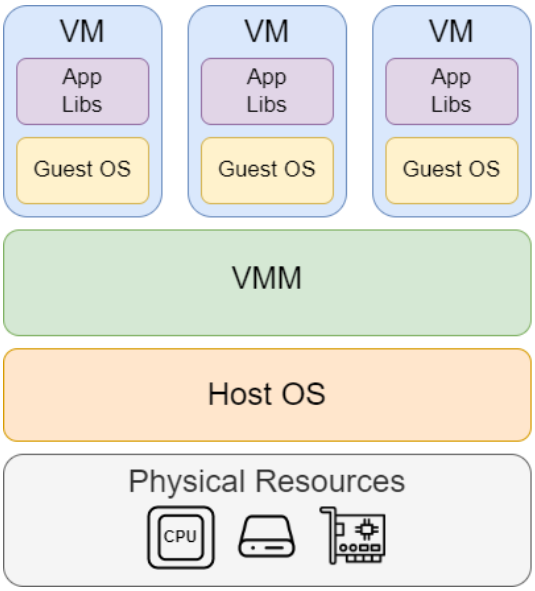
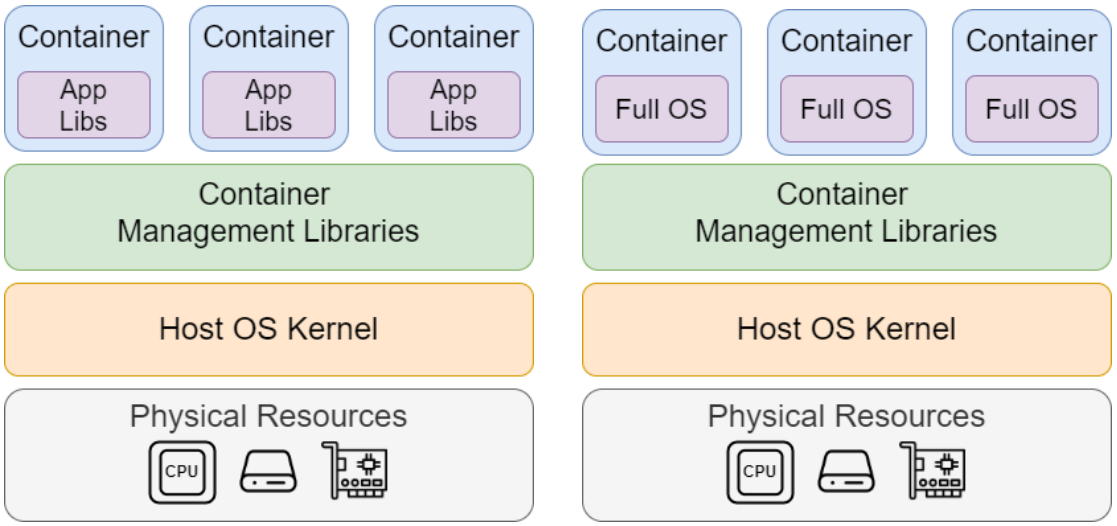


Figure 3.1: Virtual Machine Architecture

3.1 Container Virtualization Technology

In recent decades, **Container Virtualization** (or **Containerization**) has become very popular virtualization technology because it is capable of solving various challenges in software development. The Containerization[18] is **lightweight virtualization**, where software applications run on isolated computing environments, the **containers**. Each container shares the **same host machine's kernel**, avoiding the overhead of virtualization, and the host machine resources. Containers use two **Linux kernel features** to ensure resource isolation and management: **cgroups** and **namespaces**. Cgroups (or Control Groups) limit the amount of resources (like CPU, memory, network, ecc..) that a container can use. Instead, namespaces run multiple isolated instances of system resources, which means that a process in a namespace can not interact directly with other processes.



(a) Application Container Architecture

(b) System Container Architecture

Figure 3.2: Container Architectures

There are two container categories[18]:

- **Application Container** (Figure 3.2a): It contains and runs a single application per container (along with its dependencies, libraries, etc...), each application does not interfere with other applications. The application container guarantees easier portability and deployment of the applications, reducing concerns about compatibility on different platforms.

- **System Container** (Figure 3.2b): By running a full operating system and multiple processes at the same time, a system container is similar to a virtual machine.

Compared to a virtual machine, a container is **smaller in capacity** and **faster to start**. On the other hand, containers do not emulate

	Virtual Machine	**Container**
Boot Time	20 mins	2 secs
App Development Complexity	* Deploy in minutes * Lives for weeks	* Deploy in seconds * Live for minutes/hours
Development Complexity	Need to know: * Operating System * Runtime Enviroment * Application Code	Need to know: * Runtime Enviroment * Application code
Scaling	Takes our	Takes seconds

Table 3.1: Performance Comparison between Virtual Machine and Container

physical devices and do not provide the same level of isolation as a virtual machine (Table 3.1).

3.1.1 Application Container: Docker

An example of an application container is the **Docker Containerization Technology**[19]. Docker is an open platform for developing and running applications, which includes everything an application needs in containers. The docker architecture is a **client-server architecture** (Figure 3.3): **Host**, **Registry** and **Client**

The first one acts as an environment for running applications. It includes different **docker objects**:

- **Image**: A **hierarchical read-only template** to build containers. An image is divided into layers, where only the top-most layer can be modified. Developers can build their own images by writing a **Dockerfile**, and each dockerfile instruction creates

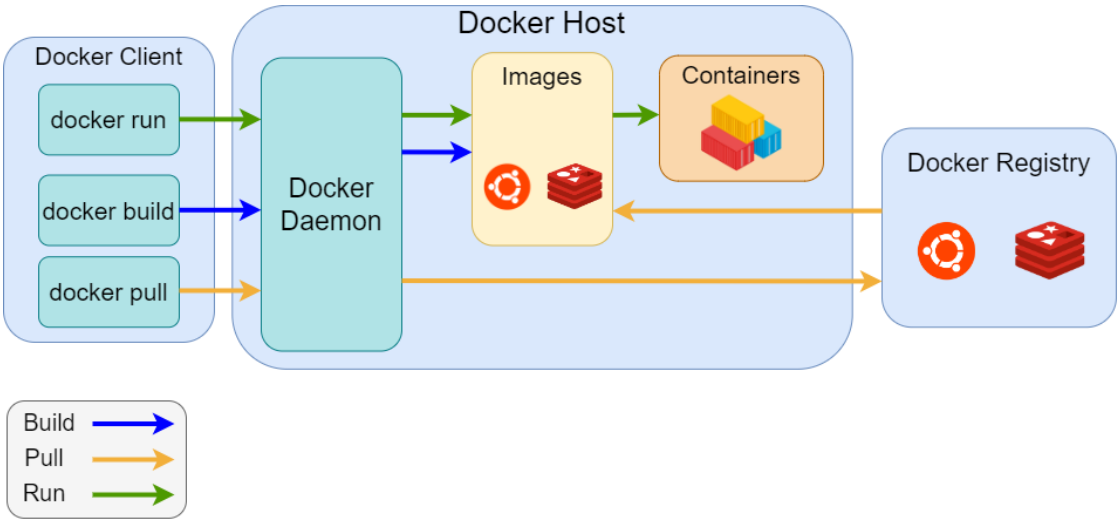


Figure 3.3: Docker Container Architecture

an image layer.

- **Daemon**: A background process that manages containers, images, networks, etc...
- **Networks**: Enable **communication between containers**.
- **Storage**: Defines the property of storing data in the writable layer of the container. Writing into the layer requires a storage driver and the data are erased when the container stops working, so docker provides volumes and bind mounts as an alternative.
- **Containers**

The **registry** is a **stateless server-side application** which stores and shares **images**. The container technology provides a default public registry with stock images (known as **Docker Hub**), moreover users can create their own registry.

The purpose of the **docker host** is to allow developers to work with the **Platform**, by sending docker commands (like docker build, docker run, etc...) to the daemon. Another important function of the client is to make easy the images retrieval from the registries and to run them on the host. A client can be located on the same host machine or on a remote machine of a daemon. Furthermore, it can communicate with more than one daemon.

Another important docker component is the **Docker Engine**[20], an **open-source client-server technology** for building, shipping and running containerized applications. The engine includes a server with **dockerd daemon process**, **REST API** and **client-side command-line interface** (**CLI**). Using the command-line interface, a developer can interact with the dockerd, which contains hosts images, containers, networks and storage volumes.

Docker not only manages the resources of its containers but also their file system, through the **Overlay file system** (or **OverlayFS**)[21]: when a container is started, Docker adds a new layer to the container's file system on top of the base image. In this way, each container has its **own file system**, isolated from the host system and other containers.

3.1.2 System Container: LXC and LXD

Opposed to application containers, there are system containers like **LXC** and **LXD**[2223] (implementation of Linux containers). The first

one serves as a low-level user space interface for the Linux kernel containment features, including commands, templates, library. Its main purpose is to create an environment that resembles a standard Linux installation, without a separate kernel. Instead lxd, usually confused with lxc, uses lxc to create and manage the containers. Like docker, lxd also features a **daemon** running in the background.

3.2 Physical Device Virtualization

The virtualization of a physical device allows to share a single input/output device across multiple VMs[2425]. The device sharing can be performed either in **software** (Figure 3.4) or in **hardware**. In the second case through two different techniques: **Device Passthrough** (Figure 3.5) and **Single Root I/O Virtualization** (Figure 3.6).

3.2.1 Software-based Sharing Approach

The **Software Approach**[2425] applies **emulation** to supply an I/O device to virtual machines and all the traffic, issued by the guest driver, is caught by the hypervisor (Figure 3.4). There exist two models based on the software-based sharing:

- **Device Emulation Model**: Replicates real devices and leverages the drivers in the guest OS. The hypervisor emulates the device, to maintain the compatibility, and handles I/O opera-

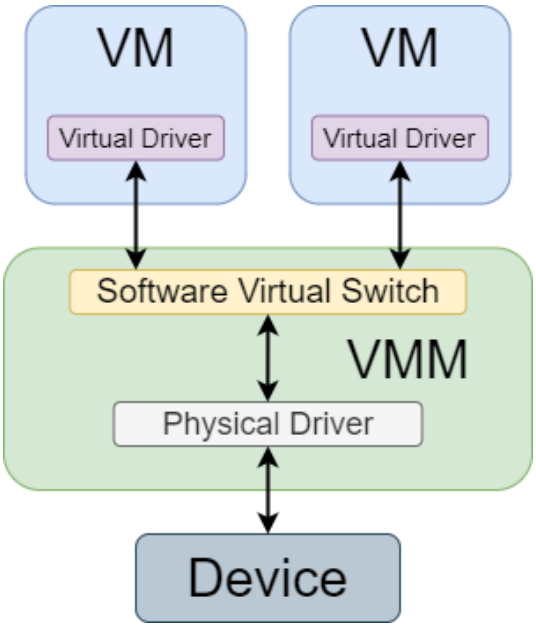


Figure 3.4: Device Software-based Sharing

tions before forwarding them to the physical device.

- **Split-Driver Model**: Uses a front-end driver, within the guest, that operates alongside a back-end driver (communicates with the physical device) in the VMM.

Both device emulation and split-driver offer a portion of the physical hardware functionalities. Additionally, the **virtual software-based switch**, implemented by the virtual machine monitor to route packets to and from virtual machines, may require a CPU overhead (reducing the device throughput).

3.2.2 Device Passthrough

By bypassing the hypervisor, the **Device Passthrough** or **Direct Assignment**[24] (Figure 3.5) ensures the exclusive use of a device by

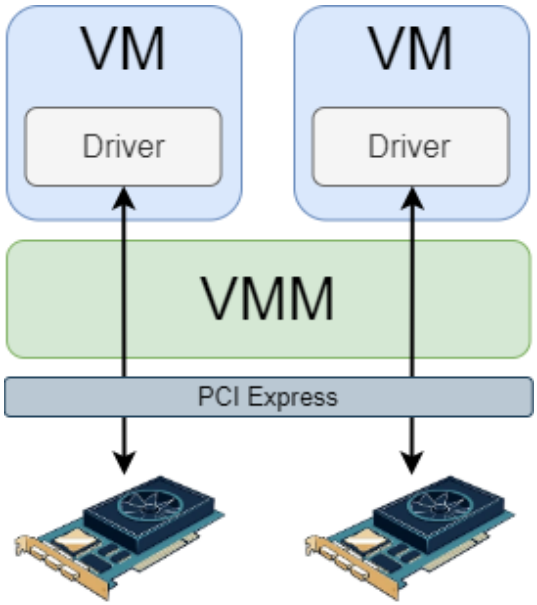


Figure 3.5: Device Passthrough

a virtual machine. To take advantage of direct assignment, hardware support is required; Intel[25] and AMD contribute with their solutions. Intel brings up **Intel VT-x** and **Intel VT-d** hardware support technologies: the first one allows the virtual machine device driver to write directly to registers of the I/O device (such as configuring DMA descriptors). With the Intel VT-d, direct writing is performed by the device in the memory space of the VM. Instead, AMD introduces **AMD-Vi**, which raises the capacity to manage memory and I/O devices directly.

Device Passthrough achieves **near-native performance**, but at the same time has limited scalability because a device can be assigned to one VM.

3.2.3 Single Root I/O Virtualization

The **Single Root I/O Virtualization (SR-IOV)**[25][26] is a hardware specification to allow PCI Express devices to appear as **multiple separate physical devices** (Figure 3.6). SR-IOV leverages Intel VT-x/VT-d and AMD-Vi technologies, just like direct assignment, and also takes advantage of **Virtal Funtion I/O** (**VFIO**)[27]. VFIO driver is a Linux framework, used to provide direct device access to userspace in a secure environment. It handles devices at the **IOMMU groups level**, where the IOMMU group is a set of devices isolated from all other devices in the system. The IOMMU groups are defined by the hardware component IOMMU, also known as **Input-Output Memory Management Unit**. IOMMU translates device-visible virtual addresses to physical addresses in memory, by establishing **memory protection** and **isolation** in systems exploiting hardware-assisted virtualization. Through IOMMU, multiple virtual machines share hardware resources in a safe way and are restrained from accessing memory regions assigned to other virtual machines.

SR-IOV includes two **functions**:

- **Physical Functions (PF)**: **PCIe functions** that manage the physical layer channels of the PCIe device.
- **Virtual Functions (VF)**: **Lightweight PCIe functions** including **part of the hardware resources** and dedicated to I/O operations. The VMM directly assigns one or more VFs

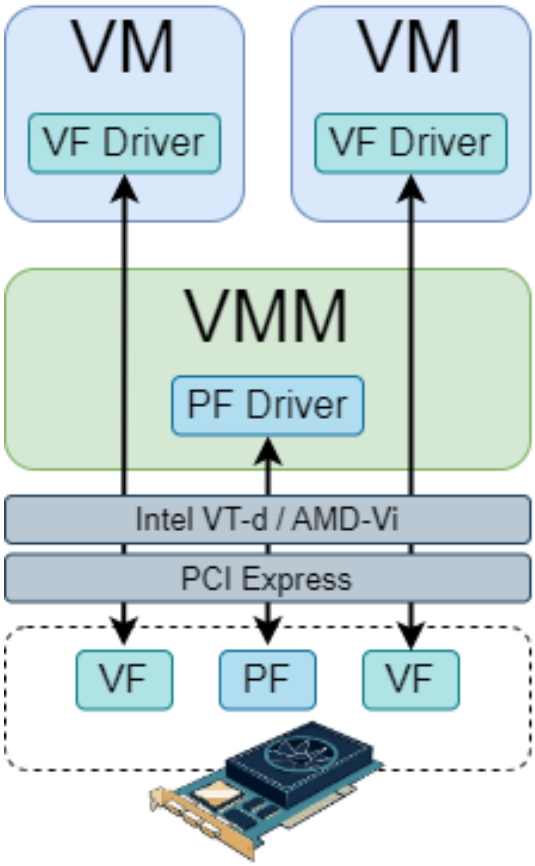


Figure 3.6: Single Root I/O Virtualization

to a virtual machine, granting the latter the access to hardware resources.

This specification brings up **near-native performance** for VMs, reducing the overhead associated with I/O virtualization. Security is improved, as SR-IOV ensures **strong isolation** between VMs sharing the same physical device.

Chapter 4

Intel Open FPGA Stack

This chapter will introduce in a qualitative way the **Intel Open FPGA Stack (OFS)**, a hardware/software stack (Figure4.1) for the development on Intel FPGAs, i.e the **HiPrAcc™ NC220/C220** (**Intel Agilex Low Profile PCIe Card**).

Starting from the bottom, Intel OFS provides:

- **AFU**
- **FIM**
- **PCI Express**
- **Kernel Driver DFL**
- **Opae SDK**

AFU stands for **Accelerator Functional Unit**[28]. It is a hardware accelerator implemented in FPGA logic which offloads a com-

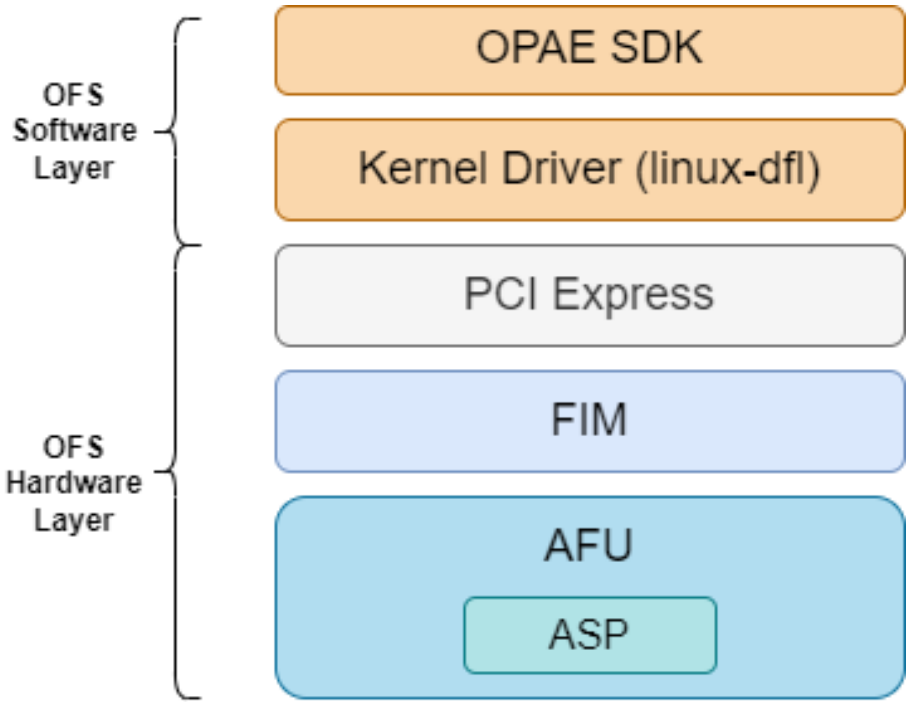


Figure 4.1: Open FPGA Stack

putational operation for an application from the CPU to improve the performance. One of the most important element of the AFU is the **oneAPI Accelerator Support Package** (**ASP**). ASP, basically a **Board Support Package (BSP)**[29], is a set of hardware and software components that guarantees the communication between **oneAPI kernel** (converted into a hardware circuit by the compiler) and the **oneAPI runtime** and **other OFS hardware/software components**. AFUs, and related ports, can be exposed through virtual function (VF) devices via SRIOV (Figure 4.2)[30]. In order to ensure the isolation, each VF only contains one port and one AFU.

The **FPGA Interface Manager** (**FIM**)[28] offers many features such as platform management and standard interfaces to host and AFUs. FIM is located in the the **static region** of the FPGA and contains

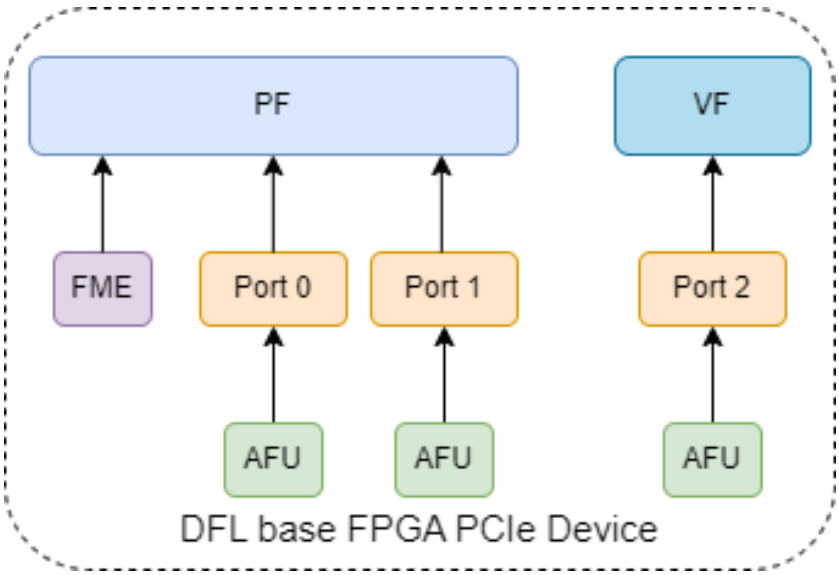


Figure 4.2: VFIO: AFUAs as VFs

the **FPGA Management Engine** (**FME**) and I/O ring.

OPAE SDK[31] is a set of libraries and tools which simplifies the development of software applications and accelerators utilizing the Open Programmable Acceleration Engine (OPAE). OPAE is a software framework designed for managing and accessing programmable accelerators (FPGAs). **OPAE C library**[32], part of OPAE SDK is a **lightweight user-space library** that abstracts FPGA resources within a computing environment. The library aims to abstract both hardware and OS specifics details, displaying FPGA resources as a group of features accessible from within software programs running on the host.

The **Device Feature List** (**DFL**)[30] FPGA framework masks the low layer hardware details, exposes unified interfaces to userspace and guarantees system level management functions such as FPGA recon-

figuration. This framework allows applications to use interfaces to configure enumerate, open and access FPGA accelerators on platforms which implement the DFL in the device memory.

Chapter 5

Virtual HDFS Cluster Architecture

In this chapter, I will introduce the characteristics of the **Physical Cluster Architecture**, which is the starting point for my thesis project. I will then describe the **Virtual HDFS Cluster Architecture** I developed, highlighting the technical aspects.

5.1 Baseline system for distributed HDFS with FPGA acceleration

The **physical cluster architecture** (Figure 5.1) follows a **master-slave paradigm**, where the **master node** coordinates the n-1 **slave nodes**. The first one hosts the **Hadoop Master Daemons** (NameNode and ResourceManager), while the slave node contains the

Hadoop Slave Daemons (DataNode and NodeManager). At the **hardware level**, each node has exclusive access to an Intel FPGA, each of which exposes its own AFU through virtual functions. At the **software level**, there is Intel OFS, which ensures development on proprietary boards. On top of this, there is a software layer, named **Thread Isolation**. Its purpose is to interconnect HDFS with the VF so that the distributed file system has exclusive access to the VF. Once the connection is instantiated, the virtual function will neither be accessible nor visible until it is released by Thread Isolation. By leveraging the **FPGA's parallel computing capabilities**[16], it is possible to accelerate the erasure coding algorithm.

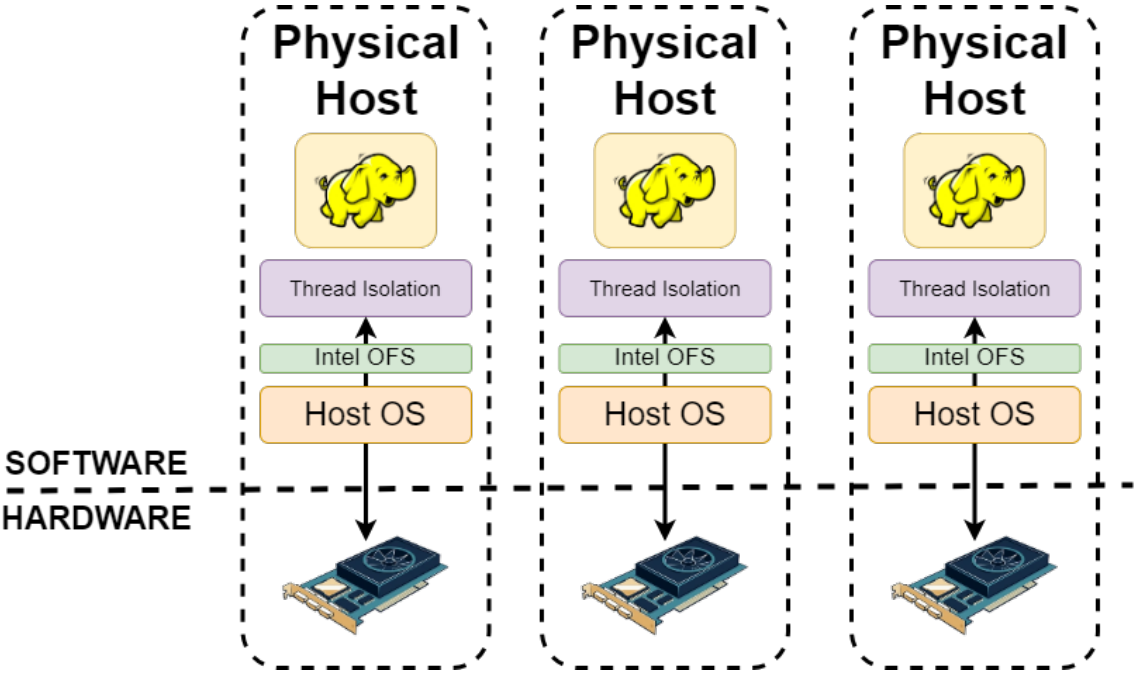


Figure 5.1: Physical HDFS Cluster Architecture

5.2 Virtualized HDFS with FPGA acceleration

Through the use of virtualization techniques, the **Virtual Cluster** differs from the physical cluster due to the presence of a **single physical host** and the use of only one accelerator (Figure 5.2). **SR-IOV** expose the FPGA as multiple VFs, The functionalities of the accelerator are ensured by dedicated hardware support: **IOMMU** and **Intel VT-d**, which provide direct access to the host's memory space by the device. The host's operating system includes the **VFIO driver**, which exposes the device's direct access to the user space and manages the devices at the IOMMU group level. The n physical hosts of the base architecture are replaced by n **Docker containers**. In this case, as well, has been implemented the master-slave paradigm.

5.3 Technical Details

From a technical point of view, the architecture was implemented through the following phases:

- **FPGA Setup**
- **Cluster Deploy**
- **Cluster Init**

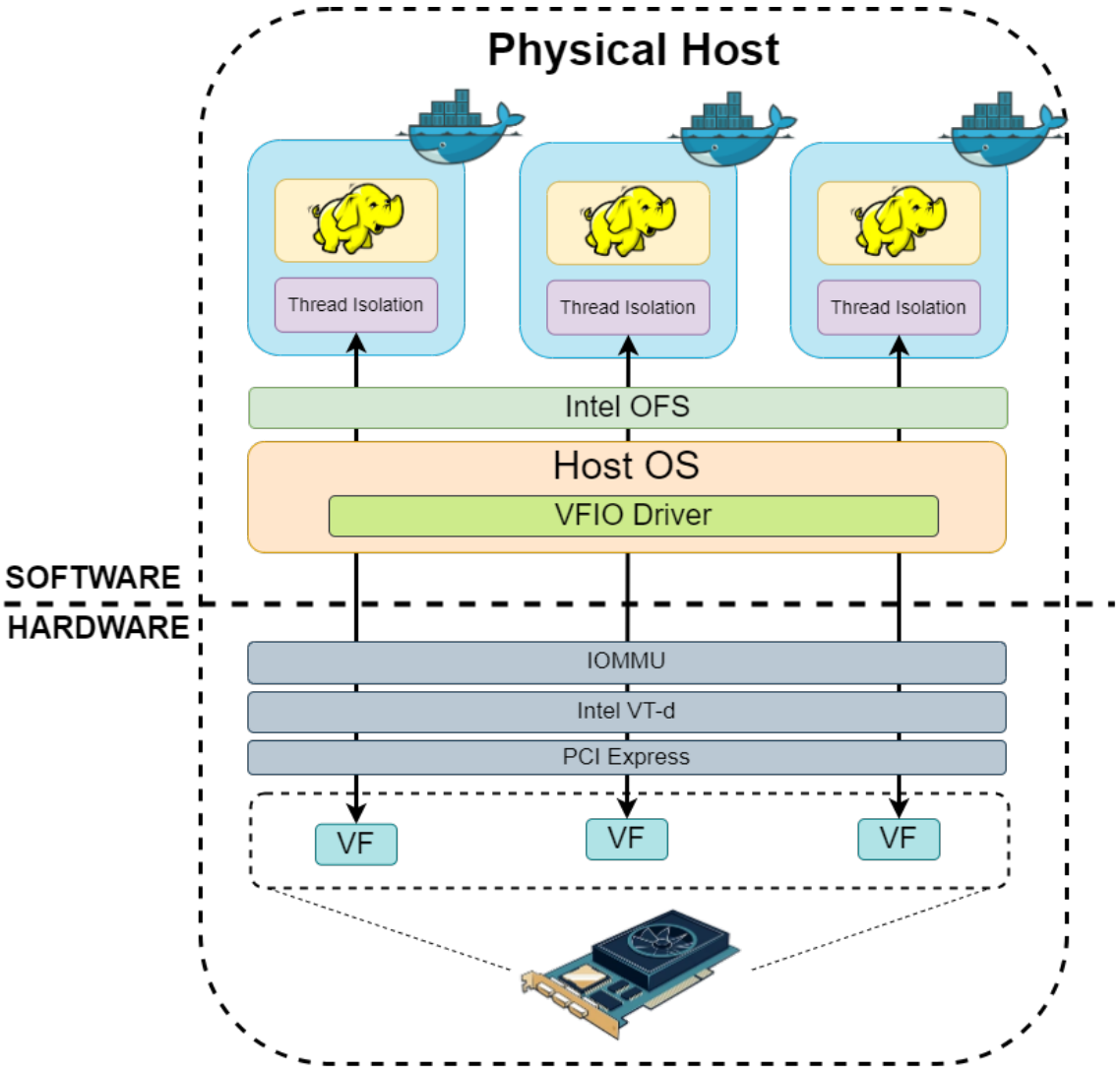


Figure 5.2: Virtual HDFS Cluster Architecture

The first and second phases are carried out outside the containers, while the third takes place inside them.

5.3.1 FPGA Set up

The **FPGA Set up Phase** guarantees the accelerator configuration, using the following OPAE SDK tools:

- *pci_device*

- *opae.io*
- *fpgainfo*
- *rsu*

pci_device one is a tool designed to assist with common tasks for **managing PCIe devices** and **drivers**, like creating or destroying virtual functions as shown in Command 5.1:

```
sudo pci_device PCI_ADDR vf NUMBER_OF_VFS
```

Listing 5.1: Create or Destroy VFs

opae.io is an interactive Python environment, used for userspace access to PCIe devices through **vfio-pci driver**. Among its functionalities, opae.io allows to:

- View the accelerator devices on the system.
- Unbinds the device from its current driver and binds it to vfio-pci.
- Release the PCIe device from vfio-pci and return it to the previous driver.

using commands in 5.2:

```
opae.io ls [-v,--viddid VID:DID]
opae.io init [-d PCI_ADDR USER:[GROUP]]
```

```
opae.io release [-d PCI_ADDR]
```

Listing 5.2: opae.io Commands

fpgainfo displays the PCI device information coming from *sysfs* files. *rsu* triggers a power cycle of the fpga using Command 5.3 :

```
rsu fpga --page=(user1|user2|factory) [PCIE_ADDR]
```

Listing 5.3: rsu Command

Below are the outputs of the commands *opae.io ls* and *fpgainfo port* before (Code 5.4) and after (Code 5.5) the VF binding:

```
$ opae.io ls
[0000:01:00.0] (0x8086:0xbcce 0x8086:0x1771)
  Intel Open FPGA Stack Platform NC220 (Driver:
   dfl-pci)
$ fpgainfo port
//****** PORT ******//
Interface                       : DFL
Object Id                       : 0xEB00000
PCIe s:b:d.f                    : 0000:01:00.0
Vendor Id                       : 0x8086
Device Id                       : 0xBCCE
SubVendor Id                    : 0x8086
SubDevice Id                    : 0x1771
Socket Id                       : 0x01
```

```
Accelerator Id                    : 56e203e9-864f
  -49a7-b94b-12284c31e02b
Accelerator GUID                  : 56e203e9-864f
  -49a7-b94b-12284c31e02b
```

Listing 5.4: VF Not Bound

```
$ opae.io ls
[0000:01:00.0] (0x8086:0xbcce 0x8086:0x1771)
  Intel Open FPGA Stack Platform NC220 (Driver:
   dfl-pci)
[0000:01:00.1] (0x8086:0xbcce 0x8086:0x1771)
  Intel Open FPGA Stack Platform NC220 (Driver:
   vfio-pci)
[0000:01:00.2] (0x8086:0xbcce 0x8086:0x1771)
  Intel Open FPGA Stack Platform NC220 (Driver:
   vfio-pci)
$ fpgainfo port
//****** PORT ******//
Interface                         : DFL
Object Id                         : 0xEB00000
PCIe s:b:d.f                      : 0000:01:00.0
Vendor Id                         : 0x8086
Device Id                         : 0xBCCE
SubVendor Id                      : 0x8086
```

```
SubDevice Id                          : 0x1771
Socket Id                             : 0x01
Accelerator Id                        : 56e203e9-864f
  -49a7-b94b-12284c31e02b
Accelerator GUID                      : 56e203e9-864f
  -49a7-b94b-12284c31e02b
//****** PORT ******//
Interface                             : DFL
Object Id                             : 0xEB00000
PCIe s:b:d.f                          : 0000:01:00.1
Vendor Id                             : 0x8086
Device Id                             : 0xBCCE
SubVendor Id                          : 0x8086
SubDevice Id                          : 0x1771
Socket Id                             : 0x01
Accelerator Id                        : 56e203e9-864f
  -49a7-b94b-12284c31e02b
Accelerator GUID                      : 56e203e9-864f
  -49a7-b94b-12284c31e02b
//****** PORT ******//
Interface                             : DFL
Object Id                             : 0xEB00000
PCIe s:b:d.f                          : 0000:01:00.2
```

```
Vendor Id                                : 0x8086
Device Id                                : 0xBCCE
SubVendor Id                             : 0x8086
SubDevice Id                             : 0x1771
Socket Id                                : 0x01
Accelerator Id                           : 56e203e9-864f
  -49a7-b94b-12284c31e02b
Accelerator GUID                         : 56e203e9-864f
  -49a7-b94b-12284c31e02b
```

Listing 5.5: VF Bound

5.3.2 Cluster Deploy

The **Cluster Deploy Phase** requires three steps:

- **Build the Docker Image**
- **Create the Docker Network**
- **Create the Docker Containers**

The first step is the process of building an image based on the **Dockerfile**. It contains the instructions to install libraries and tools, like Protobuf and OPAE SDK. The latter also plays a role within the containers because, once the VF is associated with a specific container, it is no longer visible from the host.

Once the image is ready, the next step is the creation of docker network. It is necessary to ensure communication between the Docker containers, so that the hadoop master daemons can communicate with the hadoop slave daemons.

Finally, the containers are created using *docker run* command, along with different flags to properly configure them. An example is `--device flag` to assign one or more VFs to the containers, by passing the following elements from the */dev* file system directory:

- */dev/vfio/iommu_group*: It is the IOMMU group associated with the specific VF.
- */dev/dfl-fme.0*: Part of the operating system interface for managing FPGAs, it represents an instance of FME associated with an FPGA device.
- */dev/dfl-port.0*: Part of the operating system interface for managing FPGAs, it represents an instance of a port associated with an FPGA device.
- */dev/vfio/vfio*: It provides secure and direct access to PCIe hardware resources.

The docker run command was also enhanced with

- `-mount flag`: To configure a **tmpfs filesystem** in order to use the **HugePages**, limiting the memory to 1 GB and setting the appropriate permissions.

- `--ulimit flag`: To lock unlimited amounts of memory.

In order to give access to host to the **Web UI** of the hadoop daemons, the following ports are exposed:

- Port 9870: NameNode
- Port 8088: ResourceManager
- Port 19888: MapReduce JobHistory Server

5.3.3 Cluster Init

The last phase includes the following sub-stages:

- Set up passphraseless ssh to facilitate the communication between the nodes of the Hadoop cluster.
- Start the Thread Isolation software module , to attach the VF to HDFS.
- Start the Hadoop daemons.

Chapter 6

Experimental Validation

With the aim to validate the virtual architecture, an **experimental validation** is executed through **TestDFSIO benckmark**. It is a read and write test for HDFS, used to identify file system's rate, in terms of I/O. The TestDFSIO syntax for running a test is:

```
$ hadoop jar $HADOOP_HOME/hadoop-*test*.jar
  TestDFSIO -read | -write | -clean [-nrFiles
   N] [-fileSize MB] [-resFile resultFileName
  ] [-bufferSize Bytes]
```

Listing 6.1: TestDFSIO Command

whit the following flags:

- *-write*: Initializes a write test.
- *-read*: Initializes a read test.
- *-clean*: Cleans the output directories.

- *-nrFiles*: Defines the number of files used in the test.
- *-fileSize*: Defines the size of files used in the test.
- *-resFile*: Defines the file location to save results.
- *-bufferSize*: Describes the length of the write buffer in bytes.

TestDFSIO is designed in such a way that it will use one map task per file.

6.1 Experimental Setup

The experiments were conducted using the erasure coding policy **RS-3-2-1024k**, so are instantiated **five slave containers** and **one master container**. This way, I met the minimum number of DataNodes required to use that policy. For each container are allocated 6 GB of RAM.

The **independent factors** and **their levels** chosen, are described in Table 6.1; as well as, the **response variables** chosen are described in Table6.2.

6.2 Experimental Results

The performance of the read operations are not of great interest: read operations do not trigger erasure coding (used in write and recovery scenarios), so they do not interact with the virtual functions.

Indipendent Factors	**Levels**
Number of files	4 8 16
Size of files	10 MB 100 MB 200 MB 400 MB
Number of VF per container	1 2

Table 6.1: Indipendent Factors and thier Levels

Response Variables	**Description**
Throughout	MB transmitted per seconds
Runtime	Total operation execution time
Average I/O Rate	Average DFS read/write operation

Table 6.2: Response Variables for read/write test

As for the runtime values of the write operation, we note that there is no significant variation up to 8 files. However, there is a degradation for 16 files, especially when two VFs are provided for each container.

Throughput and average I/O rate show comparable performance, so only the former performance are shown: there is a greater load tolerance for larger files with two virtual functions per container. However, in general, files larger than 400 MB are difficult to manage.

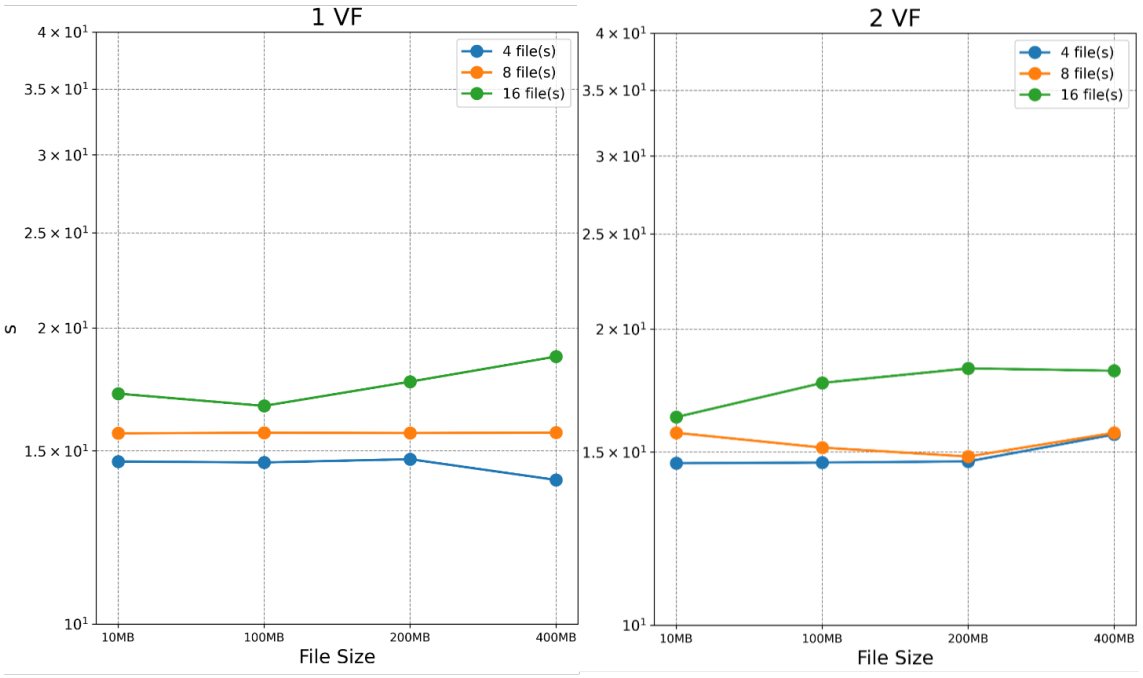


Figure 6.1: Runtime for the read operation

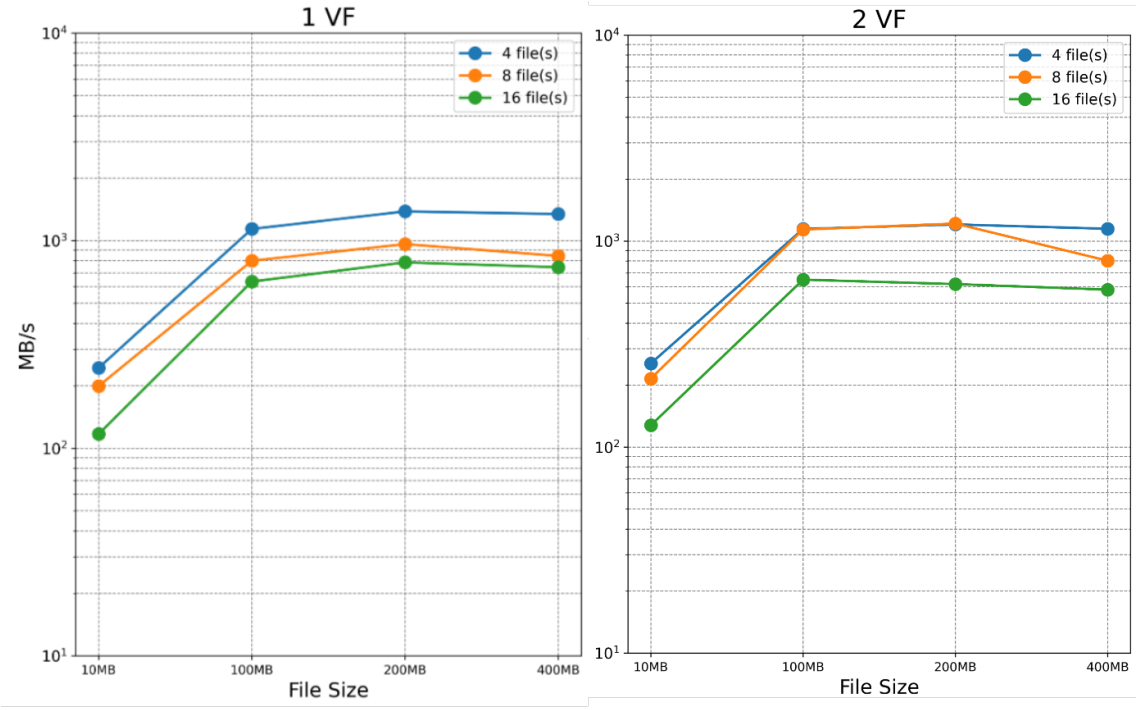


Figure 6.2: Throughput for the reasd operation

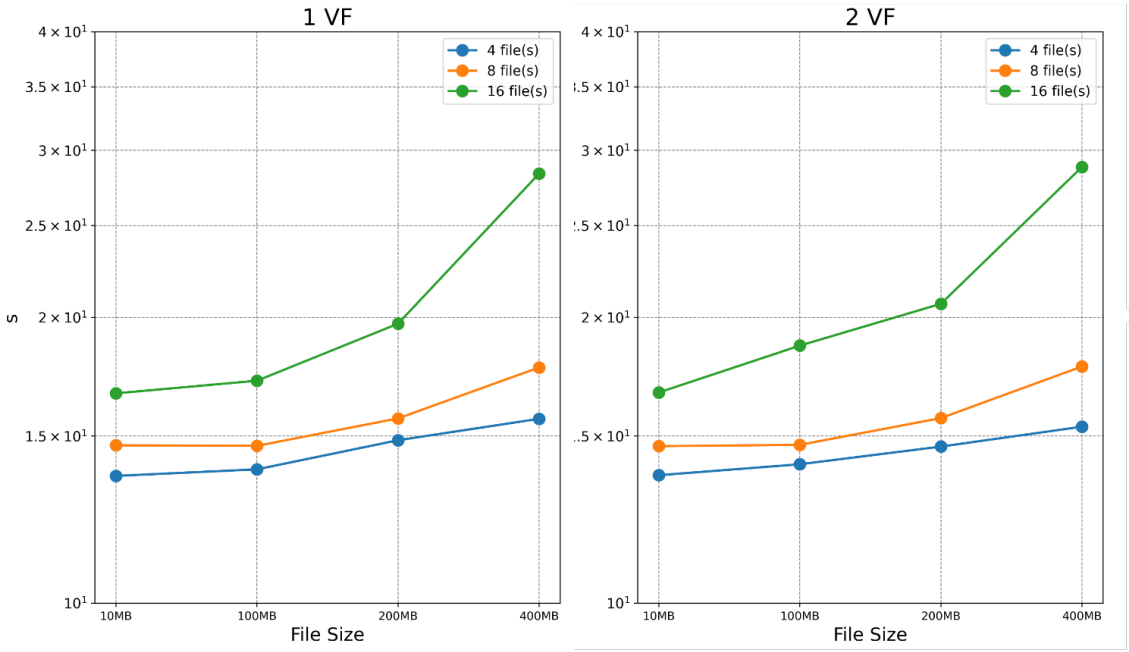


Figure 6.3: Runtime for the write operation

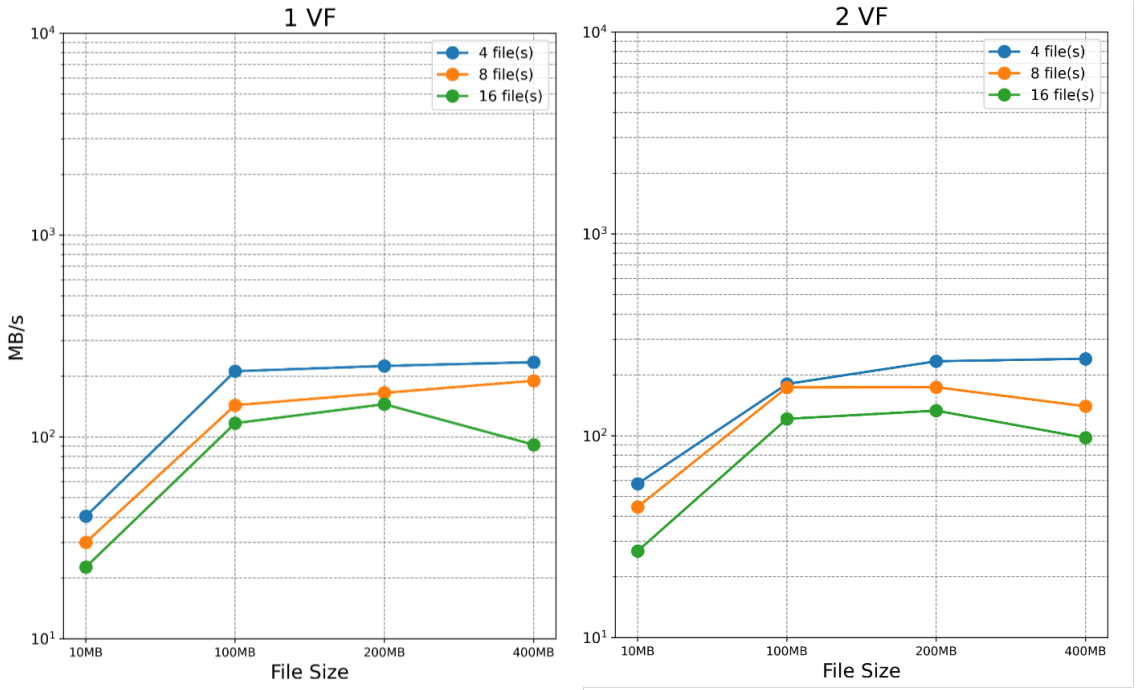


Figure 6.4: Throughput for the write operation

Chapter 7

Conclusions

This project demonstrates how it is possible to create a virtual cluster that is **functionally equivalent** to the corresponding physical device, overcoming its limitations. Indeed, through virtualization, the number of accelerators is decreased from n (where n is the number of physical hosts in the baseline architecture) to one, also reducing **the number of components**, **maintenance costs** and **energy consumption**. Additionally, due to the use of containers, **easier** and **faster development** can be achieved. However, the performance bottleneck of the virtualized architecture is the host's physical resources.

Ringraziamenti

Ringrazio i miei genitori, che attraverso i loro sacrifici, non mi hanno mai fatto mancare nulla. Mi hanno dato la possibilità di terminare questo percorso. Un percorso iniziato nel 2016 con la triennale, percorso che mi ha dato soddisfazioni ma anche tante delusioni ed amarezze. Li ringrazio per avermi spronato ad andare avanti ed a non abbandonare il mio desiderio di divenatre ignegnere. Li ringrazio per volere sempre il bene ed il massimo per me. Li ringrazio per tutti quei momenti in cui si sono impegnati nel comprendere dinamiche universitarie a loro sconosciute.

Ringrazio mia sorella Maria, che sopporta in tutto e per te. Lei che c'è sempre per aiutarmi e che condivide tante mie passioni. Lei che quando vede qualcosa di divertente me lo mostra, sperando mi faccia piacere. Le che vivendo sotto lo stesso tesso ha visto i miei deliri per via di questa facoltà, senza mia lamentarsi di nulla.

Ringrazio la mia ragazza Melania, colei che è entrata nella mia vita attraverso un sammer campus per ingegneri. Lei che mi fa stare bene anche con sorriso. Lei che mi rende felice anche con i più piccoli gesti.

Lei che si preoccupare sempre per me. Lei che c'è sempre per me. Lei che mi fa tornare bambino. Lei che porta tanta dolcezza nella mia vita. Lei che mi è stata vicina in tanti momenti e mi ha ascoltato nel mio lamentarmi dell'università, senza mai scocciarsi e continuandomi a dare consigli. Lei che da il massimo per me. Con lei, che quando stiamo insieme il tempo passa davveri veloce.

Ringrazio i miei amici dell'università, che anche se abbiamo prseo strade diverse parliamo lo stesso, ridiamo lo stesso. Loro, con cui ho vissuto i viaggi della speranza per andare all'università. Con loro che ho seguito, studiato, vissuto le sessioni invernali ed estive. Loro che mi hanno accompagnato agli esami. Loro che hanno condiviso con me la pausa caffè tra le lezioni. Con loro che ho condiviso appunti e dispense, per passare gli esami.

Ringrazio il mio migliore amico Carlo, che ci conosciamo dal secondo anno di superiori. Tanti anni in cui abbiamo fatto tanti compiti insieme, tante uscite insieme e tante mangiate insieme. Lui che hai visto la mia crescita e come sono cambiato. Con lui che ho passato tanti sabati nelle vie di Caserta, passando serate con desiderio di staccare da settimane di stress e studio.

Grazie a tutti voi per avermi supportato e sopportato, spesso di più la seconda. Grazie per esserci stati e per essere qui oggi per festeggiare questo nuovo tragurado.

Bibliography

[1] George Coulouris et al. *Distributed Systems: Concepts and Design.*

[2] Khushwant Kaur Anu Kaul Zmeena Gupta. *USING VIRTUALIZATION FOR DISTRIBUTED COMPUTING.* Aug. 2015.

[3] The Apache Foundation. Apache Hadoop. URL: https://hadoop.apache.org/.

[4] Pratit Raj Giri and Gajendra Sharma. Apache Hadoop Architecture, Applications, and Hadoop Distributed File System. In: (2022).

[5] The Apache Foundation. HDFS Architecture. URL: https://hadoop.apache.org/docs/r3.3.5/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[6] The Apache Foundation. HDFS Users Guide. URL: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html.

[7] The Apache Foundation. Apache Hadoop YARN. URL: https://hadoop.apache.org/docs/r3.3.5/hadoop-yarn/hadoop-yarn-site/YARN.html.

[8] The Apache Foundation. Hadoop: Capacity Scheduler. URL: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[9] The Apache Foundation. Hadoop: Fair Scheduler. URL: https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[10] The Apache Foundation. MapReduce Tutorial. URL: https://hadoop.apache.org/docs/r3.3.5/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[11] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: https://doi.org/10.1145/1327452.1327492.

[12] The Apache Foundation. Hadoop Cluster Setup. URL: https://hadoop.apache.org/docs/r3.3.5/hadoop-project-dist/hadoop-common/ClusterSetup.html.

[13] The Apache Foundation. Hadoop: Setting up a Single Node Cluster. URL: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html.

[14] Xiaohong Jiang Yanzhang He et al. Scalability Analysis and Improvement of Hadoop Virtual Cluster with Cost Consideration. In: (2014). URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6973791.

[15] The Apache Foundation. HDFS Erasure Coding. URL: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[16] Fan Lei et al. FPGA-Accelerated Erasure Coding Encoding in Ceph Based on an Efficient Layered Strategy. In: (Jan. 2024).

[17] Matt Ruan. Reed-Solomon Erasure Codec Design Using Vivado High-Level Synthesis. In: (Feb. 2016). URL: https://docs.amd.com/v/u/en-US/xapp1273-reed-solomon-erasure.

[18] Rui Quieroz et al. Container-based Virtualization for Real-time Industrial Systems—A Systematic Review. In: (Oct. 2023). URL: https://doi.org/10.1145/3617591.

[19] Docker Inc. What is Docker? URL: https://docs.docker.com/get-started/docker-overview/.

[20] Docker Inc. Docker Engine overview. URL: https://docs.docker.com/engine/.

[21] Docker Inc. OverlayFS storage driver. URL: https://docs.docker.com/engine/storage/drivers/overlayfs-driver/.

[22] What's LXC? URL: https://linuxcontainers.org/lxc/introduction/.

[23] About lxd and lxc. URL: https://documentation.ubuntu.com/lxd/en/latest/explanation/lxd_lxc/.

[24] Malek Musleh et al. Bridging the Virtualization Performance Gap for HPC Using SR-IOV for InfiniBand. In: (2014).

[25] Intel. *PCI-SIG SR-IOV Primer*. Jan. 2011.

[26] PCI-SIG. Specifications | PCI-SIG. 2024. URL: https://pcisig.com/specifications.

[27] VFIO - "Virtual Function I/O". URL: https://www.kernel.org/doc/html/latest/driver-api/vfio.html.

[28] Intel Corporation. oneAPI Accelerator Support Package(ASP) Reference Manual: Open FPGA Stack. URL: https://ofs.github.io/ofs-2023.2/hw/common/reference_manual/oneapi_asp/oneapi_asp_ref_mnl/.

[29] Intel Corporation. FPGA BSPs and Boards. URL: https://www.intel.com/content/www/us/en/docs/oneapi/programming-guide/2023-0/fpga-bsps-and-boards.html.

[30] FPGA Device Feature List (DFL) Framework Overview. URL: https://docs.kernel.org/fpga/dfl.html.

[31] Welcome to the OPAE SDK source code repository. URL: https://github.com/OFS/opae-sdk.

[32] Quick Start Guide of Opae SDK. URL: https://opae.github.io/latest/docs/fpga_api/quick_start/readme.html.