NOHOST: A New Storage Architecture for Distributed Storage Systems

by

Chanwoo Chung

B.S., Seoul National University (2014)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

© Massachusetts Institute of Technology 2016. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
August 31, 2016

Certified by .....
Arvind
Johnson Professor in Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Leslie A. Kolodziejski
Professor of Electrical Engineering
Chair, Department Committee on Graduate Students

NOHOST: A New Storage Architecture for Distributed Storage Systems

by

Chanwoo Chung

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2016, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

This thesis introduces a new NAND flash-based storage architecture, NOHOST, for distributed storage systems. A conventional flash-based storage system is composed of a number of high-performance x86 Xeon servers, and each server hosts 10 to 30 solid state drives (SSDs) that use NAND flash memory. This setup not only consumes considerable power due to the nature of Xeon processors, but it also occupies a huge physical space compared to small flash drives. By eliminating costly *host* servers, the suggested architecture uses NOHOST nodes instead, each of which is a low-power embedded system that forms a cluster of distributed key-value store. This is done by refactoring deep I/O layers in the current design so that refactored layers are light-weight enough to run seamlessly on resource constrained environments. The NOHOST node is a full-fledged storage node, composed of a distributed service frontend, key-value store engine, device driver, hardware flash translation layer, flash controller and NAND flash chips. To prove the concept of this idea, a prototype of two NOHOST nodes has been implemented on Xilinx Zynq ZC706 boards and custom flash boards in this work. NOHOST is expected to use half the power and one-third the physical space as compared to a Xeon-based system. NOHOST is expected to support the through of 2.8 GB/s which is comparable to contemporary storage architectures.

Thesis Supervisor: Arvind
Title: Johnson Professor in Computer Science and Engineering

Acknowledgments

I would first like to thank my advisor, Professor Arvind, for his support and guidance in the first two years at MIT. I would very much like to thank my colleague and leader in this project, Dr. Sungjin Lee, for the numerous guidance and insightful discussion. I also extend my gratitude to Sang-Woo Jun, Ming Liu, Shuotao Xu, Jamey Hicks, and John Ankcorn for their help while developing a prototype of NOHOST. I am grateful to Samsung Scholarship for supporting my graduate studies at MIT.

Finally, I would like to acknowledge my parents, grandmother, and little brother for their endless support and faith in me. This work would not have been possible without my family and all those close to me.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction **13**

1.1 Thesis Contributions 14

1.2 Thesis Outline 16

2 Related Work **17**

2.1 Application Managed Flash 17

2.1.1 AMF Block I/O Interface 18

2.1.2 AMF Flash Translation Layer (AFTL) 19

2.1.3 Host Application: AMF Log-structured File System (ALFS) . 20

2.2 BlueDBM 20

2.2.1 BlueDBM Architecture 21

2.2.2 Flash Interface 22

2.2.3 BlueDBM Benefits 23

3 NOHOST Architecture **25**

3.1 Configuration and Scalability:

NOHOST vs. Conventional Storage System 26

3.2 NOHOST Hardware 27

3.2.1 Software Interface 28

3.2.2 Hardware Flash Translation Layer 29

3.2.3 Network Controller 30

3.2.4 Flash Chip Controller 30

3.3 NOHOST Software 31

3.3.1 Local Key-Value Management 32
3.3.2 Device Driver Interfaces to Controller 33
3.3.3 Distributed Key-Value Store 35

4 Prototype Implementation and Evaluation **37**
4.1 Evaluation of Hardware Components 39
4.1.1 Performance of HW-SW communication and DMA data transfer over an AXI bus 39
4.1.2 Hardware FTL Latency 39
4.1.3 Node-to-node Network Performance 40
4.1.4 Custom Flash Board Performance 40
4.2 Evaluation of Software Modules 41
4.3 Integration of NOHOST Hardware and Software 44

5 Expected Benefits **45**

6 Conclusion and Future works **47**
6.1 Performance Evaluation and Comparison 47
6.2 Hardware Accelerators for In-store Processing 48
6.3 Fault Tolerance: Hardware FTL Recovery from Sudden Power Outage (SPO) 50

List of Figures

2-1 AMF Block I/O Interface and Segment Layout 18
2-2 BlueDBM Overall Architecture 21
2-3 BlueDBM Node Architecture 22

3-1 Conventional Storage System vs. NOHOST 26
3-2 NOHOST Hardware Architecture 28
3-3 NOHOST Software Architecture 31
3-4 NOHOST Local Key-Value Store Architecture 33
3-5 NOHOST Device Driver 34

4-1 NOHOST Prototype 38
4-2 Experimental Setup 42
4-3 I/O Access Patterns (reads and writes) captured at LibIO 43
4-4 Test Results with *db_test* 43
4-5 LibIO Snapshot of NOHOST with integrated hardware and software . 44

6-1 In-store hardware accelerator in NOHOST 49

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

4.1 Hardware FTL Latency 40

4.2 Experimental Parameters and I/O summary with RebornDB on NOHOST 42

5.1 Comparison of EMC XtremIO and NOHOST 45

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

A significant amount of digital data is created by sensors and individuals every day. For example, social media have increasingly become an integral part of people's lives and Instagram reports that 90 million photos and videos are uploaded daily [9]. These digital data are spread over thousands of storage nodes in data centers and are accessed by high-performance compute nodes that run complex applications available to users. These applications include the services provided by Google, Facebook, and YouTube. Scalable distributed storage systems, such as Google File System, Ceph, and Redis Cluster, are used to manage digital data on the storage nodes and provide fast, reliable and transparent access to the compute nodes [6, 27, 14].

Hard-disk drives (HDDs) are the most popular storage media in distributed settings, such as data centers, due to their extremely low cost-per-byte. However, HDDs suffer from high access latency, low bandwidth, and poor random access performance because of their mechanical nature. To compensate for these shortcomings, HDD-based storage nodes need a large power-hungry DRAM for caching data together with an array of disks. This setting increases the total cost of ownership (TCO) in terms of electricity cost, cooling fee, and data center rental fee.

In contrast, NAND flash-based solid-state drives (SSDs) have been deployed in centralized high-performance systems, such as database management systems (DBMSs) and web caches. Due to their high cost-per-byte, they are not as widely used as HDDs for large-scale distributed systems composed of high capacity storage nodes. How-

ever, SSDs have several benefits over HDDs: less power, higher bandwidth, better random access performance, and smaller form-factors [22]. These advantages, in addition to the dropping price-per-capacity of NAND flash, make an SSD an appealing alternative to HDD-based systems in terms of the TCO.

Unfortunately, existing flash-based storage systems are designed mostly for independent or centralized high-performance settings like DBMSs. Typically, in each storage node, an x86 server with high-performance CPUs and large DRAM (e.g. a Xeon server) manages a small number of flash drives. Since this setting requires deep I/O stacks from a kernel to a flash drive controller, it cannot maximally exploit the physical characteristics of NAND flash in a distributed setting [17, 18]. Furthermore, this architecture is not a cost-effective solution for large-scale distributed storage nodes due to high cost and power consumption of x86 servers, which only manage data spread over storage drives. It is expected that flash devices paired with the right hardware and software architecture can be a more efficient solution for large-scale data centers in the current flash-based systems.

1.1 Thesis Contributions

In this thesis, a new NAND flash-based architecture for distributed storage systems, NOHOST, is presented. As the name implies, NOHOST does not use costly *host* servers. Instead, it aims to exploit the computing power of embedded cores that are already in commodity SSDs to replace host servers and show comparable I/O performance. The study on Application Managed Flash (AMF) showed that refactoring flash storage architecture dramatically reduces flash management overhead and improves performance [17, 18]. To this end, the current deep I/O layers have been assessed and refactored into light-weight layers to reduce workloads for embedded cores. Among data storage paradigms, a key-value store has been selected as the service provided by NOHOST due to its simplicity and wide usage. Proof-of-concept prototypes of NOHOST have been designed and implemented. Note that a single NOHOST node is a full-fledged embedded storage node, comprised of a distributed

service frontend, key-value store engine, device driver, hardware flash translation layer, network controller, flash controller, and NAND flash. The contributions of this thesis are as follows:

- **NOHOST for a distributed key-value store:** Two NOHOST prototype nodes have been built using FPGA-enabled embedded systems. Individual NOHOST nodes are autonomous systems with on-board NAND flash, but they can be combined to form a huge key-value storage pool in a distributed manner. RocksDB has been used as a baseline to build a local key-value store, and for a distributed setting, Redis Cluster runs on top of the NOHOST local key-value store [5, 14]. NOHOST is expected to save about 2x in power and 3x in space over standard x86-based server solutions as detailed in Chapter 5.

- **Refactored light-weight storage software stack:** The RocksDB architecture has been refactored to get rid of unnecessary software modules and to bypass deep I/O and network stacks in the current Linux kernel. Unlike RocksDB, the NOHOST local key-value store does not rely on a local file system and kernel's block I/O stacks, and it directly communicates with underlying hardware. This architecture facilitates NOHOST software to run on a resource-constrained environment like ARM-based embedded systems and to offer better I/O latency and throughput.

- **HW-implemented flash translation layer:** To further reduce I/O bottlenecks and software latency, a hardware-implemented flash translation layer has been adopted. The hardware FTL maps logical page addresses to physical (flash) addresses, manages bad blocks, and performs simple wear-leveling.

- **High-speed serial storage network to combine multiple NOHOST nodes into a single NOHOST cluster:** For scalability, a high-speed serial storage network has been devised to combine multiple NOHOST nodes into a single NOHOST Cluster (NH-Cluster), which is seen by compute nodes as a single NOHOST node. The node-to-node network scales the storage capacity without increasing network overheads in a data center.

- **Compatibility with existing distributed storage systems:** To enable NOHOST nodes to be seamlessly integrated into data centers, NOHOST supports a popular key-value store protocol, a Redis Serialization Protocol (RESP) [14]. Redis Cluster clients work with the NOHOST local key-value store.

The preliminary results show that each design component in a NOHOST prototype correctly behaves as intended. In addition, it is confirmed that the components are integrated to provide a distributed key-value store service. However, the optimization and evaluation of NOHOST as a distributed key-value store remain to be future work for NOHOST.

1.2 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 summarizes important works that have affected the development of this thesis. Chapter 3 presents the new NOHOST architecture. Chapter 4 introduces the implementation of a NOHOST prototype and its evaluation. Chapter 5 estimates the benefits of NOHOST over existing storage systems. Finally, Chapter 6 concludes the thesis and introduces the future work for NOHOST.

Chapter 2

Related Work

2.1 Application Managed Flash

NAND flash SSDs have become the preferred storage media in data centers. SSDs employ a flash translation layer (FTL) to give an I/O abstraction and provide interoperability with existing block I/O devices. Due to the abstraction, host systems are not aware of flash characteristics. An FTL manages overwriting restrictions of flash cells, I/O scheduling, address mapping, address re-mapping, wear-leveling, bad blocks, and garbage collection. These complex tasks, especially address re-mapping and garbage collection, require software implementation with CPUs and DRAM. Commodity SSDs use embedded cores and DRAM to implement an FTL [8].

However, the abstraction makes a flash storage highly unpredictable in that high-level applications are not aware of inner-workings and vice versa. The unpredictability often results in suboptimal performance. Furthermore, an FTL approach suffers from the duplication of tasks when the host applications manage underlying storage in a log-like manner. For example, log-structured file systems always append new data to the device and mostly avoid in-store updates [26]. If a log-structured application runs on the FTL, both modules work to prevent in-place updates redundantly. This not only wastes hardware resource but also incurs extra I/Os [32].

To resolve the problems of an FTL approach, Application Managed Flash (AMF) allows host applications, such as file systems, databases, and key-value stores, to

directly manage flash [18]. This is done by refactoring the current flash storage architecture to support an AMF block I/O interface. In AMF, the device responsibility is reduced dramatically because it only has to expose the AMF interface and the host software that uses the AMF interface manages flash. The AMF performs light-weight mapping and bad block management internally. The refactoring dramatically reduces DRAM needed for flash management by 128x, and the performance of the file system improves by 80 % over commodity SSDs. This idea of refactoring is adopted for NOHOST. The AMF architecture and operation are presented next in detail.

2.1.1 AMF Block I/O Interface

The block I/O interface of AMF exposes a linear array of fixed size *logical pages* (e.g., 4 KB or 8 KB, equivalent to a flash page) which are accessed by existing I/O primitives, **READ, WRITE,** and **TRIM**. Contiguous logical pages form a larger unit, a *segment*. A segment is physically allocated when writing to the first page of a segment, and it is deallocated by TRIM. The granularity of a READ or WRITE command is a page while it is a segment for a TRIM command.

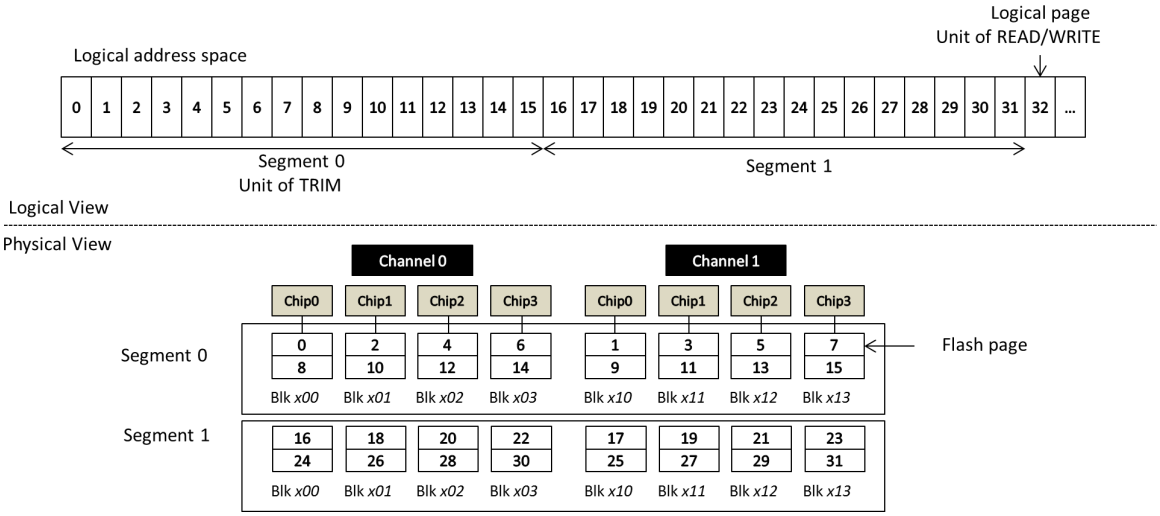


Figure 2-1: AMF Block I/O Interface and Segment Layout

A segment exposed to software is a *logical segment*, while its corresponding physical form is a *physical segment*. A logical segment is the unit of allocation; it is allocated a physical segment composed of a group of flash blocks spread over flash

channels and chips. The pages within a logical segment are statically mapped to flash pages within a physical segment using an offset. Figure 2-1 shows the AMF block I/O interface with logical and physical layouts of a segment in a setting of 2-channel, 4 chips/channel, and 2 pages/block flash. Numbers in boxes denote the logical page address (logical view) and its mapped location in real flash (physical view). The physical block labels (e.g. Blk *x12*) do not denote actual physical block numbers; they are mapped by a very simple block mapping algorithm.

Since flash cells do not allow overwrites, software using the AMF block interface must issue I/O commands accordingly in an appending manner. Many real-world applications, such as RocksDB, use derivatives of log-structured algorithms that inherently exploit the flash characteristics with little modification [5].

2.1.2 AMF Flash Translation Layer (AFTL)

Although AMF aims to remove redundancy in host software and a conventional FTL, AMF still needs some FTL functionalities: a block mapping, wear-leveling, and bad block management. It does not require address re-mapping to avoid in-place updates and expensive garbage collection. The AMF flash translation layer (AFTL) is a very lightweight FTL and similar to block-level FTL [2]. The following describes AFTL functionalities.

- **Block-mapping:** A logical segment is mapped to a physical segment. The block-granularity of AFTL ensures that the mapping table is small. If a WRITE command is issued to an unallocated segment, AFTL maps physical flash blocks to the logical segment. AFTL translates logical page addresses into physical flash addresses. The AMF mapping exploits the parallelism of flash chips by assigning flash pages on different channels and ways to consecutive logical pages.

- **Wear-leveling:** To preserve the lifetime and reliability of flash cells, AMF takes into account the least worn flash block when allocating a new segment. Furthermore, AFTL can exchange the most worn-out segment with the least worn-out segment.

- **Bad block management:** When allocating flash blocks to a segment, AMF ensures no bad blocks are mapped. This can be done by keeping track of bad blocks. AFTL learns if a block is a bad block by erasing the block. Wear-leveling and bad block management require a small table that records the program-erase cycle and status of all physical blocks.

AFTL is very lightweight and hence uses as small as 8 MB of memory for a 1 TB flash device, depending on flash chip configurations [18].

2.1.3 Host Application: AMF Log-structured File System (ALFS)

A flash-aware F2FS filesystem is modified to implement an AMF Log-structured File System (ALFS) [16]. The difference is that ALFS appends the metadata as opposed to updating it in-place, supporting the AMF block I/O interface without violating write restrictions. ALFS is an example work to show the advantages of AMF. AMF with ALFS reduces memory requirement for flash management by 128x, and the performance of the file system improves by 80 % over commodity SSDs.

2.2 BlueDBM

Big Data analytics is a huge economic driver in IT industry. One approach to Big Data analytics is RAMCloud, where a cluster of servers has enough DRAM collectively to accommodate the entire dataset in DRAM [24]. This, however, is an expensive solution due to the cost and power consumption of DRAM. Alternatively, BlueDBM is a novel and cheaper flash storage architecture for Big Data analytics [11]. BlueDBM supports the followings:

- A multi-node system with large flash storage for hosting Big Data workloads
- Low-latency access into a network of storage devices to form a global address space.
- User-defined in-store processors (accelerators)

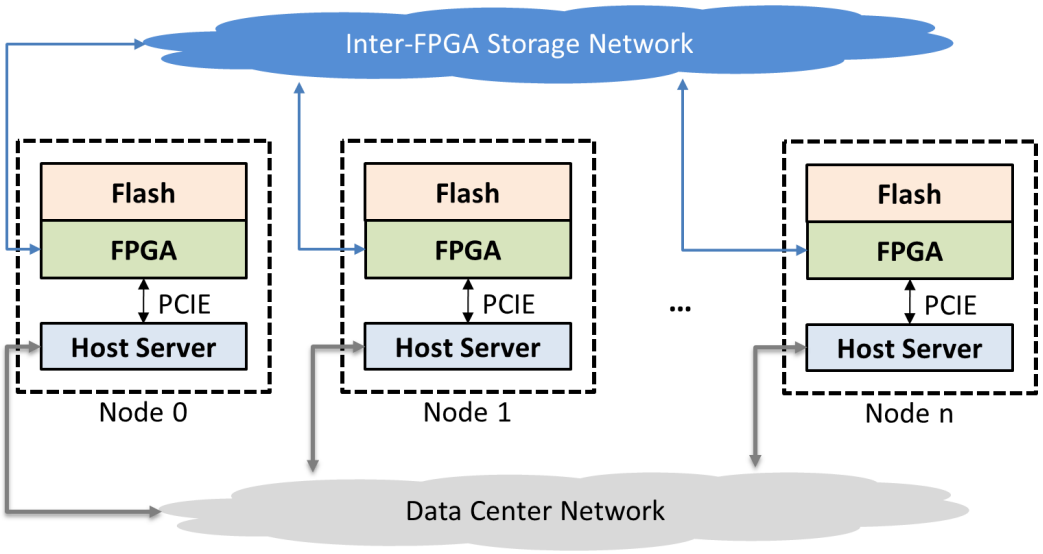


Figure 2-2: BlueDBM Overall Architecture

- Custom flash board with the special controller whose interface exposes **ReadPage**, **WritePage**, and **EraseBlock** commands using flash addresses.

2.2.1 BlueDBM Architecture

The overall BlueDBM architecture is shown in Figure 2-2. BlueDBM is composed of a set of identical BlueDBM nodes, each of which contains NAND flash storage managed by FPGA, which is connected to an x86 server via a fast PCIe link. Host servers are connected to form a data center network over Ethernet. The controllers in FPGAs are directly connected to other nodes via serial links, forming an inter-FPGA storage network. This sideband network gives us uniformly low-latency access to other flash devices and a global address space. Thus, when a host wants to access remote storage, it can directly access the remote storage over the storage network, instead of involving remote hosts. This approach improves performance by removing the network and storage software stacks.

Figure 2-3 shows the architecture of a BlueDBM node in detail. A user-defined in-store processor is located between local or remote flash arrays and a host server. The in-path accelerator dramatically reduces latency. Components in the green box are implemented on a Xilinx VC707 FPGA board [30]. A custom flash board with a flash

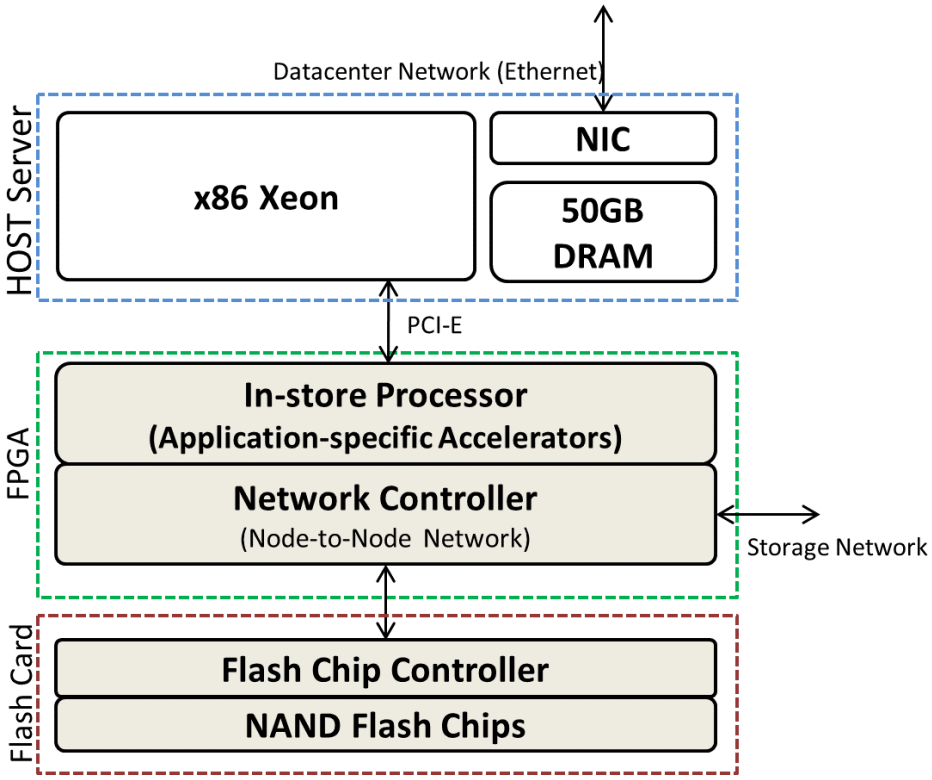


Figure 2-3: BlueDBM Node Architecture

chip controller on a Xilinx Artix-7 FPGA and 512 GB of flash chips was developed in the BlueDBM work. The custom flash board is denoted by a red box. This custom board with a flash chip controller and NAND flash chips is used in this thesis.

2.2.2 Flash Interface

The flash chip controller exposes a low-level, fast, and bit-error-free interface. The flash controller internally performs bus/chip-level I/O scheduling and ECC. The supported commands are as follows:

1. *ReadPage(tag, bus, chip, block, page):* Reads a flash page.
2. *WritePage(tag, bus, chip, block, page):* Writes a flash page, given that the page must be erased before being written. Otherwise, an error is returned.
3. *EraseBlock(tag, bus, chip, block):* Erases a flash block. Returns an error if the block is bad.

2.2.3 BlueDBM Benefits

BlueDBM improves system characteristics in the following ways.

- *Latency:* BlueDBM achieves extremely low-latency access to distributed flash devices. The inter-FPGA storage network removes Linux network stack overhead. Furthermore, the in-store accelerator reduces processing time.
- *Bandwidth:* Flash chips are organized into many buses for parallelism. Multiple chips on different nodes can be accessed concurrently over the storage network. In addition, data processing bandwidth is not bound by the software performance because in-store accelerators can consume data at device speed.
- *Power:* Flash storage consumes much less power than DRAM does. Hardware accelerators are also more power efficient than x86 CPUs. Furthermore, data moving power is reduced since it is not required to move data to hosts for processing.
- *Cost:* The cost-per-byte of flash storage is much less than that of DRAM.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

NOHOST Architecture

NOHOST is a new distributed storage system composed of a large number of nodes. Each node is a full-fledged embedded key-value store node that consists of a key-value store frontend, operating system, device driver, hardware flash translation layer, flash chip controller, and NAND flash chips, and can be configured as either a master or a slave. A NOHOST node replaces existing HDD-based or SSD-based storage node where a power-hungry x86 server hosts several storage drives.

The refactored I/O architecture of NOHOST is derived from Application Managed Flash (AMF) [18]. NOHOST hardware supports the AMF block I/O interface, and NOHOST software must be aware of flash characteristics and directly manages flash.

The hardware of a NOHOST node includes embedded cores, DRAM, FPGA, and NAND flash chips. The NOHOST software, which runs on the embedded cores, consists of an operating system, device driver, and key-value store engine. The software communicates with the hardware, manages key-value pairs in flash chips and exposes a key-value interface to users. Thus, the hardware and software must interact with each other closely to provide a reliable service.

To illustrate the overall architecture of NOHOST, this chapter begins by a comparison of NOHOST with the conventional storage system from the point of view of scalability and configuration. Then, the hardware and software of NOHOST are described in detail.

3.1 Configuration and Scalability: NOHOST vs. Conventional Storage System

Figure 3-1 shows the conventional storage system with compute nodes and the proposed NOHOST system. It is assumed that storage nodes are separate from compute nodes, which run complex user applications, just like in the conventional architecture. From the perspective of the compute nodes, NOHOST behaves exactly like a cluster of the conventional storage system. The compute nodes access data in NOHOST or the conventional system over a data center network.

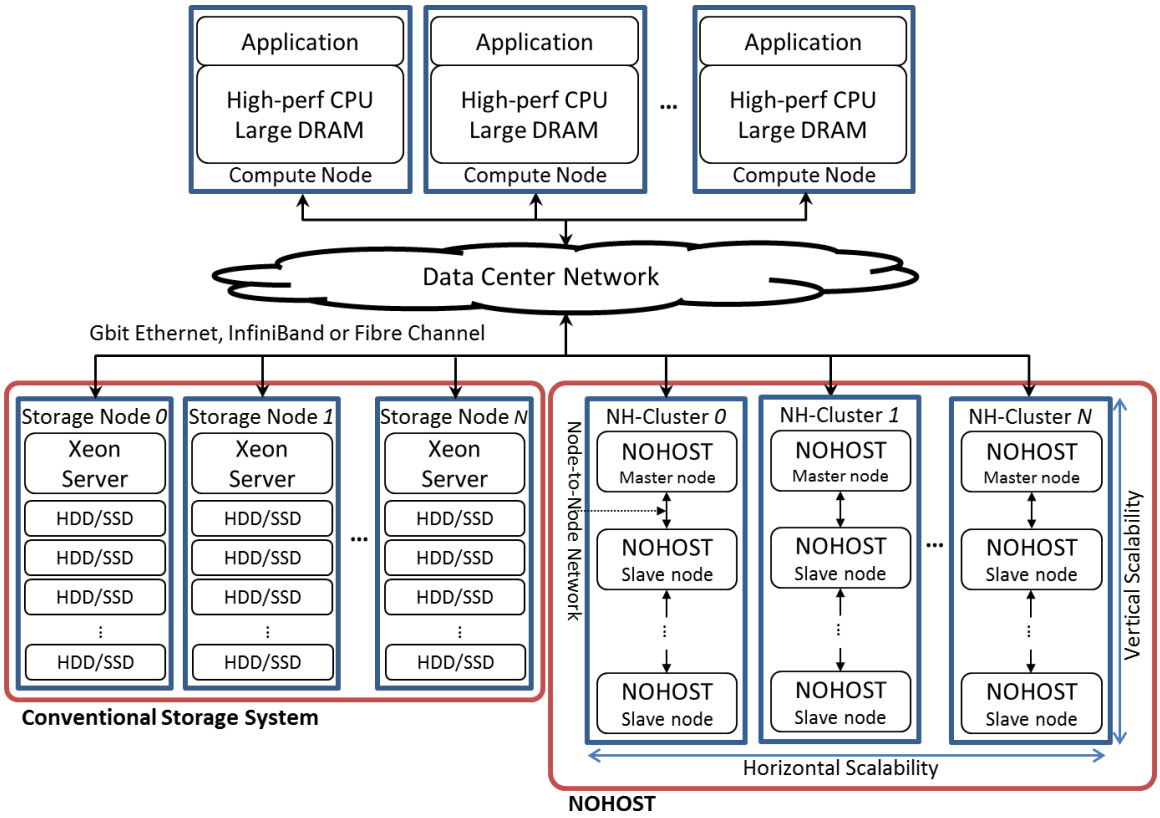


Figure 3-1: Conventional Storage System vs. NOHOST

In the conventional system architecture, denoted by the left red box of Figure 3-1, a single node consists of an Intel Xeon server managing 10 to 20 drives, either HDDs or SSDs. The Xeon server, which occupies a great deal of rack space and consumes considerable power, runs storage management software such as a local and distributed key-value store or file system. While the local key-value store manages key-value pairs

in local drives in a single node, the distributed key-value store runs on the top of the local key-value store and provides compute nodes with a reliable interface for accessing key-value pairs spread over multiple nodes. Each storage node is connected to a data center network using commodity interfaces such as Gigabit Ethernet, InfiniBand, and FiberChannel. In terms of scalability, a new server (node) needs to be installed to achieve more capacity because a single server cannot accommodate as many drives as system administrators want due to I/O port constraints. Furthermore, it is worth noting that each off-the-shelf SSD used in the conventional system is already an embedded system with ARM cores and small DRAM for managing flash chips.

In contrast, a single node is an autonomous embedded storage without any host servers. As shown in Figure 3-1, a NOHOST master node and a number of slave nodes are connected "vertically" to make a NOHOST cluster (NH-Cluster), which is analogous to a single server in the conventional system. NH-Cluster scales by adding more nodes vertically (*vertical scalability*). Only the master node is connected to the data center network via commodity network interfaces. Due to physical limitations on the number of I/O ports on a single node, expanding the capacity of a node by adding more flash chips is not a scalable solution. Thus, vertical scalability plays a crucial role in increasing the capacity of the storage system without burdening the data center network by connecting additional nodes to the network directly. Furthermore, the network port of NH-Cluster can be saturated when multiple nodes in NH-Cluster work in parallel. The number of nodes in NH-Cluster is optimized using bandwidth of the data center network and each node. NOHOST can also scale "horizontally" by adding more NH-Clusters to the data center network (*horizontal scalability*). This process is similar to installing new Xeon servers in the conventional system.

3.2 NOHOST Hardware

The NOHOST hardware is composed of several building blocks as shown in Figure 3-2. The hardware includes embedded cores and DRAM on which software runs. A network interface card (NIC) connects a NOHOST node to a data center network.

In addition, a software interface is needed for communication between the software and the hardware. The hardware also hosts NAND flash chips, where data bits are physically stored. Furthermore, the hardware has three main building blocks: a hardware flash translation layer (FTL), network controller, and flash chip controller. These principal components have special functionalities, and they are explained in detail below. Three dotted boxes (black, green, and red) on the master node side of Figure 3-2 denotes the implementation domain of a NOHOST prototype, which is presented in Chapter 4.

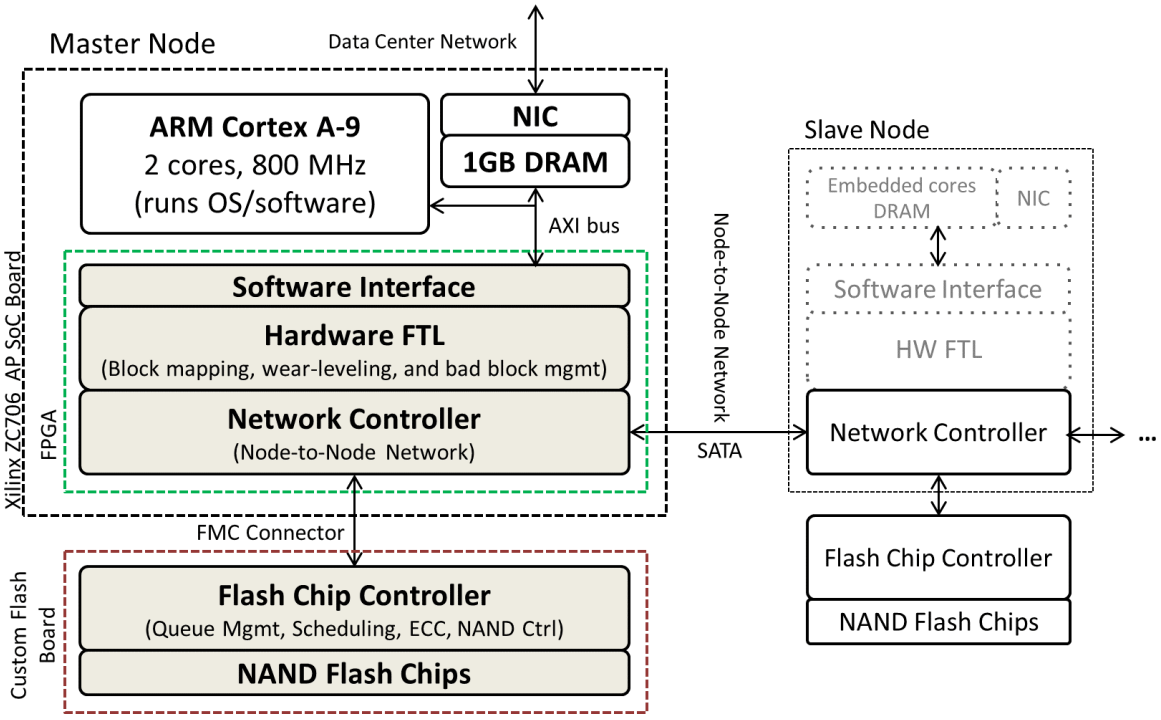


Figure 3-2: NOHOST Hardware Architecture

3.2.1 Software Interface

The software interface is implemented using Connectal, a hardware-software codesign framework [13]. Connectal provides an AXI endpoint and driver pair, allowing users to facilitate communication between software and hardware easily. The AXI endpoint transfers messages to and from hardware components. For high bandwidth data transfers, NOHOST hardware needs to read or write host system memory directly.

Data transfer between host DRAM and the hardware is managed by DMA engines in the AXI endpoint from the Connectal libraries.

3.2.2 Hardware Flash Translation Layer

The hardware flash translation layer (hardware FTL) is a hardware implementation of a light-weight AMF Flash Translation Layer (AFTL) [18]. This layer exposes the AMF Block I/O interface to software interface. It should be noted that software must be aware of flash characteristics and issue append-only-write commands. The primary function of the hardware FTL is block-mapping a logical address used by software to a physical flash address needed by hardware modules, but it also performs wear-leveling and bad block management. The basic idea of block-mapping is that a logical block is mapped to a physical flash block, and the logical page offset within a logical block is identical to a physical page offset within a physical block. If there is no valid mapping for the logical block specified by a given logical address, the FTL allocates a free flash block to analogous logical block (*block mapping*). Furthermore, when choosing a flash block to map, the FTL ensures that no bad block is allocated (*bad block management*) and selects the least worn block, that is, the free block with the lowest program-erase (PE) cycle (*wear-leveling*). Bad block management and wear-leveling enhance the lifetime and reliability of a flash storage.

To support above functionalities, the hardware FTL needs two tables: a block mapping table and a block status table. The first table stores whether a logical block is mapped and if so, a mapped physical block address. The second table keeps the status and PE cycles of physical blocks. In the NOHOST prototype implementation, each table requires only 512 KB (total 1 MB) per a 512 GB flash device. The size of tables increases linearly as the custom flash boards are added in NH-Cluster.

The hardware FTL exposes the following AMF Block I/O interface to software via a device driver. An lpa denotes a logical page address.

1. *READ(tag, lpa, buffer pointer):* Reads a flash page and store the data in host buffer.

2. *WRITE(tag, lpa, buffer pointer):* Writes a flash page from the host buffer, given that the page must be erased before being written. Otherwise, an error is returned.

3. *TRIM(tag, lpa):* Erases a flash block that includes a page denoted by the lpa. Returns an error if the block is bad.

3.2.3 Network Controller

The network controller is essential for the vertical scalability of NOHOST. This controller is adopted from BlueDBM [11, 12]. The network controllers of nodes that comprise NH-Cluster are connected with serial links to form a node-to-node network. As previously mentioned, NH-Cluster is composed of one master node and a number of slave nodes. Slave nodes work as if they are expansion cards to increase the capacity of NH-Cluster. Commands from a master node are routed to an appropriate node via the network. The network controller exposes a single address space for all nodes to the master node. Thus, software and hardware stacks above the network controller are not needed in slaves; the master is in charge of managing data in NH-Cluster. However, these components may be used to off-load the computation burden of the master node. The optional blocks are represented by dotted gray boxes in Figure 3-2.

3.2.4 Flash Chip Controller

The flash chip controller manages individual NAND flash chips. It forwards flash commands to chips, maintains multiple I/O queues and performs scheduling so that the whole NOHOST system maximally exploits the parallelism of multiple channels of flash chips. Furthermore, it performs error correction using ECC bits. Thus, the controller provides us with a robust and error-free access to NAND flash chips. The flash chip controller was developed for minFlash and BlueDBM studies, and the supported commands are presented in Section 2.2.2 [20, 11].

3.3 NOHOST Software

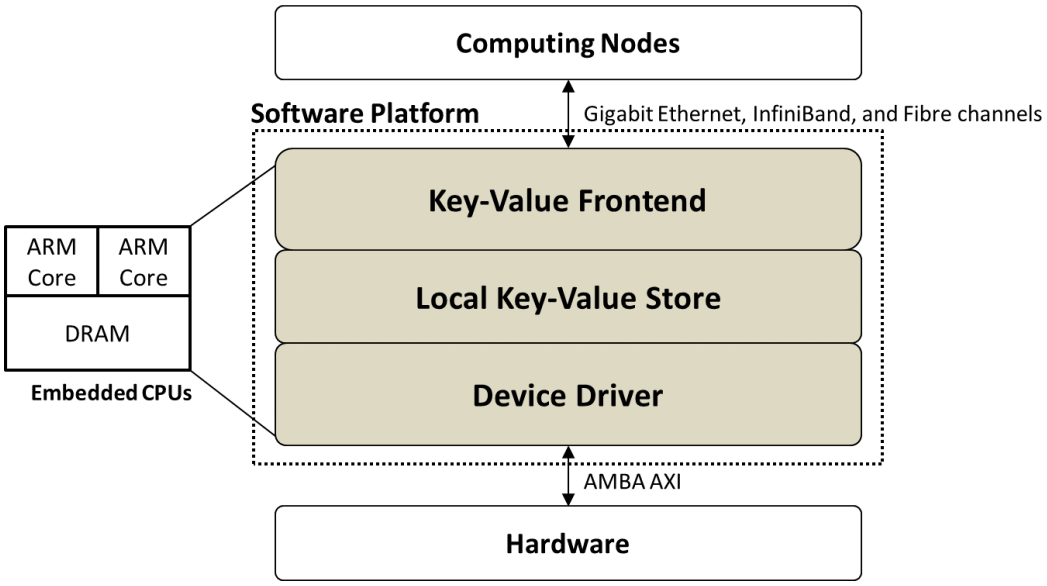


Figure 3-3: NOHOST Software Architecture

Figure 3-3 shows the architecture of NOHOST software. The NOHOST software runs on the top of a resource-constrained environment. Our primary design goal is thus to build a light-weight key-value store while maintaining its performance.

To meet such requirements, NOHOST software is composed of three principal components: a frontend for a distributed key-value store, local key-value store, and device driver. The frontend works as a manager that allows a single node to join a distributed key-value storage pool and provides users access to distributed key-value pairs. The local key-value store manages key-value pairs present on a local flash storage. For better compatibility with existing systems, NOHOST uses a REdis Serialization Protocol (RESP), a *de-facto* standard in key-value stores [14, 3].

The local key-value store manages key-value pairs in a local flash storage. Instead of building it from scratch, Facebook's RocksDB was selected as a baseline key-value store [5]. Because of its versatility and flexibility, RocksDB is widely used in various applications. Unlike the existing RocksDB, the NOHOST local key-value store does not rely on a local file system and a kernel's block I/O stack and directly communicates with underlying hardware. To this end, RocksDB has been refactored extensively to

implement the NOHOST local key-value store. This is discussed in detail later in this section.

The device driver is responsible for communication with the hardware FTL and the flash controller. In addition to this, the device driver provides a single address space so that the local key-value store directly accesses remote stores in the same NH-Cluster over the node-to-node network. This hardware support enables software modules to communicate with remote nodes, bypassing deep network and block I/O stacks in the Linux kernel.

3.3.1 Local Key-Value Management

The NOHOST local key-value store is based on RocksDB that uses an LSM-Tree algorithm [23, 5]. Figure 3-4 compares the architecture of the NOHOST local key-value store with the current RocksDB architecture. In designing and implementing NOHOST, the flash-friendly nature of the LSM-tree algorithm has been leveraged.The existing software modules for B-tree and LSM-tree algorithms are not modified at all. Instead, a NOHOST storage manager is added to RocksDB. The new manager filters out in-place-update writes coming from upper software layers and sends only out-place-update writes (append-only writes) to the flash controller. Due to the characteristics of the LSM-tree algorithm, almost all of I/O requests are append-only. This enables us to eliminate the need for the use of the conventional FTL, greatly simplifying the I/O stack and controller designs. A small number of in-place-update writes are required for logging a history and keeping manifest information, and the manager filters and send them to another storage device like SD cards in a NOHOST node.

While the current storage managers of RocksDB run on the top of a local file system and access storage devices through a conventional block I/O stack, NOHOST bypasses all of them. Instead, NOHOST relies on two light-weight user-level libraries, LibFS and LibIO, that completely replace file systems and block I/O layers. This approach minimizes the performance penalties and CPU cycles caused by redundant layers.

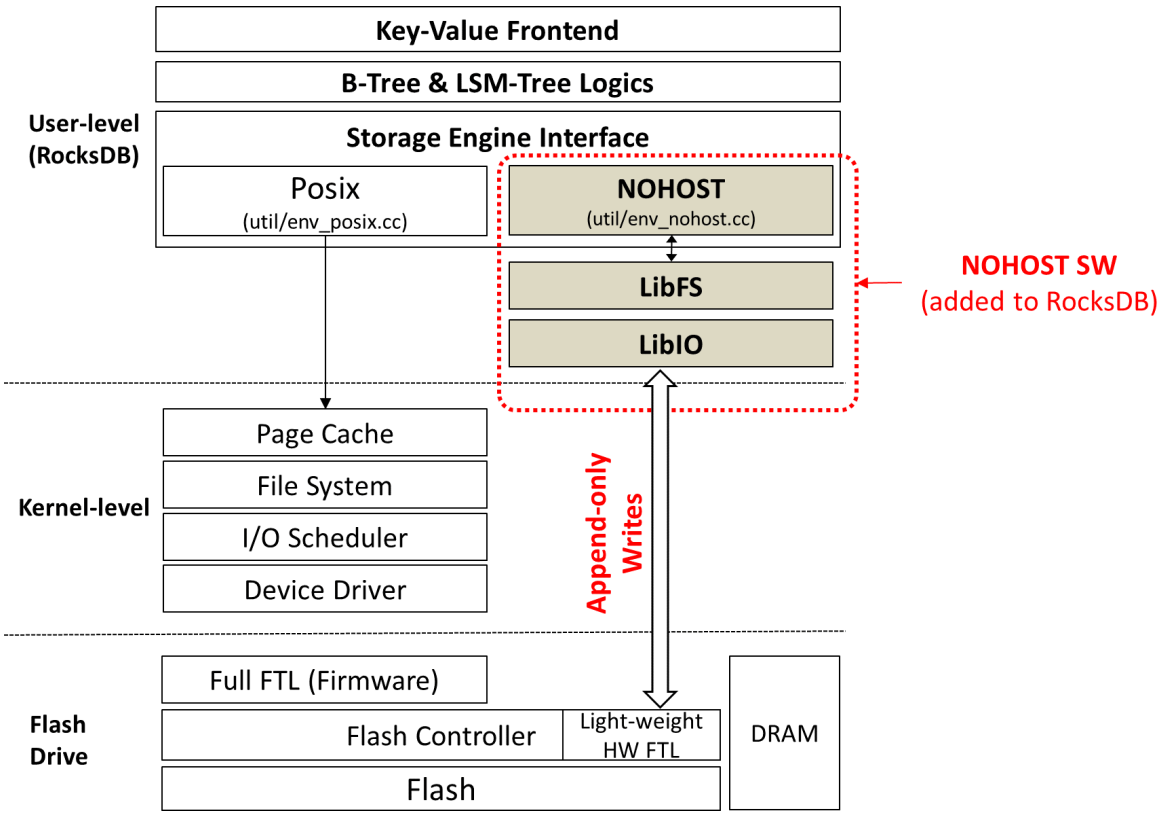


Figure 3-4: NOHOST Local Key-Value Store Architecture

LibFS is a set of file system APIs for the storage manager of RocksDB, which emulates a POSIX file system interface. LibFS minimizes the changes in RocksDB and gives the illusion that the NOHOST key-value store still runs on a conventional file system. LibFS simply forwards commands and data from the storage manager to LibIO. LibIO is another user-level library and emulates kernel's block I/O interfaces. LibIO preprocesses incoming data (e.g. chunking and aligning) and sends I/O commands to the flash controller.

3.3.2 Device Driver Interfaces to Controller

As previously mentioned, NOHOST uses a kernel-level device driver provided by Connectal [13]. Figure 3-5 summarizes how the device driver interacts with other system components. The main responsibility of the device driver is to send I/O commands from the key-value store to the hardware controller. Since NOHOST's hardware supports essential FTL functionalities, the device driver just needs to send

simple READ, WRITE, and TRIM commands with a logical address, I/O length, and data buffer pointer.

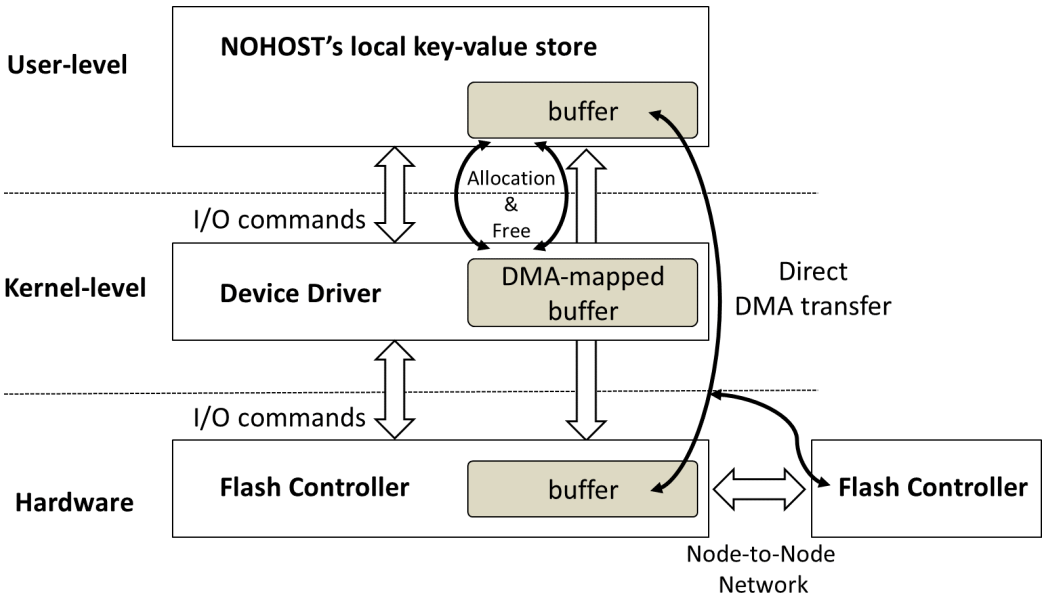


Figure 3-5: NOHOST Device Driver

Transferring data between user-level applications and hardware controller often requires extra data copies. To eliminate this overhead, the device driver provides its own memory allocation function using Linux's memory-mapped I/O subsystem. The device driver allocates a chunk of memory mapped to DMA and allows the user-level application to get the DMA-mapped buffer. The buffer allows data transfer from and to the hardware controller without any extra copying.

Another unique feature of the NOHOST device driver is that it supports direct access to remote nodes in the same NH-Cluster over a node-to-node network. This feature removes a latency from complicated Linux network stacks. From user-applications' perspective, all nodes belonging to the same NH-Cluster is seen as a single unified storage device. This makes it much simpler to handle multiple remote nodes without any concerns on data center network connections and their management.

3.3.3 Distributed Key-Value Store

To provide a distributed service, NOHOST uses RebornDB on the top of its local key-value store [25]. RebornDB is compatible with Redis Cluster, uses Redis's RESP, the most popular key-value protocol, and provides distributed key-value pair management. Since RebornDB supports RocksDB as its backend key-value store, combining RebornDB with the NOHOST local key-value store has been done in an easy manner.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

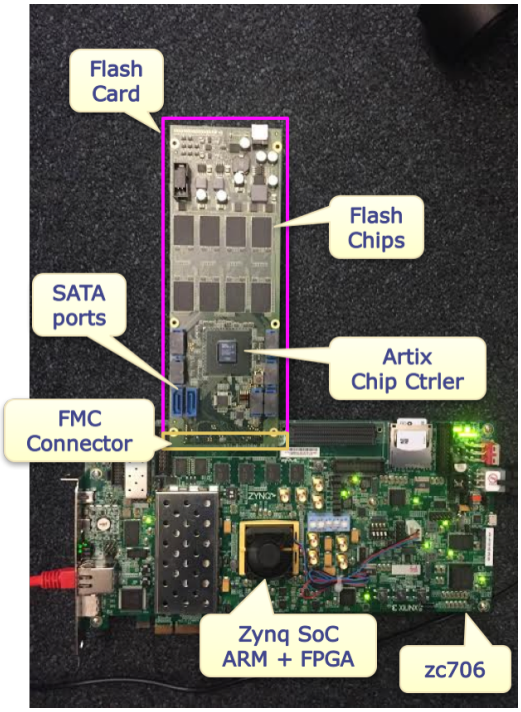
Prototype Implementation and Evaluation

A Xilinx ZC706 board has been used to implement a prototype of a NOHOST node [31]. The ZC706 board is populated with a Zynq SoC that integrates two 32-bit ARM Cortex-A9 cores, AMBA AXI interconnects, system memory interface, and programmable logic (FPGA). Thus, the board is an appropriate platform to implement an embedded system with hardware accelerators.

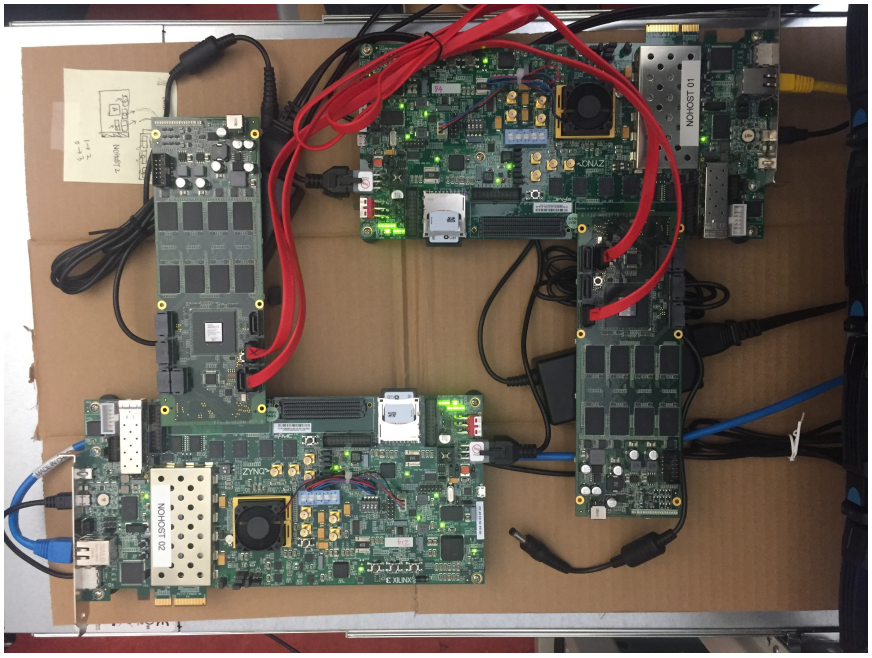
In a NOHOST node, Ubuntu 16.04 (Linux kernel 4.4) and software modules including a RocksDB-based key-value store run on the embedded cores. As shown in Figure 3-2, hardware components are implemented on the FPGA of a Zynq SoC (green box) and a custom flash board (red box).

The custom flash board (BlueFlash board) has 512 GB of NAND flash storage (8-channel, 8-way) and a Xilinx Artix-7 chip on which the flash chip controller is implemented [19]. The custom boards were developed for the previous study on BlueDBM and minFlash [20, 11]. The flash board plugs into the host ZC706 board via the FPGA Mezzanine Card (FMC) connector. The Zynq SoC communicates with the flash board using Xilinx Aurora 8b/10b transceiver [29]. Our node-to-node network controller is implemented using the Xilinx Aurora 64b/66b serial transceiver and uses SATA as a cable interface [28]. Each NOHOST prototype includes a fan-out of 8 network ports and supports ring-based simple network configuration.

Figure 4-1 shows photos of (a) a single-node NOHOST prototype and (b) a two-node NOHOST configuration.



(a) A single node



(b) Two-node Configuration

Figure 4-1: NOHOST Prototype

In this chapter, the performance of hardware components and software components is evaluated separately. Then, software and hardware modules are combined to confirm that the NOHOST prototype provides a key-value store service. Optimization and assessment of NOHOST as a "distributed" key-value store will be conducted in the future.

4.1 Evaluation of Hardware Components

4.1.1 Performance of HW-SW communication and DMA data transfer over an AXI bus

As previously mentioned, software and hardware communicate with each other over a pair of an AXI endpoint and a driver implemented with Connectal libraries. Connectal adds 0.65 µs latency (HW → SW) and 1.10 µs latency (SW → HW) [13]. Assuming a flash access latency of 50 µs, such a communication only adds 2.2 % latency in the worst case.

The data transfer between host DRAM and hardware (FPGA) is initiated by Connectal DMA engines connected to AXI bus. The ZC706 board supports 4 high-performance AXI DMA ports to work in parallel. When all DMA ports are fully utilized, our prototype supports up to 2.8 GB/s of read and write bandwidth measured by software.

4.1.2 Hardware FTL Latency

As noted in Section 3.2.2, the hardware FTL requires 1 MB of a mapping table and block status table per a 512 GB flash board. In the NOHOST prototype, tables may reside in either block RAM (BRAM) that is integrated with FPGA or external DRAM. The BRAM on the ZC706 board is as small as 2,180 KB and not expandable, but it shows less latency. The external DRAM is currently 1 GB and can be upgraded up to 8 GB, but suffers from higher latency. Table 4.1 summarizes the latency to translate logical page addresses to physical flash addresses for both implementations.

There are two scenarios: a physical block is already mapped, or a new physical block needs to be selected from free blocks and allocated. The prototype hardware operates with 200 MHz clock, so each cycle is equivalent to 5 ns.

Table 4.1: Hardware FTL Latency

	Block Already Allocated	**New Block Allocated**
BRAM	4 cycles / 20 ns	140 cycles / 700 ns
DRAM	42 cycles / 210 ns	214 cycles / 1070 ns

Even if DRAM implementation is used, the worst case translation latency is 1.07 µs. Assuming a flash access latency of 50 µs, such an address translation adds 2.1 % latency in the worst case.

4.1.3 Node-to-node Network Performance

The performance of the NOHOST storage-to-storage network is measured by transferring a stream of 128-bit data packets through NOHOST nodes across the network. The network controller was implemented using a Xilinx Aurora 64b/66b serial transceiver, and SATA cables are used as links to connect the transceivers [28]. The physical link bandwidth is 1.25 GB/s with protocol overhead, pure data transfer bandwidth is 1.025 GB/s, and per-hop latency is 0.48 µs. Each NOHOST node includes 8 network ports so that each node can sustain up to 8.2 GB/s of data transfer bandwidth across multiple nodes.

The end-to-end network latency over serial transceivers is simply a multiple of network hops to the destination [11, 12]. In a naive ring network of 20 nodes with 4 links each to next and previous nodes, the average latency to a remote node is 5 hops or 2.4 µs. Assuming a flash access latency of 50 µs, this network will only add 5 % latency, giving the illusion of a uniform access storage.

4.1.4 Custom Flash Board Performance

As noted at the beginning of this chapter, the custom flash boards developed for BlueDBM and minFlash are used in NOHOST [11, 20, 19]. The board plugs into

the host ZC706 board via the FMC connector. The communication is managed by a 4-lane Xilinx Aurora 8b/10b transceiver on each FPGA [29]. The link sustains up to 1.6 GB/s of data transfer bandwidth at 0.5 µs latency.

The design of the flash controller and flash chips provides average 1,260 MB/s of read bandwidth with 100 µs latency and 461 MB/s of write bandwidth with 600 µs latency per each board. The bandwidth is measured by software issuing page read/write commands that initiate transfers data between system memory and flash chips. The node-to-node network, FMC connection and DMA transfer can sustain full bandwidth of flash chips on each board. Multiple flash boards connected by the node-to-node network may keep all the DMA engines in the master node busy to sustain up to 2.8 GB/s of data transfer bandwidth to and from software.

4.2 Evaluation of Software Modules

A set of evaluations has been performed to confirm the behaviors of NOHOST software, including its functionalities without the FTL, direct access to a storage device with minimal kernel supports, and its ability as distributed key-value store. For a quick software evaluation, all of the software modules runs with a DRAM-emulated flash storage implemented as a part of kernel's block device driver.

Figure 4-2 shows the experimental setting. RebornDB combined with NOHOST's RocksDB-based key-value store is running on the NOHOST node. Even though DRAM-emulated flash is used instead of NOHOST flash, NOHOST software uses the same LibFS and LibIO to access the storage media. Over the network, Redis Cluster clients communicate with RebornDB on NOHOST using RESP [25, 14]. Since the goal is to check the correctness of NOHOST behaviors, 50 Redis clients, running concurrently, induce network and I/O traffics to NOHOST. In the observation, all of the software stacks including the distributed key-value frontend, local key-value store, and user-level libraries, perform correctly without any functional errors. Table 4.2 lists a summary of I/O requests with experimental parameters.

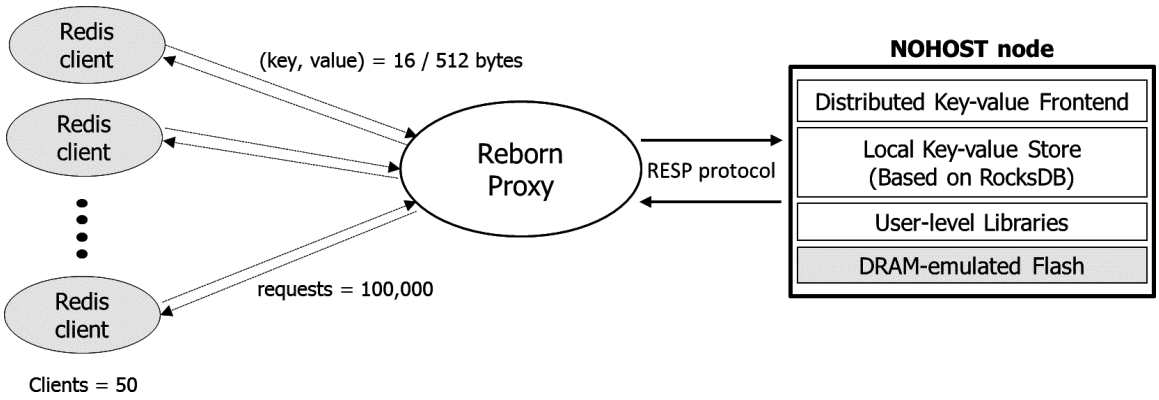


Figure 4-2: Experimental Setup

Table 4.2: Experimental Parameters and I/O summary with RebornDB on NOHOST

(a) Parameters

Paramaters	Clients	Requests	Data Size	Test Type	Reqs per Client
Values	50	5,000,000	512 Bytes	Set	100,000

(b) Results

I/O	Create	Delete	Open	Write	Read	Size per File	Total Written
Counts	51	44	102	11,169	4,908	5.3 MB	271.1 MB

To evaluate how well the local key-value store works without supports from a conventional FTL, I/O access patterns sent to the storage device are captured at LibIO. It is confirmed that all of the write requests are sequential and append-only, and there are no in-place-updates to the storage device. RocksDB performs its own garbage collection, also called compaction, to reclaim free space, thereby eliminating the need for garbage collection at the level of the FTL. Figure 4-3 shows an example of I/O patterns sent to the storage device.

Finally, NOHOST is evaluated under various usage scenarios for a key-value store. For this purpose, *db_ test* application that comes with RocksDB is used. As depicted in Figure 4-4, NOHOST software passes all of the test scenarios with DRAM-emulated flash.

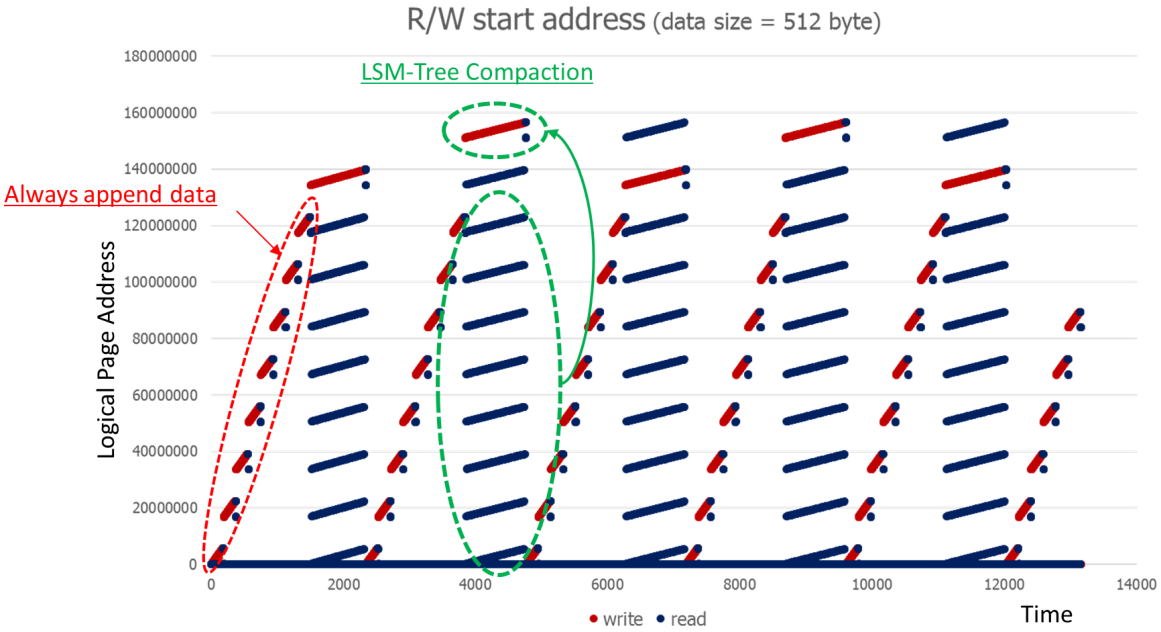


Figure 4-3: I/O Access Patterns (reads and writes) captured at LibIO

```
max_length: 1048576, left: 4739072
max_length: 1048576, left: 3690496
max_length: 1048576, left: 2641920
max_length: 1048576, left: 1593344
done
max_length: 1048576, left: 4739072
max_length: 1048576, left: 3690496
max_length: 1048576, left: 2641920
max_length: 1048576, left: 1593344
[       OK ] DBTest.LargeBatchWithColumnFamilies (7145 ms)
[ RUN      ] DBTest.FlushesInParallelWithCompactRange
[       OK ] DBTest.FlushesInParallelWithCompactRange (933 ms)
[ RUN      ] DBTest.DelayedWriteRate
[       OK ] DBTest.DelayedWriteRate (326 ms)
[ RUN      ] DBTest.HardLimit
[       OK ] DBTest.HardLimit (160 ms)
[ RUN      ] DBTest.FailWhenCompressionNotSupportedTest
[       OK ] DBTest.FailWhenCompressionNotSupportedTest (60 ms)
[ RUN      ] DBTest.PrevAfterMerge
[       OK ] DBTest.PrevAfterMerge (41 ms)
[ RUN      ] DBTest.DeletingOldWalAfterDrop
[       OK ] DBTest.DeletingOldWalAfterDrop (68 ms)
[ RUN      ] DBTest.UnsupportedManualSync
[       OK ] DBTest.UnsupportedManualSync (36 ms)
[ RUN      ] DBTest.SSTsWithLdbSuffixHandling
[       OK ] DBTest.SSTsWithLdbSuffixHandling (175 ms)
[ RUN      ] DBTest.PauseBackgroundWorkTest
[       OK ] DBTest.PauseBackgroundWorkTest (267 ms)
[----------] 91 tests from DBTest (159200 ms total)

[----------] Global test environment tear-down
[==========] 91 tests from 1 test case ran. (159201 ms total)
[  PASSED  ] 91 tests.
```

Figure 4-4: Test Results with *db_test*

4.3 Integration of NOHOST Hardware and Software

After evaluating NOHOST software and hardware separately, they were integrated into a full system, and it is confirmed that the NOHOST software system runs on a real hardware system. Figure 4-5 shows a snapshot of LibIO of the NOHOST system running on a ZC706 board. Since the current NOHOST implementation is not mature enough to run the *db_test* bench, the integrated NOHOST system has been tested using synthetic workloads that issue a series of read and write operations to a flash controller. An enhancement of NOHOST to run more complicated workloads (e.g. *db_test*) will be conducted in the future.

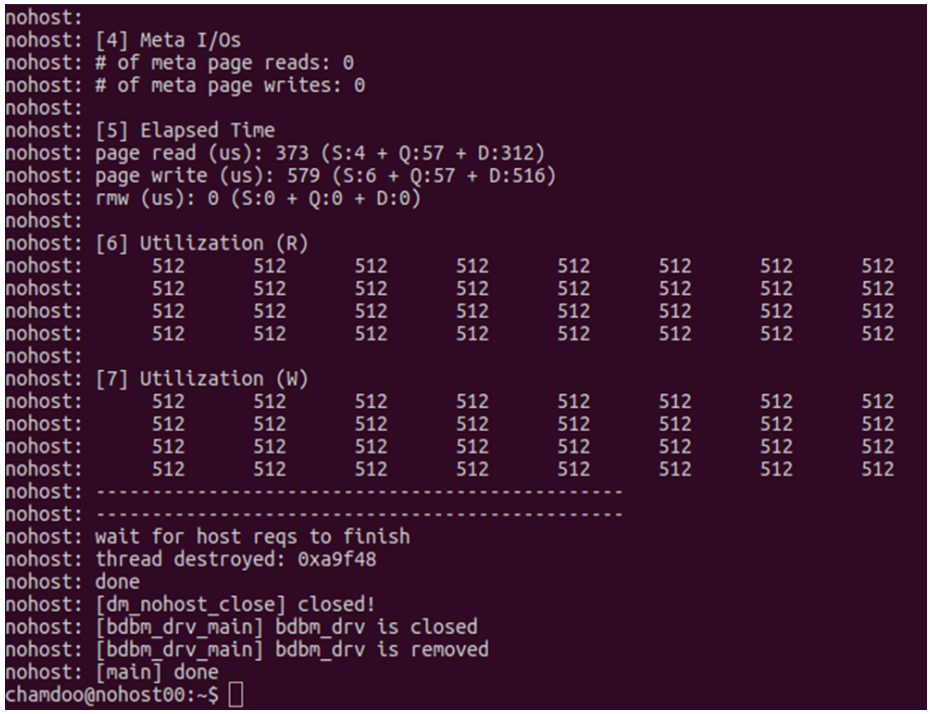


Figure 4-5: LibIO Snapshot of NOHOST with integrated hardware and software

Chapter 5

Expected Benefits

In this chapter, the expected benefits of NOHOST are discussed. NOHOST is expected to have several advantages, in terms of cost, energy, and space, over conventional storage servers. For a fair comparison, NOHOST is compared with EMC's all-flash array solution, XtremIO 4.0 X-Brick [4]. Note that, for NOHOST, its performance, power consumption, and space requirements are estimated based on the evaluation of NOHOST design components presented in Chapter 4.1 and the previous study on BlueDBM [11]. Table 5.1 compares NOHOST and the XtreamIO in terms performance, power, and space requirement.

Table 5.1: Comparison of EMC XtremIO and NOHOST

	XtremIO 4.0 X-Brick	**NOHOST**
Capacity	40 TB	40 TB
Hardware	1 Xeon server + 25 SSDs	40 nodes
Max. Bandwidth	3 GB/s	2.8 GB/s
Power	816 W	400 W
Rack Space	6 U	2 U

EMC's XtremIO 4.0 X-Brick is an all-flash array storage server. Similar to other all-flash arrays, it is dedicated to data access and nothing else, but it is also a powerful server with high-performance Intel Xeon processors. According to its specifications, the XtremIO requires 816 W and 6 U rack space [4]. Its total capacity is 40 TB with 13 SSDs. The XtremIO offers 3.0 GB/s maximum throughput with 0.5 ms latency and provides 4 Fibre Channel ports and 2 Ethernet ports.

The custom flash board consumes 5 W per a card (512 GB) [11, 20]. Assuming 20 W of the power consumption of a Xilinx ZC706 board, a 1-TB NOHOST prototype with two flash boards consumes 30 W. This power value is measured using the prototype based on Xilinx evaluation boards equipped with redundant components. Thus, its actual power consumption might be much lower than 30 W. Hitachi's Accelerated Flash employs four 1-GHz ARM cores with at least 1 GB DRAM, which would be similar to the hardware specification of our NOHOST node. The medium-capacity model consumes 7.8 W per 1 TB [8]. Thus, it is reasonable to assume that a NOHOST node requires 10 W per 1 TB. The power consumption of the 40 TB NOHOST cluster would be about 400 W, which is about 2x lower than the XtremIO. If a NOHOST node requires the similar space as Hitachi's Accelerated Flash, the cluster of 40 TB nodes occupies 2 U rack space, accomplishing 300 % space saving in a data center. Note that the power consumption can be lowered if a single node has a more capacity (e.g., 4 or 8 TB). It is also assumed that all the nodes are same as the master node, so the overall power consumption would be further reduced if slave nodes are implemented using simpler hardware without embedded cores.

According to Section 4.1.4, each board achieves the throughput of 1.26 GB/s with 100 µs latency for reads and the throughput of 461 MB/s with 600 µs latency for writes. Since our node-to-node network supports multiple flash boards to operate fully in parallel, the maximum throughput of the master node is limited to the DMA performance, 2.8 GB/s. This suggests that NOHOST offers the similar performance as the XtremIO. As a result, the NOHOST cluster would achieve similar performance but much less power and physical space than the EMC's XtremIO.

Chapter 6

Conclusion and Future works

In this thesis, a new distributed storage architecture, NOHOST has been presented. A prototype of NOHOST has been developed and confirmed that a RocksDB-based local key value store and RebornDB for Redis Cluster run on NOHOST. It is expected that NOHOST uses approximately half the power and one-third the physical space, while showing a comparable throughput of 2.8 GB/s as compared to Xeon-based current systems.

In the future, it is imperative to evaluate the performance of a NOHOST system in a distributed setting and optimize it to show performance comparable to the modern storage architecture. Along with these, it is planned to implement hardware accelerators in the current prototype for in-store processing and to add more advanced functionalities for fault tolerance. In this chapter, the future works on improving NOHOST are discussed in detail.

6.1 Performance Evaluation and Comparison

RocksDB comes with *db_ bench*, a benchmark suite with configurable parameters such as a dataset size, key-value size, software compression scheme, read/write workload, and access pattern. It provides us with useful performance measurements such as a data transfer rate and I/O Operations Per Second (IOPS). Future studies on the evaluation and comparison of NOHOST will be done as follows.

- **Identification of Software Bottleneck:** NOHOST is designed to offer raw flash performance to compute nodes, fully utilizing available network bandwidth. The evaluation goal is thus to measure the end-to-end performance from NOHOST nodes to computing clients and to identify potential bottlenecks. To understand the effect of embedded ARM cores on performance, a comparison study will be conducted with x86 processors that run the same NOHOST software on BlueDBM machines. Since BlueDBM machines use the same custom flash board, software-level bottlenecks under ARM cores will be clearly identified.

- **Effects of System-level Refactoring:** Using previously developed software modules to mount a file system on the custom flash boards, original RocksDB can run on NOHOST nodes without any modifications [11, 20]. Comparing NOHOST system with the original RocksDB on NOHOST hardware will help us to identify how much software overheads are eliminated by software refactoring, in addition to useful information that shows which layers or modules still act as bottlenecks and can be further refactored and optimized.

- **Comparison with Commodity SSDs mounted on an x86 Server:** This setting is a conventional flash-based storage architecture where the server mounts a file system and manages several SSDs. The original RocksDB is already configured to run on this setting. Since the goal is to build a distributed storage system with comparable performance, it is critical to compare NOHOST with the conventional architecture.

6.2 Hardware Accelerators for In-store Processing

NOHOST benefits from its distributed setting and possible in-store processing. The BlueDBM study has demonstrated the effectiveness of distributed reconfigurable in-store accelerators in many applications such as large-scale nearest neighbor searching [11, 10]. It is expected that in-store accelerators are still effective in NOHOST,

just like in host server-based BlueDBM. Since NOHOST hardware is also reconfigurable, in-store accelerators, which process data directly out of local and remote flash chips, can be easily added. Figure 6-1 shows an integrated hardware accelerator and data paths from flash to software. The accelerator is placed in-path between the node-to-node network and software to process data stream from flash without adding additional latency. Furthermore, a well-designed hardware accelerator outperforms software and consumes much less power. In a resource-constrained NOHOST environment, hardware accelerators are essential.

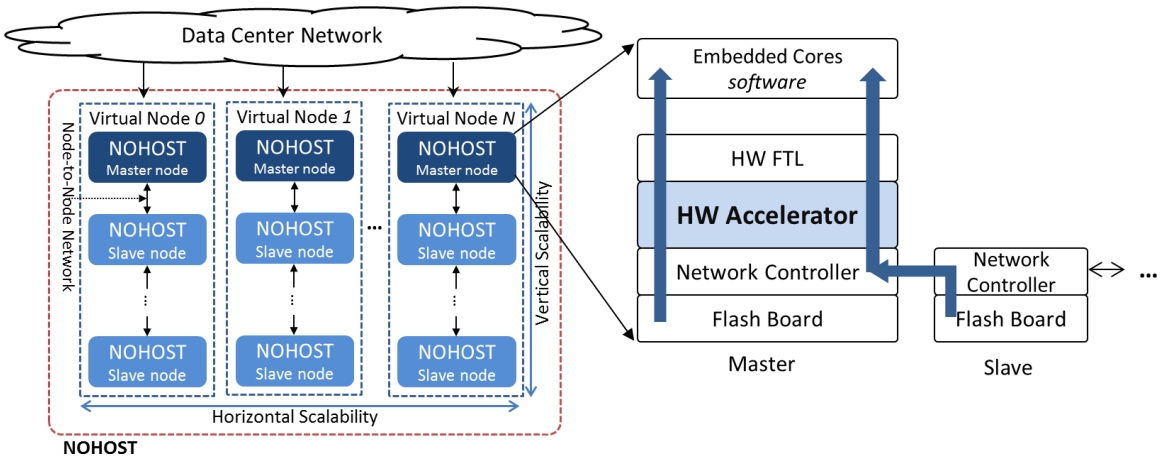


Figure 6-1: In-store hardware accelerator in NOHOST

Several applications for hardware acceleration in NOHOST are as follows:

- **Bloom filter:** RocksDB uses an algorithm to create a bit array called a Bloom filter from any arbitrary set of keys. A Bloom filter is used to determine if the file may contain the key that a user is looking for [5]. Because operations on Bloom filters are known to shine in hardware implementation, these operations can be offloaded to a hardware accelerator [21].

- **Compression:** Many open-source projects including Cassandra, Hadoop, and RocksDB use the Snappy library for a fast data compression and decompression [5, 1, 15, 7]. It is expected that software-implemented compression algorithm may not be feasible on resource-constrained embedded systems like NOHOST.

A compression algorithm suitable for hardware implementation can be used with RocksDB to support compression without performance degradation in NOHOST.

- **Deduplication:** Most enterprise storage servers support data deduplication. Integrating hardware assisted deduplication in NOHOST adds more functionalities.

6.3 Fault Tolerance: Hardware FTL Recovery from Sudden Power Outage (SPO)

Even though key-value pairs are stored in nonvolatile flash memory, hardware FTL tables reside in volatile RAM in NOHOST. Unlike a commodity SSDs whose firmware uses complex mechanisms to ensure fault tolerance, current NOHOST prototype cannot recover the tables from power failure. To improve NOHOST more fault-tolerant, it is planned to add a mechanism that recovers FTL tables from a sudden power outage (SPO). When implementing the FTL recovery mechanism, it is assumed that NOHOST nodes have capacitors that will keep the backup modules working for a short amount of time in case of SPO. Followings are several implementation ideas.

- **Using the spare area of a flash page:** The flash page in our flash chip is 8,224 Bytes including 32 Bytes as spare area (8,192 + 32 Byte). Hardware FTL table data can be recorded on these spare areas. Whenever NOHOST writes to a new flash page, NOHOST records logical address, current block PE cycles, and other relevant information in spare area.

- **Table backup in flash:** Table snapshots are stored in flash. In this case, it is required to keep the flash address that points where the most recent snapshot is stored. If a few blocks for storing these address are designated, only these blocks need to be scanned to find the valid most recent snapshot address.

In the event of SPO, the first mechanism ensures that the block mapping table can be recovered by scanning all the spare areas, but it cannot not recover PE cycle information of already erased blocks. The second mechanism can be complementary to the first. The second scheme does not require a full scan of flash. However, there might be a case where the last snapshot is incomplete due to power failure while writing tables. In this case, the second-to-last snapshot can be used to estimate PE cycles to rebuild the block status table, and scanning whole flash spare area recovers the block mapping table.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] Apache. The Apache Hadoop software library. [Online]. Available: http://hadoop.apache.org, Accessed: Aug 1, 2016.

[2] Amir Ban. Flash file system, April 4 1995. US Patent 5,404,485.

[3] DB-Engines. Ranking of Key-value Stores. [Online]. Available: http://db-engines.com/en/ranking/key-value+store, Accessed: Aug 1, 2016.

[4] EMC. EMC XtremIO 4.0 System Specification. [Online]. Available: https://www.emc.com/collateral/data-sheet/h12451-xtremio-4-system-specifications-ss.pdf, Accessed: Aug 1, 2016.

[5] Facebook. RocksDB. [Online]. Available: http://rocksdb.org, Accessed: Aug 1, 2016.

[6] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, New York, NY, USA, 2003. ACM.

[7] Google. Snappy - a fast compressoin/decompression library. [Online]. Available: http://google.github.io/snappy, Accessed: Aug 1, 2016.

[8] Hitachi. Hitachi accelerated flash datasheet. [Online]. Available: https://www.hds.com/en-us/pdf/datasheet/hitachi-datasheet-accelerated-flash-storage.pdf.

[9] Instagram. Instagram Stats. [Online]. Available: https://instagram.com/press, Accessed: Aug 1, 2016.

[10] Sang-Woo Jun, Chanwoo Chung, and Arvind. Large-scale high-dimensional nearest neighbor search using flash memory with in-store processing. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–8, Dec 2015.

[11] Sang-Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. BlueDBM: An Appliance for Big Data Analytics. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA '15)*, pages 1–13, New York, NY, USA, 2015. ACM.

[12] Sang-Woo Jun, Ming Liu, Shuotao Xu, and Arvind. A transport-layer network for distributed FPGA platforms. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2015.

[13] Myron King, Jamey Hicks, and John Ankcorn. Software-driven hardware development. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 13–22. ACM, 2015.

[14] Redis Labs. Redis. [Online]. Available: http://redis.io, Accessed: Aug 1, 2016.

[15] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[16] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[17] Sungjin Lee, Jihong Kim, and Arvind. Refactored Design of I/O Architecture for Flash Storage. *IEEE Computer Architecture Letters*, 14(1):70–74, Jan 2015.

[18] Sungjin Lee, Ming Liu, Sang-Woo Jun, Shuotao Xu, Jihong Kim, and Arvind. Application-Managed Flash. In *14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 339–353, Santa Clara, CA, February 2016. USENIX Association.

[19] Ming Liu. BlueFlash: A Reconfigurable Flash Controller for BlueDBM. Master's thesis, Massachusetts Institute of Techonology, Cambridge MA, 2014.

[20] Ming Liu, Sang-Woo Jun, Sungjin Lee, Jamey Hicks, and Arvind. minFlash: A minimalistic clustered flash array. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE '16)*, pages 1255–1260, March 2016.

[21] Michael J Lyons and David Brooks. The design of a bloom filter hardware accelerator for ultra low power systems. In *Proceedings of the 2009 ACM/IEEE international symposium on Low power electronics and design*, pages 371–376. ACM, 2009.

[22] OCZ. SSD vs HDD | Why Solid State Drives Are Better Than Hard Drives. [Online]. Available: http://ocz.com/consumer/ssd-guide/ssd-vs-hdd, Accessed: Aug 1, 2016.

[23] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[24] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[25] RebornDB. RebornDB. [Online].
Available: https://github.com/reborndb/reborn, Accessed: Aug 1, 2016.

[26] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[27] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[28] Xilinx. Aurora 64B/66B LogiCORE IP Product Guide. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/aurora_-64b66b/v10_0/pg074-aurora-64b66b.pdf, Accessed: Aug 1, 2016.

[29] Xilinx. Aurora 8B/10B LogiCORE IP Product Guide. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/aurora_-8b10b/v11_0/pg046-aurora-8b10b.pdf, Accessed: Aug 1, 2016.

[30] Xilinx. Virtex-7 FPGA VC707 Evaluation Kit. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/VC707-Kit-Product-Brief.pdf, Accessed: Aug 1, 2016.

[31] Xilinx. Zynq-7000 SoC ZC706 Evaluation Kit. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/Zynq_ZC706_Prod_-Brief.pdf, Accessed: Aug 1, 2016.

[32] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your log on my log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*, 2014.