École Polytechnique Fédérale de Lausanne

Data Center Systems Laboratory

Master Thesis

Master in Computer Science

Storage in IX: a High Performance Storage Layer for a Dataplane Operating System

Author:
Lisa Zhou

Supervisors:
Marios Kogias
Prof. Edouard Bugnion

Saturday 8th July, 2017

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Acknowledgements

They say it takes a village to raise a child . . . here are some of the "villagers" who helped "raise" this wayward Masters student.

First and foremost, I would like to thank Prof. Ed Bugnion and Marios Kogias for granting me the opportunity to work on this thesis project. The chance to work with brilliant systems minds is rare and I hope the lessons I've learned while working with you both will remain with me well after my time at EPFL has ended.

To the lovely members of DCSL and its neighbouring lab VLSC, whose kindness and congeniality have made this experience wholly enjoyable, sometimes quirky, and in all earnestness not at all awkward.

Special thanks to those who agreed to proofread drafts of this lengthy document.

This work and all others works produced by me over the course of my life was generously supported by my mother; without her encouragement I would never have been able to journey so far, both geographically and in my studies.

And last but not least, my immense gratitude to Mssr. Y. R. Stepienik, for putting up with me.

Abstract

As storage hardware evolves, the systems and components which interact with them must adapt as well. Outdated assumptions at the operating system level necessitate a dramatic rethinking of the storage path as emerging storage technology offer performance speeds several orders of magnitude faster than traditional hard disks. This thesis investigates the design space of contemporary storage systems, in particular for non-volatile memory (NVM) technology, and presents a survey of the techniques currently being employed to build systems for NVM. We identify and examine relevant systems to determine features applicable to building a storage extension for IX, a dataplane operating system. We present the design and implementation of such a system and detail the assumptions and principles we strove to incorporate.

Contents

Chapter 1

Introduction

1.1 Overview

Advances in the modern computing world have rapidly made assumptions in existing systems tragically obsolete. Improved technology such as 10-100Gb/s Ethernet and specialized processors have necessitated a redesign of traditional operating systems. One field where this phenomenon is particularly pertinent is storage, in which rapidly advancing hardware innovations are forcing a re-examination of storage abstractions and interfaces. New non-volatile memory (NVM) technologies promising performance closer to DRAM will require novel approaches to the existing storage paradigm. Similarly, while Flash-based solid state drives (SSD) have existed for more than a decade, the interface with which operating systems interact with these devices still assume outdated disk-based mechanisms.

Simultaneously, event-driven, asynchronous approaches to programming and data management have risen in popularity. The asynchronous strategies outlined in the original C10k problem have found applicability in everything from popular programming language foundations and frameworks [35, 52] to widely deployed web applications [47, 66]. Additionally, the key value abstraction has become as ubiquitous as the relational databases of yesteryear, and is emerging as the preferred form with which low-latency web applications interact with data.

Tails at Scale and Killers at Large

The recent growth in large-scale multi-server distributed web applications has caused latency, and in particular tail latency, to seize the attention of both industry professionals and researchers. In the case of many Software-as-a-Service (SaaS) offerings, providers face stringent service level agreements (SLAs) in which failures to meet objectives, sometimes at the microsecond scale, have economic repercussions [24]. The ubiquity of cloud computing, made possible by the contributions of [13], have pushed latency to the foreground of systems concerns; no longer a less marketable trait, latency holds arguably more importance than bandwidth in the current computing climate [62].

Many of the coping strategies for improving latency proposed a decade ago [62] have yielded adequate results. Indeed replication and caching are among the most common strategies applications and web services architects employ to mitigate latency. However, these strategies come at a significant cost. Both replication and caching require more resources, typically in the form of fast volatile memory. In the need for low micro-second scale latencies, some applications have taken the approach of over-provisioning large amounts of DRAM to attain their latency requirements and relying on a background process to periodically back-up data to persistent storage [66, 22, 76]. While this two-tiered strategy has worked well so far, it is cost inefficient and even unreliable in certain cases.

Moreover, it is not clear how to "solve" these latency concerns at the microsecond scale. As noted recently by Barroso et. al. [7], strategies to mitigate nanosecond and millisecond level latencies do not apply at the microsecond scale. Moreover, for large-scale services, seemingly insignificant microsecond overheads build up to a "death by 1000 cuts", making it critical for systems to mitigate or eliminate these "killer" microseconds.

From Linux to in-IX

Systems designers have an imperative to harness the capabilities of the latest state-of-the-art technologies to allow users to experience the performance improvements of low-latency hardware without imposing overheads that plague existing operating systems. As Nanavati et. al. [50] note, the increased cost of emerging non-volatile memory technologies coupled with their improved throughput capabilities spell a complete reversal of the existing I/O design

assumptions that storage is slow and cheap while "compute" resources are comparatively fast and more expensive. To build effective systems for these technologies, a redesign of the conventional I/O stack is needed; modifying Linux is not an option. As one Linux filesystem developer notes [21]:

> "tuning Linux filesystems to work well on solid-state storage devices is a lot like working on an old, clunky car. Lots of work goes into just trying to make the thing run with decent performance. Old cars may have mainly hardware-related problems, but, with Linux, the bottleneck is almost always to be found in the software. It is ... hard to give a customer a high-performance device and expect them to actually see that performance in their application."

Incorporating these concerns and observations, our project sets out to explore the performance possibilities of a new storage paradigm built within IX [9], a high performance dataplane operating system. Just as IX sought to reinvent the networking stack to provide low latency and high throughput with minimal tradeoffs, we examine the impact of applying the same design principles to I/O processing. As storage devices promise ever-decreasing microsecond-scale latencies, we examine the possibility of building a storage layer which employs an asynchronous event-based model to minimize I/O latencies and maximize throughput. We seek to leverage the latest commercially available hardware and interfaces to complement our design.

1.2 Outline

In this work, we set out to understand the issues and trade-offs in the modern storage space when working with the latest hardware. We perform a survey of research endeavours related to NVM and explore some state-of-the-art storage designs. We investigate a few existing projects in the storage space to draw inspiration for our design and justify our decision to implement a key-value abstraction as our interface. We then outline our storage extension for IX, a state-of-the-art networking dataplane kernel, and discuss the considerations and inspirations that led us to our ultimate implementation. We try to provide a modest evaluation, and explain the limitations that affected these evaluations. Lastly, we discuss other relevant works in academia as well

as how our work can be extended to other applications, and conclude with the lessons we learned and insights we gained.

Chapter 2

Motivational Background

Before we are able to discuss our work, we must first examine the context in which it came to be. First we notice the divergence between existing operating system assumptions and the latest developments in storage hardware. We dissect some subtleties between state-of-the-art technologies in the "non-volatile memory" sphere in an attempt to provide a clear backdrop for our work and dispel any confusion. We then take an in-depth look at two research endeavours that exemplify the kind of approach needed to tackle the problem of designing a storage system. Lastly, we identify some design trends and motivate our ultimate design directions.

2.1 Contemporary Storage — the Hardware-OS mismatch

Historically, storage systems could safely assume that the underlying devices operate at millisecond-scale latencies. Operating systems would submit I/O operations to disk, and then context switch to execute other processes while waiting for an interrupt from the storage interface to notify that the request was completed. Moreover, most operating system I/O handling is designed to be tightly coupled with the characteristics of the underlying storage device, and I/O requests may be reordered and/or batched to obtain better sequentiality on disk or exploit ever-growing hard drive disk (HDD) bandwidths [38].

These assumptions may remain largely stable if systems continue to rely on disks; latencies of HDDs have not dramatically changed over the years

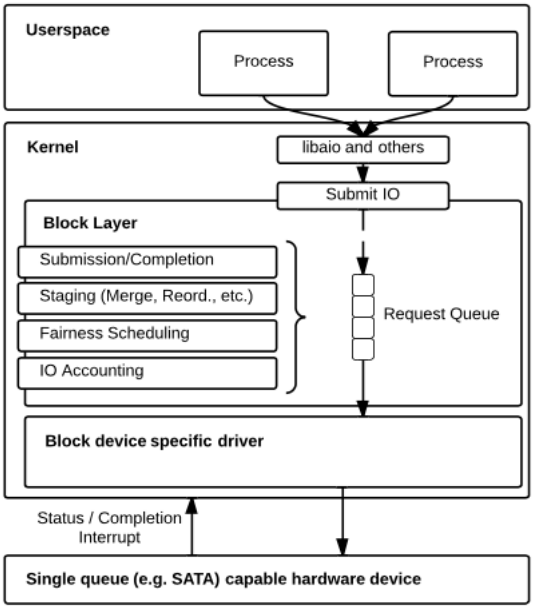

Figure 2.1: The layers within Linux's storage stack (Source: [10])

[48]. But the advent of various flash-based storage technologies requires a re-examination of OS-level I/O processing. While Flash technologies and solid state drives (SSDs) have existed for a long time, operating system design has not caught up to sufficiently make use of new hardware capabilities. NVMe, or "Non-volatile Memory Express", is a protocol specification for SSDs over PCIe which is poised to replace legacy disk-based interconnects (SATA/SAS). However, even though it has been under development since 2007, NVMe has yet to see widespread adoption, at least at the datacenter level, making up only 15% of enterprise SSD sales in the last year [55, 70]. Emerging "non-volatile memory" hardware (also varyingly referred to as "storage class memory") also entice with purported nanosecond-scale latencies [48], rendering even PCIe too slow to provide appropriate connectivity.

Figure 2.1 illustrates the traditional Linux storage stack [10]. In order to better understand the design decisions within a storage layer, we must understand the underlying components and their interactions. Figure 2.2 gives a breakdown of performance between the various hardware. Note that it is not currently clear what is the relationship, if any, between technologies

	DRAM	PCM	RRAM	MRAM	SSD	HDD
Read latency	60 ns	50 ns	100 ns	20 ns	25 μs	10 ms
Write latency	60 ns	150 ns	100 ns	20 ns	300 μs	10 ms
Addressability	Byte	Byte	Byte	Byte	Block	Block
Volatile	Yes	No	No	No	No	No
Energy/ bit access	2 pJ	2 pJ	100 pJ	0.02 pJ	10 nJ	0.1 J
Endurance	$>10^{16}$	10^{10}	10^{8}	10^{15}	10^{5}	$>10^{16}$

Figure 2.2: A comparison of storage hardware performance (Source: [2])

labelled as "NVM" and the NVMe protocol, which was designed with SSDs over PCIe in mind.

2.1.1 Hardware

NVM

"Non-volatile memory" (NVM) has been used as an umbrella term for an emerging set of technologies such as phase-change memory (PCM), memristors, and spin-transfer-torque magnetic RAM (STT-MRAM). NVM is purported to offer raw performance at DRAM-like speeds with much higher density, allowing for greater storage capacity. Moreover, it is designed to expose a byte-addressable interface, enabling possible DRAM-like "load-store" interfaces. While it is currently unclear what is the ideal usage case and environment for this technology, a slew of research effort has been under way to find ways to integrate NVM into systems, including but not limited to file systems [64, 74, 20] and databases [63, 58].

Flash-based SSDs

Flash technology has been around for more than a decade, but has only recently seen increased adoption in datacenters due to diminishing cost-to-capacity ratio; recent trends show that Flash capacity-per-dollar growth is almost on par with that of hard disk [38]. Flash exhibits various properties that require careful management and utilization. Typical NAND flash con-

sists of chips of single-level cells (SLC)[1] which can be programmed and read in units of pages, but only erased in units of blocks. Once a page has been written to, it must be erased to be reclaimed. However, since block sizes are typically many multiples of a page, data placement and deletion must be managed carefully. For instance, pages are usually 2-4KB while blocks may be 128-256KB [1]. Moreover, each cell has a specific wear "lifetime", i.e. the number of times a cell may be erased and reprogrammed (typically referred to as P/E cycles).

These characteristics mean that flash devices must be managed carefully to optimize data placement and prevent cells from wearing unevenly. The software component in SSDs, typically referred to as the Flash Translation Layer (FTL), is not only responsible for translating logical block requests into physical page numbers, but must also consider such "wear-leveling" issues. The FTL must also perform garbage collection to reclaim sparsely used blocks in the case of a space shortage, and may regroup existing pages and rewrite them to other blocks. A technique commonly employed by FTLs is to manage data in a log-structured way to leverage sequentiality for better performance [68].

Another interesting property exhibited by SSDs is "internal parallelism." To cope with asymmetric read/write performance (writes are slower) and per-chip bandwidth limitations, SSD architects introduced multiple channels to enable reading and writing to several chips in parallel with isolated performance [16]. In addition to increasing overall device bandwidth capabilities, this also allows high latency procedures such as erase to be masked behind other operations occurring in parallel. New "open-channel SSDs" offer exposition of this parallelism directly to the overlying host system [57].

Applications designed for SSDs must be wary of issues such as write amplification, in which certain write workloads patterns may cause the FTL to trigger garbage collection or reorganize data. Conversely, applications may also exploit internal parallelism inherent in SSD devices, provided that they have a deep understanding of the internal SSD mechanisms [16].

[1]Multi-level and triple-level cells that can encode more bits per cell also exist [1]

2.1.2 Interfaces

SATA/SAS

These two interfaces are the modern descendants of ATA and SCSI but are commonly used interfaces for SSDs today. Originally designed with hard disk drive semantics in mind (and in the case of SCSI, even older and more diverse devices), these interfaces were made forward-compatible for SSDs and typically feature a single request queue with a depth of 32.

NVMe

NVMe is the newest protocol specification for SSDs over PCIe. Among the key features exposed by this set of standards is the availability of a large number of concurrent submission/completion queues; NVMe can support 64k submission/completion queue pairs each with a depth of approximately 64k [54]. NVMe also exposes a namespace abstraction that specifies the range of valid logical block addresses (LBAs), which enables one NVMe device to support multiple namespaces [54].

Experiments conducted by others have shown that in a breakdown of time spent in software processing, systems using SATA spend roughly 20% more of their total I/O processing time in interface logic than systems using NVMe [79]. Other benchmarks have revealed that NVMe exhibits between 1.5-3x better IOPS throughput than SAS, the state-of-the-art SCSI-based interconnect [53].

2.2 Other Systems

Before we look at specific issues in the non-volatile memory and flash design space, we examine two storage designs that serve a more generic purpose in a broader scope. We encountered these systems in our quest for a suitable model for our storage extension, and while neither directly relates to non-volatile memory technologies, they both exhibit interesting system design principles. Where one tries to design a way to achieve low latency persistent storage without relying on high-performance hardware, the other depends on more advanced hardware functionality to express features typically found in software. We examine both in detail.

2.2.1 RamCloud

RAMCloud is a highly available, strongly consistent, two-tiered storage system designed to support mass-scale storage at micro-second scale latencies. The authors state that their system is design to enable commodity servers to achieve reads and writes and tens of microseconds. Reads are served from DRAM while writes are persisted to secondary storage using a multi-replication log-structured approach. Low latency writes are achieved by issuing multiple writes to a set of randomly selected replication servers, which respond that the write "completed" once the data is stored in their local DRAM while issuing asynchronous writes to their respective secondary storage. Only one copy of data is stored in each server's memory, as randomized replication over backup servers provides fault tolerance and a fast recovery process ensures high availability.

All data is stored in a log-structured fashion; 8MB log segments are sequentially written to in an append-only manner and organized in the same way in both memory and on hard drive. Tombstone entries indicate that existing keys were modified or deleted, and a log cleaner is run periodically to reclaim space that is no longer used. RAMCloud employs "two-level cleaning" in order to reclaim space from memory more frequently than cleaning the log on disk, thus imposing less overhead for cleaning while gaining better utilization from DRAM.

RAMCloud achieves fast recovery on node failure using interesting scalability properties. When a node fails, multiple backup servers each containing replica segments of the original node's data transfer these segments to multiple "recovery masters." By multiplexing both sides of the recovery process, RAMCloud is able to ensure fast recovery of data from a crashed node. RAMCloud leverages randomization in order to distribute its data and also select recovery masters in a fault tolerant manner.

RAMCloud makes some bold assumptions about the future of networking infrastructure, in particular the fact that ultra-low latency fabrics such as Infiniband will be widely available. In order to achieve low microsecond-scale latency, the authors assume RAMCloud will be deployed over Infiniband, and implement their own transport layer leveraging Infiniband's reliable queue pairs. While they also implement two other forms of transport, including a standard TCP-based one, they note that RAMCloud's extremely low latency performance relies heavily on the performance of Infiniband NICs and their ability to bypass kernel processing.

While RAMCloud leverages scalability to achieve low latency and fast recovery, scalability is also somewhat of a limitation. Consider a cluster of 10 nodes, the minimum reasonable size to run this system on. By their own (circa 2011) calculations, for a cluster of size n each node is capable of storing 500MB $\times$ n for a recovery time of 1 second (although per-node capacity scales linearly with the acceptable amount of recovery time). The total storage capacity of this cluster would be 50GB, but given standard ($2n$+1) replication standards, there should be at least 3 replicas of everything stored in the system. This means that the system would only be capable of storing 15GB of "unique" data. For small to medium-sized enterprises, this kind of infrastructure is unfeasible and expensive, as the cost of having multiple servers to store partitions is compounded by the capacity limitation of how much data can be actually stored.

2.2.2 Arrakis

Arrakis is a novel operating system that argues for the removal of the kernel from the data path by relying on hardware virtualization support to grant applications direct access to virtualized storage and networking interfaces. The authors systematically identify areas of overhead in existing kernels, noting that in both the storage and networking path much of the time is spent de-mutiplexing and routing requests, copying data and checking access control. They note that recent advances in hardware make it possible to offload some of these tasks to the hardware interface directly. In the Arrakis design, the kernel becomes a mere resource management layer that oversees allocation of virtual network interface cards (VNICs) and virtual storage interface controllers (VSICs) exposed by the underlying network and storage devices. Arrakis leverages NIC queues and relies on protection mechanisms such as transmit and receive filters to handle routing, rate limiting, and access control for requests from the application level. DMA is used to ensure zero-copy packet processing. Applications interact with their own virtualized storage areas (VSAs) without requiring request translation or access control mediation from the kernel.

The Arrakis storage design depends heavily on hardware technology such as Single-Root I/O Virtualization (SR-IOV), which allows multiple guests on a physical host to have direct access to a PCIe connected device. Applications can thus directly interact with the virtual device exposed by SR-IOV, referred to as a Virtual Storage Area (VSA) in the system, without needing to go

through the kernel for I/O. Access control and address translation is also handled by hardware (using IOMMU).

The authors evaluate their implementation with both a "native" asynchronous interface as well as a POSIX-compliant one, noting that the POSIX version performance slightly worse (9% less throughput). However, Arrakis is not currently implementable on existing hardware; the authors admit that they have had to emulate the desired VSIC behaviour in their prototype. While they note that modern storage controllers, including FTLs in SSDs and NVMe, do have most of the features they require, they are missing the protection mechanism that enables separation of VSAs.

2.3 Suggestions and Solutions

Researchers have approached the problem of how best to leverage new low-latency storage hardware from various angles, with a dizzying array of assumptions and goals. Mittals and Vetter [48] have conducted a survey in which they examine and categorize over 100 recent research publications involving "non-volatile memory" by overall design goals and objectives. 11 different areas were identified, ranging from energy efficiency to checkpointing and error correction, although performance improvement was by far the most popular. It is interesting to note from their study that there were roughly the same number of proposals involving (NAND) Flash versus other NVM technologies, even though only the former is currently commercially available.

In order to leverage the low latency capabilities of NVM hardware, many have advocated eliminating the kernel from the data path altogether; a number of storage applications have been proposed that directly interface with hardware without intermediate abstraction layers [45, 20, 36]. However, recent work [43] detailing the efforts of porting `memcached` to a persistent memory interface offers a cautionary tale to those who underestimate the effort of migrating existing programming paradigms to persistent memory technologies. The authors note the immense difficulty of refactoring an application with strong coupling of data structures and ingrained assumptions about hardware behaviour, as well as detail their struggles determining what data to persist and how to maintain backwards compatibility. The need for an abstraction layer is palpable.

Moreover, with the volatility of hardware trends, it is unlikely that many

of these research prototypes will remain stable in the future. Systems designers must cope with the same issue in any case: allowing users to experience the performance improvements of low-latency hardware without imposing overheads that plague existing operating systems (i.e. Linux). One opinion that does seem to meet some consensus is that modifying Linux to adapt to new technologies is impractical. While some attempts have been made to make Linux more NVMe/SSD-friendly [10, 11], they do not address the fundamental issues in the I/O path in Linux [71]. For example, the asynchronous I/O completion model that Linux uses to perform other tasks while waiting for slow disk operations is completely incompatible with SSDs and faster storage media.

While there seems to be a lack of agreement what exactly the bright future of storage technology may bring, several key design principles and recommendations have been proposed. Although some of these points are not very consistent or compatible with each other, we do our best to characterize key approaches and outline how we have applied them to our system.

2.3.1 Abstract or Expose

On the spectrum of abstracting to exposing interfaces, there exist a variety of approaches depending on the goals of the system. Storage disaggregation systems [51, 37] tend to prefer abstracting and virtualizing storage resources, while novel key-value stores designed for high performance favour awareness of underlying hardware semantics [45, 6]. Some have even advocated exposing the device driver interface directly to application developers [36], echoing a continuing trend of applications demanding transparency in order to implement specialized functionality based on their needs.

How closely our design should be tied to the underlying hardware semantics was an issue with which we struggled. While a deep understanding of the hardware is important in order to make informed design decisions, we were hesitant to base too many components of our system on behaviour of specific hardware technologies. The tradeoff between specificity of hardware and being flexible to potentially support different types of storage media was a difficult decision but we did our best to find a suitable middle ground (we will detail the specific tradeoffs we made in subsequent sections).

2.3.2 Asynchronous or Synchronous

There have been opposing opinions voiced about where and how best to apply asynchronous processing. Some have pointed out that in the face of very low latency storage devices, synchronous polling for I/O completion at the operating system level is preferable to handling interrupts from the device driver [80]. On the other hand, Barroso et. al. [7] argue that asynchronous processing should be pushed down to the operating system level to ease the burden of programmers having to struggle with asynchronous programming paradigms.

We have opted to continue using the asynchronous semantics originally implemented by IX, which we will describe in the following section. Although asynchronous programming has its challenges, it is interesting to recall that the event-based asynchronous paradigm was at some point promoted as a simpler, more developer-friendly alternative for managing concurrency[59].

2.3.3 Byte or Block

Byte addressability poses new challenges that researchers from a variety of fields are trying to tackle [5, 82, 20]. However, without actual physical devices to experiment with, it is arguable how effective these largely theoretical and emulated efforts are.

While many in the community seem to be advocating abandonment of the traditional block-addressed interface and embracing byte-addressability [61, 19, 15], it is unclear what dangers might arise in building storage systems atop persistent hardware that exposes a byte-addressable interface. For instance, more careful error detection must employed, as any memory corruption a program might introduce will be persistent in NVM. While new programming models and data structures are being proposed [69, 75, 19] to cope with potential persistence issues, without validation on the actual technology, it is unclear how effective they will be.

We opted to work with NVMe, a more practical option, even though it has not sparked as much interest in the research community as its namesake[2]. We felt that it was more valuable to obtain a prototype on an already accessible technology rather than join the numerous speculators[3]. As IX,

[2] a cursory search of the ACM digital library shows that the difference in number of results returned between "NVM" and "NVMe" is roughly 10x in favour of the former.

[3] the cruel irony that this technology was ultimately not attainable to us is something

the technical foundation of our work, is designed for datacenters running commodity hardware, basing our storage extension on some state-of-the-art technology that has yet to reach mass market seems unsuitable.

that will be discussed in later sections.

Chapter 3

Design Background

Here we describe two systems which influenced our design and implementation and the specific design aspects we adopted. We first describe IX, the system which forms the architectural foundation of our work, and MICA, a key-value store from which we borrowed specific design elements.

3.1 IX

3.1.1 Overview

IX is the main foundation on which our work is built and whose goals we seek to extend to the storage plane. IX is a state-of-the-art dataplane operating system that provides simultaneous high throughput and low-latency while preserving traditional kernel isolation and security guarantees [9]. Its authors propose a reorganization of OS functionality by separating resource management from network processing, delegating each to the control plane and dataplane respectively. IX leverages Dune [8] to provide secure access (non-root ring 0) to dataplane instances, and uses VT-x virtualization features to grant applications access to hardware resources while providing three-way isolation between the control plane, dataplane and application layer.

IX aggressively trades coarse-grained resource allocation for synchronization-free primitives, thus achieving multi-core scalability. Cores are assigned dedicated NIC queues, memory pools and system call and user event arrays. System calls are batched to reduce kernel crossings. In order to achieve low latency, all packet processing "runs to completion" to eliminate overheads

from interrupts and improve throughput. Adaptive batching at "every stage of the network stack" allows the system to exploit gains from cache locality and prefetching while avoiding pitfalls such as waiting for batches and starving transmit queues. This is achieved by batching only during periods of congestion, and imposing a limit on batch size. The combination of these two techniques also results in a overall reduction of queueing within the system, instead pushing queue build-up to NIC edges.

IX is complemented by a library of event-based system calls that allow applications to perform asynchronous networking processing. Similar to `libevent` or `libuv`, `libix` provides a set of non-blocking system calls with zero-copy functionality. IX relies on this zero-copy API for communication between the dataplane and application layers. The dataplane maps packets to read-only message buffers for the application to access; conversely the application sends "scatter/ gather" lists of memory locations containing immutable content for transmission. A polling-based mechanism allows IX to obtain responses to application or networking layers without relying on interrupt mechanisms. Lastly, flow-consistent hashing of incoming traffic eliminates synchronization and coherence between cores as each hardware thread is given a dedicated receive and transmit NIC queue.

3.1.2 Design Influences

In addition to being the architectural foundation in which our storage abstraction resides, IX also embodies key design principles that our storage implementation aspires to encompass:

event-based library to allow an asynchronous programming interface for applications

batching for better throughput

dedicated resource allocation to reduce sharing/synchronization

We believe that these characteristics apply not only to the networking stack, but are also relevant in the building of an efficient I/O path. The extended queueing functionality exposed by NVMe lends itself suitably to IX's resource allocation strategy; each core is assigned a dedicated submission/completion queue for I/O, a direct analogue to a dedicated networking receive/transmit queue.

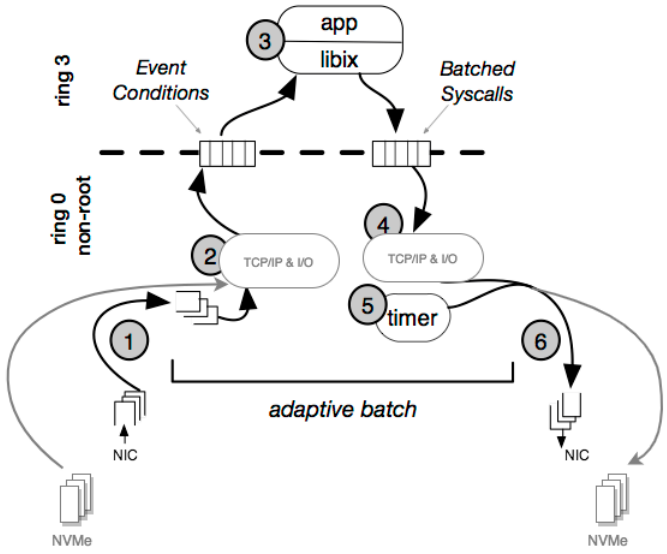

Figure 3.1: The IX run-to-completion loop (additions in grey).

3.2 MICA

3.2.1 Overview

Another system that heavily influenced our design was the MICA key-value store [39]. MICA is a "holistic" in-memory key-value store designed to optimize request processing at every stage, from partitioning data for per-core access to kernel bypass for networking processing. The authors of MICA felt that existing (TCP-centric) approaches to handle network processing do not offer sufficient performance, and note that strategies such as batching improve throughput at the cost of higher latency. They instead propose a co-designed solution using UDP in which clients have direct access to NICs and can encode "filtering" information in the UPD port number of each packet they send. Clients are assigned dedicated NIC queues to allow efficient packet processing by routing according to the filtering information provided.

MICA allows for both "cache" and "store" modes, in which the former trades random key eviction for performance while the latter preserves all data unless explicitly removed by the user. To accomplish the former, MICA

uses circular logs and "lossy" hashing which exhibit better insertion and hash performance at the cost of potential key eviction. Circular logs insert new items at the tail and evict or remove old or deleted items from the head, thus occupying a constant amount of storage and eliminating the need for expensive garbage collection or defragmentation mechanisms. Similarly, the hash index may evict entries if its hash buckets are full for a particular index value. The authors argue that key eviction is acceptable for an in-memory cache system as many other existing applications operate under the same principle.

The authors of MICA enable their cache implementation to support "store" mode by extending their hash index with "bulk chaining" to prevent evictions. The hash index maintains spare buckets to chain to the main buckets when an overflow occurs. In store mode, MICA uses segregated fits to manage memory allocation for key-value pairs, and boundary tags to coalesce free space.

MICA partitions all data between available cores, and can grant per-core EREW (Exclusive Read Exclusive Write) or CREW (Concurrent Read Exclusive Write) access to key-value pairs depending on the desired read vs. write performance. EREW eliminates inter-core coordination to serve requests, while CREW handles (read) skew better by allowing multiple cores to access the same data. Versioning is used in CREW mode to ensure correctness in concurrent accesses.

3.2.2 Design Influences

While MICA is predominantly designed for in-memory key-value operations on small data (it does not support key-value pairs larger than one MTU), we felt some of its design choices were applicable to our storage implementation. We noticed that the MICA design was fairly analogous to the existing IX architecture as both systems feature a zero-copy networking stack that largely bypasses kernel processing; drawing from their structure for the storage layer would integrate well into the existing IX architecture. We also observed that MICA's per-core EREW/CREW design would be suitable to apply to NVMe; partitioning the range of available LBAs into per-core storage regions would allow better parallelism and scalable storage.

We also drew inspiration from MICA's "store" mode, specifically the use of segregated fits for address allocation. Segregated fits is a common memory management mechanism in which memory is divided into size classes and

allocation needs are determined by finding a free entry in the appropriate size class [78]. We initially believed that this would be appropriate to apply to our system, as we want to support variable sized object storage, which requires an efficient way to handle placement within the unstructured space of contiguous logical blocks exposed by NVMe. While this would have given us a relatively simple allocation strategy, we later realized that relying on segregated fits alone was ill-suited for SSDs due to the nature of the underlying hardware and flash management mechanisms.

Chapter 4

Design

4.1 Design Overview

As mentioned in the previous section, we build our storage abstraction on top of IX's architectural foundation and principled design trade-offs. Apart from infrastructure concerns, we also contended with numerous design decisions regarding specific storage semantics. We considered a variety of storage paradigms before deciding on a key-value abstraction. We felt that a simple key-value interface would be more appropriate in today's application landscape rather than the traditional set of file system abstractions [83]. Most modern applications prefer to have finer-grain control over I/O access [3]. Moreover, user-space file systems have become more popular in recent years [26, 73], as well as file systems implemented over a cluster of distributed machines abstracted into one storage device [30, 27, 14]. We felt that exposing a key-value abstraction with simple `put` and `get` operations would be sufficiently generic; more complex functionality can be built by composing these operations.

Our storage system translates each `get/put` request for keys into requests to read/write to the corresponding LBAs, and enqueues these requests to be submitted to the device. Figure 3.1 shows the extension of the original IX run-to-completion loop with storage components included. Like IX, our system assumes one (trusted) application as the "user". While security is an interesting and important issue, we consider it outside of the scope of this project and leave this area for future improvement. Moreover, we allow the keyspace to be entirely determined by the user, thus avoiding the need to

Figure 4.1: The metadata components of a block.

manage namespaces and handle potential key collisions.

Our system uses an in-memory index of metadata to handle translations from user-provided keys to logical block addresses (LBAs). Aside from the in-memory index, nothing else is stored in memory to limit the memory footprint. Additionally, we make the following key design decisions in our system:

No Caching Since we are building a persistent storage system, we are less interested in providing caching for performance when there is a plethora of (in-memory) key-value cache options already in existence. We were more curious to see what sort of performance we could obtain by just relying on the hardware/NVMe interface alone.

No Replication As the authors of [51] note, most applications implement their own replication scheme depending on specific reliability/performance requirements. Replication at the storage level introduces complexity and reduces capacity without providing universal benefits.

Variable Length Key/Values this is a key usability feature for most applications and is critical to enable others to build on top of our abstraction.

4.2 Data Organization

We organize data and metadata on device as illustrated by Figure 4.1. We choose a block size of 512 bytes, however NVMe allows this value to be configured anywhere from 512B to 4KB [54]. Each metadata-data set is block aligned, although data may span multiple contiguous blocks. The metadata mainly tracks the key to LBA mapping. Metadata is currently limited to 128 bytes, as we felt any more would overwhelm the storage block.

We chose to co-locate the metadata with the data segment to reduce write overhead and avoid "read-modify-write" scenarios. More specifically, decoupling metadata from data would likely lead to key-value pairs with small values causing poor performance and potentially fragmentation, due to our block-alignment design choice. Each update of a key-value pair would require at least 2 I/Os in this scenario, and performance would be degraded by issuing multiple writes. Decoupling metadata and data also complicates the crash recovery process, as we will discuss in a later section.

As mentioned earlier, we follow MICA's example and partition the keyspace into per-core regions to enable EREW/CREW access. We believe that the data partition model is both suitable to IX's multi-core scalability design goal as well as enabling us to leverage SSD parallelism. So far we have implemented EREW mode.

4.3 Batching

Batching plays a major role in our design. As we use the same system call batching mechanisms as IX, this allows us to apply batching to writes in order to obtain better sequentiality. In order to avoid small, random writes, we aggregate all writes issued within one system call batch into one write request to device. Within each batch, we accumulate the write requests in a scatter-gather list and wait until the entire batch has been processed in order to issue one large write. Figure 4.2 provides a pictorial representation of the scatter-gather list within a write batch. Note that to ensure block alignment, data segments are zero-padded before writing to device.

Conventional wisdom argues that small, random writes are harmful for SSD workloads, and that sequential writes should be favoured. Although a few recent studies [16, 31] have questioned the absoluteness of this long-held theory, the generally accepted view is that random writes are best avoided in most circumstances. SSDs read and write in page units of typically 2-4KB but erase in blocks, which are usually 128-256 KB [1]. However trends in growing page sizes have resulted in pages as large as 16KB [28]. Because our naive design mapped keys to logical blocks, in the event of a workload with multiple writes of small values, SSD performance would quickly degrade.

Although it is possible that we lose the chance to take advantage of some "internal parallelism" in many high performance SSDs by only issuing one large write request at the end of every batch, we rely on our multi-core design

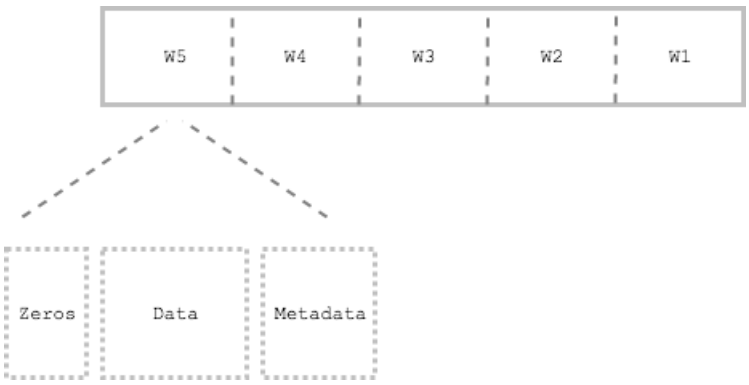

Figure 4.2: The scatter-gather components of a batched write.

to exploit parallelism in SSDs. It is also possible that we lose some performance in terms of reporting to users that their write requests completed; however our batch processing of system calls renders this scenario less likely. Future work could look at adaptively batching for write requests.

Since random reads are not problematic for flash devices, we do not currently batch reads.

4.4 Persistence

Although persistent storage systems typically write metadata or log updates before actually applying the writes to storage [29, 49], our design makes simultaneous updates for metadata and data on device. While this makes us vulnerable to potential consistency issues, we feel this is the right tradeoff for our system. We leverage the asynchronicity of our semantics and the nature of flash to mitigate consistency issues. Suppose we are updating an existing key. We only modify the in-memory index on the arrival of a successful completion event, so if the write did not complete successfully, then the previous LBA associated with the older version of this key remains in the index as well as on the device. As we will describe in detail later, we do not issue explicit erasures to the device; LBAs are reclaimed in the in-memory allocation mechanism and redistributed as needed. Since flash does not overwrite unless explicitly told so or when an LBA has been modified, the stale value will remain on the device until the LBA is reassigned. Though this may be problematic for index reconstruction in crash recovery, we use

the version numbers to resolve conflicts.

Because we do not reclaim the previously used LBAs, the SSD will maintain their logical-to-physical mapping until we reallocate the LBAs.

We eschew in-place updates for a modified version of append-log writes, the details of which we will describe later.

4.5 Crash Semantics

Currently our system supports only crash-fail semantics and a best-effort variant of recovery in the event of corruption. Let us examine the possible crash scenarios in our system:

Crash occurs before request is submitted to the NVMe queue Since we keep no records of pending requests, it is up to the user to resubmit the request.

Crash occurs while device queue is being processed We have no guarantees on whether the request, if it was a write, was flushed to the device as this depends on specific hardware semantics.

Crash occurs after request completed Since the write has been persisted, we will be able to reconstruct the in-memory index correctly using versioning.

Upon a system crash, our storage layer will try to reconstruct the in-memory index by scanning the entire storage device. Since all metadata is coupled with its corresponding data and identifiable by a block-aligned magic value (as noted in Figure 4.1), we can do a check for data corruption by checking the recorded value length against the actual length of the data read from storage and validating the checksum stored in the metadata. For example, if the stored length is different than the "actual" amount of data found on the device, we report this to the user and continue scanning until we find the next metadata block. Storing metadata and data together somewhat eases the corruption detection process; in the case that metadata and data are decoupled it would be probable to encounter situations where we are unable determine where one logical data block started and another ended. In the worst case, the integrity of the entire storage space could be compromised if one metadata entry reported an incorrect value.

Versioning also helps with providing some level of consistency. As mentioned previously, we do not explicitly overwrite anything on device until the LBAs are reassigned to a new value. If the failure occurred some time mid-way during a write and a corrupted version of some key-value pair is persisted, we can ignore it and "rollback" to the previous version of the key-value pair. Since LBAs are not logical freed until a write has been reported successful, the previous version is guaranteed to be still on-device.

Unfortunately, we do not offer stronger data recoverability semantics like other storage systems [60, 14]; this is an area for future improvement.

Chapter 5

Implementation

Here we describe the additions to the IX ecosystem for our storage extension. Our implementation adds approximately 1.5K lines of code to the existing IX code base [23].

5.1 API Overview

We extend the existing user-level library `libix` as was implemented in original IX; Table 5.1 shows the exact list of added system calls and user events. Both `get` and `put` operations are "zero-copy" by design, in keeping with IX's goals to eliminate performance overheads from copying data between user- and kernel-space. Read requests asynchronously return a pointer to a kernel-managed in-memory buffer containing the result of the read; users indicate that this memory may be reclaimed with the `ixev_get_done` system call. Similarly, `ixev_put` asynchronously returns the pointer to user-allocated memory from whence the contents of the write were obtained, to ease memory management from the user's perspective (i.e. the user is thus not required to keep track of which pointers correspond to what asynchronous write requests that were submitted but not confirmed completed).

We based our device-related interactions on the functions exposed by SPDK [72], in which read/write operations on the device only require a LBA and LBA count, as well as appropriate callbacks for when the operation completes. SPDK also supports vectored reading and writing, whereby a scatter-gatter list of entries is specified from which to read/write. We currently only make use of vectored writes, but see applicability for vectored reads in the

System Calls		
Type	**Parameters**	**Description**
get	key	Reads data associated with key
get_done	addr	Indicates addr can be reclaimed
put	key, value, len	Writes specified value associated with key

Event Conditions		
Type	**Parameters**	**Description**
read	key, data, len	Returns results of read at data for length len.
wrote_done	key, value	Indicates write completed and buffer holding value can be reclaimed

Table 5.1: The I/O system calls and event conditions added to the IX API.

case of extremely large values which may have issue being buffered in a contiguous block of memory. The scatter-gather list contains three entries for each key-value update: the metadata entry in the in-memory index, the user-specified pointer corresponding to the data being written, and a zero-buffer for padding to block-alignment. As mentioned previously, we batch all writes to the device, thus the scatter-gather list contains entries for all the key-value pairs being updated.

5.2 Index Management

As mentioned previously, we do not cache any values in memory but we maintain an index of existing keys and associated metadata, as well as track the list of free LBAs. Key length is currently restricted to 110 characters/bytes maximum in order to keep the metadata storage overhead low. Keys are placed in the index according to their hash value modulo the size of the index (which is bounded to limit memory usage). We use Google's CityHash [18] for fast, well-distributed hashing of strings, and employ chaining to handle collisions.

As illustrated by Figure 4.1, the length (in bytes) of the data portion is also stored in the metadata, as well as a 32-bit CRC of the entirety of the value and a version number. We had initially chosen a 16-bit checksum for lower storage overhead but due to performance impact we adopted a faster version of the CRC algorithm with precomputed 8-byte slices [12].

We briefly describe the key aspects and mechanisms of the index.

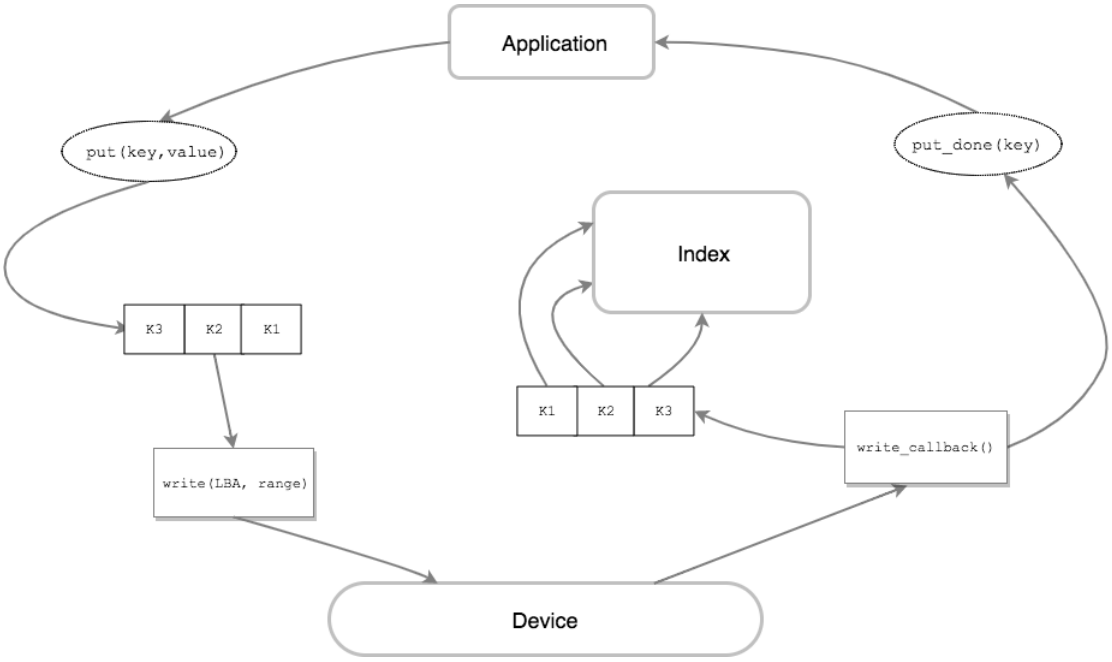

Figure 5.1: Workflow for a put request

Index Creation

Upon start-up or recovery, the system must reconstruct the in-memory index by scanning through the entire storage device. Our current implementation loads one block at a time from device, checking that the metadata is well-formed and validating the stored CRC. Although this may incur a second read request for values that turn out to span multiple blocks (as an initial reading of the metadata will reveal the exact length of the value), we make this trade-off for simplicity and leave room for improvements later on. The system must also examine the version number associated with each key, in case stale values remain that have yet to be overwritten (as mentioned previously, we currently do not explicitly erase anything on the device).

Index Updates

In order to provide some consistency semantics, we ensure that the in-memory index is not updated until the write has been persisted. For each write request, a new metadata entry is created with the requisite informa-

tion, including an incremented version number if the key already exists in the index. Once the write completes, the write callback updates in the index by removing the old metadata entry associated with the key (if one exists) and inserting the new metadata. As mentioned previously, if the write did not propagate successfully on device, then the index (and underlying storage) maintains the old value associated with an existing key.

Free Space Allocation

Since we batch all writes to the device, we make opaque LBA allocations when all writes within the batch have been processed. Thus, the LBA "offset" that each key-value pair occupies within the batched write must be computed when the write completes. Although this slightly complicates the index management, this is a design trade-off we were willing to make in order to obtain sequentiality. However, when we update the index for an existing key, we "de-allocate" the blocks associated with it by adding them to the freelist for re-use. Since we do not explicitly "erase" blocks, the previously occupied LBA is only accessed again when it is assigned to another value. Although there is the possibility that such a strategy can lead to some sort of logical "fragmentation", but since LBAs are translated to physical pages by the SSD's Flash Translation Layer (FTL), we are less concerned with this issue. It is more important that we submit somewhat sequential requests to the device, as the wear levelling and garbage collection mechanisms in FTLs will move data anyway to reclaim erase blocks, and modify LBA to physical page mappings in the process.

While we intended to use segregated fits, we now use a form of first fit due to our batched writes mechanism. Since allocations are only made for batches of writes, segregated fits is ill-suited as size classes are not applicable. The freelist is a singly linked list of starting LBAs with an associated number of free contiguous blocks. The list is organized in ascending LBA order, and any coalescing occurs when blocks are freed. Allocations traverse the list until a large enough chunk of contiguous blocks is found; space is always allocated from the head of the block and the freelist entry is removed if completed allocated or updated if there are still blocks within the entry available. Note that we currently assume the NVMe namespace is large enough for our allocation needs; we do not currently handle the case where no freelist entry is large enough and batches must be divided to be allocated.

Chapter 6

Evaluation

6.0.1 Caveats

Before embarking on a discussion of the evaluation of our system, we must lay some explanatory groundwork. While the original intention was to experimentally determine the performance capabilities of our system on an actual NVMe device, due to logistical issues we were unable to obtain the (correct) device in a timely fashion. While this was not a complete roadblock for evaluation (as we could rely on emulation strategies much like many others' work in this field), it did pose a few limitations on our design/experimental hypotheses:

Persistence Without the device, we cannot be certain what is the exact behaviour of the hardware under a physical failure. We model crashes by interrupting and terminating the program, but it is unclear how the actual device crash behaviour would impact our design and assumptions.

Write Atomicity The NVMe specifications describe an"Atomic Write Unit Power Failure" which guarantees that a write request of this size will be persisted to device in the event of a power failure. Although we were not able to estimate a typical size for this value, having this information would have influenced our consistency semantics and affected our persistence evaluation.

Block Size Not being aware of the device page/block size also influenced our design and subsequent evaluation. Although we chose a logical

block size of 512 bytes, without being certain what this value is in relation to the device page/block size we cannot claim any realism in our results. An inappropriate logical block size could have non-trivial impact on performance. For example, recall that writes and reads are performed in units of pages but erasures are at the block level. In order to avoid write performance degradation, it is preferable to write units of pages or even blocks, but in modern devices page size may vary from 2KB to 16KB. Our logical block size should reflect these physical characteristics.

Device Performance Using the descriptions of SSD performance and behaviour in the related literature, we based our design on educated guesses of how a physical device should behave given the workloads and access patterns our system induces. For example, we assume that our batched sequential writes should not incur as many expensive background SSD processes as workload with numerous small, random writes. Without a device we can neither confirm nor refute our assumptions based on experimental evidence.

6.1 Experimental Methodology

We ran our experiments on either a Xeon E5-2637 @ 3.5Ghz or Xeon E5-2650 @ 2.6 Ghz server with 64GB of memory. Our machine is presumably configured with Intel x520 10GbE NICs (82599EB chipset), although we do not utilize any networking functionality in these preliminary experiments. The host operating system was Ubuntu 16.04.1 running on a 4.8.0-51 Linux kernel.

As we did not have the appropriate hardware, we emulated a persistent device by allocating an appropriately large RAM-backed file in system shared memory (/dev/shm).

6.2 Preliminary Findings

More Caveats

Although our initial strategy was to introduce artificial delays to mimic the behaviour of an actual device, two issues prohibited us from doing so in

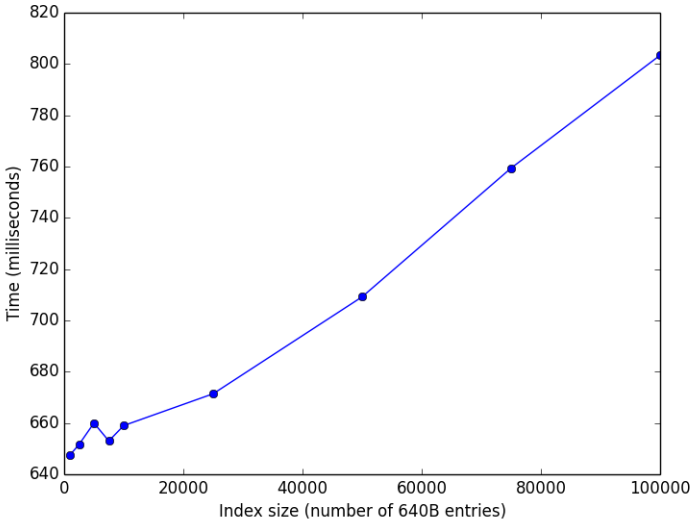

Figure 6.1: Time needed to reconstruct the index

practice. First, having a constant delay regardless of the size of the request is unrealistic; while we could have adaptively chosen a delay based on the size of the given request, it is unclear what would be an appropriate heuristic. Second, the asynchronous timers we utilized to implement the delay were unfortunately designed to be fairly coarse-grained and best effort in terms of precision. We found that we were unable to specify any delay under 100µs and have the timer actually respect this value. The combination of these factors led us to remove artificial delays from most of our experiments and instead focus on the performance characteristics and overhead of our software implementation.

6.2.1 Persistence

We performed a perfunctory evaluation for persistence by issuing a large number of writes sequentially and terminating the workload while it was in progress. We then noted the correctness of the reconstruction process by checking that the last key for which there was an acknowledgement of successful persistence is indeed on the device.

We also profiled the index reconstruction process; Figure 6.1 shows our results. Although our index initialization currently does not account for

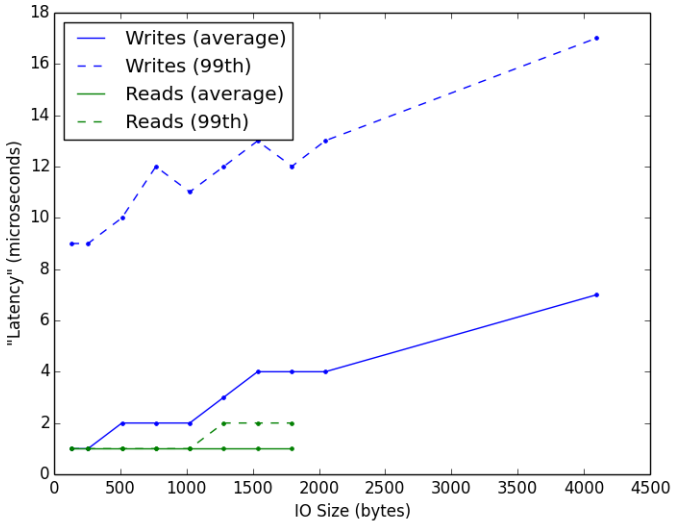

Figure 6.2: Software overhead for each type of request

device performance (as we removed all artificial delays), we note that the software overhead of the index construction process is not egregious.

6.2.2 Software Overhead

In lieu of presenting latency capabilities of our emulated system, we instead focus on measuring the software implementation overhead. Figure 6.2 shows the performance for individual read and write requests without any artificial delays imposed. Note that due to an implementation restriction, we are currently not able to issue reads larger than 2KB at this time (hence the truncated results); code infrastructure is in place to return larger read requests to user space as a scatter-gather list.

6.2.3 Batching

The performance characteristic we were most interested in for our system was the effect of batching. Although we were fairly confident batching would improve throughput, we wanted to quantitatively demonstrate the benefits. As Figure 6.3 illustrates, the throughput gains are substantial up to a batch size of roughly 1000.

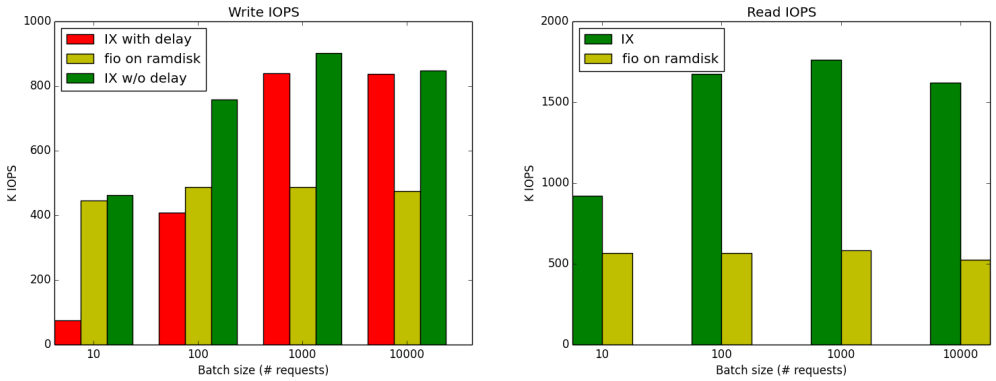

Figure 6.3: Throughput for batched reads and writes

We compare our system against running `fio` [4] on a RAM disk configured with `tmpfs`. We did our best to align the `fio` parameters with our IX configuration, choosing `libaio` as the I/O engine and restricting `fio` job execution to a single core. The results are somewhat bizarre, as it would appear `libaio` does not scale at all given larger batch sizes.

Although we evaluated writes both with the artificial delays and without, it would seem that they have minimal impact beyond 100 requests per batch and an outsized impact for batches below that threshold, and thus as mentioned before add little meaning to our evaluation. Delays were not induced for reads due to the previously mentioned precision issue.

6.3 Evaluation Remarks

Our initial findings were neither laudatory nor even logical, but after carefully examining the sources of bottleneck and adjusting the implementation, we were able to obtain some presentable results.

Due to the fact that we did not have a physical device to experiment with, we had difficulties coming up with reasonable and sound evaluation scenarios. Although raw performance, in particular latency and throughput, were most of interest to us, without a physical device these metrics are somewhat meaningless.

Chapter 7

Discussion

7.1 Related Work

Here, we outline some of the work that we felt was related to certain components in our design. As our work is at the intersection of many areas of research, we cast a broad eye in searching for research ideas that relate to our own. These are a few note-worthy items that we encountered.

7.1.1 Key Value Stores

In our quest to build a key-value storage system, we naturally examined several other contemporary key-value store implementations in the research landscape. We chose a subset of relevant systems to present here:

Echo

Claiming to be the first "fine-grained, persistent key-value store" design for SCM/NVM technologies, this system employs a two-level DRAM/NVM hybrid architecture. Threads initially process data in local stores in DRAM, and then issue "commits" to a master store on NVM, which then persists the data. Echo leverages the byte-addressability of NVM to achieve a lightweight versioning scheme and provide some semblance of snapshot isolation. Every commit of an existing key is considered a new version, and versioning is implemented by modifying pointer; thus old data is never deleted. Moreover, every `get` request for a key must be accompanied by a version number, in order to allow some level of isolation between concurrent threads operating

on the same key. We applied a similar approach of "lazily" versioning in order to ensure some consistency and crash recovery properties, but do not consider concurrency issues as we have exclusive writes.

This two-tiered approach is an interesting way of resolving the issue of managing traditionally ephemeral semantics in persistent memory. For instance, a lot of work is ongoing to investigate how pointers should be implement in persistent memory. Echo's design works around this issue by using database-like transactional semantics to separate volatile from non-volatile memory.

SILT

SILT, or "Small Index Large Table", is a flash-based key-value store that aims to minimize DRAM usage by architecting a three-tiered storage system [38]. They propose three differently structured storage systems for flash that compose into an overall well-performing key-value store with low memory utilization for the in-memory index as well as optimized writes and reads for flash wear-levelling. Writes are first issued to a "LogStore" that uses the index to map keys to offsets within the log. The LogStore has a fixed capacity to limit index-growth/memory usage, and once full will become converted to a "HashStore". Because this store is organized in a hash table structure, it does not require an index for key lookup but uses a small in-memory filter. Once several HashStores have been created, they are bulk-merged into a "SortedStore" which, as its name suggests, stores all keys in sorted order and is indexed by a novel data structure.

This system is interesting due to its multi-staging design, but also demonstrates the complexity needed to design a highly optimized system tightly coupled with the characteristics of the underlying hardware. In contrast, we opted for a single layer structure for simplicity and clarity, and instead leave performance-centric optimizations to lower layers.

Custom Key Value Stores

The solution proposed by Marmol et. al. [44] takes an interesting approach to the question of how best to integrate non-volatile memory technologies into existing applications. They argue that since the underlying hardware making up NVM is currently hard to characterize and subject to differences in specifications and details, the best approach is to abstract a key-value-

based library for applications to make use of, and then modify the library implementation based on the specific underlying hardware. They outline the fundamental features expected of key value stores and provide design principles for creating a key-value library.

While abstracting a key-value interface seems like a useful approach, this design does not address the fundamental obstacles in extracting better performance from new memory technologies: the operating system stack. Moreover, expecting application developers to adapt implementations to different device interfaces seems unlikely to become a reality. While we also expose a key-value abstraction, our approach leverages a well-developed standard like NVMe to handle device-specific issues rather than attempt to bring them into our abstraction layer.

Industry Solutions

A few hybrid key-value stores have emerged as industry responses to the need for cost-effective low-latency solutions. McDipper [46] is Facebook's implementation of an SSD-based replacement for memcached. Twitter's fatcache [25] treats SSD storage as a large cache (as its name suggests) and applies similar tactics as our system such as batching small writes and issuing them to device in a log-structured manner.

7.1.2 NVMe

As mentioned previously, NVMe has not quite inspired the same volume of research as visionary NVM technologies. In our review of current NVMe applications in research, we found two main applications of NVMe that were intriguing.

NVMeDirect

NVMeDirect [36] is a proposed user-level I/O framework that bypasses the kernel to expose the storage device interface directly to applications. The authors note that SPDK, an existing user-level library, has fundamental limitations; since it migrates the device drivers into user-space, only one application may make use of the device at a time. NVMeDirect multiplexes the device interface so that many applications may utilize the same NVMe device. Rather than directly exposing NVMe queues, NVMeDirect offers the

"I/O handles" abstraction to allow applications to specify I/O policies, such as latency guarantees or read/write prioritization. Other features include a block cache for caching LBA translations, an I/O scheduler and a dedicated completion thread to allow latency-critical applications to synchronously poll for I/O completion.

Storage Disaggregation

Two recent projects in storage disaggregation both rely on NVMe to deliver high performance storage at scale. Reflex [37] offers low-latency remote storage in order to achieve better resource utilization within a datacenter. They also build their system atop IX, but extend their system with a quality-of-service scheduler. While we are more concerned with providing applications an interface to persistent storage, Reflex's main design goal is to enable tenants in a datacenter to have differentiated service and finer-grained SLOs specifications while achieving better resource utilization across the datacenter by giving away any unused capacity to "best-effort" clients.

The authors of Decibel pursue a similar purpose by proposing a set of storage abstractions in order to allow better isolation and more flexibility in expressing performance requirements for tenant applications.

While this does not represent an exhaustive list of research endeavours utilizing NVMe, we cannot help but feel that there is a lack of exploration in the NVMe research space. To the best of our knowledge, few others have attempted integrating NVMe into their system design; we are among the first to leverage NVMe to build an operating system-level I/O abstraction.

7.1.3 Linux & Operating System I/O Paths

Some efforts have been made to address scalability issues in the Linux I/O path. In particular, the singly-locked single request queue has been replaced by multiple queues at the software level complemented by a single hardware level queue [10]. While removing this lock has reportedly improved performance, the single queue bottleneck remains due to the inherent nature of the underlying hardware, in particular SATA/SCSI device drivers which only support one I/O request queue.

A more recent development is the application of Open-Channel SSDs to Linux [11]. Open-Channel SSDs are a new class of devices which directly

expose the internals of SSDs by allowing the user to specify I/O scheduling and data placement, in essence allowing them to implement their own FTL.

While these new advancements have been integrated in Linux [41, 42], one could argue that they only exacerbate the issue of an over-burdened monolithic kernel. A study conducted almost two decades ago found that roughly 70% of the Linux source code was devoted to device drivers; as so happens this subset of the source code was also the most error-prone [17]. One can imagine that adding support for more devices to support would not help with either of those statistics.

Another approach to mitigate the OS overhead problem in I/O was proposed by Shin et. a. [71]. They designed a modified I/O path to eliminate extraneous context switching or interrupt handling by implementing a Hardware Abstraction Layer to expose I/O functionality directly to upper layers and apply an asychronous-esque model for processing I/O requests and completions.

While the performance numbers they report are somewhat uninspiring (600K IOPS), their system displays some interesting scalability characteristics. They report that they are able to support 6 SATA SSDs concurrently while Linux is supposedly unable to provide acceptable performance for more than 3 SSDs. While we have yet to consider multi-device operation in our system, our isolated cores design enables ease of extension should we pursue this idea.

7.1.4 Positions and Opinions

During our "search" phase, we encountered many position papers that ultimately shaped how we approached our design: here are but a few.

When Poll is Better Than Interrupt

As mentioned previously, contemporary operating systems typically implement asynchronous I/O completion; due to the assumption that storage devices are slow, OS's try to complete other work in the many milliseconds that pass before an I/O completion is obtained from device. This assumption no longer holds in the face of flash-based SSDs and other faster storage media. Yang et. al. experimentally determined that a synchronous I/O completion model incurs only half the latency of that of the asynchronous model, and that throughput is also substantially improved [80]. Moreover, they offer

a meticulous breakdown of kernel vs. device processing times, and provide some theoretical notions of how each relate to the others. For example, they note that for the SSDs they conducted experiments on, 1.4 µs is the threshold of kernel overhead over which no useful work will be accomplished by a context switch between device request and response time.

The Unwritten Contract of Solid State Drives

This survey details the "unwritten contract" of SSDs in an attempt to provide developers and researchers with basic guiding principles for building their systems [31]. While there are a few well-known tenets of SSDs that we described earlier, this paper outlines five "rules" for optimizing SSD performance drawn from observing the characteristics implicit in SSD operation. In particular, the authors note that writes to SSDs should either be large enough to induce FTL "striping" into smaller sub-requests or there should be enough small requests to exploit SSD parallelism; in either case individual small requests should be avoided. Other guidelines include grouping data on device according to similar "death times" (deletion time) and ensuring uniformity of data lifetime.

Although we took their request size guideline under advisement, unfortunately some of the other advice offered do not translate well into practice. For example, it is unclear how developers could design their applications such that data with similar "death times" are grouped together. Moreover, although the authors ascribe some vague sense of synergy between FTLs, file systems or the operating system-level storage interface, and applications, it is not clear how well we can depend on parties involved in each of these three layers having a deep understanding of the interactions between their layers. Interestingly, this paper makes almost no mention of NVMe and operating system-level interfaces.

Don't stack your Log on my Log

Conversely, while the previous paper aimed to provide guiding principles to advise, the authors of this paper sought to "de-bunk" some commonly held beliefs for designing applications for SSDs that were long-held to be beneficial but ultimately not performance-optimal. In a prime example of the lack of the transparency in the layers between storage and application, the authors investigate the effects of overlaying logs-structured implementations in the

FTL, file system and application layers.

Their results show that contrary to the popularly-held idea that log-structuring data is beneficial for SSDs, in specific cases log-stacking may be detrimental to performance due to duplication of processes such as garbage collection. This supports the opinions held by the authors of the previous paper that "logging is not a panacea" and should only be applied intelligently.

7.2 Future Work

7.2.1 Log-structuring

Recall from earlier sections our procedure for batching writes and our allocation strategy for the LBA freelist: we aggregates writes and continuously append to the range of LBAs unless there is a sufficient amount of reclaimed space at an "earlier" LBA number. Although we did not set out to design our system with these properties, we ultimately ended up with something resembling a log-structured system [34]. As noted before, there is some contention about whether this is appropriate, as SSD devices already feature a log-structured management system [81]. We are hoping that subsequent experimental data will convey the performance benefits of our design decision.

7.2.2 Implementation Improvements

Although we tried to be as complete as possible in our implementation, it is by no means a perfect prototype. Here are just a few ideas for modest improvements that could be made:

Key Membership Testing Although we use a fast hash to determine key membership in the index, in the event of a collision we must resort to traversing the hash chain and comparing keys using standard `strcmp`. Bloom filters are a performance-friendly way to test membership quickly without having to traverse the index.

Key storage Although our design aspires to support variable key length, we currently truncate it at 110 bytes in order to keep the metadata storage overhead low. Future enhancements could include storing only a hash or some other form of encoding the key.

Read Size Restriction As mentioned in the previous section, due to an implementation issue our system currently only supports reads up to 2KB, although writes may be arbitrarily sized. This creates a rather lopsided (and incorrect) system, as users may write large values that they cannot subsequently access. The restriction on reads may be lifted by dividing up larger reads from device into scatter-gather list entries.

Improved Integrity Check Our implementation currently uses a basic CRC (mostly adapted from [12]), as we made a tradeoff in slight favour of performance over stronger integrity. However, profiling reveals that the CRC still contributes a significant amount to the overall software overhead. More sophisticated protocols exist that try to mitigate the tradeoff between performance and integrity [65, 40, 67].

Our prototype implements EREW but can be extended to CREW using the appropriate inter-core communications to route write requests. Our long-term plan was to implement EREW mode first, and then extend our system to CREW access by partitioning the LBA address space and using inter-core communication to route write requests to the appropriate core. Indeed we would have liked to examine the throughput capabilities of this approach, as well as explore the performance impact of increased parallelism in our design, as this is a well-studied aspect of SSDs [16].

7.2.3 Potential Directions

There is an emerging set of design principles called "Offline First", in which applications are designed with loss-of-connectivity in mind [56]. Proponents of this paradigm argue that applications should be design with resilience to loss of network connectivity and better capabilities for offline operation. One could imagine that the IX infrastructure would be applicable to such design principles; IX provides a fast path for networking and now storage with our proposed design.

Another potential direction for the system could be to port directly to NVM hardware by removing the block serialization we currently use. This seems less likely; it is still unclear exactly what kind of semantics NVM will offer. In contrast, NVMe is an industry-backed set of standards that is already seeing greater adoption. For example, Intel's datacenter storage offerings make use of NVMe interfaces and their latest 3D XPoint (purportedly PCM-based) product line requires NVMe-compliant controllers [32, 33]. Moreover,

the general opinion is that PCIe will be too slow for NVM's nanosecond-scale latencies, and either NVMs should reside directly on the memory bus [69] or new protocols will be needed [77].

As mentioned previously, we assume a single trusted application is running on top of IX. Further work could be done to examine the role IX should play in terms of managing access control and permissions checking for remote requests for data.

Chapter 8

Conclusion

We have presented a prototype of a key-value abstraction which incorporates into the IX architecture, implemented using a variety of techniques both within the existing design principles of IX and borrowed from various other systems. In our search for appropriate approaches, we examined a panoply of existing work, from in-memory key-value stores to storage disaggregation. While we initially made many "sweeping simplifications" with the goal of obtaining a simple prototype on which to iterate, we soon discovered that providing persistence and some minimal guarantees of consistency were non-trivial undertakings. Pursuing a storage system without the adequate background left us vulnerable to drawbacks in our design and implementation process, and at times forced us to go back and re-examine our assumptions. Moreover, lacking the physical device around which many of our assumptions are made, we struggled with devising an appropriate performance model against which to validate our hypotheses.

However, with guidance from experts (both in literature and real life), we implemented a prototype which we hope will demonstrate the benefits of IX's design approaches to I/O processing. Moreover, we showed a novel use case for NVMe to illustrate the untapped potential of this technology. We put forward our design in the humble hopes that others will draw from the observations we made in order to build new systems for NVMe and other emerging storage innovations.

Bibliography

[1] Arpaci-Dusseau, R. H., and Arpaci-Dusseau, A. C. *Operating Systems: Three Easy Pieces*, 0.91 ed. Arpaci-Dusseau Books, May 2015.

[2] Arulraj, J., Pavlo, A., and Dulloor, S. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *SIGMOD Conference* (2015), pp. 707–722.

[3] Atlidakis, V., Andrus, J., Geambasu, R., Mitropoulos, D., and Nieh, J. POSIX abstractions in modern operating systems: the old, the new, and the missing. In *Proceedings of the 2016 EuroSys Conference* (2016), pp. 19:1–19:17.

[4] Axboe, J. Flexible I/O Tester. `https://github.com/axboe/fio`.

[5] Bailey, K., Ceze, L., Gribble, S. D., and Levy, H. M. Operating System Implications of Fast, Cheap, Non-Volatile Memory. In *Proceedings of The 13th Workshop on Hot Topics in Operating Systems (HotOS-XIII)* (2011).

[6] Bailey, K. A., Hornyack, P., Ceze, L., Gribble, S. D., and Levy, H. M. Exploring storage class memory with key value stores. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workload (INFLOW)* (2013), pp. 4:1–4:8.

[7] Barroso, L. A., Marty, M., Patterson, D., and Ranganathan, P. Attack of the killer microseconds. *Commun. ACM 60*, 4 (2017), 48–54.

[8] Belay, A., Bittau, A., Mashtizadeh, A. J., Terei, D., Mazières, D., and Kozyrakis, C. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the 10th Symposium on*

Operating System Design and Implementation (OSDI) (2012), pp. 335–348.

[9] Belay, A., Prekas, G., Primorac, M., Klimovic, A., Grossman, S., Kozyrakis, C., and Bugnion, E. The IX Operating System: Combining Low Latency, High Throughput, and Efficiency in a Protected Dataplane. *ACM Trans. Comput. Syst. 34*, 4 (2017), 11:1–11:39.

[10] Bjørling, M., Axboe, J., Nellans, D. W., and Bonnet, P. Linux block IO: introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR)* (2013), pp. 22:1–22:10.

[11] Bjørling, M., Gonzalez, J., and Bonnet, P. LightNVM: The Linux Open-Channel SSD Subsystem. In *Proceedings of the 15th USENIX Conference on File and Storage Technologie (FAST)* (2017), pp. 359–374.

[12] Brumme, S. Fast crc32. `http://create.stephan-brumme.com/crc32/`.

[13] Bugnion, E., Devine, S., Rosenblum, M., Sugerman, J., and Wang, E. Y. Bringing Virtualization to the x86 Architecture with the Original VMware Workstation. *ACM Trans. Comput. Syst. 30*, 4 (2012), 12:1–12:51.

[14] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., ul Haq, M. F., ul Haq, M. I., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K., and Rigas, L. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (2011), pp. 143–157.

[15] Caulfield, A. M., De, A., Coburn, J., Mollow, T. I., Gupta, R. K., and Swanson, S. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. In *Pro-*

ceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (2010), pp. 385–395.

[16] Chen, F., Hou, B., and Lee, R. Internal Parallelism of Flash Memory-Based Solid-State Drives. *TOS 12*, 3 (2016), 13:1–13:39.

[17] Chou, A., Yang, J., Chelf, B., Hallem, S., and Engler, D. R. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)* (2001), pp. 73–88.

[18] Cityhash, a family of hash functions for strings. `https://github.com/google/cityhash`.

[19] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)* (2011), pp. 105–118.

[20] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B. C., Burger, D., and Coetzee, D. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)* (2009), pp. 133–146.

[21] Corbet, J. Atomic I/O operations. `https://lwn.net/Articles/552095/`.

[22] Couchbase. `https://www.couchbase.com`.

[23] Danial, A. CLOC: Count Lines of Code. `http://cloc.sourceforge.net/`.

[24] Dean, J., and Barroso, L. A. The tail at scale. *Commun. ACM 56*, 2 (2013), 74–80.

[25] Twitter's fatcache. `https://github.com/twitter/fatcache`.

[26] FUSE. `https://github.com/libfuse/libfuse`.

[27] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 29–43.

[28] Goossaert, E. Coding for SSDs: Pages, Blocks, and the Flash Translation Layer. http://codecapsule.com/2014/02/12/coding-for-ssds-part-3-pages-blocks-and-the-flash-translation-layer/.

[29] Hagmann, R. B. Reimplementing the Cedar File System Using Logging and Group Commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP)* (1987), pp. 155–162.

[30] HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[31] He, J., Kannan, S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. The Unwritten Contract of Solid State Drives. In *Proceedings of the 2017 EuroSys Conference* (2017), pp. 127–144.

[32] NVMe vs SATA: Intel SSD with NVM Express outperforms SATA SSD's. https://www.intel.com/content/www/us/en/solid-state-drives/intel-ssd-dc-family-for-nvme.html.

[33] Intel optane technology. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-optane-technology.html.

[34] Johnson, N. Damn cool algorithms: Log structured storage. http://blog.notdot.net/2009/12/Damn-Cool-Algorithms-Log-structured-storage.

[35] Javascript. https://www.javascript.com/.

[36] Kim, H.-J., Lee, Y.-S., and Kim, J.-S. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2016).

[37] Klimovic, A., Litz, H., and Kozyrakis, C. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXII)* (2017), pp. 345–359.

[38] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. SILT: a memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)* (2011), pp. 1–13.

[39] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th Symposium on Networked Systems Design and Implementation (NSDI)* (2014), pp. 429–444.

[40] Lin, S.-J., Al-Naffouri, T. Y., Han, Y. S., and Chung, W.-H. Novel Polynomial Basis With Fast Fourier Transform and Its Application to Reed-Solomon Erasure Codes. *IEEE Trans. Information Theory 62*, 11 (2016), 6284–6299.

[41] Linux 3.13. `https://kernelnewbies.org/Linux_3.13`.

[42] Lightnvm support is going into linux 4.4. `https://www.phoronix.com/scan.php?page=news_item&px=Linux-4.4-LightNVM`.

[43] Marathe, V. J., Seltzer, M., Byan, S., and Harris, T. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (Santa Clara, CA, 2017), USENIX Association.

[44] Mármol, L., Guerra, J., and Aguilera, M. K. Non-volatile Memory through Customized Key-value Stores. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2016).

[45] Mármol, L., Sundararaman, S., Talagala, N., and Rangaswami, R. NVMKV: A Scalable, Lightweight, FTL-aware Key-Value Store. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)* (2015), pp. 207–219.

[46] Mcdipper: A key-value cache for flash storage. `https://www.facebook.com/notes/facebook-engineering/mcdipper-a-key-value-cache-for-flash-storage/10151347090423920/`.

[47] Memcached. https://memcached.org/.

[48] Mittal, S., and Vetter, J. S. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Trans. Parallel Distrib. Syst. 27*, 5 (2016), 1537–1550.

[49] Mohan, C., Haderle, D. J., Lindsay, B. G., Pirahesh, H., and Schwarz, P. M. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst. 17*, 1 (1992), 94–162.

[50] Nanavati, M., Schwarzkopf, M., Wires, J., and Warfield, A. Non-volatile storage. *Commun. ACM 59*, 1 (2016), 56–63.

[51] Nanavati, M., Wires, J., and Warfield, A. Decibel: Isolation and Sharing in Disaggregated Rack-Scale Storage. In *Proceedings of the 14th Symposium on Networked Systems Design and Implementation (NSDI)* (2017), pp. 17–33.

[52] Nodejs. https://nodejs.org/en/.

[53] NVMe for Absolute Beginners. https://blogs.cisco.com/datacenter/nvme-for-absolute-beginners.

[54] NVM Express 1.3 Specifications. http://www.nvmexpress.org/wp-content/uploads/NVM_Express_Revision_1.3.pdf.

[55] Dual Ports Drive NVMe SSD Uptick. http://www.eetimes.com/document.asp?doc_id=1331595.

[56] Offline first. http://offlinefirst.org/.

[57] Open-channel solid state drives. https://openchannelssd.readthedocs.io/en/latest/.

[58] Oukid, I., Booss, D., Lehner, W., Bumbulis, P., and Willhalm, T. SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In *Proceedings of the 10th International Workshop on Data Management on New Hardware (DaMoN)* (2014), pp. 8:1–8:7.

[59] Ousterhout, J. K. Why threads are a bad idea (for most purposes). http://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf.

[60] Ousterhout, J. K., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S. M., Stutsman, R., and Yang, S. The RAMCloud Storage System. *ACM Trans. Comput. Syst. 33*, 3 (2015), 7:1–7:55.

[61] Ouyang, X., Nellans, D. W., Wipfel, R., Flynn, D., and Panda, D. K. Beyond block I/O: Rethinking traditional storage primitives. In *Proceedings of the 17th IEEE Symposium on High-Performance Computer Architecture (HPCA)* (2011), pp. 301–311.

[62] Patterson, D. A. Latency Lags Bandwidth. In *Proceedings of the 23rd International IEEE Conference on Computer Design (ICCD)* (2005), pp. 3–6.

[63] Pelley, S., Wenisch, T. F., Gold, B. T., and Bridge, B. Storage Management in the NVRAM Era. *PVLDB 7*, 2 (2013), 121–132.

[64] Qiu, S., and Reddy, A. L. N. NVMFS: A hybrid file system for improving random write in nand-flash SSD. In *Proceedings of the 29th IEEE Symposium on Mass Storage Systems and Technologies (MSST)* (2013), pp. 1–5.

[65] Rashmi, K. V., Nakkiran, P., Wang, J., Shah, N. B., and Ramchandran, K. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth. In *Proceedings of the 13th USENIX Conference on File and Storage Technologie (FAST)* (2015), pp. 81–94.

[66] Redis. `https://redis.io/`.

[67] Rizzo, L. Effective erasure codes for reliable computer communication protocols. *Computer Communication Review 27*, 2 (1997), 24–36.

[68] Rosenblum, M., and Ousterhout, J. K. The Design and Implementation of a Log-Structured File System. *ACM Trans. Comput. Syst. 10*, 1 (1992), 26–52.

[69] Rudoff, A. Programming models for emerging non-volatile memory technologies. *;login: The USENIX Magazine 38*, 3 (2013).

[70] Shiah, T. The Competitive Advantage of NVMe SSDs in the Data Center.

[71] Shin, W., Chen, Q., Oh, M., Eom, H., and Yeom, H. Y. OS I/O Path Optimizations for Flash Solid-state Drives. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)* (2014), pp. 483–488.

[72] Storage performance development kit. `http://www.spdk.io/`.

[73] Tarasov, V., Gupta, A., Sourav, K., Trehan, S., and Zadok, E. Terra Incognita: On the Practicality of User-Space File Systems. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)* (2015).

[74] Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., and Swift, M. M. Aerie: flexible file-system interfaces to storage-class memory. In *Proceedings of the 2014 EuroSys Conference* (2014), pp. 14:1–14:14.

[75] Volos, H., Tack, A. J., and Swift, M. M. Mnemosyne: lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVI)* (2011), pp. 91–104.

[76] VoltDB. `https://www.voltdb.com/`.

[77] Vucinic, D., Wang, Q., Guyot, C., Mateescu, R., Blagojevic, F., Franca-Neto, L., Moal, D. L., Bunker, T., Xu, J., Swanson, S., and Bandic, Z. DC express: shortest latency protocol for reading phase change memory over PCI express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologie (FAST)* (2014), pp. 309–315.

[78] Wilson, P. R., Johnstone, M. S., Neely, M., and Boles, D. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the 1995 International Workshop on Memory Management* (1995), pp. 1–116.

[79] Xu, Q., Siyamwala, H., Ghosh, M., Suri, T., Awasthi, M., Guz, Z., Shayesteh, A., and Balakrishnan, V. Performance analysis of NVMe SSDs and their implication on real world databases.

In *Proceedings of the 8th International Systems and Storage Conference (SYSTOR)* (2015), pp. 6:1–6:11.

[80] Yang, J., Minturn, D. B., and Hady, F. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologie (FAST)* (2012), p. 3.

[81] Yang, J., Plasson, N., Gillis, G., Talagala, N., and Sundararaman, S. Don't Stack Your Log On My Log. In *Proceedings of the 2nd Workshop on Interactions of NVM/FLASH with Operating Systems and Workload (INFLOW)* (2014).

[82] Yang, J., Wei, Q., Chen, C., Wang, C., Yong, K. L., and He, B. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologie (FAST)* (2015), pp. 167–181.

[83] Zadok, E., Hildebrand, D., Kuenning, G., and Smith, K. A. Posix is dead! long live... errr... what exactly? In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)* (Santa Clara, CA, 2017), USENIX Association.