IT UNIVERSITY OF COPENHAGEN

DOCTORAL THESIS

# OX: Deconstructing the FTL for Computational Storage

*Author:*
Ivan Luiz PICOLI

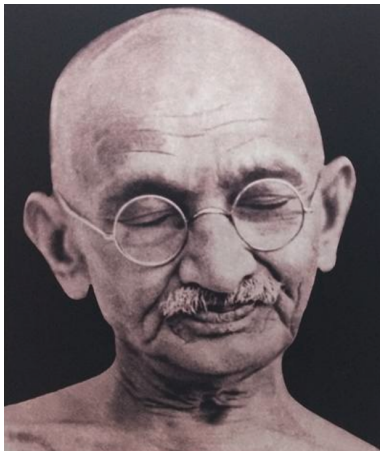
*Supervisor:*
Philippe BONNET

*A thesis submitted in fulfillment of the requirements*
*for the degree of Doctor of Philosophy*

*in the*

Data Systems Group
Department of Computer Science

July 8, 2019



*"Experience has taught me that silence is a part of the spiritual discipline of a votary of truth. Proneness to exaggerate, to suppress or modify the truth, wittingly or unwittingly, is a natural weakness of man, and silence is necessary in order to surmount it. A man of few words will rarely be thoughtless in his speech; he will measure every word."*

M. K. Gandhi

IT UNIVERSITY OF COPENHAGEN

# *Abstract*

Data Systems Group

Department of Computer Science

Doctor of Philosophy

**OX: Deconstructing the FTL for Computational Storage**

by Ivan Luiz PICOLI

Offloading processing to storage is a means to minimize data movement and efficiently scale processing to match the increasing volume of stored data. In recent years, the rate at which data is transferred from storage has increased exponentially, while the rate at which data is transferred from memory to a host processor (CPU) has only increased linearly. This trend is expected to continue in the coming years. Soon, CPUs will not be able to keep up with the rate at which stored data is transferred. The increasing volume of stored data compound this problem. In the 90s, pioneering efforts to develop Active Disks were based on magnetic drives. Today, renewed efforts fall in two groups. The first group combines Open-Channel SSDs with a programmable storage controller integrated with a fabrics front-end. The second group integrates a programmable storage controller directly onto a SSD (e.g., ScaleFlux, NGD). We focus on the former approach. At its core, our approach is based on defining application-specific Flash Translation Layers (FTLs) on storage controllers as a means to offload processing from the host. Our goal is to leverage computational storage as a means to collapse layers within the I/O stack. In this thesis, we make three contributions. First, we explore the performance characteristics of Open-Channel SSDs. Second, we introduce OX as an FTL template for programming SoC-based storage controllers on top of Open-Channel SSDs. Third, we present ELEOS, a log-based storage engine based on OX to offload storage management from the host. We evaluate ELEOS together with the LLAMA system, developed at Microsoft Research.

IT-UNIVERSITETET I KØBENHAVN

# *Resumé*

Data Systems Group

Department of Computer Science

Doctor of Philosophy

**OX: Deconstructing the FTL for Computational Storage**

by Ivan Luiz PICOLI

Computational storage gøre det muligt at behande data hvor den er lageret. Det er en måde at minimere databevægelsen og en skalerbar måde at behandle at det stigende volumen af lagrede data. I de seneste år er den hastighed, hvormed data overføres fra lageret, øget eksponentielt, mens den hastighed, hvormed data overføres fra hukommelse til en værtsprocessor (CPU), kun er øget lineært. Denne tendens forventes at fortsætte i de kommende år. Snart vil CPU'erne ikke være i stand til at følge med den hastighed, hvorpå lagrede data overføres. De stigende mængder lagerede data forværre dette problem. I 90'erne blev en banebrydende indsats for at udvikle aktive diske baseret på magnetiske drev den første form for computational storage. I dag er fornyede indsatser faldet i to grupper. Den første gruppe kombinerer Open-Channel SSD'er med en programmerbar storage controller integreret med en front-end switch. Den anden gruppe integrerer en programmerbar storage controller direkte på en SSD (fx ScaleFlux, NGD). Vi fokuserer på den tidligere tilgang. Kernen er, at vores tilgang er baseret på at definere applikationsspecifikke Flash Translation Layers (FTL'er) på storage controllers som et middel til at aflaste behandling fra host CPU. Vores mål er at udnytte computational storage som et middel til at kollapse lag i I/O-stakken. I denne sammenhæng laver vi tre bidrag. For det første, undersøger vi præstationsegenskaberne ved Open-Channel SSD'er. For det andet introducerer vi OX som en FTL-skabelon til programmering af SoC-baserede lagerkontroller oven på Open-Channe SSD'erl. For det tredje præsenterer vi ELEOS, en logbaseret lagringsmaskine baseret på OX for at aflaste lagringsstyring fra værten. Vi evaluerer ELEOS sammen med LLAMA-systemet, udviklet ved Microsoft Research.

# Contents

# *Acknowledgements*

I am grateful for the knowledge I acquired during the course of this Ph.D. Part of this knowledge is a result of the interaction with people from tens of nations. In the past years, these people helped and motivated me to reach the end of this journey.

I would like to thank

**My mother**, for the unconditional love, and for showing me the position that led to my fellowship grant. **My brother**, for the inspiration and motivation along the entire Ph.D.

**Philippe Bonnet**, a brilliant advisor. For the motivation and trust since from the very first contact via email, for advising me during this journey, for introducing me to his network of contacts around the globe, for giving me the opportunity to explore new horizons, and for sharing a table in Asian restaurants.

**Björn Jónsson** and **Sebastian Büttrich**, for the support and help during my time at ITU. **Pınar Tözün**, for the support and help with my thesis corrections in the last part of my studies. **Fabricio Narcizo**, for all the support and friendship since from my first day at ITU. **Jonathan Fürst** and **Carla Villegas**, for sharing the office and for the great moments we spent together in the first years of my Ph.D. Jonathan, thank you for the idea of traveling together in Asia. **Omar** and **Martin**, for sharing the office in the last months. **Javier González** and **Matias Bjørling**, for guidance and help. And all the other colleagues at ITU that helped and motivated me.

My flatmates, **Marina Korenevskaya**, **Danilo Luz**, **Gintare**, and **George**. For the friendship, motivation and great moments while I was living in Copenhagen. Marina, thank you for our long conversations and for illuminating my path, you showed me the purpose of this journey. My distant friends **Pozzo**, **Paulo Capeller**, **Victor Rampasso**, **Bruno Cruz**, **Silvio de Paula**, and **Andrey Prado**, who I never lost contact during the Ph.D. **Malene Søholm**, for traveling to distant places and visiting me during my stay abroad in China, and while I was an intern in the USA. I am glad that I decided to take that summer school in Athens.

**Lu Youyou**, for the invitation and organization of my stay abroad at Tsinghua University, in Beijing. **Zhang Jiacheng**, **Chan Youmin**, and my other colleagues at Tsinghua Storage group, for the support and help when I was living in Beijing. **Chen Hongyu**, for sharing her time and helping me with Chinese during my stay abroad program. **David Lomet** and **Jaeyoung Do**, for the opportunity to spend a summer at Microsoft Research Redmond, and for being my mentors during the internship. It was a great experience.

And all the other people who in some way became part of this journey.

# List of Figures

# List of Tables

# 1 Introduction

## 1.1 Context

As Marc Andreesen predicted in 2011, all companies are now software companies [3]. To support the software applications at the core of their business, companies can choose to host their own IT infrastructure on-premise, or to rely on the managed services of a cloud provider. Public cloud providers such as Google, Ali Baba or Amazon operate hyperscale data centers where (i) compute, memory, storage, and network resources are managed and composed to match workload requirements, and where (ii) applications can directly leverage hardware resources. The scale of their data centers makes it possible for these companies to consider custom-made hardware and software components, tailored to their needs. The technology from public cloud service providers is progressively made available to enterprises, that can thus reap the benefits of hyperscale data centers. Conversely, research conducted in academia can impact the design and operation of hyperscale data centers, as they struggle to provide ever lower latency, higher utilization and reduced cost. In this thesis, we focus on software systems for solid state storage in the context of such data centers.

Solid-State Drives (SSDs) have become the secondary storage of choice for data-intensive applications because they offer high-performance, and lower total cost of ownership compared to hard disks. SSDs are composed of tens of storage chips wired in parallel to a controller. Storage chips are based on non-volatile memories such as flash and PCM (Phase-Change memory). A flash chip is a complex assembly of flash cells, organized by pages (4 to 32 kilobytes per page), blocks (64 to 512 pages per block) and sometimes arranged in multiple planes (typically to allow parallelism across planes). Operations on flash chips are read, write (or program) and erase. Due to flash cells characteristics, these operations must respect the following constraints: (C1) reads and writes are performed at the granularity of a page; (C2) a block must be erased before any of the pages it contains can be overwritten; (C3) writes must be sequential within a block; (C4) flash chips support a limited number of erase cycles. The trends for flash memory is towards an increase (i) in density thanks to a smaller process, (ii) in the number of bits per flash cells (from 1 for single-level cell (SLC) to 4 for quad-level cell (QLC)), (iii) of page and block size, and (iv) in the number of planes. Increased density also incurs reduced cell

lifetime (5000 cycles for triple-level-cell flash), and raw performance decreases. This lower performance can be compensated by increased parallelism within and across chips. Z-NAND[1], the latest generation of SLC NAND, announced by market leader Samsung in 2018, trades lower density for higher performance with read and write latency of about 10 microseconds. As an alternative to NAND flash, 3D-XPoint from Intel and Micron has been available since 2017, also with read and write latency of about 10 microseconds.

SSDs initially provided the same interface as magnetic disks. It was therefore possible to replace old disks by new SSDs, but it has become a problem for high-performance workloads. In 2011, Bonnet and Bouganim [8] proposed that SSDs should expose their internals to the host computer, which would then be responsible for data placement and I/O scheduling. About five years ago, Google, Baidu, and others designed special-purpose Solid-State Drives, Open-Channel SSDs, that could expose their internal parallelism directly to applications. In a previous project at ITU, Matias Bjørling defined the Linux framework for Open-Channel SSDs and later J.Gonzalez defined the Linux Flash Translation Layer that makes it possible to use open-channel SSDs with legacy file-system based applications. As a result, Open-Channel SSDs are now becoming commodity components that can be integrated into any data center. Standardization efforts are now underway in the context of NVMe zoned namespaces. Open-channel SSDs provide predictable I/O latency, at the cost of increased CPU load on the host.

In recent years, the rate at which data can be read or written from storage and network devices has increased exponentially, while the rate at which data can be read or written from the memory of a host processor (CPU) has only increased linearly. This trend is expected to continue in the coming years. The increasing volumes of stored data compound this problem. Soon, CPUs will not be able to keep up with the rate at which data moves through storage and network. This trend is especially problematic for open-channel SSDs, that increase CPU load by design.

A way to reduce data movement is to offload processing from the host CPU to the storage controller (i.e., a processing unit embedded an all storage devices), making it possible to scale processing with the increased volume of stored data and creating an opportunity to eliminate overheads and leverage hardware acceleration. There was pioneering work on Active Disks in the 90s, but they did not address a pressing need.

More recent work on computational storage, also called near-data processing, includes pioneering work by Mueller et al. at ETH Zurich, in the context of very low-latency data processing with database systems designed in hardware on FPGAs [63]. Steve Swanson and his team at UC San Diego have explored programmable

[1] Z-NAND: https://www.samsung.com/semiconductor/ssd/z-ssd/

SSDs with Willow [78], that allows applications to drive an SSD by installing custom software on small processors running within the SSD. Also, Jun et al. [40] at MIT, explored a new system architecture, BlueDBM, which systematically pushes processing to FPGA-based SSDs.

Today, efforts in academia and industry fall in two groups. The first group combines Open-Channel SSDs with a programmable storage controller integrated into a network switch (Broadcom Stingray[2]). The second group integrates a programmable storage controller directly onto an SSD (ScaleFlux[3], NGD[4], Samsung SmartSSD[5]). For both groups, the programmable storage is a Linux-based ARM or RISC V processor and/or programmable hardware (FPGA).

Twenty years ago, Jim Gray wrote: Put Everything in Future Disk Controllers (it's not "if", it's "when") [27]. His argument was that running application code on disk controllers would be (a) possible because disks would be equipped with powerful processors and connected to networks via high-level protocols, and (b) necessary to minimize data movement. He concluded that there would be a need for a programming environment for disk controllers. In this thesis, we follow-up on Jim Gray's prediction and introduce OX, a framework for programming storage controllers.

## 1.2 Problem

Defining standard interfaces, operating system services or a programming environment for computational storage are open issues. In this thesis, we focus on how to program a storage controller. The following questions must be addressed: (1) Who programs the storage controller? (2) What is actually programmed? and (3) How are programs written?

**Who programs the storage controller?** Traditionally, SSDs storage controllers have been programmed by firmware engineers with a background in NAND flash. The complexity of FTLs has led to decoupling front-end (FTL) and back-end (storage media) SSD management, respectively focused on software and hardware engineering. The advent of pblk [6] now enables Linux kernel engineers to program host-based front-end SSD management. Whether storage controllers should be programmed by DevOps teams who program and configure their data center, or by FTL specialists is an open issue.

**What is programmed?** What does offloading processing to a storage controller actually mean? For NGD and ScaleFlux, offloading processing to a storage controller

[2]Stingray: https://www.broadcom.com/products/ethernet-connectivity/smartnic/bcm58800
[3]ScaleFlux: https://www.scaleflux.com/
[4]NGD: https://www.ngdsystems.com/
[5]SmartSSD: https://samsungatfirst.com/smartssd/

means installing application-specific code on top of a generic storage management layer. This is the model pioneered by Active Disks in the 90s. Storage devices no longer provide a fixed memory abstraction; they offer a communication abstraction based on a form of (asynchronous) RPC.

But why settle for a generic storage management layer? On flash-based devices, generic FTLs cause redundancies and missed optimization opportunities across layers on the data path [6]. Open-Channel SSDs make it possible to specialize FTLs on top of a well-defined abstraction of the physical address space [25].

Our position is that storage controller programming should be defined as the mapping from commands defined on a logical address space onto storage primitives defined over a physical address space, i.e., modifying a generic FTL rather than simply adding functionality on top of it. Mapping such commands onto a physical address space requires modifying the mapping, garbage collection or recovery functionalities of a traditional FTL.

This thesis is based on the hypothesis that *deconstructing the FTL leads to a modular architecture that can efficiently support the design of application-specific storage controller software*.

The formulation of this problem is inspired by previous work focusing on reconfigurable FTLs by C.Park [66] and others [79], which has shown the potential benefits of tuning FTL parameters to fit the characteristics of given application workloads. Here, we go beyond tuning an FTL and consider a framework for programming computational storage with application-specific FTLs. To the best of our knowledge, we are the first to propose such a framework.

**How are programs written?** The key question is whether a programming environment for storage controllers should be based on a general purpose programming language or on a domain-specific language? Put differently, the question is whether the equivalent of P4[6] can or should be defined for storage controllers[7].

In previous work at ITU, AppNVM proposed match-action rules (inspired by OpenFlow) as a means to implement an FTL [7]. Today, we see little advantages in declarative FTL programming. Indeed programming an FTL requires efficient coordination of many dependent tasks affecting a persistent state.

The question is then what kind of high-level abstractions can be defined for FTL programming and whether they warrant the definition of a domain-specific language (as opposed to a collection of libraries or templates). In their seminal work on compositional FTLs, Prof. Sang Lyul Min and his team, propose a log-based framework for representing FTLs [14]. Today, this is great for checking the correctness

[6] P4 Language website: https://p4.org/

[7] This question was first formulated by T.Roscoe (ETHZ).

of an FTL, but it does not provide a framework for generating application-specific FTLs.

In this thesis, we adopt a procedural programming approach, based on a modular architecture.

## 1.3 Approach

We adopt an experimental approach based on the design, implementation, and evaluation of software systems. This thesis was defined in the context of the DFC open-source community. The DFC community gathers a restricted group of universities, selected by Dell EMC, to experiment with a programmable SSD platform, the DFC card, that Dell EMC and NXP made available for research on near-data processing. Dell EMC donated a DFC card to IT University in January 2016. I was the first to develop software for this platform, an open-channel SSD controller, that I named OX.

The first generation of DFC card is equipped with 40GE, PCIe and a System-on-a-Chip (SoC) LS2085 based on a 8-core ARMV8 processor attached (via PCIe) to an FPGA directly connected to NVM DIMM chips (NAND flash or persistent memories) via up to eight channels. The second generation of DFC card is equipped with 40GE, PCIe and a LS2088 SoC attached to external storage devices via M.2. For more details, see section 2.4.2. Most experiments in this thesis are run on DFC platforms. Reproducing those experiments on similar platforms such as Broadcom ST-1100 is future work.

## 1.4 Contributions

We make the following contributions:

1. We designed and implemented an NVMe controller that supports accesses through PCIe or fabrics (TCP/IP), implements the Open-Channel SSD interface and can be extended with custom commands.
2. We designed and implemented a full-fledged FTL in user-space based on a modular architecture.
3. We leveraged the modular FTL architecture to design and implement an application-specific FTL exposing a log-structured SSD interface.
4. We defined OX, a framework for programming the storage controller on computational storage.

5. We conducted extensive performance evaluations of a prototype and industrial Open-Channel SSD. To this end, we designed a suite of micro-benchmarks for Open-Channel SSDs and we implemented a tool for testing and evaluating Open-Channel SSDs.
6. We evaluated several instances of the OX framework and described the lessons learned from these experiments.

About 36K lines of codes were developed in the course of this PhD. All code is available on GitHub. More specifically, the following prototypes have now been shared with the community:

- *OX: A Framework for Computational Storage*
  https://github.com/DFC-OpenSource/ox-ctrl
- *FOX: A Benchmark Tool for Open-Channel SSDs*
  https://github.com/DFC-OpenSource/fox
- *QEMU-OX: An OX Controller Emulator*
  https://github.com/DFC-OpenSource/qemu-ox

I have implemented all the code mentioned above. I have contributed to several publications throughout the course of my Ph.D. They are listed below. The first two publications are the basis for Chapter 4. The third publication is the basis for Chapter 3. Publications 4 and 5 are the result of a collaboration with Microsoft Research, where I spent 3 months as an intern in summer 2018. They are the basis of the work presented in Chapter 7. The last publication describes work in progress, based on the OX framework and lessons learned in this thesis.

1. *Beyond Open-Channel SSDs*. [69]
   I. L. Picoli, C. V. Pasco, and P. Bonnet. *NVMW '17* (poster)
2. *uFLIP-OC: Understanding Flash I/O Patterns on Open-Channel SSDs*. [71]
   I. L. Picoli, C. V. Pasco, B. Þ. Jónsson, L. Bouganim, and P. Bonnet. *ApSys '17*
3. *Programming Storage Controllers with OX*. [70]
   I. L. Picoli, P. Tözün, A. Wasowski, and P. Bonnet. *NVMW '19*
4. *High IOPS via Log Structuring in an SSD Controller*. [20]
   J. Do, D. Lomet, and I. L. Picoli. *NVMW '19* (poster)
5. *Improving SSD I/O Performance via Controller FTL Support for Batched Writes*. [21]
   J. Do, D. Lomet, and I. L. Picoli. *DaMoN '19*
6. *LSM Management on Computational Storage*. [68]
   I. L. Picoli, P. Bonnet, and P. Tözün. *DaMoN '19*

## 1.5 Structure of the Manuscript

The manuscript is organized in 8 Chapters. A background section follows this introduction and introduces in more depth the relevant aspects of Flash Translation Layers, Open-Channel SSDs, and Computational Storage. In particular, this Chapter describes the DFC platform which is used for all experiments. There is no chapter dedicated to related work. Relevant related work is incorporated in all chapters describing our contribution. The third Chapter describes the OX framework for programming storage controllers and present experimental results fixing the boundaries for the performance we can expect on the DFC platform. In the next sections, we instantiate OX in different contexts with the overall goal of exploring how a modular FTL architecture supports the design of application-specific FTL on computational storage. First, we instantiate OX to explore the characteristics of open-channel SSDs (Chapter 4), then we explore the modular design of an FTL (Chapter 5). We design, implement and evaluate a generic FTL (Chapter 6) and an application-specific FTL (Chapter 7) before summarizing our results, presenting the lessons learned and the topics for future work in Chapter 8.

# 2 Background

## 2.1 SSD Organization

SSDs are composed of arrays of flash chips wired in parallel on physical **channels**. A channel is the unit of independent parallelism: there are no interferences across channels. Several **parallel units** (PU) or flash chips are connected to the same channel. There might be interferences across PUs on the same channel (a write request on one PU might have to wait until a read request on another PU completes before it can be issued). PUs are the minimal unit of parallelism in SSDs. Each PU is capable of performing flash operations in parallel to other PUs. SSDs achieve many orders of magnitude higher throughput than a single flash chip by spreading flash operations among PUs.

PUs are organized in **chunks**, also called flash blocks. Manufacturers usually assemble SSDs with thousands of chunks within a PU. A chunk is the unit of erasure and is organized in flash **pages**, a chunk usually contains less than a thousand pages, depending on the manufacturer. A flash page is the minimum unit of writing, meaning that enough data need to be buffered before the write operation takes place. Flash pages are also split in four to eight smaller units called flash **sectors**, a sector is the minimum unit of reading and usually built with 4 KB of flash cells.

The flash chips are composed of a multi-dimensional array of flash cells, the number of bits stored in a cell determines the flash memory technology. Cells storing one, two, three and fours bits are, respectively, SLC, MLC, TLC and QLC types. Today, TLC is commodity in SSDs, however, three-dimension QLC is the focus of the industry due to a higher density and low cost. SSDs up to 16 TB are already available today. SSDs are basically split in two parts, non-volatile memory (NVM) and controller. The controller is responsible for NVM abstraction and communication to hosts. Most of the controller software is composed of the flash translation layer, described in the next section.

## 2.2 Flash Translation Layer

SSDs are widely used not only because of their high performance but also due to the support for legacy systems provided by flash translation layers (FTL). In standard SSDs, the FTL is embedded within the SSD firmware. It hides the complexity of the underlying media and enables in-place updates, which are not allowed in flash memory. Historically, computer systems were built based on hard disks that expose a block device interface: a flat space of logical block addresses on which read and write operations are supported. NAND flash is not a natural match for this interface. As we have seen, flash-based SSDs are organized in channels, PUs, chunks, pages, and sectors. In addition, NAND flash imposes a set of constraints on the operations it supports: chunks (also called flash blocks) must be erased before any page they contain can be re-written, page writes must be sequential within a chunk, the number of erase cycles per chunk is limited. Additional constraints might be introduced for SLC, MLC, TLC or QLC technologies or by NAND vendors.

The role of the FTL is to expose a logical address space that abstracts the physical address space available in an SSD. For example, the FTL of commercial flash-based SSDs exposes the same block device abstraction as hard disks. Such FTLs must support in-place updates and convert the flat space of logical block addresses onto the hierarchical physical address space of an SSD. The three FTL functionalities deriving from the logical to physical translation are mapping, garbage collection and wear leveling.

The FTL maintains an explicit mapping of logical addresses onto physical addresses. Mapping was the focus of early research efforts with block level mapping, page level mapping, and hybrid mappings. Logical write operations are issued at different granularities, the FTL is responsible for breaking data into pieces that match media boundaries. A mapping table is required to map logical addresses into physical addresses. Preferably, logical address sizes should match physical pages, but this is not a rule. If logical addresses are at a smaller size, additional metadata management is needed.

As there are no in-place updates in NAND flash, updates on a logical address require (i) that the new value is written at a new location, (ii) that the mapping table is modified to point the same logical address to the new physical address and (3) that the page containing the old value is marked as invalid. As storage fills up, chunks contain a mix of valid and invalid pages (in the general case). As a result, garbage collection is necessary before a chunk can be freed. When a block is to be erased, the valid pages it contains must be written on another chunk. Garbage collection generates read and write operations on the physical address space and creates write-amplification [58]. Also, erase operations are ten times more expensive than writes, which introduces a scheduling concern in high-performance FTL designs.

Wear leveling is concerned with making sure that all chunks are erased at approximately the same rate, or at least that chunks are not erased at a rate that would cause their early degradation. Dynamic wear leveling is concerned about which chunks should be written to next. Static wear leveling is concerned about storing cold data onto chunks that have been erased a lot. Dynamic wear leveling is part of mapping. Static wear leveling is part of garbage collection.

Many research papers have been published in the area of FTL design. Most of them focus on mapping, garbage collection or wear leveling. In 2018, Prof. Sang Lyul Min and his team published a seminal paper on FTL design focusing on what they call the *Achilles heel* of FTLs, i.e., crash recovery [14]. Indeed, crash recovery is not a direct consequence of the logical to physical translation at the heart of any FTL. However, crash recovery is of the essence to guarantee that the metadata that are necessary for mapping, garbage collection and wear leveling are maintained in a durable state in a way that guarantees their consistency in the case of failure. The key insight from this paper is that read and write operations on the logical address space should be considered as transactional contexts for all operations on the physical address space. To be more specific, all the operations on the physical address space associated with a single read or write operation on the logical address space should be considered as part of the same transaction. This is a key insight that we build on in this thesis.

Prof. Sang Lyul Min and his team describe the need for a combination of shadow paging and write-ahead logging as a means to ensure durability and atomicity for these FTL transactions. Shadow paging is an atomic switch between the old value and the new value of a data item on durable storage. This is necessary to ensure that none of the effects of aborted transactions (due to failure) are durable. Write-ahead logging ensures that log records are written to disk to support redo-based recovery in the case where volatile data structures are lost (due to failure). Note that undo is not an option because NAND flash does not allow in-place updates. We will get back to these notions of shadow paging and write-ahead logging when we describe the design of our various FTLs. We note here that the work of Prof. Sang Lyul Min assumes that the FTL is designed independently from the upper layers in the system. An assumption that we relax when we introduce cross-layer design and design an application-specific FTL.

For years, FTLs were proprietary software. The OpenSSD project [80], led by Prof. Yong Ho Song at Hanyang University, introduced sample FTLs for the Jasmine OpenSSD platform as open-source software, in 2011. The first full-fledged, industry-grade, open source FTL was designed and implemented by J.Gonzalez and released as part of the Linux kernel (4.12) open-channel subsystem in July 2017. Key design characteristics of pblk are its thread model articulated around the write cache and the introduction of the line abstraction as a means to organize data striping across channels and PUs. The availability of pblk and its clean design have inspired and

greatly facilitated our work on a modular FTL architecture, in user-space, for computational storage. Recently, Intel has released a full-fledged open-source FTL in user-space in the context of SPDK[1], which now supports open-channel SSDs. An in-depth comparison of the Intel FTL, with pblk and the FTLs we designed is a topic for future work.

## 2.3 Open-Channel SSDs

In traditional SSDs, the FTL is part of the firmware embedded on the SSD controller. The problems associated with embedding complex Flash Translation Layers are well documented: redundancies (log-on-log) [58, 86], large tail-latencies [18, 35], unpredictable I/O latency [13, 42, 45], and resource under-utilization [1, 12].

As an alternative, open-channel SSDs expose their internals thus allowing the host to manage data placement and I/O scheduling. The PPA I/O interface was originally proposed as a standard interface for open-channel SSDs, in a previous project at ITU, together with LightNVM which is now distributed as a subsystem of the Linux kernel[2]. This interface is an extension of the NVMe standard.

With the initial version of the PPA interface, FTL responsibilities such as wear leveling, logical to physical mapping, garbage collection, bad block management, among others, may be implemented on the host side or remain embedded on the SSD controller. This opens up a large design space. A point in this design space emerged as an industry consensus, primarily based on the requirement that storage devices should provide a warranty. This makes it necessary for the SSDs to maintain metadata about the media (i.e., wear leveling, bad block management, and ECC remain embedded while mapping and garbage collection are managed on the host). This led to a second version of the open-channel interface supported by LightNVM in Linux.

In the last year, standardization efforts have focused on open-channel SSDs: (i) Project Denali driven by Microsoft and CNEX Labs and (ii) zoned named devices in the context of NVMe. As mentioned previously, there is now open-channel SSD support in SPDK. In March 2019, Alibaba announced the deployment of their open channel SSD platform with their industry partners [88].

[1] https://spdk.io/doc/ftl.html
[2] http://lightnvm.io/

## 2.4 Computational Storage

### 2.4.1 Programmable Storage Controller

The term "Computational Storage" has evolved from the pioneering idea of active disks from Jim Gray [27]. Other terms such as "near-data processing" and "application-managed flash" [50] were also used in the literature for similar work as Jim Gray predicted. Later efforts on open-channel SSDs [6] allowed new designs of flash translation layers and a better understanding of flash characteristics. Today, computational storage is evolving towards hardware accelerating applications by offloading processes to the storage controller. The development of a controller is heavily guided by the understanding of media behaviors, which are deeply studied by the open-channel SSD effort. We argue that media management is crucial for computational storage, layers in the storage stack can be collapsed into a simpler streamline by merging them at the media management level, thus we believe that FTLs for computational storage are application-specific FTLs. To simplify the argument, we separate computational storage in two classes, (i) on top of generic FTLs, and (ii) on top of application-specific FTLs. (i) is the state-of-the-art, companies such as ScaleFlux and NGD Systems are pioneers on this design, (ii) is a new class we propose, instead of using generic FTLs, we collapse layers and merge them into application-specific FTLs. Figure 2.1 depicts the state-of-the-art on generic FTLs (left) and the proposed class on application-specific FTLs (right).

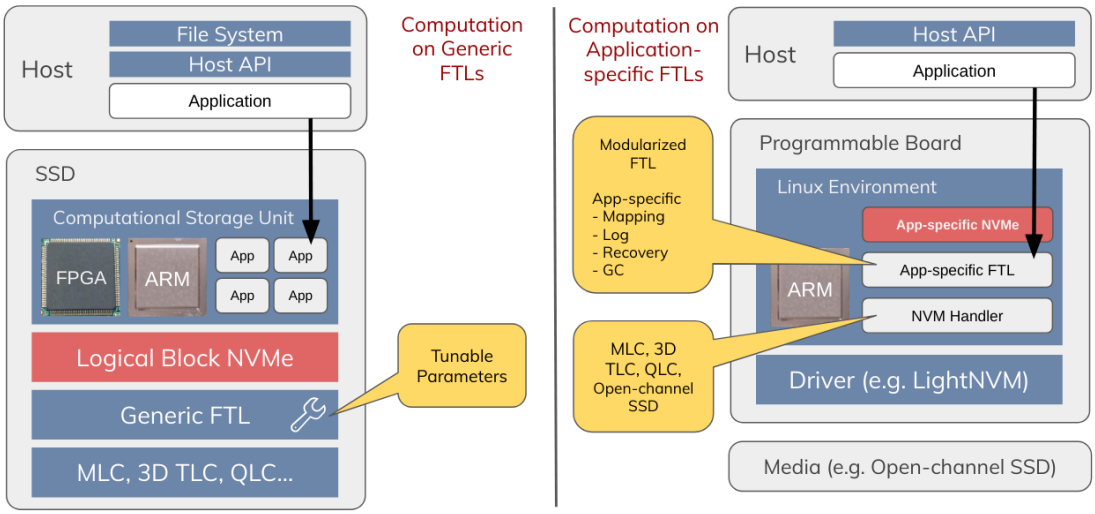



FIGURE 2.1: Computational Storage Classes

At the left, applications on host machines use computational storage libraries to offload processing to the storage controller. The storage controller is composed of either a general-purpose CPU or an FPGA where application slots are reserved for offloaded applications. The simplicity of this design is the block interface, applications access the media via standard file systems and a generic FTL is used to hide the complexity of underlying media. The advantage of offloading applications to

other CPUs naturally improves performance on host machines, however, the underlying media is still a black box and collapsing layers at the FTL level is not an option, we have to rely on hidden algorithms used by the embedded FTL. On the right of the figure, we present computation on application-specific FTLs and the block interface is no longer a requirement, for instance, the open-channel SSD interface can be used instead. For efficiency and flexibility, the architecture should allow replacement of underlying media without affecting the media management design, at the same time, media management should be modularized in a way to support replacement of FTL pieces by application code as a way to collapse layers. To support the idea of application-specific FTLs, we suggest the development of SSD controllers in three layers:

- **Transport and Parsers:** Communication to hosts should be via standard protocols such as NVMe [64]. The protocol should allow application specific commands such as custom opcodes in the NVMe specification. This layer is responsible for the implementation of standard protocols and for the parsing of application specific commands with the goal of delivering reliable and consistent messages to the media management layer.
- **Media Management:** This layer defines the logic of application-specific FTLs. The FTL serves as a mediator between the host application and the underlying media. The design of this layer should be modularized in a way to provide replacement of fine-grained FTL components. Components may be application-specific, collapsing layers in the storage stack.
- **Media Abstraction:** To support heterogeneous media, this layer should provide a standard abstraction for media addressing, thus allowing FTLs to use different forms of media without modification on its code. We strongly rely on the open-channel SSD interface as a standard addressing protocol for non-volatile memories.

The three suggested layers for application-specific FTL controllers are materialized in the context of our OX Controller, described in the next chapter of this thesis.

### 2.4.2 Dragon Fire Card (DFC)

The Dragon Fire Card was first architected by Dragan Savic at Dell EMC, then further engineered and built by VVDN Technologies. Today, NXP leads the support and new development on the platform. The DFC was the first open-source hardware for computational storage, the platform is also used as smart NIC for network processing, as well as a hardware accelerator for data compression/decompression. All firmware and software support, as well as our OX Controller, are available on the DFC Open Source Community[3].

[3]https://github.com/DFC-OpenSource

The DFC is assembled in two parts, (1) SoC board (main board), and (2) storage board. The SoC board was released in several generations, we describe the LS2088 SoC. The board is equipped with an 8-core Arm® Cortex®-A72 @ 2.0 GHz, 32 GB of DRAM with ECC enabled, and 4x10 Gbit/s Ethernet interfaces. In addition, the board has 2 PCIe endpoints and 2 PCIe root complexes used for host communication via PCIe, and registration of devices attached on the storage board, respectively. The storage board attaches to the main board via PCIe root complexes and can be replaced. Currently, two types of storage board exist, (2a) an FPGA-based design equipped with DIMM slots for custom NVM designs, and (2b) an M.2 adapter equipped with two 4-lane M.2 slots. Figure 2.2 shows a DFC equipped with an FPGA storage board, while figure 2.3 shows a DFC equipped with an M.2 adapter.

In figure 2.2, the FPGA board is equipped with custom MLC NAND DIMMs. In figure 2.3, cables M.2→U.2 and M.2→PCIe are attached to the M.2 adapter, at the bottom right we can see a CNEX Open-channel SSD.

FIGURE 2.2: DFC equipped with FPGA Board (2a) and NAND DIMMs

Later on this thesis, the setup (1,2a) is used for experimentation with the first generation of our OX Controller, and setup (1,2b) is used for the second and third generations.

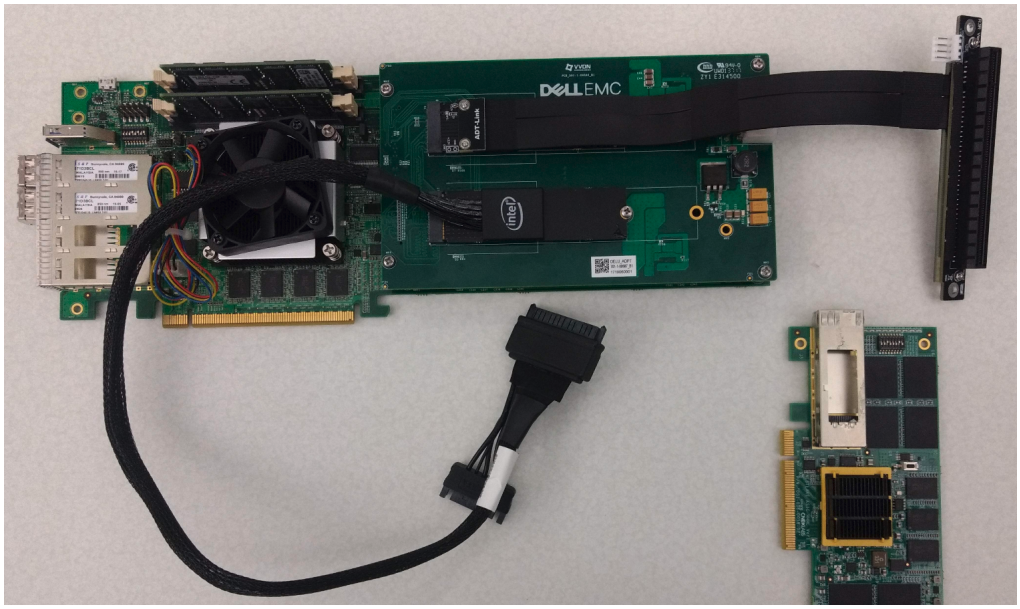



FIGURE 2.3: DFC equipped with M.2 Adapter (2b) and Open-Channel SSD

# 3 The OX System

OX is a framework for programming storage devices equipped with computational capabilities., i.e., a programmable storage controller. OX proposes a modularized architecture for computational storage. The system was developed in three generations, first, it was designed to serve as open-channel SSD controller, implementing the LightNVM specification. Then, it evolved to a second generation as a framework for FTL development, implementing OX-App and a generic FTL abstraction. The third generation arose to be a complete framework for programming storage controllers, with a host API and two full-fledged FTL implementations, OX-Block and OX-ELEOS. OX was tested and evaluated with ARMv8 and x86 architectures. Tests with other architectures are feasible but subject for future work. This chapter describes the concept and the design of OX. The first generation is detailed in chapter 4, the second is detailed in chapter 5, and the third is split into chapter 6 and 7.

OX is composed of three layers, as proposed in section 2.4.1. The bottom layer focuses on media management, responsible for abstracting various forms of underlying storage media under a common interface of the physical address space. The middle layer is responsible for the translation of logical addresses to physical addresses (exposed by the bottom layer). The upper layer is the host interface, where storage specifications and custom commands are implemented for interacting with host applications. The three layers of OX are shown in figure 3.1.

## 3.1 Physical Address Space

OX abstracts the underlying storage media with a hierarchical address space, similar to the one defined for open-channel SSDs. The physical address space is organized into a hierarchy of channels (the unit of independent parallelism), PUs (the unit of parallelism within a channel), chunks (where logical pages are written sequentially) and logical pages (the unit of read and write). Each logical page is organized in a set of sectors.

This design makes it possible for OX to incorporate open-channel SSDs as storage backends and to organize any form of NVM into a well-defined abstraction.

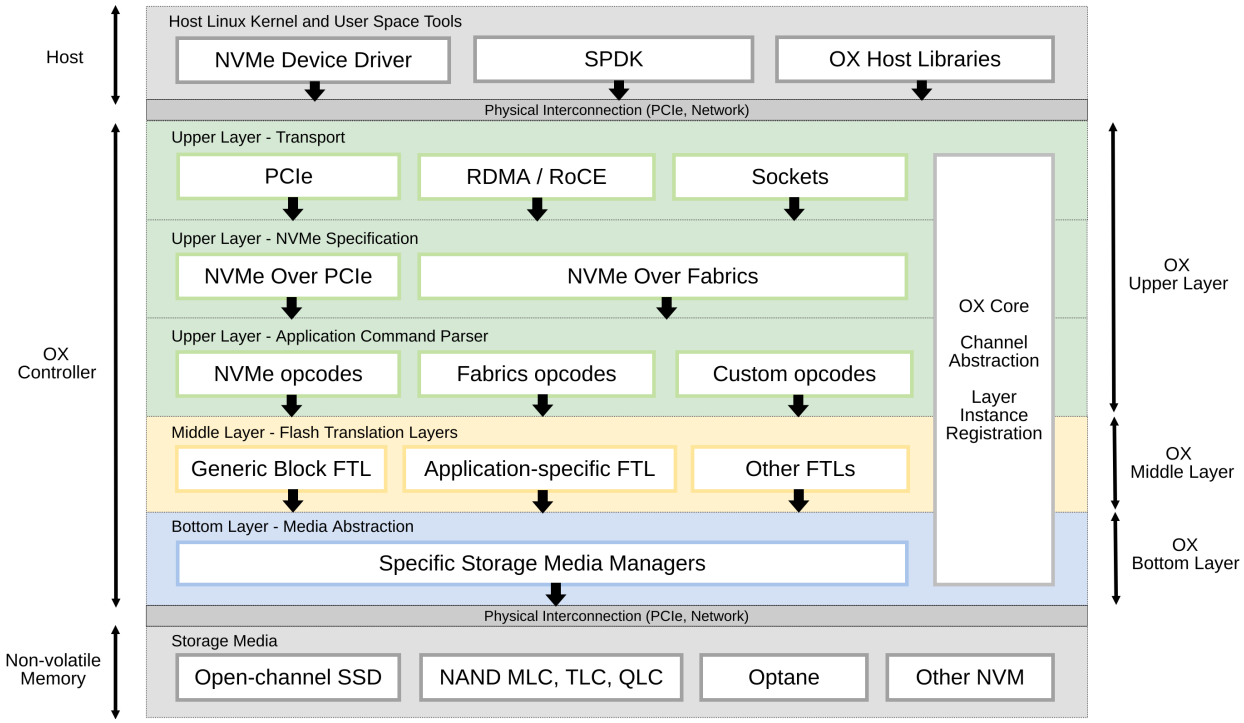



FIGURE 3.1: OX Controller Layers

The size of each level in the hierarchy is defined by the media manager that exposes the channel geometry. A 64-bit value is used for addressing any layer within a channel, meaning that the sector is the minimum addressable unit. When designing a media manager, a developer may use the media constraints to define boundaries. For instance, flash memory has different granularities for write, read, and erase commands, which could be the size of sectors, pages, and chunks, respectively. Media managers maintain metadata about the organization of each channel. This metadata is stored within the channel it describes.

## 3.2 The Bottom Layer: Media Managers

Media managers are instances of the bottom layer that allow different forms of media to be used via a standard interface. Each type of media requires an instance of the bottom layer in OX, where each instance exposes its media as a set of channels.

All channels exposed by the media managers will form a global namespace organized in an array of channels, further used by the middle layer. In case of media replacement for a new sort, a new media manager is required while the middle layer remains untouched.

A media manager is responsible for (a) exposing channels into OX, (b) receiving I/O commands from the middle layer and translate to media specifics, (c) moving data to/from hosts according to the commands, and (d) reporting I/O errors to the

middle layer. Besides exposing channels, a minimal media manager implementation contains the following functions:

- **Read:** Receives a list of sectors to be read, and moves data from the media to memory buffers.
- **Write:** Receives a list of sectors to be written and moves data from memory buffers to the media.
- **Erase:** Prepares a chunk to be used. In case of flash memory, it must erase the flash block linked to the chunk.

Note that memory buffers might not be located on the storage controller. They might be located on the host, directly attached or connected via fabric. DMA is required in the former case, and RDMA in the latter case to support accesses to the host memory. Such approaches avoid memory copies and have the advantage of bypassing the storage controller during data transfers.

## 3.3 The Middle Layer: FTLs

FTL code is located on this layer, which manages the mapping of logical to physical addresses. The term FTL comes from flash-based storage devices. However, in OX, an FTL may be designed to interact with other forms of NVM using the channel abstraction model. From the top layer's point of view, an FTL is a black box that exposes submission and completion queues. Submission queues process I/O commands submitted by the upper layer, while completion queues use callbacks for completion. The upper layer operates on logical addresses without the concern of mapping data to physical media. By design, mapping is the responsibility of the middle layer.

If we look inside the black box, we find a complex family of algorithms that deal with mapping, logging, checkpoint, recovery and garbage collection. A detailed explanation of these algorithms is given later. In this section, we only consider the middle layer as a suitable location for FTLs.

OX may implement several FTLs. Each FTL has a unique FTL identifier. During controller initialization, each channel exposed by a media manager has its first block scanned for metadata information. If metadata is found, the channel is registered to the FTL pointed by the FTL identifier, found in the first block metadata. Otherwise, the channel is registered to the standard FTL. After all channels and their associated FTLs have been initialized, the middle layer is ready to accept incoming I/Os. The I/O path across the middle layer consists of:

1. **Logical Submission:** The upper layer submits a logical I/O to the middle layer via submission queues.

2. **Mapping:** The middle layer maps the logical addresses to physical addresses.
3. **Physical Submission:** The middle layer submits physical I/Os to the bottom layer.
4. **Physical Completion:** The bottom layer completes physical I/Os via callbacks to the middle layer.
5. **Logical Completion:** When all physical I/Os complete, the middle layer completes the logical I/O to the upper layer via completion queues.

A minimum implementation of an FTL contains the following functions:

- **Submit:** Receives a logical I/O command from the upper layer.
- **Mapping:** Complex mapping strategy and metadata management. This function is responsible for sending physical I/Os to the bottom layer. A single logical I/O may generate several physical I/Os.
- **Callback:** Completes I/O commands processed by the bottom layer.

## 3.4 The Upper Layer: Host Interface

OX upper layer is divided in (i) transport, (ii) NVMe specification and (iii) command parsers. The next three subsections describe the upper layer division, respectively.

### 3.4.1 Transports

The transport is the communication between hosts and controller. OX supports network fabrics and PCI Express transport types. Examples of transports are TCP/UDP sockets and RDMA technologies for network fabrics, and PCIe messages for PCI Express. A transport implementation contains the following functions:

- **Create/Destroy Connection:** Manages end-to-end connections with hosts. A connection must handle incoming I/Os from hosts and send it to the command parser.
- **DMA/RDMA:** Transfers data to/from hosts. DMA engines may be used for PCI Express, while RDMA techniques may be used for network fabrics.
- **Complete:** Completes I/O commands processed by the middle layer. The completion is done via the same connection the I/O was submitted.

### 3.4.2 NVMe Specification

OX implements both the base and fabrics NVMe specifications, supporting PCIe and network fabrics transports. For details about the specification, please refer to [64]. During initialization, OX identifies the transport to be used. In case of fabrics, it accepts network connections. For PCI Express, it registers the NVMe controller for host probing.

The NVMe protocol is built based on submission and completion queues. A host driver initializes the storage controller by sending the 'identify' command. When the controller is ready, queues are started by a 'connect' command in network fabrics, or by a 'create queue' command in PCI Express. After the connection is established, host and controller are ready for commands, each command contains a byte called operation code (opcode), used to identify the command.

A few differences are found between NVMe over PCIe and NVMe over Fabrics, the differences are at the mechanisms the controller uses to fetch commands from hosts. For PCIe, doorbell registers are used to notify the controller about new entries in the submission queue. For network fabrics, connections receive commands encapsulated into NVMe capsules defined in the NVMe over Fabrics specification. An NVMe capsule may contain a single command or may also be followed by data. When followed by data, the capsule contains *in-capsule* data.

### 3.4.3 Command Parsers

Incoming I/O commands fetched by the transport are sent to command parsers. Parsers are organized by specifications, for instance, NVMe base, NVMe over fabrics, LightNVM, and custom commands are different types of parsers. Each parser type is allowed to manage several operation codes, but not allowed to manage a code that is already used by another parser. A command parser is responsible for (i) command interpretation, (ii) data pointers organization, and (iii) command submission to the middle layer. The responsibilities are detailed below:

- **Interpretation:** By default, a command must be 64-byte wide. The parser reads the operation code and calls a proper function.
- **Data Pointers:** Data may be represented with different data structures, examples are Physical Region Page (PRP) lists and Scatter Gather Lists (SGL). The parser is responsible to organize the data pointers into an array, required by the middle layer. A data pointer might be a host memory address used for DMA/RDMA, or an offset within the *in-capsule* data (in case of NVMe over Fabrics).

- **Submission:** Commands must be formatted to a default structure required by the middle layer. When data pointers and command structure are ready, the command is submitted to the middle layer.

### 3.4.4 Custom SSD Interfaces

The flexibility of NVMe allows custom command definition with two requirements: (i) the command must be 64-byte wide, and (ii) the first byte must contain the operation code. Application-specific code running in the storage controller interprets custom commands defined by a host application, this flexibility is key for computational storage. Later in chapter 7, we describe ELEOS, a custom SSD interface for log-structured stores.

## 3.5 OX-MQ: A Parallel I/O Library

Parallelism is the main advantage of modern SSDs and OX must execute I/Os in parallel. The upper, middle, and bottom layers of OX have different responsibilities, thus commands should cross the layers in an independent manner. We separate the layers by implementing queue pairs. Commands are submitted and completed to the next layer via submission and completion queues. Each queue has its own thread which dequeues entries in a loop, for each entry it executes user defined functions for submissions and completions. By default, an instance of OX-MQ is available in all OX layers, it is up to OX developers whether the default queues are used or not. The OX-MQ library is a way of simplifying and standardizing the development of parallel I/Os in storage controllers, therefore, threads and queues are handled by the library. An instance of OX-MQ contains multiple queue pairs, the number of queues is defined at the library instantiation, as well as the queue depth and other attributes. The attributes are passed to the library via the structure below.

```
1  struct ox_mq_config {
2      char                name[40];
3      uint32_t            n_queues;
4      uint32_t            q_depth;
5      ox_mq_sq_fn         *sq_fn;      /* submission queue consumer */
6      ox_mq_cq_fn         *cq_fn;      /* completion queue consumer */
7      ox_mq_to_fn         *to_fn;      /* timeout function */
8      uint64_t            to_usec;     /* timeout in microseconds */
9      uint8_t             flags;
10
11     uint64_t            sq_affinity[OX_MQ_MAX_QUEUES];
12     uint64_t            cq_affinity[OX_MQ_MAX_QUEUES];
13 };
```

A name for the OX-MQ instance, the number of queue pairs, the queue depth, a submission function, a completion function, a timeout function, the timeout in microseconds, flags, and the thread affinity for each queue, respectively, are the required parameters to instantiate OX-MQ. The library supports thread affinity up to 64 cores, each bit at *sq_affinity* and *cq_affinity* arrays represents a CPU core. If the bit is set, the core is part of a CPU set for the corresponding array index, the array index represents the queue identifier. The latest version of OX-MQ supports the two flags below.

```
1 #define OX_MQ_TO_COMPLETE    (1 << 0)
2 #define OX_MQ_CPU_AFFINITY   (1 << 1)
```

The first flag defines if timeout commands are completed or not by calling **cq_fn*, this avoids double completion if the command is already completed by another thread. Timeout can also be disabled by setting *to_usec* to zero. The second flag enables thread affinity, if this flag is set, the affinity described by the affinity arrays is respected.

Figure 3.2 shows the abstract model of OX. The model shows the layers separated by queue abstractions, each abstraction is an OX-MQ instance which contains several queue pairs.

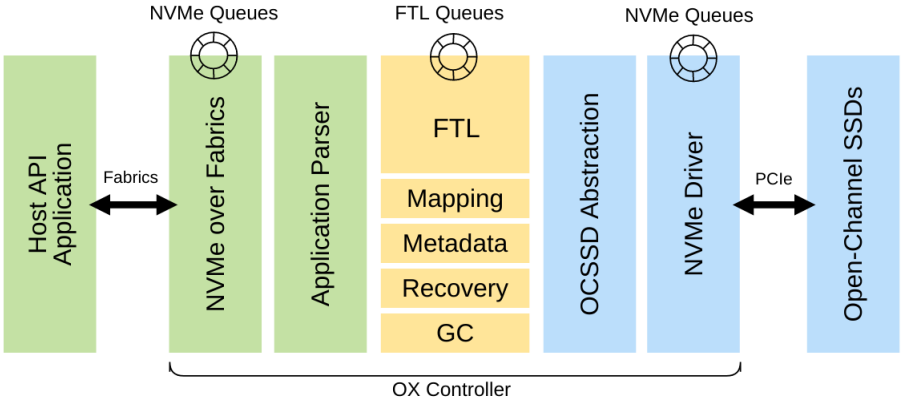



FIGURE 3.2: OX Abstraction Model

Green represents the upper layer with fabrics connection and application parser, yellow is the middle layer with FTL components, and blue is the bottom layer with open-channel SSD abstraction. The layers are interconnected via queues instantiated by the OX-MQ library. Upper and bottom layers contain queues for NVMe abstractions, NVMe over Fabrics at the upper and NVMe over PCIe at the bottom. Later in our FTL implementation, we explain that queues on the FTL are only used for writes. Reads dequeued from the upper NVMe queues are directly posted to the bottom NVMe queues to improve read latency. The number of queues in each layer is customized depending on the number of available CPU cores in the controller. Figure 3.3 shows the internals of each OX-MQ instance.

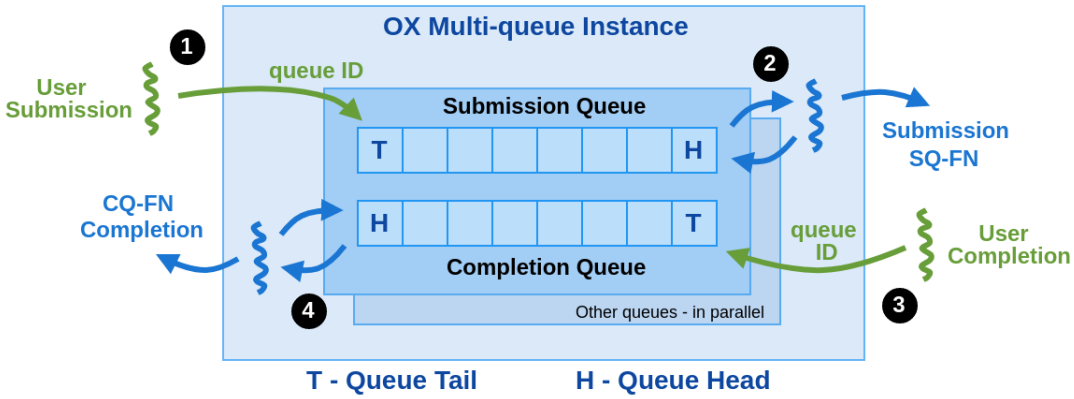



FIGURE 3.3: OX-MQ I/O Processing

The blue threads are created and built-in by OX-MQ library while green threads are created by other components. We represent in green the user threads from other layers. When several OX-MQ instances work together, blue threads might be green threads from other instances, forming a chain of queues. In figure 3.3, step 1, a user thread submits a command to OX-MQ, a queue identifier is required for correct parallelism. In step 2, a built-in thread dequeues commands from the submission queue and executes the submission function. The built-in thread stays in a loop executing commands, and sleeps if the queue is empty. POSIX conditions are implemented for waking up the thread instantaneously if a command arrives. In step 3, a user thread completes a command to OX-MQ using a queue identifier. In step 4, a built-in thread dequeues commands from the completion queue and executes the completion function, the thread behavior is similar to step 2. Queue pairs and OX-MQ instances run in parallel, however, the number of parallel units in open-channel SSDs is a lot higher (hundreds) than the number of available CPU cores, which is a concern.

We conducted an experiment to measure the performance of OX-MQ in different platforms, we run the experiment in three different systems, we chose (i) a 4-core Intel® Core™ i7-4810MQ CPU @ 3.6 GHz with two threads per core via hyperthreading, (ii) a 2-socket, 16-core Intel® Xeon® Silver 4109T CPU @ 2.3 GHz with two threads per core and two NUMA nodes, and (iii) A DFC equipped with an 8-core Arm® Cortex®-A72 @ 2.0 GHz with no hyperthreading. The goal was measuring performance of OX-MQ without interference of any other process, thus, we wrote a test application that instantiates the queues and submits null commands, without data. Commands submitted in step 2 are directly completed to step 3.

We measured the performance of a single queue pair running on two cores, one for submission and other for completion. We then increased the number of queues as well as the number of cores in a way that queue pairs are always linked to two different logical cores. We increased the number of queues until reaching the number of available cores. For instance, 16 queues are linked to 32 cores. Figure 3.4 shows the experiment in millions of IOPS.

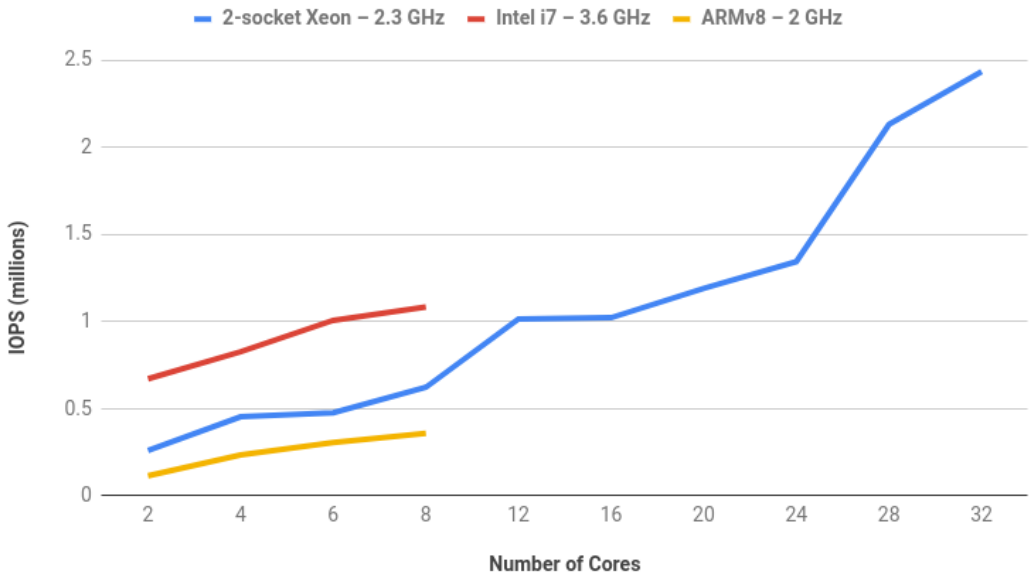



FIGURE 3.4: OX Multi Queue Performance

We submitted 4 million commands in each experiment and spread them across queues using round-robin. The ARM and the i7 scales up to its 8 cores. The Intel i7 has the best performance with a single queue and two cores but does not scale linearly. The ARM has better scalability but shows 5.8x less performance than the Intel i7, less than we expected by looking at the frequency – 2.0 GHz against 3.6 GHz. The Xeon exhibits several gaps while it scales up to its 32 cores, this is due to the NUMA node performance that is outside OX-MQ library, we divided the workload equally between both NUMA nodes. If we compare to the Intel i7, the Xeon performance with a single queue was expected by looking to its core frequency. At 8 cores, our system sustains up to 360K IOPS for ARM, 622K for Xeon, and 1.08M for Intel i7. At 32 cores, the Xeon system sustains up to 2.43M IOPS.

### 3.5.1 NVMe over Fabrics Performance

NVMe over Fabrics (NVMeoF) was introduced as a means to standardize Storage Area Network (SAN), the standard was merged as an extension to the NVMe specification and is largely used by the industry. NVMeoF introduces a way to connect NVMe queues over a network connection where data is transferred using a transport protocol. The transport can be RDMA such as InfiniBand and RoCE, or socket connections. RDMA is achieved via hardware or software, while socket connections can only be done via software. Hardware RDMA requires specialized hardware that implements the RDMA protocol, but delivers the highest performance by removing CPUs from the data path. Software RDMA is part of an open-source effort[1]

[1]Soft-RDMA: https://github.com/linux-rdma/rdma-core

| Transport Type | CPU involved | Kernel involved | Price | Flexibility |
|---|---|---|---|---|
| Hard-RDMA | No | No | $$$ | Low |
| Soft-RDMA | Yes | No | $$ | Medium |
| Sockets | Yes | Yes | $ | High |

TABLE 3.1: NVMeoF Transport Types and Characteristics

to support RDMA on commodity Ethernet interfaces, the performance is not comparable to hardware support but flexibility and lower prices are the main benefits. Table 3.1 shows three ways of implementing NVMeoF, followed by three metrics: CPU overhead, price, and flexibility.

We compare the cost of moving data through the network by CPU utilization and operating system kernel overhead. Prices are related to hardware equipment and specialized personnel. Flexibility is related to hardware migration and software updates.

Hard-RDMA bypasses both CPU and kernel to achieve higher performance, but costs of hardware and personnel are much higher compared to other techniques. In hard-RDMA, flexibility is low in terms of hardware and software changes. Soft-RDMA is cheaper in terms of hardware while specialized personnel is still required, it is more flexible than hard-RDMA in terms of hardware but software updates may not be flexible. In theory, soft-RDMA gives superior performance compared to sockets by removing the kernel from the data path but the CPU is still involved. Sockets are the cheaper and the most flexible approach, however, the expected performance is lower than RDMA because the CPU and kernel are involved.

All experiments performed with the second and third generations of OX use NVMeoF and sockets as standard transport. We measured the performance of TCP stream sockets in OX. Our system was composed of a single 8-core ARMv8 DFC equipped with 2x10 GBit/s Ethernet interfaces, the host machine was a 32-core Intel®Xeon®Silver 4109T CPU @ 2.00GHz equipped with 128 GB of DRAM and an Intel X710-DA2 Ethernet card. Host and DFC were connected via SFP+ transceivers. Figure 3.5 shows the performance of our NVMeoF using sockets.

Our goal was measuring the performance of the DFC cores for NVMe over Fabrics. As we implemented the queues via stream sockets, CPUs and kernels are involved both at the host and DFC. We first measured the performance of a single DFC core by connecting a single NVMe queue, then, we increased the number of cores and also the number of queues in a way we always had 1 queue connected per logical core.

In figure 3.5, at the top left, it shows the throughput of reads and writes. We run eight experiments for each read and write types, and increased the number of cores

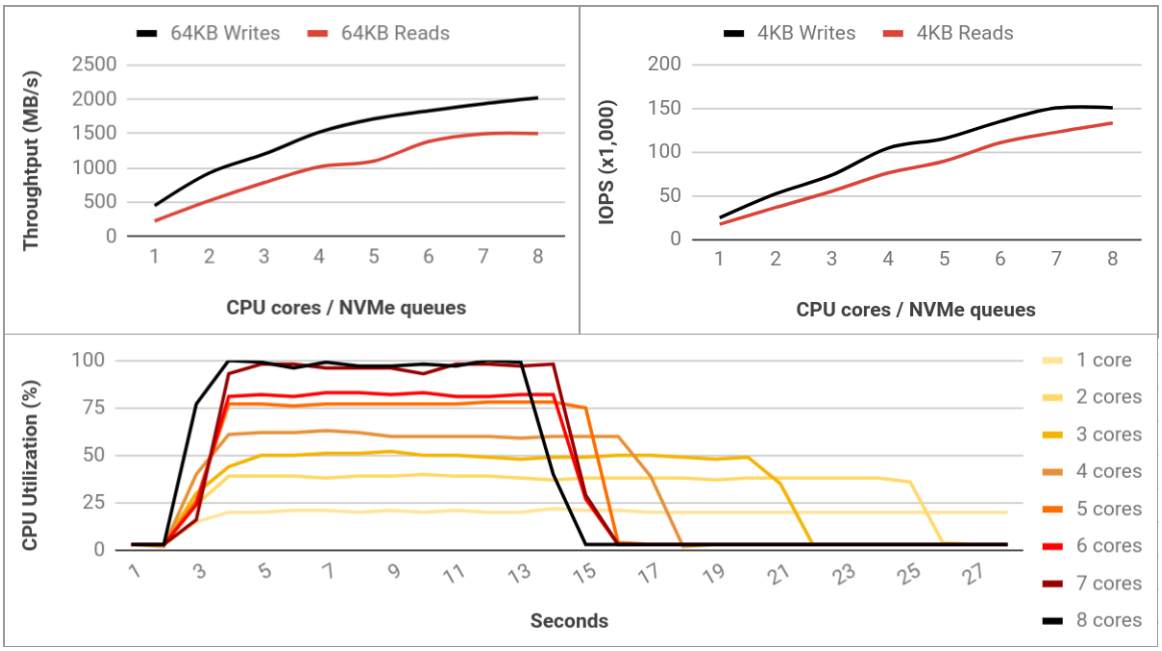



FIGURE 3.5: NVMe over Fabrics Performance using Sockets

until all DFC cores were utilized. In each experiment, we transferred 20 GB of data via 64 KB commands. On the DFC, we discarded received data for writes, and sent generated data for reads to the host, all commands were completed via completion queues.

At the bottom, the figure shows the DFC cores utilization for writes, the percentage is related to 8 cores. Surprisingly, reads consume more CPU cycles than writes and lower throughput is seen. To study this behavior, we inverted the setup by running OX on the host and the tests on the DFC, surprisingly again, reads became faster than writes. We assume that transferring data from DFC to the host via sockets consumes more CPU cycles on the DFC, this is still an open issue. The CPU utilization increases linearly up to 5 cores. For 5 and 6 cores as well 7 and 8 cores, CPU utilization is similar. We believe this is related to having only 2 ethernet interfaces connecting the host and DFC. When 6 or more cores are enabled, other threads in the kernel are assigned to OX-disabled cores, reaching 100% before we assign 8 cores to our setup.

At the right top, the figure shows the IOPS of 4 KB reads and writes. Similar to throughput, reads are slower than writes. In both left and right figures, the performance does not increase linearly, above 4 cores the overhead of sockets is clearly seen. Developing and measuring the performance of soft-RDMA and hard-RDMA are the next steps towards the next generation of OX and still an open study in our project.

Improvements in the socket approach are possible by implementing the XDP

socket type[2], XDP is optimized for high performance packet processing and was recently introduced in the Linux kernel. However, the flexibility of XDP is not as good as TCP, up-to-date drivers are required and applications need to be developed for XDP support. This is a topic for future work.

## 3.6 Conclusions and Future Work

OX is designed as a framework for programming storage controllers with application-specific FTLs (as introduced in Section 2.4.1). The three layers separate concerns between host interaction through a logical address space (upper layer), media management and organization of the physical address space (bottom layer), and translation between logical and physical address spaces (middle layer). A generic queue abstraction with its associated thread model is introduced to organize concurrent execution of the asynchronous data path.

A key question throughout this thesis is whether each layer should be considered a black box, or whether OX – as a whole – should be considered a white box. The former approach makes it possible to develop each layer independently based on well-defined interfaces. This is an efficient way to reduce the complexity of design and implementation. The latter approach enables cross-layer optimizations within OX, which might be necessary to avoid data copies, leverage hardware acceleration (e.g., a RDMA engine) and thus improve performance. We combine both approaches, by introducing well-defined interfaces across layers and considering how to organize cross-layer optimizations when necessary. How to organize cross-layer optimizations in a more systematic way is a topic for future work.

The experiment we described in Section 3.5 establishes a baseline for the performance we can expect from the ARMv8 System-on-a-Chip at the heart of the DFC platform. Regardless of how the DFC is accessed (PCIe or fabric) and regardless of the underlying media technology, we cannot expect more than 360K IOPS from the DFC platform if data is to be written/read from RAM, i,e., if any data copy is to take place on the DFC. This insight is reinforced by the experiment in Section 3.6 that shows that the mere task of transferring data back and forth with NVMe over fabric is CPU-bound when 7 cores are involved. These experiments illustrate how much attention the designers of computational storage should devote to the performance of RAM accesses from the storage controller, and how important it is to focus on using hardware acceleration to bypass the CPU when transferring data at high throughput and low latency.

In the next Chapter, we focus on OX as an open-channel SSD controller and develop a benchmark to study the performance characteristics of open-channel SSDs.

[2]XDP: https://www.kernel.org/doc/html/v4.18/networking/af_xdp.html

# 4 $\mu$FLIP-OC: The Open-Channel SSD Benchmark

Solid State Drives (SSDs) have replaced magnetic disks in data centers. Cloud providers now expect SSDs to provide predictably high performance, as well as high resource utilization, for their dynamic workloads. As traditional SSDs offer the same interface as magnetic hard drives to abstract a radically different physical storage space, however, resource utilization is suboptimal and performance is often unpredictable [45, 86]. An emerging option for fulfilling the requirements of cloud providers is based on open-channel SSDs, which expose media geometry and parallelism to the host [6]. As it is the host's responsibility to manage data placement and I/O scheduling, it becomes possible to avoid redundancies and exploit optimization opportunities in the storage stack. The question is then: how should a data system that relies on open-channel SSDs be designed? More precisely, the question is whether some I/O patterns should be favoured, while others should be avoided. This is the question studied in this chapter.

Recently, He et al. [36] discussed the "unwritten contract" of traditional SSDs, i.e., SSDs equipped with an embedded Flash Translation Layer, that provide the block device abstraction (initially defined for magnetic hard drives): a linear space of logical block addresses (LBAs) associated with read and write operations. According to He et al., systems implemented on top of SSDs should follow five rules: (i) request scale rule: submit large requests or many outstanding requests, (ii) locality rule: favour locality to minimise misses in the FTL mapping table, (iii) aligned sequentiality: write sequentially within a block, (iv) grouping by death time: group on the same blocks data that is updated or deleted together, and (v) uniform data lifetime: favour data structures where data are updated/deleted in batch. But do these five rules still apply on open-channel SSDs? And if not, then what rules do apply?

Before we can answer these questions, however, we need a tool to understand the performance characteristics of open-channel SSDs. Bouganim et al. defined the uFLIP benchmark in 2009, as a means of characterizing the performance of flash-based SSDs [9]. More specifically, the goal was to understand the impact of the FTL on the performance of simple I/O patterns. As it turned out, the benchmark showed that different SSDs behaved in different ways and that the complexity of the FTL

introduced significant performance variability. With open-channel SSDs, however, the FTL is out of the equation. Furthermore, while the simple I/O patterns defined in the uFLIP micro-benchmarks could possibly yield a performance model, the uFLIP benchmark assumes a block device abstraction which is not supported by open-channel SSDs. Note that existing papers focusing on SSD performance and error patterns, such as Meza et al. [61], Ouyang et al. [65] or Grupp et al. [29] all make similar assumptions. In Linux, LightNVM instead introduces the PPA interface, a new interface that relies on a hierarchical address space and vector data commands (each read or write command can target up to 64 addresses).

In this chapter, we redesign the μFLIP benchmark for the PPA interface. More specifically, we make the following contributions:

1. We design uFLIP-OC, a variant of the uFLIP benchmark, adapted to the characteristics of the PPA interface of open-channel SSDs.
2. We apply the uFLIP-OC benchmark on an open-channel SSD composed of the DFC equipped with the OX controller.
3. We revisit the five rules of He et al. and discuss the path towards a new performance contract for open-channel SSDs based on the uFLIP-OC benchmark.

## 4.1 OX as Open-Channel SSD Controller

In 2015, open-channel SSDs were not available. Our first effort was building a system on top of custom NAND DIMMs to act as open-channel SSD controller, while FTL design was a host responsibility. OX first generation arose to be the first open-source storage controller that exposes a programmable board as open-channel SSD. This section describes the first generation of OX controller, built for μFLIP-OC. We also present FOX, a host-based tool to submit I/O patterns on open-channel SSDs.

### 4.1.1 System Setup

The system is composed of (i) a host machine, (ii) a Dragon Fire Card, (iii) an FPGA board, and (iv) two custom MLC NAND modules. The components and connections are described below:

- **Host machine:** A single-socket machine with a 4-core Intel x86 processor is used for running FOX and the benchmark code. The machine is equipped with 32 GB of DRAM, a 16-lane PCIe 3.0 slot, and runs a Linux kernel v4.11 with LightNVM support.
- **Dragon Fire Card:** A DFC is connected to the host machine PCIe 3.0 slot. The DFC is equipped with an 8-core ARMv8 processor, 16 GB of DRAM, and two

4-lane PCIe 3.0 slots. The ARMv8 processor runs a Linux kernel v4.1.8 and OX Controller generation one.

- **FPGA board:** An FPGA board is connected to the DFC PCIe 3.0 slots. The board is equipped with an Intel Stratix V FPGA, and two DIMM slots compatible with DDR3 and custom NAND memories. The FPGA runs an NVMe controller, a DMA engine, and exposes the NAND interface to OX controller.
- **MLC NAND modules:** Two custom MLC NAND DIMM modules are connected to the FPGA board. Each module has 4 chips, each chip is organized in 4 PUs containing 1024 blocks each, the blocks are then organized in 512 pages with 4 sectors per page. A chip has a capacity of 64 GB, a DIMM module has 256 GB, and OX controls two modules with a total capacity of 512 GB of MLC NAND. The interface to access the NAND is proprietary, however, we simplified it to three main functions defined as 'write page', 'read page' and 'erase block'. These functions are available to OX controller via PPA addressing.

The described DFC setup is shown in figure 2.2, in chapter 2.

### 4.1.2 OX Design - First Generation

OX first generation implements the bottom, middle, and upper layers and exposes the NAND modules as open-channel SSD to the host machine. The layers are described below:

- **Bottom Layer:** A media manager called 'NAND FPGA' interacts directly with the NAND modules, it was developed and instantiated by the OX bottom layer. As described in chapter 3, media managers abstract the NAND as a set of channels and provide access to physical media via PPA addressing.
- **Middle Layer:** Open-channel SSDs do not need address translation. A minimalistic FTL providing support for vectored I/Os was developed. The function here is parsing large vectors of physical addresses into I/Os that match the required geometry boundaries. Then, tracking the completion of bounded I/Os and complete the full vectored I/O to the upper layer.
- **Upper Layer:** The upper layer is composed of a PCIe transport that links OX to the host benchmark, NVMe queues for command submission and completion, and the LightNVM command parser. The NVMe commands used are 'identify controller', 'create queue', 'destroy queue', 'erase block', 'physical write', and 'physical read'. The commands are explained in the Open-channel SSD specification v1.2.

## 4.2 The Benchmark

In the original µFLIP benchmark the addresses are defined in the logical block address (LBA) space exposed by SSDs with embedded FTLs. We propose µFLIP-OC, and revisit the µFLIP benchmark in the context of open-channel SSDs and its PPA interface. The requirements of µFLIP-OC are derived from the characteristics of open-channel SSDs:

- With open-channel SSDs, media characteristics are exposed to the host. We should explore their impact on performance.
- The PPA interface supports vector I/Os. We should compare the parallelism obtained with vector I/Os to the parallelism obtained with a number of concurrent outstanding requests.
- I/Os are partitioned in the PPA space across channels and PUs. We should explore the characteristics of **intra**-channel and **inter**-channel parallelism.

We define µFLIP-OC as a collection of four micro-benchmarks, organized in two thematic groups that focus on: (i) media characteristics and (ii) parallelism. Each micro-benchmark consists of a sequence of I/Os, at page granularity, on a given block. The blocks involved in a benchmark are erased prior to its execution. We do not consider random patterns, as they are not directly supported on open-channel SSDs. Table 4.1 summarizes the µFLIP-OC benchmark. Note that while the table defines a specific range of values that apply to the device under study, the micro-benchmarks can be adapted to any device geometry. The term LUN is equivalent to PU.

| Name | Parallelism | Pattern | Threads | Pages | **Definition** (":" = loop, "()" = grouping, ";" = order, and "\|" = choice) |
|---|---|---|---|---|---|
| $\mu OC_0$ | No | Sequential | 1 | 1 | Block=0−63: Page=0−511: ( WWWW \| WWWR \| WRWR \| WRRR \| RRRR ) |
| $\mu OC_1$ | No | Sequential | 1 | 1 | Loop: (Erase; Page=0−511: W; Page=0−511: R) |
| $\mu OC_2$ | Multi-LUN | Round-Robin | 1 | 1 | Block=0−63: Page=0−511: LUN=0−3: ( W \| R ) |
| | Threads | Round-Robin | 1, 2, 4 | 1 | Block=0−63: Page=0−511: LUN=$T_i$: ( W \| R ) − $T_i$ is the thread identifier |
| $\mu OC_3$ | Vector I/Os | Round-Robin | 1 | 1, 2, 4, 8 | Block=0−63: Page=0−511: Channel=0−7: ( W \| R ) − I/Os are issued as vectors |
| | Threads | Round-Robin | 1, 2, 4, 8 | 1 | Block=0−63: Page=0−511: Channel=$T_i$: ( W \| R ) − $T_i$ is the thread identifier |

TABLE 4.1: µFLIP-OC micro-benchmark overview. Unspecified geometry components can be selected arbitrarily.

### 4.2.1 Media Characteristics

There is significant heterogeneity in the various non-volatile memories that compose the storage chips at the heart of open-channel SSDs. We thus design two micro-benchmarks to identify these characteristics.

- **$\mu$OC 0 - Read/Write Performance of a single PU:** This first micro-benchmark is executed by a single thread, accessing a single PU. There is no form of parallelism. We focus on the latency and throughput of reads and writes at page granularity. We consider various mixes of reads and writes ranging from 100% read, to 100% writes with three intermediate mixes of reads and writes (25% reads/75% writes, 50% reads/50% writes, 75% reads/25% writes). Our goal is twofold. We aim at characterizing (i) latency variance on a PU with different mixes of read and write operations, and (ii) throughput variance across PUs on the SSD.
- **$\mu$OC 1 - Impact of Wear:** This micro-benchmark sacrifices a block to study the impact of wear on performance. It focuses on a single block, accessed by a thread that loops through cycles of erase, writes and reads on the entire block, until an erase fails and the block is definitely classified as a bad block. Our goal is to trace the evolution of erase, write and read latency as a function of erase cycles, as well as the number of failures of page reads/writes.

### 4.2.2 Parallelism

Parallelism is the essence of SSDs: storage chips are wired in parallel onto each channel, several channels are wired in parallel to the controller, and the controller is multi-threaded. We design two micro-benchmarks to characterize the impact of parallelism on performance.

- **$\mu$OC 2 - Intra-Channel Parallelism:** This micro-benchmark focuses on parallelism across PUs, within a channel. A single thread issues write I/Os at page granularity on a number of PUs within a channel in round-robin fashion. The number of PUs targeted, as well as the modality of the I/Os (read or write) are the factors in this experiment. The measurements focus on throughput.
- **$\mu$OC 3 - Inter-Channel Parallelism:** This micro-benchmark focuses on parallelism across channels. A number of threads issue I/Os at page granularity on a single PU per channel. There are three factors in this experiment: the number of submitting threads (from one to the number of channels), the number of PPAs targeted in each I/O (ranging from the number of PPA per page to 64 PPA addresses), and the modality of each I/O (read or write).

## 4.3 FOX: A Tool for Testing Open-Channel SSDs

We have defined a tool, called FOX, to submit the $\mu$FLIP-OC I/O patterns on open-channel SSDs. FOX is a user-space tool that relies on the liblightnvm library [59] to submit I/Os to open-channel SSDs via LightNVM. Liblightnvm relies on IOCTL calls for submitting commands. As a result, each I/O is synchronous. We

| (<PU>, <block>, <page>)<br>*nb*: number of blocks, *np*: number of pages | | | | |
|---|---|---|---|---|
| | **Column 0** | **Column 1** | **Column 2** | **Column 3** |
| **Row 0** | (0,0,0) | (1,0,0) | (2,0,0) | (3,0,0) |
| **Row 1** | (0,0,1) | (1,0,1) | (2,0,1) | (3,0,1) |
| **Row 2** | (0,0,2) | (1,0,2) | (2,0,2) | (3,0,2) |
| | | ... | | |
| **Row** $np$ | (0,1,0) | (1,1,0) | (2,1,0) | (3,1,0) |
| **Row** $np+1$ | (0,1,1) | (1,1,1) | (2,1,1) | (3,1,1) |
| **Row** $np+2$ | (0,1,2) | (1,1,2) | (2,1,2) | (3,1,2) |
| | | ... | | |
| **Row** $nb \cdot np$ | $(0, nb-1, np-1)$ | $(1, nb-1, np-1)$ | $(2, nb-1, np-1)$ | $(3, nb-1, np-1)$ |

TABLE 4.2: Rows and columns distribution across four PUs and *nb* blocks

thus rely on commands submitted by multiple threads to obtain concurrent outstanding I/Os. FOX is open-source and available to the community[1].

### 4.3.1 I/O Engines

The distribution of I/O commands in FOX is defined by I/O engines. A user may develop an engine with a custom I/O pattern following the definition of nodes, rows, and columns:

- **Node:** Thread assigned to submit I/Os to a set of PUs. A PU is never assigned to multiple nodes to avoid interference.
- **Row:** Flash pages that share the same page and block identifiers across the PUs assigned to the node. Each row is composed of $n$ pages, where $n$ is the number of PUs assigned to the node.
- **Column:** Page offset within a row. The column also refers to a PU identifier.

Table 4.2 shows a distribution of flash pages across four PUs. Table 4.3 shows the I/O sequence for sequential (S-<sequence>) and round-robin (R-<sequence>) engines.

We expected to materialize the four μFLIP-OC micro-benchmark using three I/O engines that are embedded in FOX. The engines are listed below:

- **Sequential:** Sequential pattern of a mix of reads and writes.

[1] Available in https://github.com/DFC-OpenSource/fox

| Number of blocks: **2**, Number of pages: **3** | | | | |
|---|---|---|---|---|
| | **Column 0** | **Column 1** | **Column 2** | **Column 3** |
| Block 0 | | | | |
| **Row 0** | S-01 ⁞ R-01 | S-07 ⁞ R-02 | S-13 ⁞ R-03 | S-19 ⁞ R-04 |
| **Row 1** | S-02 ⁞ R-09 | S-08 ⁞ R-10 | S-14 ⁞ R-11 | S-20 ⁞ R-12 |
| **Row 2** | S-03 ⁞ R-17 | S-09 ⁞ R-18 | S-15 ⁞ R-19 | S-21 ⁞ R-20 |
| Block 1 | | | | |
| **Row 3** | S-04 ⁞ R-05 | S-10 ⁞ R-06 | S-16 ⁞ R-07 | S-22 ⁞ R-08 |
| **Row 4** | S-05 ⁞ R-13 | S-11 ⁞ R-14 | S-17 ⁞ R-15 | S-23 ⁞ R-16 |
| **Row 5** | S-06 ⁞ R-21 | S-12 ⁞ R-22 | S-18 ⁞ R-23 | S-24 ⁞ R-24 |

TABLE 4.3: I/O sequence [S - Sequential, R - Round-robin]

- **Round-robin:** Round-robin pattern across PUs with a mix of reads and writes.
- **Isolation:** Dedicated PUs for 100% reads and 100% writes with round-robin pattern. With isolation, reads are never blocked by writes.

## 4.4 Experimentation

In this section, we present the results of the $\mu$FLIP-OC benchmark applied to the DFC equipped with OX first generation. We start by discussing the impact of media characteristics and then the impact of parallelism.

### 4.4.1 Media Characteristics

#### 4.4.1.1 $\mu$OC-0: Latency Variance

We apply $\mu$OC-0 and measure latency for various mixes of read and write operations. Recall that each I/O is executed at page granularity. A mix of 25%R and 75%W corresponds to a sequence of three writes followed by one read. In this micro-benchmark, a single thread submits I/Os to a single PU and a given channel. Based on the data sheets of the NAND chips, we expect that writes take between 1.6 and 3.0 msec while reads take approximately 150 $\mu$sec. The write characteristics are due to the nature of the MLC NAND chip, which stores two bits per cell, and the first ("low") bit must be written before the second ("high") bit. As the MLC chip exposes pairs of pages encoded on the same cells, the consequence is that (i) pairs of pages must be written together and (ii) the "low page" is written before the "high page".

So, we expect that write latency will oscillate between two values and that read latency will be stable and low. Existing work on open-channel SSDs [6] suggests that reads will be slowed down by writes.

Figure 4.1 presents the results of µOC-0 for the various mixes of read and write operations. In each experiment, the number of I/Os submitted is equal to 32768, which corresponds to writing or reading 64 blocks. For 100% reads, we observe stable latency but much higher than what could be expected from the data sheet. This suggests that the overhead associated with reads (essentially ECC check) is significant. For 100% writes, we observe three bands: (i) from 2.2 to 3.0 msec, (ii) from 1.0 to 1.3 msec and (iii) around 800 µsec. The first two bands correspond to what we expect for high and low pages. The third band corresponds to write latency below what is expected from the NAND chip. Our hypothesis is that some form of write-back is implemented within the DFC. Write performance for mixes of reads and writes confirm this hypothesis. Any mix of reads and writes reinforces this third band, which is faster than NAND. As soon as the ratio of reads is greater than 50%, then the latency of writes is stable below 1.0 msec while read latency increases dramatically. Reads are blocked by writes and the cost of NAND writes is reflected in the latency of reads. At 50% reads, read latency can reach above 4 msec. Such degradation in performance does not result from the hidden cost of writes alone. We observe here the result of read disturbances, where reads must be retried because of interference from writes.

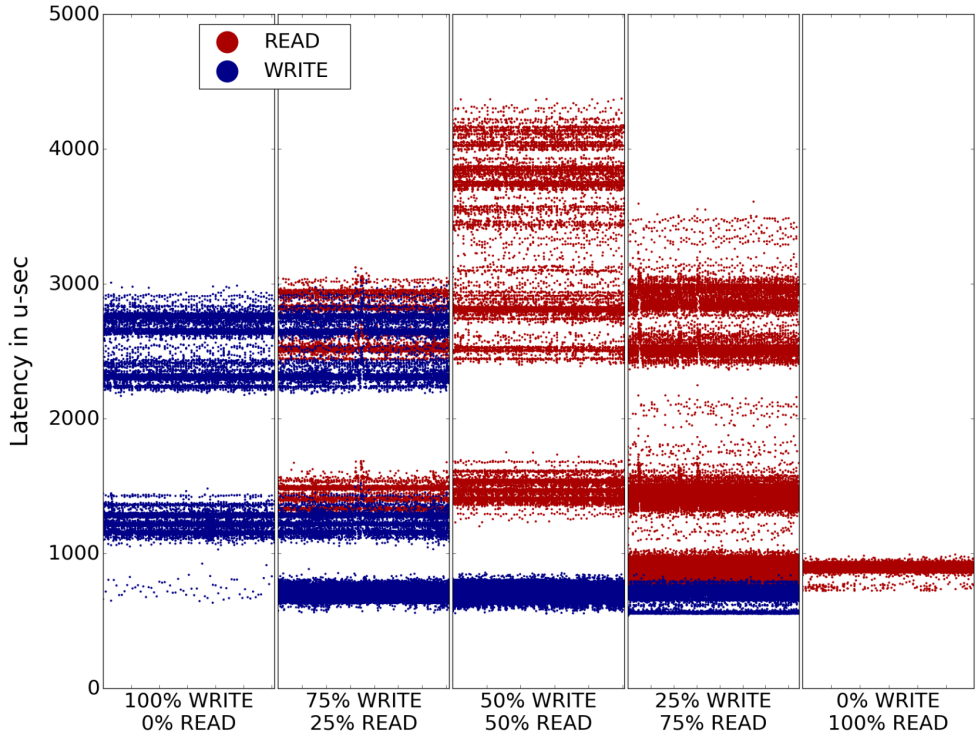



FIGURE 4.1: µOC-0: Impact of read/write mix on latency.

#### 4.4.1.2 $\mu$OC-0: Throughput Variance

We apply $\mu$OC-0 on every PU for all channels and measure throughput, FOX sequential engine was used. We focus on 100% read and 100% write workloads. Our goal is to visualize variance across PUs in the SSD. Figure 4.2 shows a heatmap to represent the result. We feared that performance would be uneven because we tend to experiment mostly with channel 0 and PU 0 on each channel. But the results show little variance across PUs. Throughput is stable at 16 MB/sec per PU for writes and 38 MB/sec per PU on reads. Note that this throughput is the result of sequential, synchronous I/Os on one PU at a time. There is no form of parallelism involved.

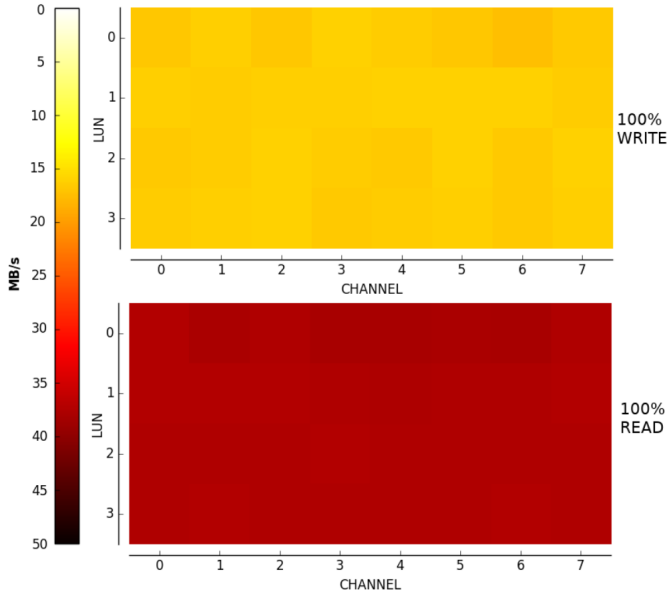



FIGURE 4.2: $\mu$OC-0: Heatmap of throughput for the entire SSD

#### 4.4.1.3 $\mu$OC-1: Wear

In order to measure the impact of wear (i.e., the number of erases performed on a block) on performance, we sacrifice a block and conduct $\mu$OC-1 . While we do not really know the state of the block we choose for this experiment, it is one of the less used blocks of the system. The NAND flash data sheet indicates a guarantee of 3,000 erase cycles per block. Based on Cai's outstanding study of NAND flash errors [10], we expect that the open-channel SSD will exhibit a low number of failures up to a point where the number of failures will increase steeply and negatively impact the performance of all operations.

Figure 4.3 shows the result of $\mu$OC-1 . We observe the first read failure only after 5,872 erases, or almost double the factory guarantee of the underlying NAND. This shows that ECC introduces high latency for reads but provides perfect error correction until wear reaches a given threshold. We remark that this tradeoff between read

latency (due to ECC) and read failure rate is a key characteristic of open-channel SSDs. On the other hand, the erase process must apply increasingly larger voltages to avoid failure.

The voltages are applied in a stepwise fashion, so the cost of erase operations increases regularly throughout the experiment. So, on the DFC equipped with the current generation of NAND chips, there is a correlation between erase latency and failure rate. This result suggests that it might be possible to assume that reads never fail until erase latency reaches a given threshold. This would have a major impact on the design of host-based FTLs or application-specific FTLs that today assume that I/Os might fail at page, block or die level and deploy considerable engineering resources to design failure handling mechanisms.

A surprising outcome of this experiment is that write latency remains constant and unaffected by wear. This is a worrying characteristic that can be linked to a write-back mechanism enabled in the DFC. Writes always complete fast, and failures are only identified on reads. Note, however, that this behavior makes sense under the assumption that reads never fail. Finally, note that we stopped the experiment after 17,000 erase cycles; at this point, the failure rate for reads had reached 10%.

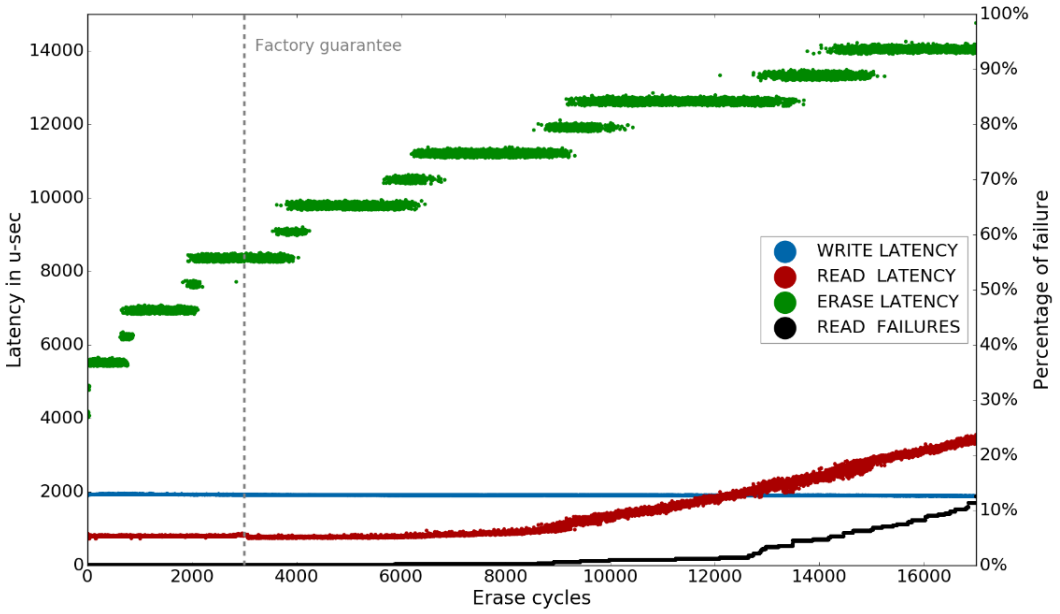



FIGURE 4.3: µOC-1: Impact of wear (erase cycles) on latency (left axis) and read failures (right axis)

| PUs | 100% Writes Multiple Threads | | 100% Writes Single Thread | | 100% Reads Multiple Threads | |
|---|---|---|---|---|---|---|
| | Throughp. (MB/s) | Scaling Factor | Throughp. (MB/s) | Scaling Factor | Throughp. (MB/s) | Scaling Factor |
| 1 | 16.81 | – | 16.81 | – | 37.09 | – |
| 2 | 27.22 | 1.62 | 23.47 | 1.40 | 49.77 | 1.34 |
| 4 | 37.85 | 2.25 | 33.45 | 1.99 | 49.73 | 1.34 |

TABLE 4.4: $\mu$OC-2: Impact of **intra**-channel parallelism on throughput.

### 4.4.2 Parallelism

#### 4.4.2.1 $\mu$OC-2: Intra-channel Parallelism

We first focus on parallelism within a channel with $\mu$OC-2. Write or read I/Os at page granularity are sent to a varying number of PUs within one channel in a round-robin manner, either using one thread or multiple threads. We expect that requests are executed in parallel on the different PUs and that throughput increases in proportion to the number of PUs until the channel becomes a bottleneck (i.e., writes are blocked until the channel is ready). Table 4.4 shows the throughput of $\mu$OC-2 when targeting 1, 2 and 4 PUs within a channel. Consider first write requests issued by multiple threads. Performance for 1 PU is the same as in $\mu$OC-0 at approximately 16 MB/sec. While throughput increases nearly linearly with the number of PUs, it does not increase by a factor corresponding to the number of PUs. We believe that this must be due to overhead on the DFC and OX controller. With only a single thread issuing synchronous writes, throughput is nearly the same, due to write-back on the DFC; control is given back to the thread very quickly, but when the thread returns to PU 0, it must wait for the completion of all previous writes.

For reads, the story is different, as synchronous reads must be completed before handing back control. With one thread (not shown) the throughput is not affected by the number of PUs considered. With multiple threads, throughput is increased when two threads issue read requests in parallel; as the maximal throughput per channel is 50MB/s, further threads do not increase throughput.

#### 4.4.2.2 $\mu$OC-3: Inter-channel Parallelism

We now turn to parallelism across channels with $\mu$OC-3. We first explore the impact of vector I/Os (a single command applied to up to 64 PPAs) and compare it to the impact of outstanding concurrent I/Os submitted by different threads. On the DFC, equipped with MLC NAND, page granularity corresponds to 8 PPAs (1 PPA per sector, 4 sectors per page and 2 pages per plane). We thus experiment with

vector I/Os applied to multiples of 8 PPAs (8, 16, 32 and 64). Each group of 8 PPAs corresponds to a page located on a separate channel, so our experiment targets 1, 2, 4 and 8 channels. We consider outstanding concurrent I/Os submitted by a thread dedicated to a given channel. We experiment with 1, 2, 4 and 8 threads so that potential **inter**-channel parallelism is the same for vector I/Os and concurrent I/Os. When the number of targeted channels is less than 8, the experiment targets each channel in turn in round-robin fashion.

Table 4.5 shows the write throughput for vectored and concurrent I/Os. First, we observe that even when a single channel is targeted at a time (8 PPAs or a single thread), throughput is more than 40 MB/sec, i.e., better than the throughput obtained with **intra**-channel parallelism. This is because targeting each channel in a round-robin fashion effectively hides a significant portion of the time spent writing on NAND. Less time is spent waiting for a PU to become available and as a result throughput is increased. As expected, both vector I/O and concurrent I/O take advantage of **inter**-channel parallelism. The throughput when targeting two channels, with 16 PPAs or two threads, is twice the throughput obtained with 1 channel, with 8 PPAs or 1 thread. When targeting four or eight channels throughput is increased up to 130 MB/sec, but not by a factor of two when doubling the number of channels targeted. The throughput obtained with vector I/O and concurrent I/Os is similar. With 32 threads, each targeting a PU (there are 8 channels and 4 PUs/channel on the DFC), we reach 300 MB/sec throughput for writes and 400 MB/sec throughput for reads. So, reaching maximum throughput for the device requires some level of concurrent I/Os, either due to asynchronous I/Os issued from the kernel (e.g. pblk) or multiple threads in user space via liblightnvm.

Figure 4.4 shows the latency obtained for various mixes of reads and writes issued with concurrent I/Os. More specifically, each thread issues either read or write on a separate PU. We observe that read latency remains low, stable and unaffected by writes. As suggested by previous work [6] separating reads and writes leads to minimal latency variance. An interesting effect is observed, however, with 100%W, where the writes have a much less predictable latency than when mixed with reads, while throughput is not affected.

| | Vectored I/O (Single Thread) | | | Parallel I/O (Multiple Threads) | | |
|---|---|---|---|---|---|---|
| Total I/O size | Pages | Throughput (MB/s) | Scaling Factor | Threads | Throughput (MB/s) | Scaling Factor |
| 32 KB | 1 | 44.37 | — | 1 | 45.86 | — |
| 64 KB | 2 | 80.35 | 1.81 | 2 | 94.37 | 2.06 |
| 128 KB | 4 | 117.84 | 2.66 | 4 | 119.90 | 2.61 |
| 256 KB | 8 | 128.59 | 2.90 | 8 | 127.23 | 2.77 |

TABLE 4.5: $\mu$OC-3: Impact of **inter**-channel parallelism on write throughput: Vector I/Os vs Multiple threads.

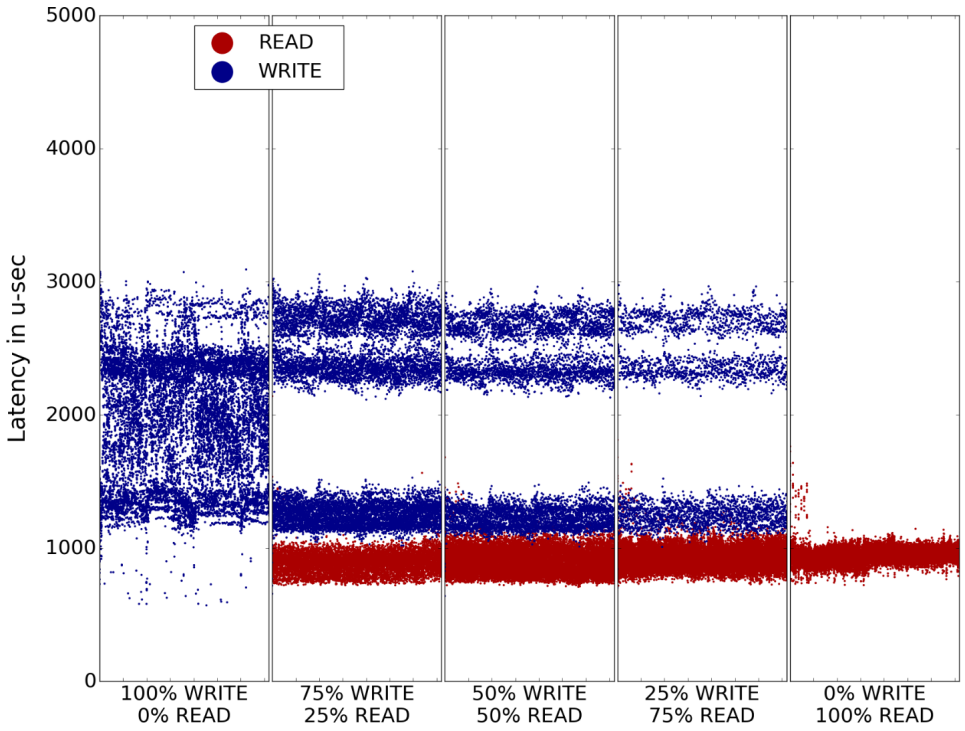



FIGURE 4.4: $\mu$OC-3: Impact of parallelism on latency for mixes of reads and writes.

### 4.4.3 Industry-grade Open-Channel SSD

In this section, we switch the DFC-based open-channel SSD prototype for an industry-grade open-channel SSD. Instead of a DFC equipped with OX, we now consider a Westlake SSD developed by CNEX Labs connected to an x86-based host via PCIe. We run $\mu$FLIP-OC again and compare the results. The host machine runs Linux with the LibLightNVM library (which we used to run our FOX application). The Westlake SSD is composed of 128 PUs spread across 16 channels, each PU contains 1,024 blocks composed of 512 pages of 32 KB in size. The total capacity is 2 TB. We compare latency and parallelism via $\mu$OC-0, $\mu$OC-2, and $\mu$OC-3. We start

with µOC-0 and measure the impact of mixed I/Os on latency. Figure 4.5 shows the CNEX system results, it compares to the results obtained with OX shown in Figure 4.1.

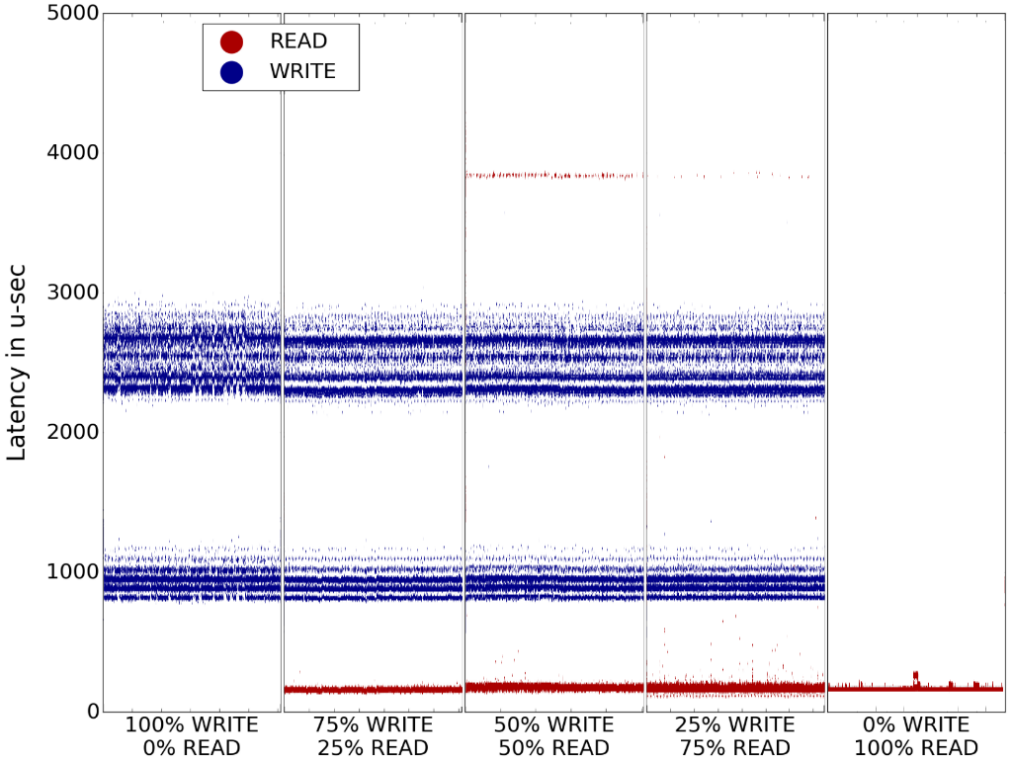



FIGURE 4.5: µOC-0: Latency on industry-grade OCSSD.

As expected, the industry-grade OCSSD delivers lower read latency compared to the FPGA system, in all cases the CNEX system delivered read latencies below 200 µs while the FPGA exhibits 800 µs for 100% reads. Differently than the FPGA, the Westlake SSD does not implement write-back mechanisms and write latency is not reflected on reads, however, a small group of reads in 50/50% and 25/75% are located at around 4 ms. We believe this is due to interference of reads and writes at the chip level. For writes, similar latency bands are seen between the two systems which is caused by similar NAND chips, in both systems MLC was used as media backend. At 50/50% and 25/75%, the write latency remains the same for CNEX while the FPGA delivers low write latency due to the write-back mechanism at the cost of unstable reads.

We compare **intra**-channel parallelism by applying µOC-2 to the CNEX system. The results at table 4.6 compare to the FPGA in table 4.4. In a single PU, the CNEX system is only 1 MB/s faster on writes. On reads the scenario differs, CNEX delivers 156.30 MB/s in a single PU while the FPGA delivered 37.09 MB/s, 4.21x of difference. As we scale to 4 PUs and 4 threads the **intra**-channel parallelism is evident. On multi-threaded writes, we see linear scalability of 3.99x at the CNEX while the FPGA scales 2.25x, however, for a single thread, CNEX does not scale at all while the FPGA scales 1.99x. This is due to the write-back mechanism implemented in the

| PUs | 100% Writes Multiple Threads | | 100% Writes Single Thread | | 100% Reads Multiple Threads | |
|---|---|---|---|---|---|---|
| | Throughp. (MB/s) | Scaling Factor | Throughp. (MB/s) | Scaling Factor | Throughp. (MB/s) | Scaling Factor |
| 1 | 17.81 | – | 17.81 | – | 156.30 | – |
| 4 | 71.10 | 3.99 | 17.78 | 0.99 | 302.65 | 1.93 |

TABLE 4.6: $\mu$OC-2: **Intra**-channel parallelism on industry-grade OC-SSD.

| Threads | 100% Writes | | 100% Reads | |
|---|---|---|---|---|
| | Throughp. (MB/s) | Scaling Factor | Throughp. (MB/s) | Scaling Factor |
| 1 | 17.81 | – | 156.30 | – |
| 4 | 73.84 | 4.14 | 627.39 | 4.01 |
| 16 | 301.16 | 16.94 | 2,368.59 | 15.15 |

TABLE 4.7: $\mu$OC-3: **Inter**-channel parallelism on industry-grade OC-SSD.

FPGA. On reads, both systems do not scale linearly, with 4 PUs and 4 threads the CNEX system scales 1.93x while the FPGA scales 1.34x.

For **inter**-channel parallelism we applied $\mu$OC-3 to the CNEX system. In this experiment, we scale the number of channels as well as the number of threads, we target 1 PU per channel. Table 4.7 shows that the CNEX system scales linearly for writes and almost linearly for reads. Differently, the FPGA does not scale linearly at table 4.5.

The Westlake SSD shows similar latency behaviors while it delivers superior performance. By observing the write latency bands exhibited by MLC NAND chips in both systems, we conclude that write latency is heavily linked to the type of media installed in open-channel SSDs. Read latency is stable if write-back mechanisms are not enabled. In mixed workloads, reads are severely affected by the write-back, thus we conclude that open-channel SSD manufacturers should avoid this type of mechanism in order to sustain stable read latency. We decided to move from the FPGA system to industry-grade open-channel SSDs, we used the CNEX open-channel SSD for the experiments conducted with the second and third generations of OX.

## 4.5 Conclusions and Future Work

The unwritten SSD contract and the five rules identified by He et al. [36] were defined for SSDs equipped with embedded FTL. Let us revisit how these five rules

apply to open-channel SSDs in light of the results of the µFLIP-OC benchmark applied on the DFC equipped with the OX controller.

- **Request scale rule:** Our results show that there is a tension between max throughput (that requires a queue of outstanding requests on each PU) and low latency variance (that requires separation of writes from reads). The request scale rule does not allow to cope with this trade-off. It is up to system designers to consider data placement and I/O scheduling strategies that strike an appropriate balance for generic or application-specific FTLs.
- **Locality rule:** Locality might lead to interferences between reads and writes on the same PU and thus high latency variance. This rule is thus not applicable.
- **Aligned sequentiality rule:** This rule still holds, and is indeed trivial to enforce with the PPA address space. Alignment within a block requires that writes start at page 0 in a given block.
- **Grouping by death time rule & Uniform data lifetime rule:** These rules focus on requirements for data placement; open-channel SSDs make it possible for the host to take such decisions without impediment. Our benchmark results, however, show another requirement on data placement: reads and writes should be isolated to preserve low latency variability.

Note that none of these rules account for media characteristics. Our results indicate that aggressive assumptions can be made about the absence of read failures in the upper layers of the system. More work is needed to generalize the results and identify whether a set of design rules, favoring specific I/O patterns, can be derived for open-channel SSDs in general. In particular, an open question is how different types of media, different generations of storage chips or even different ECC design decisions will impact the performance of an open-channel SSD.

Our work on OX started with a narrow focus on the performance of Open-Channel SSDs. We could have leveraged OX to study the design space of Open-Channel SSDs (e.g., should the write cache be based on the host or the SSD, what are the implications of implementing error correction on the host rather than the SSD, pros and cons of the nameless write abstraction). Instead, we chose to focus on the challenges of designing application-specific FTLs. In the next section, we present the modular FTL architecture that we designed for this purpose.

# 5 OX-App: Programming FTL Components

$\mu$FLIP-OC and OX show that misleading data placement decisions have a considerable impact on performance [71]. The complexity increases with heterogeneous NVM types, it forces developers to redesign the FTL for each type of memory, making media management a non-trivial task [73]. On the other hand, open-channel SSDs bring a common interface that allows the same FTL design to be reused, opening space for common components that run on top of any sort of NVM. In an open-channel system, FTL tasks can be separated in media-related (e.g. ECC, block metadata, data retention) and application-related (e.g. mapping table, garbage collection, write caching), media-related tasks remain in the storage controller while application-related ones are implemented on hosts.

At the datacenter scale, open-channel SSDs bring additional bottlenecks [24]. Tasks running on hosts such as garbage collection and media recovery waste network bandwidth and consume CPU cycles, hardware resources that could be spared if such tasks were offloaded to application-aware storage controllers. We argue that the FTL should be deconstructed into a set of tasks, and applications should decide whether a task runs on hosts (close to applications) or into storage controllers (close to data).

This chapter presents the second generation OX, equipped with OX-App, a framework that decomposes the FTL into a set of primitive components. Each component is to be an intrinsic part of media management. By using components, OX-App allows a modularized FTL to be built based on application-specific tasks. Applications may decide (i) whether a component is implemented or not, and (ii) where it is implemented, into a storage controller or at the host.

## 5.1 The FTL Components

Efforts on building FTLs [6, 32, 13] and FTL evaluations [15] bring examples of necessary internal tasks and how these tasks are interconnected – often via metadata that must be recovered after a failure or shutdown. In OX-App, we want to separate

| Primitive | Component Name | Function |
|---|---|---|
| P1-BAD | Bad Block Management | Tracks media bad blocks |
| P2-BLK | Block Metadata Management | Maintains media block metadata |
| P3-PRO | Block Provisioning | Delivers media addresses across PUs |
| P4-MPE | Persistent Mapping | Maintains metadata for mapping recovery |
| P5-MAP | In-memory Mapping | Manages the in-memory mapping table, that maps logical to physical addresses |
| P6-LOG | Log Management | Maintains a recovery log |
| P7-REC | Checkpoint-Recovery | Recovers the device from a checkpoint, after a clean shutdown or failure |
| P8-GC | Garbage Collection | Recycles media blocks and reclaims space |
| P9-WCA | Write-caching | Caches data and guarantees atomic writes |

TABLE 5.1: OX-App primitive components

FTL tasks and define primitives that can be programmed as components. Components should interact with each other via common calls that guarantee consistency and durability. Table 5.1 describes OX-App primitive components.

Figure 5.1 shows the dependencies and interactions of OX-App components while OX serves user requests. Dark yellow circles represent OX-App primitive components (refer to table 5.1).

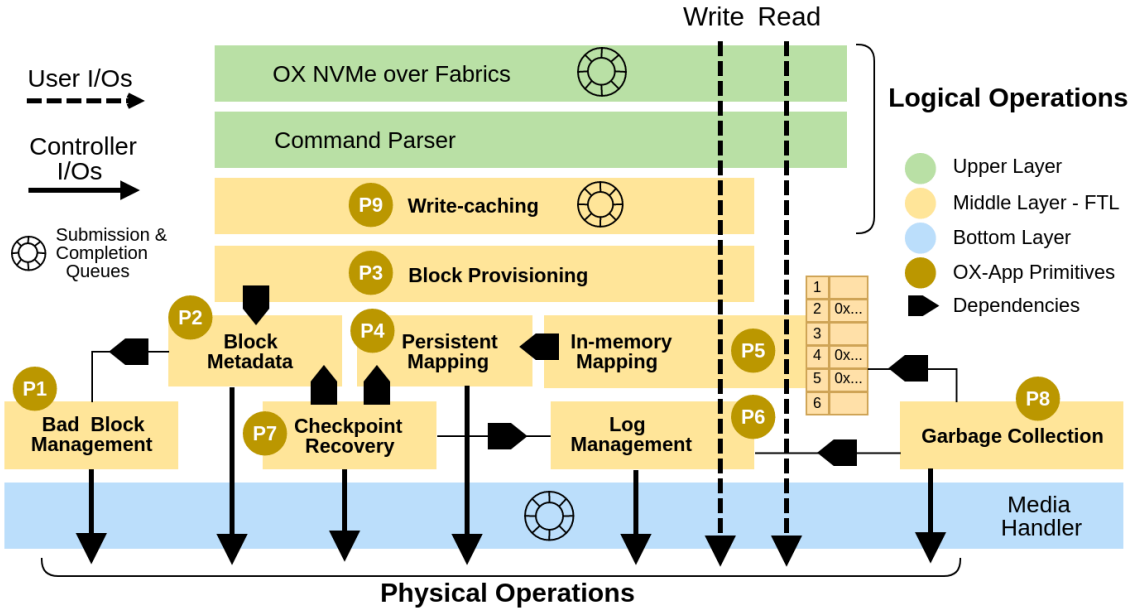



FIGURE 5.1: Interactions of OX-App components within OX Controller

Controller I/Os represented by solid lines, are executed synchronously and generated by several components: (i) garbage collection may read and write to media

as data get invalid, (ii) recovery log may be persisted according to atomic requirements, (iii) mapping and block metadata may be persisted during checkpoint process, (iv) mapping information may be read and persisted by caching mechanisms, and (v) bad block information may be updated at any time. User I/Os, represented by dashed lines and executed asynchronously, are queued in the upper and bottom layers. I/O queues on the upper layer must match queues on the host and preferably be executed in parallel by the controller processor cores. User writes are parsed in the upper layer and cached in P9, I/Os are completed by the cache if a write-back mechanism is enabled, otherwise, the I/O is completed after data is persisted. The write-cache must contain a queue and a single thread that calls P3 (get a physical address) and P6 (append a log), then, P5 (mapping table updating) is called on the submission or completion, depending on atomic constrains. If multiple threads are used for user writes, concurrency mechanisms must be implemented within the write path. User reads, after parsed by the upper layer, must cross only P5 (mapping table read) and the code must be as light as possible to achieve low latency.

Each OX-App component provides a function set as a programmable interface. A component shall follow the dependencies in figure 5.1. Regardless of the dependencies, a component should be coded independently and allow its entire code to be replaced without changing other components. The following subsections present OX-App components and its respective function sets.

### 5.1.1 Bad Block Management (P1-BAD)

A media block is considered bad if data is unrecoverable or if an erase command fails. If a bad block is found, it should be removed from the provisioning and retired from any further usage. NVM on all its sorts have a life cycle, for instance, the MLC NAND used in the experiments of section 4 has a life cycle of 3,000 erases per physical block. NAND chips often come with bad blocks caused by failures during the fabrication process, these blocks are marked by the manufacturer and should be identified by the FTL before the chip is available for usage. If bad block information is maintained by the chip, providing a bad block list to the FTL is a responsibility of OX bottom layer. By its nature, open-channel SSDs maintain bad block information, for that reason, FTL developers may choose if the component P1-BAD maintains metadata or not. Table 5.2 shows P1-BAD function set.

### 5.1.2 Block Metadata Management (P2-BLK)

An FTL must maintain metadata related to physical blocks. A block is defined as a range of physical addresses of a certain size that belongs to the same parallel unit. The sizes and boundaries of a block are defined by the geometry exposed by the OX bottom layer. Block metadata may contain, but not limited to, the life of the

| Function | Description |
|---|---|
| CREATE | Used for bad block identification and creation of metadata.<br>If metadata is maintained by the media, this function is not used. |
| GET | Returns the state of a media block identified by its address.<br>Blocks that are not in bad state must be added to provisioning. |
| LOAD | Reads bad block information from the media.<br>Media specific commands or an FTL-defined process may be used. |
| PERSIST | Used to persist bad block metadata that is cached in memory.<br>If metadata is not cached, this function is not used. |
| UPDATE | Updates a single block state.<br>"Bad" and "non-bad" values are required, other values may be defined. |

TABLE 5.2: OX-App P1-BAD function set

block as number of erases, the write address where data is currently being appended to, the block type, the block address, the amount of invalid data, and a bit vector representing the invalid data pages. The information described is used for processes such as wear-leveling and garbage collection. Table 5.3 shows P2-BLK function set.

### 5.1.3 Block Provisioning (P3-PRO)

Parallelism is a key for SSDs, data pages must be spread equally among parallel units. A P3-PRO component must be responsible for data placement by providing physical addresses in a certain order that guarantees full SSD parallelism. In flash memory, blocks must be written sequentially, which requires this component to follow certain media constraints. Provisioning of blocks is required at the FTL, which may be a list of available and free blocks ready for user writes. The provisioning component is responsible for block metadata updating, such as the current write address and the status of the block. A block status may be, but not limited to, free, open, or closed. When block metadata is updated, it must be marked as dirty, then, it can be persisted by a checkpoint process. An FTL may have a single or multiple provisioning lists depending on block type requirements. A block type may be, but not limited to, hot data block, cold data block, log block, or metadata block. Table 5.4 shows P3-PRO function set.

### 5.1.4 Persistent Mapping (P4-MPE)

Mapping logical to physical addresses is the essence of the FTL. Pairs of logical and physical addresses are placed sequentially to assemble a table that represents the entire address space. The size of this table depends upon the storage capacity and the granularity of each entry – a physical address may represent 4,096 bytes,

| Function | Description |
|---|---|
| CREATE | If metadata is not found, this function creates it.<br>If metadata is maintained by the media, this function is not used. |
| GET | Retrieves a structure containing metadata of a single block.<br>This function is to be used for reading and updating information. |
| INVALIDATE | Used to mark a data page as invalid if a bit vector is implemented.<br>In flash memory, data pages become invalid due to updates. |
| LOAD | Reads block information from the media.<br>Media specific commands or an FTL-defined process may be used. |
| MARK | If metadata is cached in memory, it marks the block metadata as dirty.<br>This avoids persisting metadata that has not been changed. |
| PERSIST | Used to persist block metadata that is cached in memory.<br>If metadata is not cached, this function is not used.<br>This function shall be called during checkpoint process. |

TABLE 5.3: OX-App P2-BLK function set

| Function | Description |
|---|---|
| INIT | Initialize the provisioning component at startup.<br>The component must be ready for GET calls after initialization. |
| EXIT | Safely closes the provisioning component before a shutdown.<br>In-flight writes should be completed and resources should be deallocated. |
| CLOSE | Closes a block after all its pages have been written.<br>Metadata must be updated and the block must be included in the proper lists. |
| PUT | Informs the provisioning that a block is no longer used and can be erased.<br>Garbage collection may call this function while recycling blocks. |
| GET | Returns a list of physical addresses to be written, following media constraints.<br>Information of in-flight writes must be kept. |
| FREE | Notifies that a write, previously requested by GET, is completed.<br>When FREE is called, information of in-flight writes must be updated. |
| CHECK | Checks if a set of blocks need to be garbage collected.<br>The set of blocks may be the entire device, or a single or multiple PUs. |

TABLE 5.4: OX-App P3-PRO function set

| Function | Description |
|---|---|
| CREATE | If the secondary table is not found, this function creates it.<br>If the FTL does not implement the secondary table, P4-MPE is not used. |
| GET | Returns the physical address of a given mapping table fragment index.<br>The entire secondary table should be in DRAM for higher performance. |
| PERSIST | Persists the secondary table, if it is dirty.<br>Depending on the secondary table size, it may be persisted in fragments.<br>This function shall be called during checkpoint process. |
| LOAD | Loads the entire secondary table to the controller main memory.<br>If the secondary table is fragmented, a third and tiny table may be used. |
| UPDATE | Updates the physical address of a secondary table entry.<br>The entry should become dirty and marked for persistence. |

TABLE 5.5: OX-App P4-MPE function set

for instance. The table size decreases in half if the first logical address is zero and the table is an array of physical address, then the index of the array represents the logical address. In order to achieve high performance on operations, the mapping table needs to be loaded to main memory within the storage controller, which is a concern if we consider a limited amount of DRAM available to controllers. The table may grow tens of gigabytes in size, we thus assume that loading the entire table at once is not viable, and that persisting it sequentially would degrade performance drastically. A reasonable solution is splitting the table in fragments that match the boundaries of a flash page. If we spread the fragments across parallel units, we can read and write mapping pages in parallel and independently. A secondary and smaller table is then needed, it stores the physical addresses which the larger table fragments are written to. The secondary table is smaller enough to be persisted sequentially and loaded entirely to main memory at once. P4-MPE component is responsible for the secondary table creation, persistence, and availability. Table 5.5 shows P4-MPE function set.

### 5.1.5 In-Memory Mapping (P5-MAP)

Loading the entire mapping table to the controller main memory is not viable. Independent mapping fragments that are aligned to the media page boundaries can be flexibly cached, which allows us to read and update mapping entries as fast as possible. P5-MAP component shall implement caching mechanisms that support loading and eviction of mapping pages. For loading a page, the read address is provided by P4-MPE GET function. For page eviction, a new write address is provided by P3-PRO GET function. Table 5.6 shows P5-MAP function set.

| Function | Description |
|---|---|
| INIT | Initializes the mapping table in-memory component.<br>After initialization, READ and UPSERT are ready for calls. |
| EXIT | Closes the component safely.<br>Cached pages must be evicted and persisted before it returns. |
| CLEAR | Forces persistence of all in-memory dirty pages.<br>This function may be called during checkpoint process. |
| READ | Returns the physical address of a given logical address.<br>If the logical address was never written, it returns zero. |
| UPSERT | Inserts or updates the physical address of a given logical address.<br>A lock or lock-free technique must be used to avoid concurrency. |

TABLE 5.6: OX-App P5-MAP function set

| Function | Description |
|---|---|
| INIT | Initializes the component.<br>After initialization, P6-LOG must be ready for APPEND calls. |
| EXIT | Closes the component safely.<br>All cached logs must be persisted before this function returns. |
| APPEND | Appends a log. If logs are stored in data pages, it may not be used.<br>Logs may be appended in main memory for higher performance. |
| PERSIST | Persists all logs that are stored in main memory.<br>To guarantee durability, logs must be persisted before completions. |
| TRUNCATE | Updates the head of the log, a physical address to the first entry.<br>This function is called after a checkpoint is completed. |

TABLE 5.7: OX-App P6-LOG function set

### 5.1.6 Log Management (P6-LOG)

The FTL must provide consistency and durability. In case of failure and unexpected shutdown, completed updates must be durable and recoverable, while uncompleted updates must be aborted and metadata must be consistent. Large logical writes generate multiple physical writes as part of a single command, this command must be atomic – either all updates are durable, or none are. Atomicity, consistency, and durability are properties of database transactions [85, 28] and we believe that such guarantees should be provided by any sort of FTL. To guarantee such properties, any change performed in metadata must be recorded in a persistent log, which can be truncated after the changes are persisted. FTL designers must rely on recovery requirements to decide whether log is stored within data pages as out-of-bound information or in dedicated media blocks. P6-LOG component shall guarantee a persistent log that is used for recovery. Table 5.7 shows P6-LOG function set.

| Function | Description |
|---|---|
| INIT | Initialize the component.<br>After initialization, GET, SET, RECOVER, and REPLAY are ready. |
| EXIT | Closes the component safely.<br>Any ongoing checkpoint must be completed before it returns. |
| CHECKPOINT | Creates a checkpoint for durability and log truncation.<br>The checkpoint must persist dirty metadata and be incremental.<br>The checkpoint data must be persisted to a known location. |
| GET | Returns a pointer to a keyed checkpoint data.<br>The checkpoint data is available in memory by the RECOVER call. |
| SET | Appends keyed data to be persisted within the next checkpoint.<br>New information is added to the checkpoint by key and value. |
| RECOVER | Loads the checkpoint data from storage to main memory.<br>After it returns, checkpoint data must be available via GET. |
| REPLAY | Restores the FTL to a consistent state during startup.<br>It may use the logs persisted by P6-LOG.<br>The log head should be available within the checkpoint. |

TABLE 5.8: OX-App P7-REC function set

### 5.1.7 Checkpoint-Recovery (P7-REC)

The FTL should recover to a consistent state after a failure or shutdown, however, recovery time is a concern. Applications should be allowed to request data from secondary storage as soon as possible, thus checkpoint techniques are required for fast recovery. The checkpoint guarantees durability of all updates occurred before it started by persisting dirty metadata and truncating the recovery log. During checkpoint, several seconds might pass while P2-BLK and P4-MPE persist its dirty metadata pages. Waiting for checkpoint completion is not viable, thus user updates should be allowed during the checkpoint process. All updates occurred during checkpoint are not durable, thus log entries should be persisted for those updates. Metadata pages persisted during checkpoint are also not durable and must also persist logs. Log entries created after checkpoint completion must be appended to the logs created during the process, forming a single linked list that is used for recovery. This technique is called incremental or fuzzy checkpoint [62]. If log entries are persisted at the out of bounds space of each data page, then recovery should implement its own mechanism to restore the FTL to a consistent state. P7-REC component is responsible for checkpoint and recovery implementation, it may use the logs created by P6-LOG. Table 5.8 shows P7-REC function set.

### 5.1.8 Garbage Collection (P8-GC)

Flash memory requires a block to be erased prior to writes. During updates, new data is written to a newly allocated block while pieces of data within the old block become invalid. As more and more updates arrive, the amount of invalid data increases, eventually, we run out of space. Inevitably, we need garbage collection. Whatever technique used in this component will face concurrency issues. We advise the writing of P8-GC data to different blocks rather than mixing with user data written by other components. This guarantees that concurrent writes to the same physical block are done in the correct sequence, it also guarantees media endurance by not moving cold data often.

Another concurrency problem is at the metadata update. When P8-GC moves data to a new address $new_{GC}$ it needs to update the mapping table physical entry $map_{phy}$ at P5-MAP. Meanwhile, a concurrent write thread at P9-WCA component may try to update $map_{phy}$ to a different address $new_{WCA}$. The issue is that $new_{GC}$ points to an older version $page_{v0}$ of the data, while $new_{WCA}$ points to a newer $page_{v1}$ version. Even if locking strategies are used, we cannot guarantee whether the final value of $map_{phy}$ will be $new_{GC}$ or $new_{WCA}$. Who has the priority? The mapping should always point to newer versions, thus we assume that user writes at P9-WCA have priority over P8-GC. In the case described, P9-WCA always replaces the mapping entry with $new_{WCA}$, while P8-GC uses a modified version of optimistic locking [52, 54] to validate the version of $map_{phy}$. The original optimistic locking is applied to readers, we also apply on $new_{GC}$ writes in addition to a standard fine-grained spinlock on mapping entries. The spinlock protects the entry during updating, and the optimistic lock guarantees the version priority. Prior of holding the spinlock, the GC reads $map_{phy}$ to $map_{old}$. After holding the spinlock it reads again and compares $map_{old}$ to $map_{phy}$, if the values are equal, then $map_{phy}$ is replaced by $new_{GC}$, otherwise, nothing is done. For flexibility, we advise the implementation of the lock and the optimistic lock at P5-MAP UPSERT function, then, GC is responsible only for reading and providing $map_{old}$ to the UPSERT function.

P8-GC component is responsible for garbage collection and supports custom designs, FTL developers may rely on their needs when architecting P8-GC. Table 5.9 shows P8-GC function set.

### 5.1.9 Write-Caching (P9-WCA)

P9-WCA is the FTL entry point for user writes. It controls all other components thus it is considered performance critical. The design of P9-WCA relies on data caching that will be persisted by mechanisms in P2-BLK, P3-PRO, P5-MAP, and P6-LOG. At the cache level, data may be parsed and processed according to the application needs, it is up to FTL designers to decide when other components are

| Function | Description |
|---|---|
| INIT | Initializes the component. A thread should be prepared for calling TARGET and RECYCLE whenever P3-PRO CHECK function identifies the need for garbage collection. |
| EXIT | Closes the component safely. Any ongoing garbage collection should be completed. |
| TARGET | Given a set of physical blocks, it targets blocks in need for GC. A block may be targeted by checking the amount of invalid data. |
| RECYCLE | Recycles a physical block. It moves valid data to new locations, updates metadata at P5-MAP and P2-BLK, and calls P3-PRO PUT function. |

TABLE 5.9: OX-App P8-GC function set

| Function | Description |
|---|---|
| INIT | Initializes the component. It creates the write cache and starts the threads responsible for processing writes. After this function returns, the FTL is ready for processing writes. |
| EXIT | Closes the component safely. No writes are allowed after this call. The write cache must be empty and all writes must be completed. |
| SUBMIT | Submits a logical write command to the FTL. It processes user data and enqueues commands for submission. Calling other components to ensure durability is responsibility of write threads. |
| CALLBACK | Threads from the bottom layer call this function for command completion. If long processes are implemented in the callback, this function should enqueue the completed command to a completion queue, and then return. A completion thread is responsible for completing the command to hosts. |

TABLE 5.10: OX-App P9-WCA function set

called. If the transaction mechanisms described in P6-LOG are used, then the mapping table at P5-MAP should be updated after data is persisted, in addition, a *commit* log should be appended at P6-LOG before host completion (completion does not depend upon persistence of *commit* logs, which potentially degrades performance). If multiple write threads are defined and metadata is updated concurrently, multiple provisioning lists at P3-PRO are required. Table 5.10 shows P9-WCA function set.

### 5.1.10 Built-in Functions

Multiversioning is a property of out-of-place updates in FTLs, it consumes physical space while the logical space remains the same. User writes must be allowed if logical space is available, thus P8-GC needs to be fast enough to guarantee the availability of physical free blocks at P3-PRO. At the same time, P8-GC should not affect

P9-WCA write performance – user writes have priority over garbage collection. We want to garbage collect enough blocks and guarantee free space, but P9-GC throughput should not hurt performance of user writes, unless, and only if physical free space is critical. We use a channel abstraction to guarantee stable throughput within the controller. A set of built-in functions provide write and read access to common structures containing the state of each channel. The FTL may use this information for channel interleaving and synchronization between P8-GC and P9-WCA. An instance of each structure is created per channel. The structures and its purposes are described below:

- **Channel Switch:** This structure tells if the channel is enabled for user writes. If enabled, user reads and writes are allowed. We do not recommend garbage collection in a channel that is currently enabled for user writes. If disabled, only user reads are allowed and P3-PRO GET should not return addresses of disabled channels. FTL designers may use the channel switch structure to balance the throughput of P9-WCA user writes. The channel may be disabled at P8-GC for garbage collection, or if free space is critical at P3-PRO CHECK function.
- **Channel Status:** An FTL may set the status of a channel to *need gc* if the free space is critical. If P8-GC is designed to garbage collect channels, it may check this structure to find channels that need garbage collection. *Channel status* and *channel switch* may work together to balance P8-GC and P9-WCA write throughput.
- **Channel Contexts:** This structure counts the number of ongoing write contexts in a channel, a write context is created when physical addresses are reserved by P3-PRO GET function. If addresses from multiple channels are reserved in a single P3-PRO GET call, then multiple contexts are created (one per channel included in the reservation). When a new context is created in a channel, *channel contexts* should be incremented by one. A context is completed when all reserved addresses have been written. When a context is completed, *channel contexts* should be decremented by one by calling P3-PRO FREE function. The *channel contexts* structure may work together with *channel switch*, a process may disable a channel and wait until *channel contexts* is zero before execution.

Table 5.11 shows the built-in functions. All functions get the channel identifier as parameter.

## 5.2 OX Design - Second and Third Generations

Based on OX-App design, we built the second generation of OX controller. From an open-channel SSD controller showed in Chapter 4, OX evolved into a controller

| Function | Description |
|---|---|
| SWITCH ENABLE | Enable a channel for user writes. |
| SWITCH DISABLE | Disable a channel for user writes. Only reads are allowed. |
| SWITCH READ | Returns the *channel switch* value. |
| STATUS SET | Sets the *channel status* value. |
| STATUS GET | Returns the *channel status* value. |
| CTXS ADD | Increments the *channel contexts*. |
| CTXS REMOVE | Decrements the *channel contexts*. |
| CTXS READ | Returns the *channel contexts* value. |

TABLE 5.11: OX-App built-in function for channel management

equipped with a framework for custom FTL design and development. The open-channel support implemented in the middle layer of OX was kept, and OX can still be used as open-channel controller. OX-App was introduced as an instance of the middle layer, allowing media channels to be managed by OX-App components. The new layer organization is described below.

- **Bottom Layer:** Industry-grade open-channel SSDs were finally available at the time OX was updated to version two. The 'NAND FPGA' media manager was replaced by an open-channel SSD manager, while the channel abstraction exposed to the middle layer remained the same. Industry-grade open-channel SSDs provide a higher performance compared to our previous FPGA setup.
- **Middle Layer:** Ox-App was implemented in the middle layer. OX second generation does not implement any FTL component, but the basis for a guided FTL development is available in OX-App. FTL instances were developed with OX-App for the third generation of OX controller, described in the next chapters.
- **Upper Layer:** Custom command parsers were introduced in the upper layer, which means any 64-byte command can be parsed in OX. Applications may define their own NVMe commands and implement their own command parser at OX upper layer. We also introduced NVMe over Fabrics [64] and OX can be configured for data transfer over the network. All experiments performed with OX second and third generations used NVMe over Fabrics as the standard transport.

After designing and developing OX-App, we architected FTL instances. We developed two FTLs, (i) OX-Block as a page-level and generic FTL that exposes a block interface for legacy file systems, and (ii) OX-ELEOS as application-specific FTL that exposes a log structuring store interface. In (ii), log-structuring functionality was implemented as OX-App components, allowing file systems and databases to

offload components to the storage controller. The third and last generation of OX emerged equipped with OX-Block and OX-ELEOS. Both FTLs are described in the next chapters, together with experimental evaluation.

## 5.3 Related Work

Efforts on building programmable SSDs include Willow [78] and KAML [38] at UCSD, and the KV-SSD at Samsung [76]. These efforts are based on computation on top of generic FTLs. To the best of our knowledge, our OX-App framework is the first attempt at defining a framework for computational storage based on application-specific FTLs.

Some efforts on FTL techniques for metadata and parallelism management include P-BMS [44], a bad block management scheme that maximizes parallelism on SSDs by delaying I/Os to identified bad blocks; Hydra [37], an SSD architecture that increases parallelism by considering intra and inter-channel parallelism at the provisioning; PGIS [31], an I/O scheduler that also leverages the intra-channel parallelism via hot data identification. In [11], Chang presents techniques for specializing a mapping table to translate application semantics directly into flash. We focus on generic components for our FTL, thus we are inspired by specializations of metadata and parallelism techniques, but our design choice is simplistic and robust.

For garbage collection, Prof. Jihong Kim and his team presented JIT-GC [34], a just-in-time GC technique that predicts future write demands and triggers background garbage collection only when necessary. This technique can be applied to avoid garbage collecting data that will probably be invalid in the near future.

In databases, Caetano Sauer and Goetz Graefe introduced instant restore [77], a technique that permits databases to proceed with transactions after a failure and before the transactions are completely recovered. Common designs, such as the generic recovery we implemented in our FTL, wait until all metadata have been recovered before allowing new transactions.

## 5.4 Conclusions and Future Work

In this chapter, we zoomed in on the middle layer and defined the FTL framework that we will instantiate in the coming chapters. Ideally, we can implement various types of FTL by specializing the modules we identified in this chapter. This is a hypothesis we will test with a generic FTL in the next chapter and an application-specific FTL in Chapter 7. We will also revisit the question of black-box vs. white box in the context of the middle layer. Can the interfaces defined above be respected when instantiating an FTL or do we need cross-modules optimizations?

# 6 OX-Block: A Page-Level FTL for Open-Channel SSDs

Before we focus on application-specific FTLs, we developed generic FTL components based on our hypothesis that *deconstructing the FTL leads to a modular architecture that can efficiently support the design of application-specific storage controller software*. We built OX-Block, a full-fledged FTL that implements all the nine OX-App components. Some components will likely remain unmodified in future application-specific FTLs while others will be replaced by application code. OX-Block maintains a 4KB-granularity mapping table and exposes open-channel SSDs as a block device compatible with file systems, we assume 4 KB as the minimum read granularity in open-channel SSDs. Durability of metadata and mapping information is entrusted by write-ahead logging (WAL) [46], checkpoint, and recovery mechanisms. For garbage collection, FTL components mark channels for collection, then, background threads recycle open-channel SSD blocks. The data path within OX-Block follows OX-App dependencies shown in figure 5.1.

## 6.1 Design and Implementation

This section describes OX-Block as a set of nine components, we explain each component in separated subsections. For a complete component definition and details about component interactions, refer to chapter 5.

### 6.1.1 P1-BAD

In open-channel SSDs, a bad block table is maintained either by the device or FTL. We maintain this information as metadata at the FTL level. Our design is simple, an array of bytes represents the whole set of chunks in a device where bytes store the current state of a chunk. We have the definition of planes for compatibility with older versions of open-channel SSDs. Flash chunks may be composed of several blocks that are internally written together at the NAND level, these blocks are addressed by planes within its chunk. Planes no longer exist in open-channel SSD v2 or newer. When planes are higher than one, a chunk state is defined by several

bytes, each containing the state of its internal plane blocks. Figure 6.1 represents the bad block table structure.

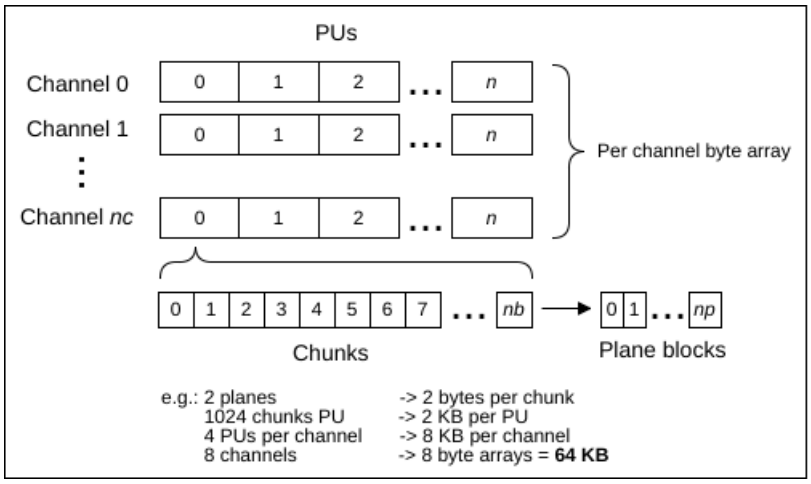



FIGURE 6.1: OX-Block bad block table structure

Following the channel abstraction, each channel maintains its own bad block table that is stored in a fixed chunk (e.g. PU 0, chunk 1). Figure 6.1 depicts an 8-channel device, each channel stores an 8 KB table containing 4096 chunks, each chunk has 2 plane blocks. If a page is 16 KB in size, a channel persists its whole bad block table in a single write. At first, OX-Block stores a version of the table at page 0, when the table is updated, a new version is written at the next page, when the chunk is full, the chunk is erased and page 0 is used again. It is recommended that the table is replicated to other blocks as backups to ensure durability in case of failures while erasing the chunk. OX-Block assumes that bad block table updates are always persisted and does not maintain a version in volatile memory.

### 6.1.2 P2-BLK

Components such as provisioning and garbage collection require per-chunk information such as the current write pointer, block type, and a bit vector for invalid sectors. Chunk information is loaded from storage and kept in volatile memory as an array, where updates take place. Each entry in the array follows this structure:

```
struct chunk_entry {
    uint16_t    flags;
    uin64_t     ppa;
    uint32_t    erase_count;
    uint16_t    current_page;
    uint16_t    invalid_sec;
    uint8_t     bit_vector[VECTOR_SIZE];
};
```

*ppa* represents the chunk address, *current_page* stores the next page to be written, *invalid_sec* is a summary of *bit_vector* that avoids scanning it every time we need the

number of invalid sectors in the chunk. *bit_vector* is a bit sequence where each bit represents a sector in the chunk, if the bit is set, the sector contains invalid data. A 16-bit field is reserved for *flags*:

```
enum chunk_flags {
    CHUNK_USED = (1 << 0),
    CHUNK_OPEN = (1 << 1),
    CHUNK_LINE = (1 << 2),
    CHUNK_AVLB = (1 << 3),
    CHUNK_COLD = (1 << 4),
    CHUNK_META = (1 << 5)
};
```

If *CHUNK_USED* is set, the chunk is full. *CHUNK_COLD* or *CHUNK_META* tells if the chunk contains cold data moved by garbage collection, or logging information, respectively. If *CHUNK_OPEN* is set, the chunk is partially written, *CHUNK_LINE* tells if the chunk is currently being written by the provisioning. If *CHUNK_AVLB* is set, the chunk is empty and ready to be written.

If a chunk has 4096 sectors, *VECTOR_SIZE* is set to 512 bytes – each byte represents 8 sectors. The total size of a chunk entry, following the structure *chunk_entry*, is 530 bytes. If a device contains 1024 chunks per PU, and 8 PUs per channel, then each channel maintains 4,240 KB of chunk metadata table ($chunk_L$). Large capacity devices may contain hundreds of channels which gives hundreds of megabytes to be maintained. To achieve high performance, all $chunk_L$ tables must be loaded to volatile memory where updates take place, eventually, the tables are persisted to storage. However, persisting hundreds of megabytes when only a few bytes are modified is not viable. $chunk_L$ tables are split into flash pages and spread across PUs. When information is updated, we mark the corresponding flash page as modified – a flash page contains a range of *chunk_entry* structures, if a structure is modified, the page is individually persisted.

We also keep a smaller table $chunk_S$ that contains physical addresses of $chunk_L$ pages, when a dirty $chunk_L$ page is persisted we update $chunk_S$ with the new address. If a flash page is 32 kilobytes in size, it accommodates 61 *chunk_entry* structures and requires 135 addresses (1,080 bytes) in $chunk_S$ per channel. $chunk_S$ tables are stored as part of the checkpoint, described at the M7-REC component. When the FTL starts, all tables can be recovered by reading the checkpoint entry that contains $chunk_S$. The most important processes are described below:

- **Startup:** Each channel loads its $chunk_S$ from a checkpoint entry, it contains physical addresses to pages of $chunk_L$. $chunk_S$ is scanned and all flash pages are loaded into volatile memory. At the end, the entire $chunk_L$ is built and chunk information can be retrieved/updated in main memory.
- **Chunk update:** When a chunk is updated, the flash page corresponding to the chunk is marked as dirty by setting a bit in $chunk_S$.

- **Checkpoint and shutdown:** During checkpoint, dirty pages are persisted to new physical locations, $chunk_S$ is then updated with new addresses. When all dirty pages are persisted, a copy of $chunk_S$ is set to checkpoint. In case of shutdown, a checkpoint is performed and the shutdown proceeds.

### 6.1.3 P3-PRO

Our provisioning strategy is based on the experiments of chapter 4, that shows the impact of intra and inter-channel parallelism on open-channel SSDs. We designed the provisioning in two levels. First, at the channel-level, an algorithm selects chunks to be part of an abstract definition of a *line*, chunks in the *line* are marked as *CHUNK_OPEN* and *CHUNK_LINE* on its metadata. Second, at global-level, an algorithm selects active channels and requests chunk addresses that are to be written. The global provisioning requests the same amount of addresses for each selected channel, then, each channel provisioning returns addresses from the current *line* chunks. Figure 6.2 shows the global provisioning and figure 6.3 shows the channel provisioning.

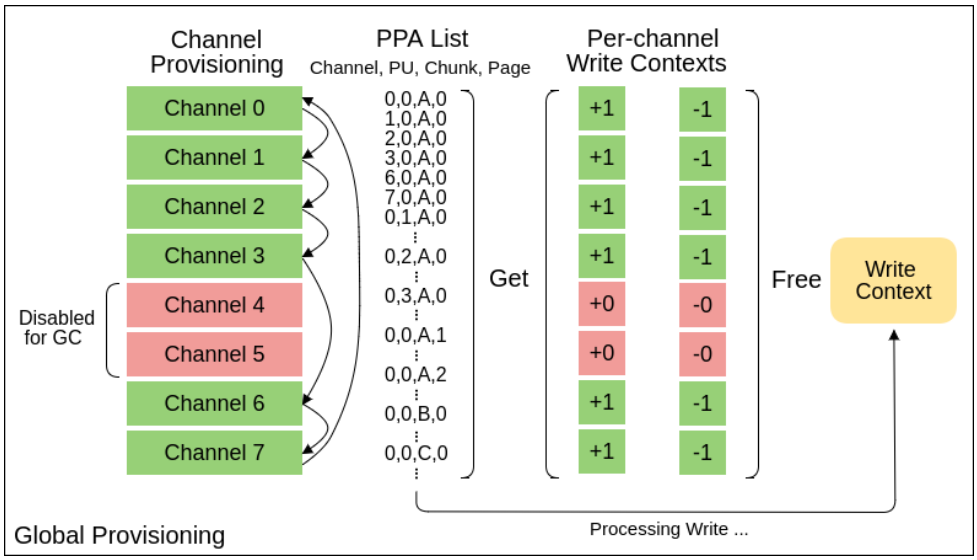



FIGURE 6.2: OX-Block Global Provisioning

FTL components request physical addresses where data is written. To achieve maximum parallelism, a single flash page per channel is reserved at a time, then, channels are selected in a round-robin fashion until the amount of requested addresses have been reserved. In figure 6.2, at the left, green channels are selected using round-robin while red channels are disabled for garbage collection – a component may disable channels using the built-in functions of section 5.1.10. In the middle, the function P3-PRO GET reserves a list of addresses and increments *channel contexts* of each channel. At the right, the function P3-PRO FREE decrements *channel contexts* when data has been successfully written to storage.

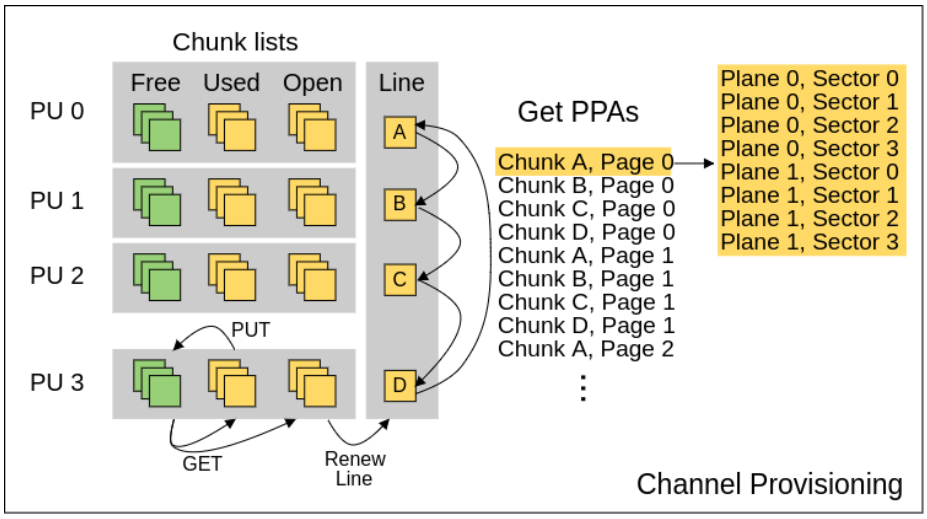



FIGURE 6.3: OX-Block Channel Provisioning

In figure 6.3 at the left, three list types are shown, free, used and open. During startup, these lists are created per PU at the channel provisioning. Free lists contain empty chunks, used lists contain chunks that are fully written with valid data, and open lists contain chunks that are partially written. During startup or when chunks in the *line* are full, the *line* is renewed by open chunks, a chunk per PU is selected to create a fully intra-channel parallelized *line*. Chunks that no longer contain valid data – recycled by garbage collection – are put back to free lists. When free chunks are requested and added to open lists, an erased command is issued to the chunk. Bad blocks are excluded from provisioning and not added to any list by checking the bad block table. At the right of the figure, a set of physical addresses spread among the *line* are returned to the global provisioning. At the PPA address, we kept the plane abstraction for compatibility with older open-channel SSD versions.

The provisioning is also responsible for marking channels for garbage collection. Every time a *line* is renewed, the channel is checked using the following:

```
if (free_chunks < GC_MIN_FREE_CHUNKS) {
    channel_switch_disabled (channel);
}

if ((float) 1 - (free_chunks / total_chunks) > GC_THRESHOLD) {
    channel_status_set (channel, NEED_GC);
}
```

If the number of free chunks in a channel is lower than GC_MIN_FREE_CHUNKS, the channel is disabled and writes are no longer allowed. A few chunks remain free, allowing garbage collection to move data and recycle chunks in the future, if no free chunks were left, then the channel would be out of provisioning. If the percentage of used chunks is higher than GC_THRESHOLD, the *channel status* is marked for garbage collection.

Writes are generated by several FTL processes such as user writes, garbage collection, checkpoint, and logging. Some of these processes require independent chunks where data is not mixed with other processes. We defined three types of provisioning, that concurrently serve address requests. Each provisioning has its own *line* and returns addresses of chunks that are not part of other provisionings. For instance, chunks of a log provisioning will contain only logs while chunks of a cold provisioning will contain only data moved by garbage collection. We defined a general, log, and cold provisionings. General chunks store hot data written by users, mapping table pages, and chunk metadata. Log chunks store write-ahead log entries only, and cold chunks store data moved by garbage collection that were previously located in general chunks. By using several provisionings we avoid concurrency between write threads that could lead to wrong write sequence within an open-channel SSD chunk.

The last responsibility taken by our provisioning is to guarantee that chunks recycled by garbage collection are not erased before a checkpoint is completed. In case of failure and further recovery, some write transactions might be aborted and older data should be still available. If recycled chunks are erased, we might recover the FTL to a state where the mapping table points to an erased block, which is undesirable. To fix this issue, our provisioning keeps a list of chunks in a temporary stage between used and free, when a checkpoint is completed, all chunks in that state are safely added to free lists.

### 6.1.4 P4-MPE

For a better presentation, a summary of variables utilized in this section is provided in table 6.1

| Variable | Value | Description |
|---|---|---|
| $es$ | 8 bytes | Mapping entry size |
| $ep$ | 4,096 | Mapping entries per flash page |
| $map_L$ | - | Large mapping table |
| $map_S$ | - | Small mapping table |
| $map_T$ | - | Tiny mapping table (checkpoint) |
| $map_{Le}$ | 536,870,912 | Large mapping table entries |
| $map_{Se}$ | 131,072 | Small mapping table entries |
| $map_{Te}$ | 32 | Tiny mapping table entries |
| $map_{Ls}$ | 4 GB | Large mapping table size |
| $map_{Ss}$ | 1 MB | Small mapping table size |
| $map_{Ts}$ | 256 bytes | Tiny mapping table size |

TABLE 6.1: OX-Block Mapping Variables

Durability of mapping information is the main role of a persistent mapping component. OX-Block maps logical blocks of $2^{12}$ bytes (4 KB), if we look at a $2^{41}$ bytes (2 TB) device then the mapping table $map_L$ would contain $2^{41\text{-}12}$ entries ($map_{Le}$). The mapping table entry size ($es$) is $2^3$ bytes (8 bytes), which give us a mapping table size ($map_{Ls}$) of $2^{32}$ bytes (4 GB):

$$\begin{aligned} es &= 2^3 \\ map_{\text{Le}} &= 2^{41-12} \Rightarrow 2^{29} \\ map_{\text{Ls}} &= map_{\text{Le}} \cdot es \Rightarrow 2^{29} \cdot 2^3 \Rightarrow 2^{29+3} \Rightarrow 2^{32} \end{aligned} \tag{6.1}$$

As proposed in section 5.1.4, the **large** 4-GB $map_L$ should be fragmented into flash pages and cached by M5-MAP. We follow this proposal in OX-Block and create a **small** table $map_S$ that contains the physical addresses of $map_L$ pages. If a flash page is $2^{15}$ bytes (32 KB) in size, it accomodates $2^{15\text{-}3}$ $map_L$ entries per page ($ep$), thus $map_S$ contains $map_{Le}$ divided by $2^{15\text{-}3}$ entries ($map_{Se}$). $map_S$ entries are also $es$ bytes longer which give us a $map_S$ table size ($map_{Ss}$) of $2^{20}$ bytes (1 MB):

$$\begin{aligned} ep &= 2^{15-3} \Rightarrow 2^{12} \\ map_{\text{Se}} &= \frac{map_{\text{Le}}}{ep} \Rightarrow \frac{2^{29}}{2^{12}} \Rightarrow 2^{29-12} \Rightarrow 2^{17} \\ map_{\text{Ss}} &= map_{\text{Se}} \cdot es \Rightarrow 2^{17} \cdot 2^3 \Rightarrow 2^{17+3} \Rightarrow 2^{20} \end{aligned} \tag{6.2}$$

Even for larger devices, the size of $map_S$ will not grow more than a few megabytes, which is a viable size to maintain $map_S$ entirely in volatile memory. When $map_L$ entries are accessed, $map_S$ is read and the corresponding flash page is loaded into cache. Eventually, $map_L$ pages that were modified need to be persisted to new locations, which causes an update in $map_S$, now we have a dirty $map_S$ table that needs to be persisted during checkpoint. Even for a few megabytes, neither persisting $map_S$ sequentially in a chunk nor persisting parts that are not modified are viable. Thus, we need a **tiny** $map_T$ table. The process is the same we used for $map_L$, which give us a $map_T$ table size ($map_{Ts}$) of $2^8$ bytes (256 bytes):

$$map_{\text{Ts}} = \left(\frac{map_{\text{Se}}}{ep}\right) \cdot es \Rightarrow \left(\frac{2^{17}}{2^{12}}\right) \cdot 2^3 \Rightarrow 2^{17-12} \cdot 2^3 \Rightarrow 2^{5+3} \Rightarrow 2^8 \tag{6.3}$$

$map_T$ is small enough to be persisted in a single flash page, but we persist $map_T$ as part of a checkpoint. During startup and recovery, we load $map_T$ from the last checkpoint and rebuild $map_S$ in volatile memory. Accesses to $map_L$ require a single check in $map_S$ which is entirely cached. A bit located at $map_S$ entries shows whether the $map_L$ page is cached in M5-MAP or not. If the page is cached, the $map_S$ entry contains a cache pointer, if not cached, then it contains a physical page address and the page needs to be loaded from open-channel SSD to the cache. Figure 6.4 depicts the mapping table levels. Figuratively, blue squares are mapping table entries, white

stars are addresses to flash pages that contains several higher level entries.

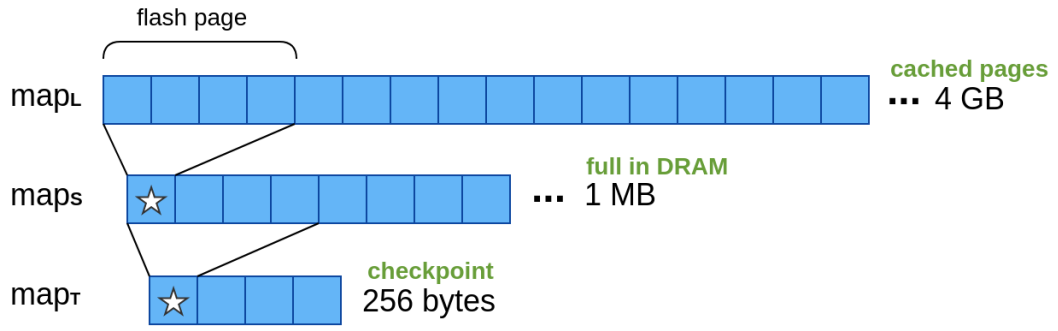



FIGURE 6.4: OX-Block Persistent Mapping Table Levels

Making $map_L$ durable requires (i) persisting modified $map_L$ pages that are cached, (ii) persisting modifed $map_S$ pages – $map_S$ pages are modified when $map_L$ pages are persisted, and (iii) sending a copy of $map_T$ to the checkpoint. When the checkpoint is completed, $map_L$ is durable. Persisting pages requires appending a log entry at P6-LOG, then a recovery process can redo changes at the mapping levels.

### 6.1.5 P5-MAP

While P4-MPE component is responsible for mapping table durability, P5-MAP manages the in-memory mapping, a cache component that loads and evicts flash pages from DRAM. Each flash page contains a range of mapping entries, used by user threads that may read and update its values. The cache component is not aware of durability, but responsible for reading and updating $map_S$ during page loading and eviction. If $map_S$ is not updated properly, durability cannot be guaranteed. Multiple threads are allowed to access the cache, however, if a page is being loaded or evicted, the working thread holds a lock to guarantee consistency. It also avoids the same page being loaded twice by concurrent threads. Figure 6.5 shows the cache mechanisms.

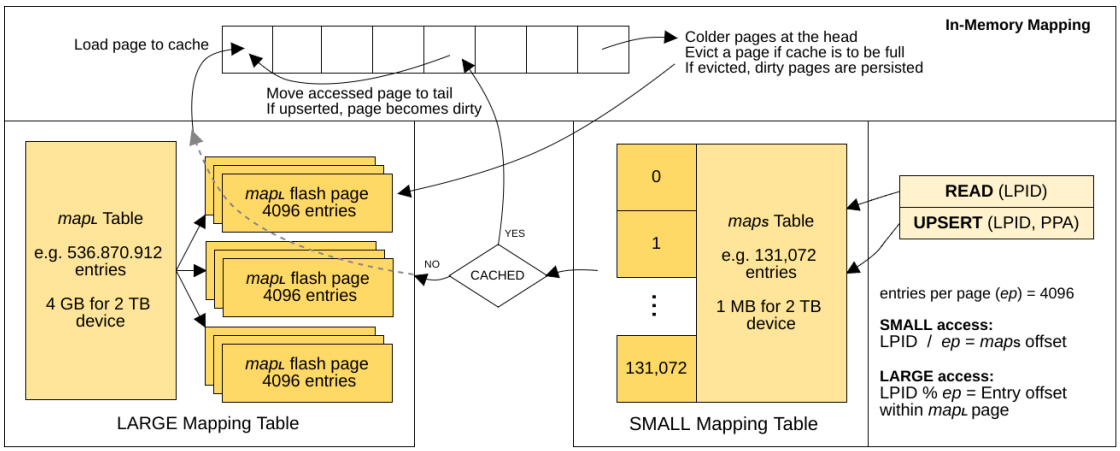



FIGURE 6.5: OX-Block In-Memory Mapping

At the right, read and upsert requests access $map_S$ at first. *LPID* is the logical address that starts at zero to the end of the total storage capacity, each address contains 4,096 bytes of data. Accessing $map_S$ requires a simple division of *LPID* by 4,096 (*ep*). Once we have the $map_S$ entry, it either points to a cached page or to an open-channel SSD location. If the page is not in cache, a read is issued and the page is loaded. The cache is structured as a tail queue where loaded pages are added to the tail and pages are evicted from the head, to simplify the structure we create a queue per channel and load pages from channels to their respective cache. When a page is already cached, after its access, we move it to the tail, it allows us to always evict the coldest pages from the head while hottest pages are located at the tail. Upserted pages are always marked as dirty to be persisted during eviction. During upsert, we check if the thread is performing a user write or moving data due to garbage collection (GC). If GC is identified, we make a comparison. We compare the current mapping value to a GC address (address that GC is moving data from), if the comparison fails, then a concurrent user write has modified the value, we ignore the GC upsert and return – user writes have priority. Refer to section 5.1.8 for details about mapping concurrency and our implementation of optimistic locking strategy.

During checkpoint, the cache component will be requested to persist all cached pages that were modified since last checkpoint, it is important that pages are persisted but not evicted, eviction of the entire cache is done only during shutdown. Also, persisted pages require appending a log entry at P6-LOG to ensure consistency during recovery. A last concern is after a startup, the cache is empty and every first access to a page requires a read from storage, we do not warm up the cache by design, smarter techniques of access prediction are needed for further improvements after startup.

### 6.1.6 P6-LOG

A recovery log is necessary to ensure durability of metadata information. Even if a clean shutdown is performed, some mapping entries and chunk metadata may not contain the latest values, this is due to updates during an incremental checkpoint at the shutdown. After startup, we recover the metadata to a consistent state by reading the recovery log and applying updates that were recorded after the checkpoint. If the shutdown was caused by failure, then the recovery log contains all changes performed after the latest durable checkpoint. All changes in metadata including in-memory mapping, persistent mapping, and chunk metadata must be recorded in a sequential log list, that is traversed during recovery. Each log entry is 64-byte in size and follows the structure in table 6.2.

The log component is built around a circular buffer that maintains an append (*AP*) pointer and a flush (*FP*) pointer. The buffer is empty when *AP* is equal to *FP*, if incrementing *AP* makes it to point *FP* then the buffer is full and needs to be flushed

<table>
<tr><th>Byte</th><th>Description</th></tr>
<tr><td>7:0</td><td>Timestamp: 64-bit integer that represents the time when the log entry was appended to the list.</td></tr>
<tr><td>8</td><td>Event: The type of the log. There are 12 log types:
<table>
<tr><td>Padding</td><td>Padding data. Used when a flash page is partially written by valid data, the rest is padding.</td></tr>
<tr><td>Chain Pointer</td><td>Not a log, but contains pointers to the next flash pages in the log chain. Used by recovery when traversing the chain.</td></tr>
<tr><td>User Write</td><td>User data, written by the user thread.</td></tr>
<tr><td>Mapping Large</td><td>Mapping entries in $map_L$, written by eviction or checkpoint.</td></tr>
<tr><td>Mapping Small</td><td>Mapping entries in $map_S$, written by checkpoint.</td></tr>
<tr><td>GC Write</td><td>User data, moved by garbage collection.</td></tr>
<tr><td>GC Mapping</td><td>Mapping entries in $map_L$, moved by garbage collection.</td></tr>
<tr><td>Ammendment</td><td>A write error has ocurred. A chunk must be closed.</td></tr>
<tr><td>Commit</td><td>Commits a set of writes, for atomicity.</td></tr>
<tr><td>Chunk Metadata</td><td>Chunk metadata, written by checkpoint.</td></tr>
<tr><td>Abort Write</td><td>Last write in a chunk was not performed, but the write pointer was incremented.</td></tr>
<tr><td>Recycle Chunk</td><td>A chunk was recycled by garbage collection.</td></tr>
</table></td></tr>
<tr><td>15:9</td><td>Reserved</td></tr>
<tr><td>63:16</td><td>Log Data: For writes, it contains information such as logical address, old physical address, and new physical address. For other types, it contains information such as log chain pointers, transactions, and chunk identifiers.</td></tr>
</table>

TABLE 6.2: OX-Block Log Entry Structure

before incrementing *AP*. If the buffer is full, no appends are allowed until *FP* is incremented. Our FTL design avoids this situation by flushing the buffer before it gets full. Appends are still allowed while the buffer is flushing, which avoids freezing the entire FTL. P6-LOG APPEND function allows multiple logs to be appended at the same time if space is available in the circular buffer. P6-LOG PERSIST function flushes all logs up to *AP*.

For better performance, we wait until enough logs are buffered and fill an entire flash page (a set of 512 logs are written in a 32 kilobyte flash page – 1 log for the chain pointer and 511 for FTL events). If a write fails, the previous log page would contain a wrong chain pointer, and the log chain would be broken. We fix this issue by storing the addresses of the next three pages at the chain pointer, if the write fails more than three times, we force a checkpoint and the log is truncated. The head of the log is stored as part of the checkpoint as a physical address. Figure 6.6 shows the circular buffer and the log chain stored on flash pages.

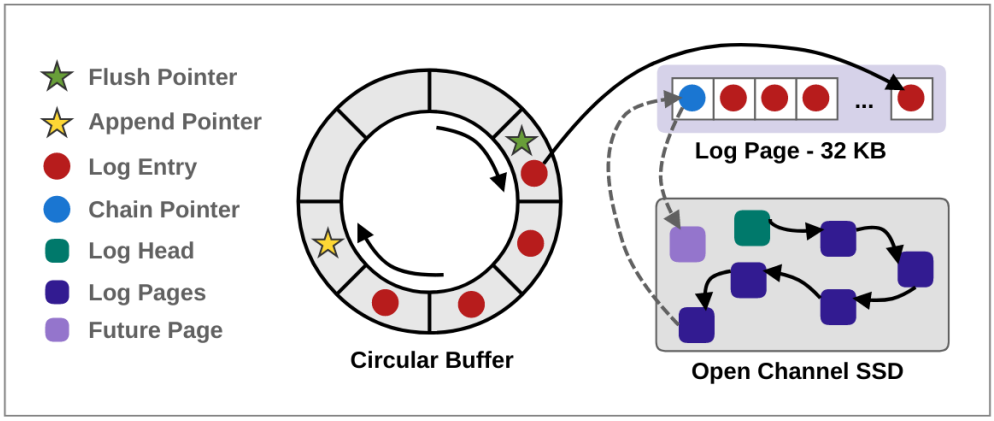



FIGURE 6.6: OX-Block Circular Log Buffer and Log Chain

A chain pointer is added at the beginning of the flash page, then log entries located at *FP* are added sequentially until the page is full. When the page is persisted, *FP* is moved to the next position after the last flushed log, or at the same position as *AP* if the buffer is empty. Dotted lines represent links to future pages, that are not physically persisted yet, but the addresses are given to previous pages. Logs are not mixed with other metadata or user data, instead, they are written to exclusive chunks selected by the metadata provisioning, we call them *log chunks*. Log chunks cannot be garbage collected, otherwise, the log chain would be broken. When a checkpoint is completed it is safe truncating the log, meaning that log chunks created before the checkpoint can be safely erased and reused to store other information.

| Key | Entry Size | FTL Component | Description |
|---|---|---|---|
| CHUNK | 17,280 bytes | P2-BLK | Chunk metadata $chunk_S$ tables. |
| MAPPING | 256 bytes | P4-MPE | Mapping metadata $map_T$ table. |
| LOG HEAD | 8 bytes | P6-LOG | Physical address to the first recovery log page. |
| TIMESTMP | 8 bytes | P7-REC | Checkpoint timestamp. Used to check if chunks and logs were created or modified after the latest checkpoint. |
| PROV | 32 bytes | P3-PRO | Current write channel. Keep the round-robin state after recovery. |

TABLE 6.3: OX-Block Checkpoint Entries

### 6.1.7 P7-REC

This component is divided in two parts, checkpoint and recovery from log. The checkpoint provides durability of metadata while recovery from log uses the checkpoint as the starting point in the process of rebuilding metadata in volatile memory.

#### 6.1.7.1 Checkpoint

Checkpoint information is designed to be as short as possible, depending on the device capacity a checkpoint is stored in a single or in a few flash pages. The structure of a checkpoint is variable, FTL components are allowed to include entries into the checkpoint by using the P7-REC SET function. Checkpoint entries are appended to a list and identified by a key, if the key already exists then the entry is updated. The entries stay in volatile memory until the next checkpoint is triggered, at the end of the checkpoint process the list is persisted. During recovery, the checkpoint list is loaded into volatile memory and entries become available via P7-REC GET function. To retrieve a checkpoint entry, FTL components are required to use the same key used for insertion. Table 6.3 lists the chekpoint entries inserted by FTL components, we assume a 2 TB device with 16 channels as example.

A checkpoint of 17,584 bytes that fits in a single flash page is all we need to ensure durability of metadata and start a recovery process. The checkpoint page must be stored in a fixed chunk that is read when the FTL starts. We chose a chunk in channel 0, PU 0 to store the checkpoint, we then replicate the checkpoint to other PUs as backups. The first checkpoint is written to page zero and subsequent checkpoints are written sequentially within the chunk. A magic byte is added to the out-of-bounds area of each checkpoint page and allows the recovery to identify the latest

checkpoint. If the checkpoint chunk gets full, an erase is issued and the checkpoint is written again to page zero. The checkpoint process is described step-by-step below:

- **Interval:** A custom interval $cp_i$ in seconds is defined. Checkpoint is triggered by a specific thread every $cp_i$ seconds.
- **Timestamp:** A timestamp is captured. The checkpoint process ensures that metadata is durable up to this timestamp, updates occurred after the timestamp and before the checkpoint is completed are not durable and will be recovered by a log redo process.
- **Log Head:** P6-LOG provides the current log tail address (address of the next log page to be written), this address is inserted into the checkpoint as the log head. During recovery, log redo starts from the log head to the end of the chain.
- **$map_L$ Cache:** P5-MAP component is required to persist all modified versions of $map_L$ pages that are cached. $map_S$ stores cache pointers to $map_L$ pages and contains dirty bits that represent if pages were modified. $map_S$ tables are scanned once and dirty pages are persisted, but not evicted from the cache. Pages that are modified after the dirty bit check are not durable and can be recovered by log redo.
- **Durable $map_S$:** After pesisting the $map_L$ cache, entries at $map_S$ are modified and need to be persisted. P4-MPE is required to make an immutable copy of all $map_S$ tables before the writes, it avoids modifications in pages that are being written and ensure metadata consistency. After creating an immutable copy of $map_S$, we can safely scan $map_T$ for dirty bits and persist $map_S$ pages that were modified. After this process, the immutable tables are deleted from memory. Finally, P4-MPE inserts a copy of $map_T$ into the checkpoint, this copy contains the latest version of $map_S$.
- **Durable $chunk_L$:** The same process is done for $chunk_L$. $chunk_S$ tables are checked for dirty bits and $chunk_L$ pages are persisted. Then, a copy of $chunk_S$ is inserted into the checkpoint, this copy contains the latest version of $chunk_L$.
- **Provisioning Channel:** P3-PRO provides the identifier of the next channel, the provisioning continues the round-robin selection normally after recovery, instead of starting from channel identifier zero again.
- **Durable Checkpoint:** Finally, the checkpoint list is persisted at the fixed chunk. We scan for the latest written page in the chunk by checking the magic byte stored in the out-of-bounds area of each page. When the magic byte is not present, the page is empty and the checkpoint is written.

- **Log Truncation:** The checkpoint is persisted, we can safely discard all log pages in the chain before the log head, the log head was stored in the checkpoint. We discard pages by allowing garbage collection to recycling only chunks that were written before the checkpoint timestamp. We identify when the chunk was written by storing a timestamp in the chunk metadata every time a write is issued to it. This process guarantees that the log chain is never broken by GC and old log chunks are marked as invalid for collection.
- **Collected Chunks:** In case of failure, recovery may need to restore metadata that points to chunks recycled by GC. To avoid inconsistency, chunks collected by GC should not be erased until a checkpoint is completed, instead, the provisioning keeps a list of chunks in a state between used and free. When the checkpoint is completed, P3-PRO is allowed to free these chunks and reallocate them for new writes.

#### 6.1.7.2 Recovery from Log

The FTL is ready for requests when the log redo process is completed. Log redo is performed only once, during the startup. After channels are registered within OX-Block and all components are started, log redo is called. At this point, the checkpoint is already loaded and the log head is available. Log pages are loaded sequentially to volatile memory following the log chain and starting at the log head. Log entries within pages are applied sequentially until the entire chain is processed. At the end of the process, the mapping and chunk information are recovered to its latest version. Depending to the checkpoint interval and the amount of FTL events, the log chain may grow to millions of logs but even very large chains are applied only in a few seconds. Log redo is a replay of an FTL event. For each event, a sequence of actions must be performed. The FTL events are listed in table 6.2 and the actions are detailed below.

- **Padding:** The log entry is ignored.
- **Chain Pointer:** Next flash page in the chain. If no chain pointer is found, the chain has been completely processed.
- **User Write:** It updates mapping and chunk metadata, this log contains the *old* and *new* physical addresses. First, it checks the chunk state:
  - **Chunk is not open and *current_page* is zero:** It updates *current_page* to one, sets *bit_vector* values to zero, sets *invalid_sec* to zero, set the flags CHUNK_USED and CHUNK_OPEN (opens the chunk), and increments *erase_count*.

  - **Chunk is open and *current_page* is the last page in the chunk:** It unsets the flags CHUNK_LINE and CHUNK_OPEN (closes the chunk), and increments *current_page*.
  - **Chunk is open and *current_page* is not the last page:** If *new* page is higher or equal than *current_page*, increment *current_page*.

  For all states above, we mark the respective page at $chunk_S$ as dirty. If transactions are enabled, a commit log should be found before applying changes in the mapping table, it guarantees atomicity. The log contains the transaction identifier that is repeated in all logs that belong to the same transaction, we maintain a list of transactions in memory containing the buffered logs. When a commit log is found, for each log in the transaction we update $map_L$ and invalidate the *old* address at P2-BLK (set a bit in *bit_vector*).

- **Mapping Large ($map_L$):** A $map_L$ page was persisted. It follows the same as *User Write* but transactions do not apply, $map_S$ updating and invalidation of *old* are done when the log is processed. Note that $map_S$ is updated, instead of $map_L$.
- **Mapping Small ($map_S$):** A $map_S$ page was persisted. It follows the same as *Mapping Large* but does not update any mapping information. There is no need to update $map_T$ as it is used only for checkpoint, and not for accessing $map_L$.
- **GC Write:** User data was moved by garbage collection. It follows the same as *User Write*, transactions apply, if enabled. If a GC write concurrency was found – refer to Section 5.1.8 – then $map_L$ is **not** updated and *old* is **not** invalidated.
- **GC Mapping:** A $map_L$ page was moved by garbage collection. It follows the same as *Mapping Large*, but transactions apply, if enabled. GC write concurrency may happen at $map_S$ as well, then the update is discarded and *old* is **not** invalidated.
- **Ammendment:** A write error occurred and a flash page was not written. We close the chunk at P3-PRO and invalidate all pages starting from the failed page to the last page of the chunk.
- **Commit:** A transaction is committed (set of writes). If transactions are enabled, mapping information is not updated when logs are processed but when a commit is found. Logs are organized by transaction identifiers, commit logs contain the identifier to be committed. If at the end of the log chain logs are still buffered, and the respective commits were not found, we discard the logs and assume that those transactions are aborted.
- **Chunk Metadata:** A $chunk_L$ page was persisted. It follows the same as *User Write* but transactions do not apply, $chunk_S$ updating and invalidation of *old*

are done when the log is processed. Note that $chunk_S$ is updated, instead of $chunk_L$.

- **Abort Write:** A write to a *page* was aborted. If the chunk is open and *page* is not the last page in the chunk, we update *current_page* value to *page*. If *page* is the last page in the chunk, the chunk is already closed and does not need a fix.
- **Recycle Chunk:** A chunk was recycled by garbage collection. P3-PRO PUT function is called.

To finalize the recovery, a checkpoint is performed and the FTL is ready to accept requests.

### 6.1.8 P8-GC

Garbage collection guarantees free space by recycling chunks and moving valid data to new locations. Our design is based on a channel abstraction where channels being collected are disabled for writes. The number of channels being collected is customized by *gc_slots*. By updating the number of *gc_slots* we balance the write throughput between user and GC. If the device is getting full, *gc_slots* should be increased, and decreased as chunks are recycled. Our GC starts with a single thread that checks for *channel status*, when a channel needs GC a *gc_slot* is assigned to the channel, each channel is collected by a separated thread. Figure 6.7 is the GC flowchart.

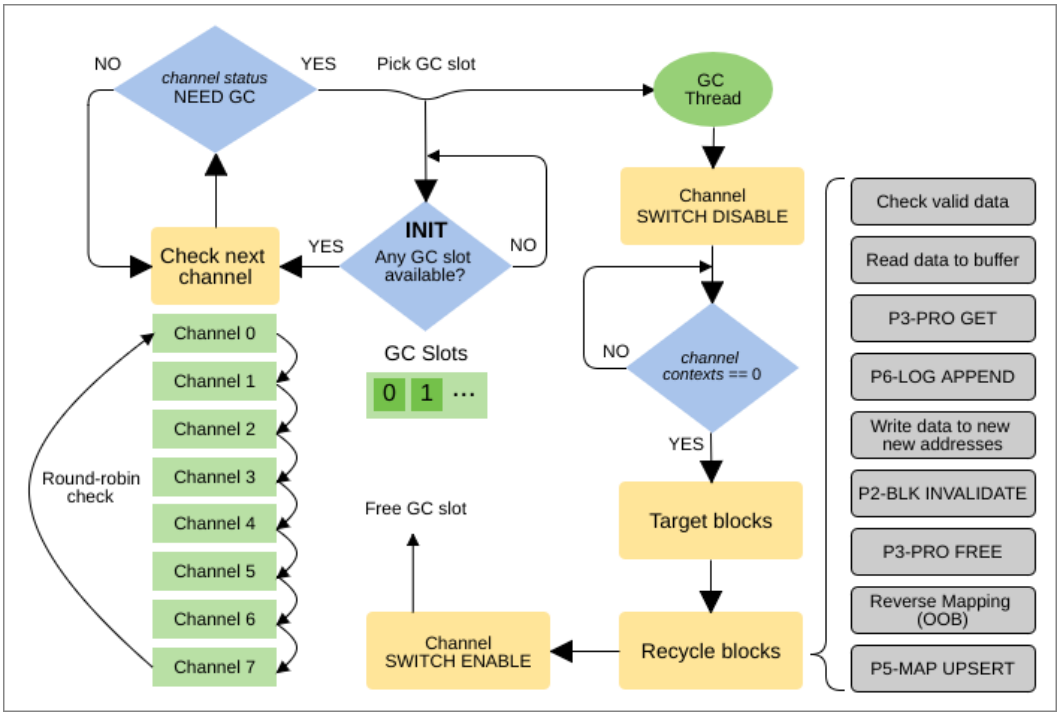



FIGURE 6.7: OX-Block Garbage Collection Flowchart

At the left, the main thread checks *channel status* in a round-robin fashion, if a channel has a *need gc* status, a *gc_slot* is assigned to the channel and the GC thread starts the process. After assigning the *gc_slot*, the main thread checks whether *gc_slots* are available for other channels, this process is repeated during the entire FTL runtime.

The GC thread initiates by calling the built-in function *SWITCH DISABLE* and waits until *channel contexts* is decreased to zero. Chunks are targeted for collection when all write contexts have been completed. To target a chunk, we check the *invalid_sec* value at the chunk metadata, if the amount of invalid data is higher than the minimum invalid rate (*mir*) then the chunk is targeted. A target rate (*tr*) defines the maximum value of *mir*, thus all chunks containing a percentage of invalid data equals or higher than *tr* are targeted for collection. To calculate *mir* in a channel, we define a few variables:

- **Chunks per channel (*cc*):** 8,192 chunks are equivalent to 8 PUs.
- **Sectors per chunk (*sc*):** 4,096 sectors.
- **Used chunks (*uc*):** Chunks that contains the flag *CHUNK_USED*. As example, 6,144 used chunks represent that 75% of the channel is used.
- **GC Threshold (*th*):** Minimum percentage of *uc* compared to *cc* required to triggering garbage collection. We set *th* as 70%, represented by the value 0.7.
- **Minimum Invalid Rate (*mir*):** Minimum amount of invalid sectors a chunk must contain to be targeted for garbage collection. *mir* is variable according to runtime calculations.
- **Target Rate (*tr*):** Percentage of invalid sectors in a chunk. Chunks containing invalid data equal or higher than *tr* are always targeted for garbage collection. *tr* is the maximum value of *mir* and is constant.
- **Warning Rate (*wr*):** Warns the GC algorithm about channel usage, this variable is used to calibrate the GC aggressiveness.

In this example we set *tr* to 90%. We always collect chunks which *invalid_sec* is higher than *tr*, thus we start setting *mir* by multiplying *sc* by *tr*:

$$mir = sc \cdot 0.9 \Rightarrow 4,096 \cdot 0.9 \Rightarrow 3,686 \tag{6.4}$$

We then measure the channel usage by calculating the warning rate (*wr*), *wr* is the percentage of used chunks beyond *th* – the GC threshold. For instance, for *th* 0.7, *wr* is a number between 0 and 0.3. *wr* is calculated as follows:

$$wr = \frac{uc}{cc} - th \Rightarrow \frac{6,144}{8,192} - 0.7 \Rightarrow 0.75 - 0.7 \Rightarrow 0.05 \tag{6.5}$$

Now we finally calculate the real *mir* using *wr*:

$$mir = mir \cdot \left(1 - \frac{wr}{1-th}\right) \Rightarrow 3,686 \cdot \left(1 - \frac{0.05}{1-0.7}\right) \Rightarrow 3,686 \cdot 0.8333 \Rightarrow 3,072 \quad (6.6)$$

The example above will target all chunks that contain 3,072 invalid sectors or more. The goal of this method is collecting chunks more agressively when channels are getting full, but avoid moving valid data if the channel still has space left. If *wr* is zero, chunks with at least 90% of invalid data are collected (channel still has 30% of free space). If *wr* is 0.3, the channel is full and all chunks with at least one invalid sector are collected. An aggressive GC is required if the channel is full, however, large amounts of data will be moved.

After calculating *mir*, a bucket sort algorithm sorts the targeted chunks by its *invalid_sec*, so chunks that require less data movement are recycled before. Chunks are recycled sequentially following the grey steps at the right of figure 6.7. *bit_vector* is used to identify valid data to be read into buffers, after preparing the buffers we request physical addresses to the provisioning and append a proper GC log to the log chain. After valid data is moved to new locations, the old addresses are invalidated at the chunk metadata and the write context is decremented at the provisioning. The last step is updating $map_{<L,S>}$ depending of the data type, the logical address is found at the out-of-bounds space of each page in a process called reverse mapping. When garbage collection is completed, we call the built-in function *SWITCH ENABLE*, user writes are allowed and the *gc_slot* is freed.

For safety and to avoid the situation where channels are full, we limit the visible namespace by setting an over provisioning of 30%. For instance, a device capacity of 2,048 GB would be exposed as 1,434 GB to the user. Figure 6.8 shows the device capacity.

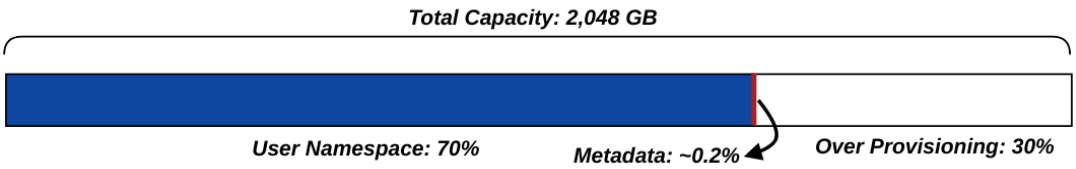



FIGURE 6.8: OX-Block Device Capacity and Namespace

The metadata for the example above considers a log chain created by 60 seconds of writes at 1 GB/s rate, mapping table, and chunk metadata. Note that the namespace size is 70%, which is the same value as *th*.

### 6.1.9 P9-WCA

Writing data consistently requires a complex synchronism among FTL components, the write cache is responsible for abstracting this complexity. User writes

involve the components P2-BLK, P3-PRO, P5-MAP, and P6-LOG, while user reads involve only P5-MAP. Reads do not change the state of metadata and are completed as fast as possible, providing lower latencies. Differently, writes must guarantee that changes on metadata are consistent and durable, in addition to this overhead, writes have higher latencies compared to reads. We do not implement a write-back mechanism due to the write cache being located in volatile memory, if the programmable board is equipped with persistent memory or batteries then write-back strategies are possible. Write-back consists on completing the write command to hosts after copying the data to the write cache but before persisted it to flash. It avoids media latencies being reflected to users.

User writes vary in size. We cache writes of variable logical sizes and parse them into 4 kilobyte sectors that are added to a single queue (*wq*). A single writing thread (*wth*) dequeues sectors from *wq* and groups them into flash aligned pages – a 32 kilobyte flash page contains 8 sectors. We then write aligned pages in parallel according to the P3-PRO round-robin strategy. But before writing the aligned page, a sequence of calls to FTL components must be respected. First, we request physical addresses from the provisioning, addresses for several pages are requested at the same call to save CPU cycles during provisioning. Next step is appending *User Write* logs at P6-LOG. As a last step, we store the logical addresses at the out-of-bounds area of each flash page. This is useful for reverse mapping performed by the GC. *wth* submits the flash pages asynchronously to OX bottom layer and maintains the write context for completion.

The bottom layer submits pages to the open-channel SSD in parallel. When the write completes, a status is returned to OX-Block and the write context is recovered. If the write fails, an *Ammendment* log must be appended and the chunk must be closed at the provisioning. If the write is successful, $map_L$ is updated at P5-MAP. Updating the mapping will trigger invalidation of pages at the chunk *bit_vector* at P2-BLK, *invalid_sec* is then incremented for future GC targeting.

In addition to the steps described above, transactions may be required. Writes are accepted in different granularities ranging from 4 kilobytes to several megabytes, these writes are then split into 4-kilobyte units and log entries are appended for each unit. For hosts, a large logical write is seen as a single command that must be either successful or failed, however, for the FTL, a logical write is divided into multiple, flash aligned, and independent commands. In a logical write, a page might fail while other pages succeed, if a failure is returned to the host then all pages composing the logical write must be aborted. We need to guarantee atomicity, either all pages succeed or we abort them all. We have designed a transaction mechanism that guarantees atomicity, consistency, and durability of user writes.

If transactions are enabled, recovery will discard all logs which a *commit* log was not found. If a logical write is successful we update $map_L$, append a *commit* log

containing the transaction identifier, and return a successful status to the host. If a logical write fails, we do **not** update $map_L$, we close all chunks which addresses where reserved at the provisioning, we do not append any log, and return a failure status to the host. In case of power failure in the middle of logical writes, the *commit* log is not appended and recovery will abort the transaction.

We guarantee atomicity of writes at the page level by updating $map_L$ atomically only after the new page is successfully persisted, this technique is based on shadow paging [28]. However, shadow paging guarantees atomicity of single pages for non-transactional writes. In transactional writes, multiple pages striped across PUs are part of the same atomic group, all entries in $map_L$ should be atomically updated together. In addition to shadow paging we use WAL with a commit log to guarantee atomicity of transactional writes. In OX-Block, we may use shadow paging only, or enable transactional writes with WAL commit.

## 6.2 Experimentation

This section presents the evaluation of OX-Block. Our system was composed of a single Dragon Fire Card equipped with an OX Controller and the OX-Block FTL, the DFC was connected to a host machine via 2x10-Gbit Ethernet ports. At the host, we used a 20-Gbit Intel X710-DA2 Ethernet card with two SFP+ transceivers. At the DFC we used the built-in ports with two SFP+ transceivers. The maximum network bandwidth was 2.4 GB/s approximately. The host machine was a 32-core Intel® Xeon® Silver 4109T CPU @ 2.00GHz, equipped with 128 GB of DRAM and the Intel X710-DA2 Ethernet card. For data transfer between host and DFC we used our NVMe over Fabrics implementation with in-capsule data, no form of hardware RDMA was involved, which means that host and DFC CPUs were used to transfer data via stream sockets. In terms of throughput, the general performance of the system was rated at around 450 MB/s for sequential writes and 1 GB/s for random reads. In terms of latency, it exhibits around 7ms for 1 MB sequential writes (in transaction mode), around 900$\mu$s for 256 KB random reads, and around 500$\mu$s for a 4 KB read. In the next sections, we measured memory utilization, garbage collection, and recovery including the log chain and checkpoint.

### 6.2.1 Memory Utilization

In this section, we present the memory footprint of OX-Controller. We started OX and measured the memory utilization in the DFC. We collected the information a few seconds after the startup when OX entered in a idle state waiting for incoming commands. Table 6.4 shows the memory utilization per component inside OX Controller.

| OX Core and Layers | | |
|---|---|---|
| **Component** | **Memory Utilization (MB)** | **Memory Utilization** |
| OX Core | 86.22300 MB | 90411372 bytes |
| OX Bottom Layer | 8.03442 MB | 8424704 bytes |
| OX Middle Layer | 32.51634 MB | 34095856 bytes |
| **Network Components - Admin Queue Only** | | |
| OX Upper Layer | 165.39738 MB | 173431720 bytes |
| NVMe Fabrics | 512.03367 MB | 536906216 bytes |
| **OX-Block FTL** | | |
| $map_{<S,T>}$, $chunk_{<L,S>}$ | 152.56003 MB | 159970784 bytes |
| Provisioning | 10.27722 MB | 10776448 bytes |
| Transactions | 384.14075 MB | 402800768 bytes |
| Log Buffer | 0.31308 MB | 328288 bytes |
| Recovery | 0.03555 MB | 37276 bytes |
| Garbage Collection | 0.00966 MB | 10128 bytes |
| **FTL Caches** | | |
| Write cache | 1496.12805 MB | 1568803968 bytes |
| $map_L$ Cache | 4108.00085 MB | 4307551104 bytes |
| **Total - OX Controller** | | |
| With Caches | **6955.67001 MB** | **7293548632 bytes** |
| Without Caches | **1351.54109 MB** | **1417193560 bytes** |

TABLE 6.4: OX Controller Memory Utilization (Admin queue only)

| Network Components - Four I/O Queues | | |
|---|---|---|
| **Component** | **Memory Utilization (MB)** | **Memory Utilization** |
| OX Upper Layer | 826.94941 MB | 867119304 bytes |
| NVMe Fabrics | 2560.15625 MB | 2684518400 bytes |
| **Total - OX Controller** | | |
| With Caches | **9665.34461 MB** | **10134848400 bytes** |
| Without Caches | **4061.21571 MB** | **4258493328 bytes** |

TABLE 6.5: OX Controller Memory Utilization (Four I/O queues)

Then, we run a few workloads and measured the memory utilization again. Before the runs, only the admin queue was allocated. Then, four new I/O queues were allocated. We can see difference in memory utilization at the network components. Table 6.5 shows the new values for the network after running the workloads.

After creating I/O queues we see a memory utilization of around 2.7 GB higher. This is due to buffers created by OX for handling incoming data from the network. We set the I/O queue depth to 4,096, a high value set on purpose for measuring the memory consumption, for lower values the memory footprint is also lower. A buffer to store submission and completion data is created for each slot in the queue. We created four queues, which give us 16,384 buffers. We prepare the buffers when the queue is created, it avoids the overhead of allocating memory.

As computational storage becomes popular, programmable controllers are increasing volatile memory capacity to accommodate larger applications. From a few megabytes found in legacy solid-state drives in the past, today, we find programmable devices such as the DFC equipped with 32 GB of DRAM or more. With this growth in memory capacity we can expect memory-heavy applications such as application-specific FTLs to become common in storage controllers in the near future.

### 6.2.2 Garbage Collection

In this experiment, we measure performance of garbage collection based on (a) impact on user writes, (b) write-amplification and (c) reclaimed space. As a parameter, we vary the target rate (*tr*), this parameter controls the aggressiveness of our algorithm that targets chunks. As explained in section 6.1.8, *tr* is the maximum value for the minimum invalid rate *mir*, our algorithm uses *mir* for targeting chunks. Thus, by manipulating *tr* we control the aggressiveness. We run the experiment five times, we set *tr* to 10, 25, 50, 75, and 90 %, respectively. Each experiment is composed of four phases, the insertion phase inserts a 5 GB dataset to OX-Block, the *User Only*

and *User + GC* phases issue 20 GB of uniform updates, and the GC Only phase occurs after the updates are completed while GC is still active. The *User Only* phase is over when GC is triggered, we set the threshold *th* to 10 GB which is twice the size of the dataset. We also set *gc_slots* to 8, allowing half of the channels to be garbage collected concurrently – our open-channel SSD has 16 channels.

On the host machine, we use 4 threads and create 4 NVMe over Fabrics queues. Figure 6.9 (a) shows the five experiments timeline, we do not include the insertion phase. Figure (b) shows the write amplification at time *T-1* (at the end of *User + GC*) and *T-2* (at the end of *GC Only*). Figure (c) shows the device utilization at *T-1* and *T-2*.

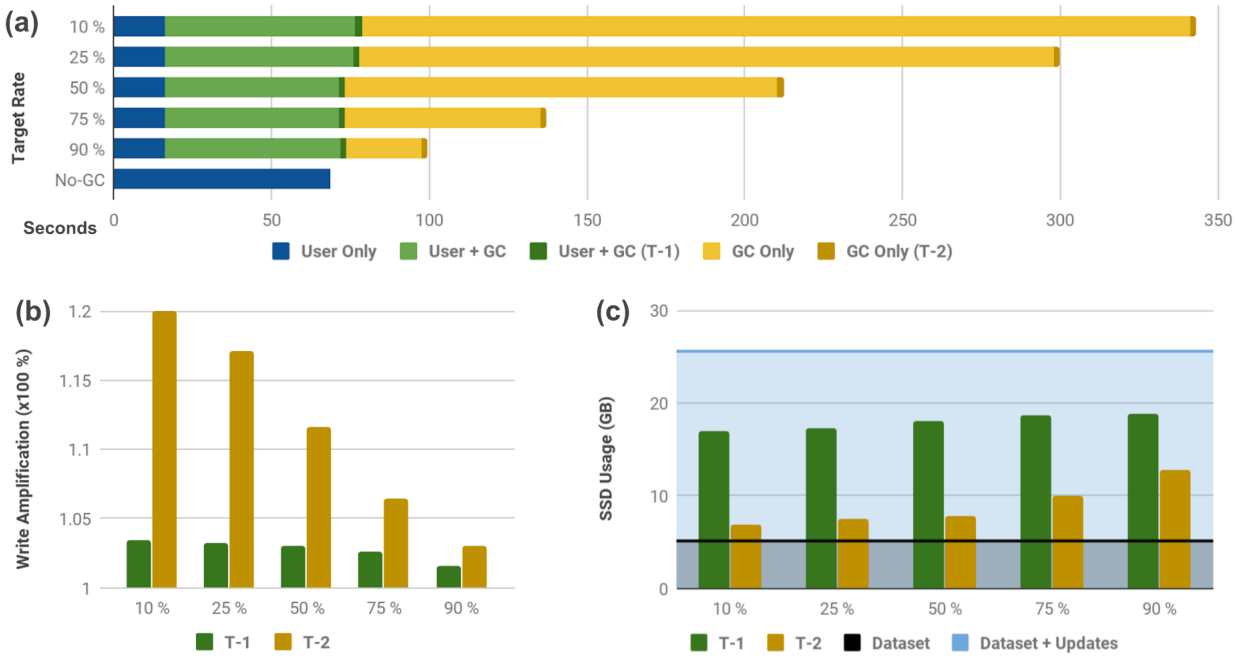



FIGURE 6.9: Garbage Collection Performance

- **(a) Impact on user writes:** In figure (a), we compare different *tr* values against the *No-GC* experiment. The blue bars are the *User Only* phase. In *No-GC* it composes the entire experiment because GC is never triggered. The user writes are completed at the host at *T-1*, or at the end of *User Only* in case of *No-GC*. The experiment continues in OX until *T-2* but at the host the experiment is completed.

  At the host, *No-GC* completes at 68 seconds, while *tr* 50%, 75% and 90% complete at 73 seconds. *tr* 25% completes at 78 seconds and *tr* 10% at 79 seconds. For non-aggressive options (*tr* >= 50%), the impact on user writes is approximatelly 7.4%, while for aggressive options (*tr* < 50%) the impact is around 14%.

- **(b) Write amplification (*wa*):** In figure (b), we compare *wa* over several *tr* values. *wa* shows how many times data is duplicated on the SSD due to garbage collection. Higher values wear out the media and decrease the device lifetime. We collected *wa* at *T-1* to see the impact on wear during high update rates, we then collected *wa* at *T-2* to see the impact on wear during the *GC Only* phase.

  At the dark green bars user updates are high. For *tr* 90%, *wa* is 1.016x while *tr* 10% gives 1.034x, an almost unnoticed difference between agressive and non-agressive options. This is due to random updates invalidating data so frequently that collected chunks always have high values in *invalid_sec*. At the yellow bars, updates no longer invalidate data and GC threads are running alone, *tr* 90% wears out the media 1.03x while *tr* 10% wears out 1.2x, an increase of 17%. In aggressive options, the *GC Only* phase is much longer and OX remains actively recycling chunks to reclaim space, in non-agressive options most of targeted chunks are recycled before *T-1* and OX enters an idle state much faster.

- **(c) Reclaimed Space:** In figure (c), we compare reclaimed space. The black line shows the dataset size of 5 GB, the blue line shows the amount of data written by user insertions and updates before *T-1*. In total, 20 GB of data became invalid by updates. At *T-1*, 42.4% of invalid space was reclaimed when *tr* was set to 10 while only 33.3% was reclaimed for *tr* 90. At *T-2*, the differences are higher. For *tr* 10, 91.3% of invalid data was reclaimed while *tr* 90 reclaims only 62.6 %.

In general, aggressive options reclaim more space at the cost of higher write amplification. GC aggressiveness does not impact wear during random updates but helps to reclaim more space at the cost of 7% less performance in user writes. For the *GC Only* phase, aggressive options reclaim more space but keeps the controller busy much longer and wears out the media. The optimal setup is a variable *tr* based on the current user update rate. An open study is the value of *gc_slots*, which might impact the optimal performance.

### 6.2.3 Checkpoint, Log, and Recovery

In this section, we add the checkpoint process and measure the impact on performance for write-heavy workloads. More specifically, we study how different checkpoint intervals ($C_i$) impact user writes performance and recovery time after a failure. We run the experiments with an active garbage collection, *tr* set to 50% and *gc_slots* to 8.

In the first experiment, we insert 5 GB of data into OX-Block, then we issue random updates equivalent to 50 GB. We set the checkpoint interval to 10 and 30 seconds, then we compare the performance against a disabled checkpoint. Table

| | | | Persisted Metadata (MB) | | |
|---|---|---|---|---|---|
| Interval ($C_i$) | Completion | Checkpoints | $map_L$ | $map_S$ | $chunk_L$ |
| No | 189.13 s | – | – | – | – |
| 30 s | 206.89 s | 6 | 100.09 | 2.81 | 276.75 |
| 10 s | 250.91 s | 18 | 313.34 | 8.38 | 838.22 |

TABLE 6.6: Impact of checkpoint on 50 GB updates

6.6 shows the completion time and the amount of metadata persisted during the checkpoints.

Degradation on performance is visible when checkpoint is enabled. The experiment takes 9.39% longer when $C_i$ is set to 30 seconds, and 32.66% longer for $C_i$ equals to 10 seconds. User writes are impacted when smaller checkpoint intervals are used. This is related to the amount of metadata that was persisted during the checkpoints. 72.9% of metadata are $chunk_L$, 26.4% are $map_L$ and 0.7% are $map_S$. These percentages are linked to the workload we used, a 50 GB update-only workload in a dataset of 5 GB.

In the second experiment, we simulate a fatal failure by killing OX in the middle of the experiment. All metadata in volatile memory is lost, some updates since last checkpoint might not be persisted, and the OCSSD is left inconsistent. Such a failure forces OX to rely on its recovery log to reconstruct the metadata and bring the OCSSD back to a consistent state. Before the failure, we use random logical writes of up to 1 MB in size; each logical write is a transaction. After the failure, we restart OX and wait until the FTL is ready, we expect an increasing restart time for longer runs. To cause the failure, we kill OX with *sudo kill -9 <process>*. We vary the point in time at which the failure occurs. We consider six different points in time *T1-6*. Figure 6.10 shows the experiment for disabled checkpoint, $C_i$ 10, and $C_i$ 30 seconds.

The blue line represents the expected recovery time if the checkpoint is disabled. Dots are the experiments at the time of failure. Without checkpoint the recovery time is close to 100 seconds at *T6*. This is due to the size of the log chain. Time is consumed for reading the log pages and applying the changes to metadata. When checkpoint is introduced, the recovery time decreases dramatically for longer experiments. We also see that recovery time oscillates up and down and remains constant, this was expected due to the checkpoint process truncating the log at a certain interval. The difference between $C_i$ 10 and $C_i$ 30 is not much, at *T2* and *T4* both recovered at the same speed. Table 6.7 shows the log statistics and recovered transactions at *T1-6*.

In table 6.7, we clearly see the benefits of checkpoint at the number of recovered transactions, at *T-6* 206K transactions had to be recovered when checkpoint was disabled, against 16.6K and 23.3K when checkpoint was enabled, this impacts recovery time. We also see the impact of checkpoint on updates, the update rate decreases

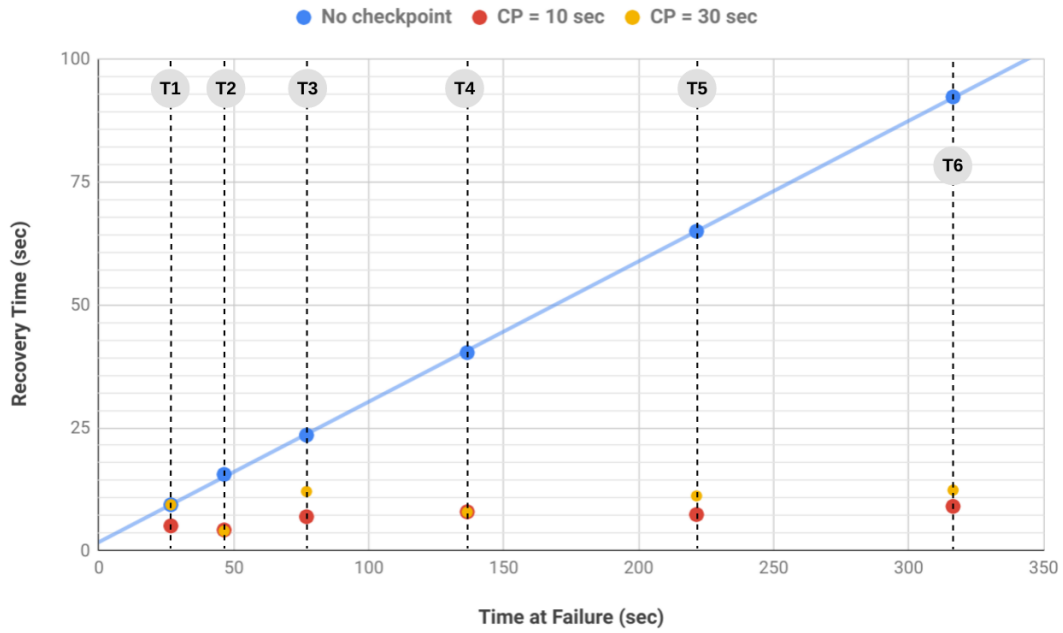



FIGURE 6.10: Impact of Checkpoint Intervals on Recovery Time

as we set $C_i$ to lower values. At the table, updates are composed of user writes, GC writes, and persisted metadata.

The log chain grows at the rate of user writes and garbage collection activity. Thus determining the optimal $C_i$ requires a runtime analysis on the FTL events such as updating mapping table and provisioning requests. The write rate varies depending on SSD usage and garbage collection constraints. Therefore, an optimal solution could be defining a variable $C_i$ where the interval changes according to the SSD usage. For a scenario such as figure 6.10, $C_i$ 10 is not viable compared to $C_i$ 30, we only see a small improvement in recovery time at the cost of a higher impact on write performance.

## 6.3 Related Work

For FTL components, related work is at section 5.3. This section presents related work on high performance components in OX, such as NVMe/OX-MQ queues and NVMe over Fabrics.

Lee et al. [49] shows the benefits of isolating reads and writes on NVMe queues. In our design, we separate reads and writes at the middle layer to avoid interference of writes at the queue level. In [39], Joshi et al. implements weighted-round-robin-with-urgent-priority (WRR) on the NVMe driver in the Linux kernel. This approach allows I/Os to be executed with priority even if the SSD usage is high. In our design, we used a standard round-robin mechanism for the NVMe queues.

| | | Checkpoint Interval & **Recovery Log** | | | | | |
|---|---|---|---|---|---|---|---|
| | | No checkpoint | | 10 seconds | | 30 seconds | |
| Failure | Time | Logs | Size | Logs | Size | Logs | Size |
| T1 | 26.7s | 2.1M | 182 MB | 583K | 54 MB | 2.1M | 182 MB |
| T2 | 46.4s | 3.6M | 308 MB | 202K | 16 MB | 215K | 16 MB |
| T3 | 76.9s | 5.6M | 433 MB | 884K | 64 MB | 2.4M | 171 MB |
| T4 | 136.4s | 9.9M | 759 MB | 1,111K | 83 MB | 1.1M | 78 MB |
| T5 | 221.4s | 16.4M | 1,198 MB | 988K | 70 MB | 2.1M | 153 MB |
| T6 | 316.4s | 23M | 1,691 MB | 1,317K | 98 MB | 2.2M | 159 MB |

| | Checkpoint Interval & **Transactions (R: Recovered, A: Aborted)** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | No checkpoint | | | 10 seconds | | | 30 seconds | | |
| Failure | Updates | R | A | Updates | R | A | Updates | R | A |
| T1 | 7.8 GB | 8.1K | 4 | 7.5 GB | 2.3K | 3 | 7.8 GB | 8.1K | 4 |
| T2 | 13.6 GB | 20K | 136 | 11.9 GB | 3.1K | 130 | 12.9 GB | 2.9K | 102 |
| T3 | 21.2 GB | 39K | 291 | 18.6 GB | 10K | 300 | 20.5 GB | 23.9K | 163 |
| T4 | 37.7 GB | 81K | 183 | 32.1 GB | 12.9K | 497 | 35.2 GB | 12.4K | 123 |
| T5 | 61.9 GB | 142K | 165 | 50 GB | 11.1K | 222 | 56.8 GB | 23.6K | 4 |
| T6 | 86.8 GB | 206K | 126 | 70.7 GB | 16.6K | 519 | 78.2 GB | 23.3K | 166 |

TABLE 6.7: Log Chain Statistics at Recovery

RDMA techniques were studied in [89, 30]. The implementation of RDMA over Ethernet shows improvement over TCP sockets, as we predicted in our work. A study of NVMe over Fabrics performance can be found in [33]. In Titan [81], a detailed study of the Linux networking stack shows the unfairness of packet scheduling for high-speed and hardware optimized network and addresses ways of reducing latency. In our network implementation, we rely on standard network protocols available in stable versions of the Linux kernel.

## 6.4 Conclusions and Future Work

In this Section, we showed the feasibility of putting together a generic FTL based on our modular framework. We detailed the design of the data structures and functions at the heart of an FTL. We zoomed in on the performance of garbage collection and recovery and on memory utilization. The results show that the techniques we propose have low overhead and are viable. This is a strong basis for the design of an application-specific FTL.

# 7 OX-ELEOS: Improving Performance of Data Caching Systems

In databases, in-memory systems [19, 48, 82, 43] deliver superior perfomance by retaining all data in volatile memory. They improve latency to nanosecond scale by avoiding access to secondary storage. Such systems sustain impressive performance, however, storing data in main memory is expensive in terms of hardware resources and energy consumption. Data caching systems such as Microsoft's Deuteronomy [56, 55] and LeanStore [51] reduce costs by keeping cold data in secondary storage and hot data in main memory as explained by David Lomet at MSR [57]. Main memory systems are necessary for hot data management but today's largest datasets are primarily composed of cold data [67, 5]. We need high performance cache layers.

The cost of flash memory per byte is decreasing quickly. SSDs equipped with high density QLC NAND are now becoming common and prices will decrease even faster. Unlike flash, DRAM prices do not decrease at the same rate and large main memory setups are becoming orders of magnitude more expensive than SSD arrays. Today, data caching systems leverage the speed and capacity of modern NVMe SSDs at lower costs, but latency of flash is still tens of times higher than DRAM and techniques to minimize SSD access are required. This chapter presents log structuring techniques used by Microsoft's BwTree/LLAMA and introduces OX-ELEOS, a modified version of OX-Block that supports log structuring interface. We offload BwTree components to OX-ELEOS and entrust log structuring resposibility to OX controller. Our goal is removing overhead from host CPUs and minimizing data movement between hosts and SSDs, this approach considerably improves performance of the BwTree's cache layer.

This work is a collaboration with Jaeyoung Do and David Lomet in the context of a summer internship at Microsoft Research Redmond and further work [21].

## 7.1 Modern Data Caching Systems

Many modern data caching systems [55, 22] use a log structuring approach [75] to minimize I/Os to secondary storage, large buffers containing database updates are flushed to the SSD using large and sequential I/Os. Log structuring avoids concurrency by appeding new versions of data to the log and avoiding in-place updates. It also minimizes the CPU cache misses by avoiding eviction of cache lines in multi-core systems. Log buffers in volatile memory serve as cache and a mapping table is required to map from logical pages to offsets within the buffers. If the page is not in cache, a read is issued to the SSD and the page is loaded. As nature of a logging approach, several versions of the same logical page are appended to buffers, thus garbage collection is necessary to free space for new data. Durability of mapping table is also a concern, it requires write-ahead logging and checkpoint mechanisms to ensure durability and proper recovery. All these techniques increase the overhead in host CPUs, but are necessary.

Two main approaches have been proposed for log structuring stores. At Microsoft Research, the BwTree [55] appends updates called "deltas" to log structured store (LSS) buffers, a chain of delta updates represent multi versions of logical base pages. Garbage collection is performed on LSS buffers, it frees space invalidated by page consolidation. A page is consolidated when delta updates are applied to the base page and deltas are removed from the chain. On the other hand, at Facebook, RocksDB relies on the LSM-Tree [16], it appends updates to sorted sequence tables (SST), which are organized in levels. A new level is created and filled when the previous and smaller level is full. Each level stores exponentially more data. Garbage collection is performed by a process called compaction, which consists of removing duplicated keys from SSTs. Compactions are triggered following leveling or tiering techniques, or smarter techniques such as lazy leveling proposed by Niv Dayan et al. [17]. In both BwTree and LSM-Tree, performing garbage collection requires loading data from SSDs to main memory. This data movement could be avoided if the SSD controller was aware of application requirements. In the context of computational storage, application-specific FTLs could implement consolidation/compaction-aware garbage collection techniques.

Besides garbage collection, the BwTree maintains a mapping table of logical pages, and ensures durability/recovery via write-ahead logging and checkpoint mechanisms. In OX-ELEOS, we offload garbage collection and recovery components to the SSD controller by implementing OX-App components based on log structuring requirements. An irony of log structuring on databases is that standard SSDs already implement a form of GC and recovery on its controller. By installing BwTree components on the SSD we are replacing generic FTL components by application-specific ones, and hence OX-ELEOS deduplicates components and shrinks layers on the storage stack.

## 7.2 BwTree and LLAMA Log-Structured Store

Microsoft's Deuteronomy architecture includes separation of transaction components (TC) and data components (DC). We focus on LLAMA, which is a DC component implemented for BwTree. LLAMA guarantees persistence and availability of LSS buffers generated by BwTree updates. We thus describe what is in the buffer and how it is organized. Then, we describe the mechanisms of mapping table which is affected by page updates, LLAMA's garbage collection, and when LLSs are persisted to flash.

BwTree uses logical page identifiers (LPID) to track data pages. At LLAMA, logical pages are composed of a **base page** ($P_B$) and a sequence of **delta updates** ($P_D$). $P_B$ is appended to the LLS buffer when first created. Subsequently, updates to $P_B$ are not in-place, instead, $P_D$ updates are logically prepended to $P_B$ and physically appended to the end of an LSS buffer. Pointers at $P_B$ and $P_D$ form a sequence called *delta chain*. The main benefit of this design is that parts of $P_B$ that are not modified do not need to be persisted, only $P_D$ updates that contain modified data are written to flash.

The mapping table maintains translations of LPIDs to the most recent $P_D$ at the page *delta chain*, the map entry contains a memory address if the delta is cached, or an LSS file offset if not cached. Updates to the mapping table are lock-free and performed via an atomic compare-and-swap (CAS) instruction [60]. Figure 7.1 shows examples of *delta chains* and sketches LSS buffers in main memory as well as in the SSD.

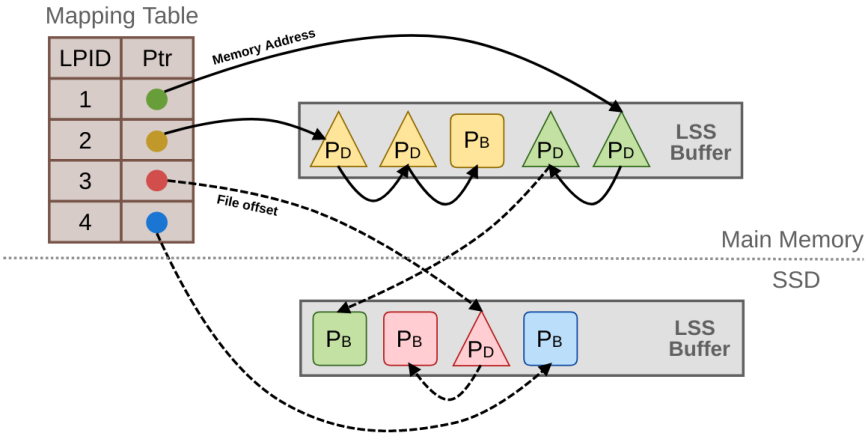



FIGURE 7.1: BWTree/LLAMA Mapping and Delta Chains

Figure 7.1 shows four examples of *delta chains*. LPID 2 (yellow chain) is completely in main memory and all pointers are memory addresses. When the LSS is persisted the entire LPID 2 is written to flash. LPID 3 (red chain) is completely in flash and all pointers are file offsets. If LPID 3 is requested, the page needs to be loaded from flash to main memory. LPID 1 (green chain) has $P_D$s in volatile memory, when the LSS is persisted, only modified pieces of LPID 1 are written to flash.

LPID 4 (blue chain) has no deltas, and $P_B$ is on flash, thus the mapping table contains a file offset to $P_B$.

A transaction component running above LLAMA performs write-ahead logging and sends a signal to LLAMA containing the end of stable log (ESL). This signal triggers persistence of LSS buffers. The transactional aspects of the system requires a checkpoint at certain interval, it guarantees durability of mapping table by writing all required metadata to flash. Checkpoint is not the only overhead, a garbage collection scans LSS buffers and free space that previously belonged to consolidated pages. In order to scan a buffer, the entire LSS file is loaded from flash to main memory. In OX-ELEOS, we want to remove from LLAMA the following components:

1. **File offsets** from the mapping table. We want to use LPIDs to load pages directly from the SSD. LLAMA will no longer know where the page is physically located. This approach saves CPU cycles by minimizing atomic calls to CAS when updating the mapping.
2. **Checkpoint**. If the SSD controller guarantees durability, then the mapping table containing only memory addresses can be safely restored by requesting LPIDs directly from the SSD.
3. **Garbage Collection**. If the SSD controller is aware of physical LPID locations, then LSS buffers can be scanned within the SSD controller and buffers are no longer moved from flash to LLAMA's main memory.

### 7.2.1 Fixed, Variable, and Multi-Piece Pages

To simplify the implementation process, we defined three steps. First, we want to implement a system for fixed-size pages that can be used not only for BwTree but for any page-oriented data caching store. For fixed-size, BwTree pages are 4 KB in size and updates do not create deltas but consolidates the page, thus LPIDs always match with fixed blocks of 4 KB. Second, we want to introduce variable-size pages and support updates of smaller granularities, we do not create deltas updates but the base page size is variable. Third, we want to introduce multi-piece pages by allowing delta updates and a *delta chain*, this version supports BwTree on its full design.

For fixed-size pages, we can reuse all OX-Block components but P9-WCA write cache. To support LPIDs in OX, we need to parse LSS buffers in the SSD controller. The correct place to implement the parser is at the write cache component. For variable-size pages, changes at P5-MAP and P8-GC are also required, the granularity of logical pages neither matches with flash sectors nor with the fixed 4 KB granularity of our mapping table. An extended physical address that contains an offset and the page size is required. For GC, invalid flash pages may still contain valid data

of smaller granularities, which need to be tracked and moved to new locations. Alternatively, P2-BLK *bit_vectors* can be modified to support multiple granularity of page invalidation. For multi-piece pages, new changes at P5-MAP are required. The mapping table stores the location of the first delta in the chain. Then, other structures need to be created to support an in-memory management of *delta chains*.

In this thesis, we present OX-ELEOS design for **fixed-size pages**. We implemented and evaluated BwTree without checkpoint, GC, and file offsets, but all pages are fixed to 4 KB and no delta updates are allowed. Implementation and experimentation of variable and multi-piece pages are still a work in progress. We intend to publish results in a future conference paper.

### 7.2.2 Batching: A Log Structuring SSD Interface

To remove file offsets, checkpoint, and GC from LLAMA, our SSD controller needs to support a new interface other than block-oriented addresses. We designed and implemented the batching interface, we modified OX upper layer to support larger I/Os of several megabytes, then we created two new NVMe commands to support flushing ($flush_{LSS}$) of entire LSS buffers and batches of reads ($read_{BATCH}$). For flushes, we move the entire LSS buffer to OX-ELEOS FTL where it parses the buffer, separates the fixed-size pages, and persists them to the open-channel SSD. For reads, we support random LPIDs within the same command, allowing larger I/Os for random reads of smaller granularities. It differs from standard NVMe reads where a starting logical address and a data size are provided. Fixed 4 KB logical pages match with OX-Block mapping granularity, thus most of OX-Block components can be reused. Figure 7.2 depicts the batching interface and its compatibility with current OX-Block components.

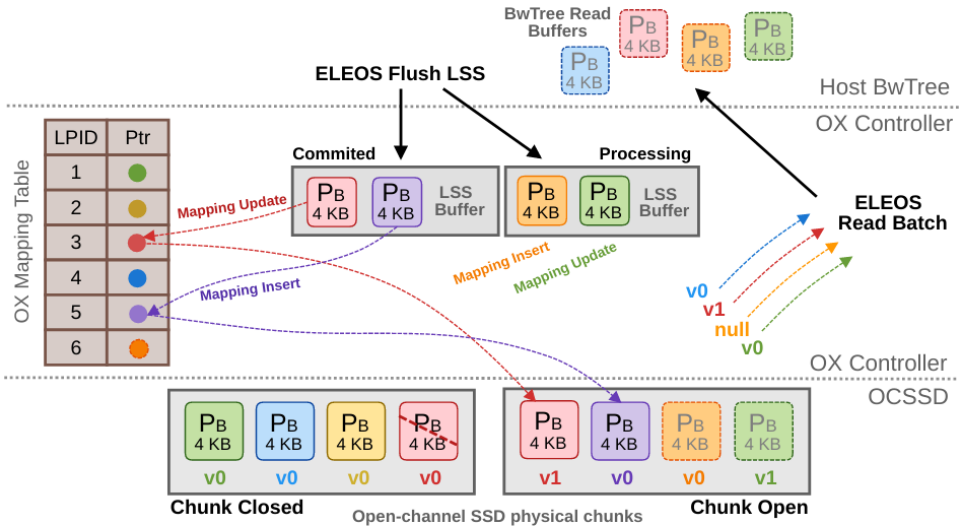



FIGURE 7.2: ELEOS Batching Interface and Transactional Buffers

At the left of figure 7.2, OX mapping table ($OX_{MT}$) mirrors the BwTree mapping table ($BT_{MT}$). At the host, $BT_{MT}$ translates LPIDs to cache pointers. At the controller, $OX_{MT}$ translates LPIDs to physical flash addresses. Fixed-size pages are represented by $P_B$ and the color represents a specific LPID. At the middle of the figure, two $flush_{LSS}$ commands are issued, one already committed via OX transaction mechanisms described in section 6.1.9. LPID 3 (red) updates the mapping table and a new version 1 of the page is written to an open OCSSD chunk. The old version 0 is then invalidated. At the same buffer, LPID 5 (purple) is new at the mapping table and version 0 is written to the open chunk. At the same time, another LSS buffer is being flushed and is not committed yet. At the right, a $read_{BATCH}$ command requests four LPIDs. Reads see only committed transactions, thus, LPID 1 (green) reads at version 0 and LPID 6 (orange) reads a null value.

We enforce atomicity of transactions at the LSS buffer and reuse the OX-Block WAL, checkpoint, and recovery to ensure durability and consistency of LSS buffers. We use transactional recovery with WAL commit log. Note that concurrency control is enforced above LLAMA by a transaction component, as a result OX-ELEOS does not need to implement shadow paging [28] across PUs to enforce before-or-after atomicity for LSS buffers.

## 7.3 FTL Design and Implementation

This section describes OX-ELEOS as a modification of OX-Block. We reused eight components and reimplemented P9-WCA, the new write cache supports LSS buffers and transactions at the buffer level. At OX upper layer, we added $flush_{LSS}$ and $read_{BATCH}$ commands as part of our ELEOS application parser. Refer to chapter 5 for a complete definition of other OX-Block components.

### 7.3.1 P9-WCA for SSD Transactions

We modified the write cache in OX-Block to support multiple flushes of LSS buffers. We then modified BwTree to use an ELEOS host library that implements the $flush_{LSS}$ command. Concurrent calls to $flush_{LSS}$ are allowed by allocating multiple buffers at the write cache. Then, incoming buffers are treated as transactions and processed by concurrent threads created by OX-MQ in the middle layer. When buffers are completely copied to the write cache, a sequence of transactional phases are performed. The phases are described below.

1. **Buffer Parsing**: Logic extracted from the original BwTree identifies valid LPIDs and data offsets within the LSS buffer. For instance, a 1 MB buffer may contain up to 252 LPIDs of 4 KB each, the rest of the space is reserved for metadata that describes the buffer.

2. **New Transaction**: A new transaction context is created. The context tracks the status of each LPID in the buffer and the status of each physical write performed to the OCSSD. New transactions have the status *pending*. To enforce isolation we serialize transactions. *Pending* transactions are added to a *serial queue* and must be committed or aborted in the same sequence they were added to the queue. Serializability is necessary to guarantee isolation of concurrent transactions.

3. **Physical Page Aligment**: LPIDs are grouped in flash-aligned pages, for instance, 8 LPIDs for a 32 KB flash page. A byte vector tracks the status of each flash page. The byte value is "zero" for processing, "one" for successfully written, and other values if an error has occurred.

4. **Write Submission**: Flash aligned pages are submitted to a single write queue. Concurrent transactions share the same queue and flash pages are mixed. A write thread dequeues pages, requests physical addresses from P3-PRO, appends *User Write* logs to P6-LOG, and submits the page to OX bottom layer. We do not wait until the write-ahead log is persisted. To minimize write latency we write the pages concurrently with the log. This is not a problem if we make sure the logs are persisted before the transaction is committed.

5. **Write Completion**: Each completed write sets the corresponding byte in the vector. When the vector is completely set, the transaction status is changed to *persisted*. For every persisted transaction we check the *serial queue*, we commit all transactions in order from the head of the queue until we find a transaction in *pending* state. This way, we guarantee serializability.

6. **Transaction Commit**: Persisted transactions that have not failed are committed, otherwise, they are aborted. The commit process consists of updating the mapping and chunk metadata, then appending a *Commit* log. All $map_L$ entries corresponding to the LPIDs are updated via P5-MAP and old physical addresses are invalidated by P2-BLK. If the steps above succeed, the *Commit* log is appended at P6-LOG and the transaction buffer is freed for new transactions. At this point, we **must wait** until the *Commit* log is persisted. To improve latency, we might not wait, but BwTree must be aware of a small probability of not recovering a recently committed transaction if a fatal failure occurs. We call P6-LOG PERSIST with the transaction timestamp as parameter, from this point, our log management component is responsible for completing $flush_{LSS}$ to the host.

   At the commit phase, $map_L$ updating is at page level, each entry (LPID) is updated individually. This causes a situation where readers may see some *old* versions, while other pages are already updated to the *new* version. This situation might happen before the *Commit* log is appended and its safe for recovery.

A transaction component above LLAMA also guarantees consistency in case of concurrent writers and readers.

7. **Transaction Abort**: A transaction is aborted if a write to persistent storage fails, or if a metadata update fails during the commit. To abort a transaction we close all physical chunks (at P3-PRO) which the transaction wrote into, we do not update any metadata, and we do not append the *Commit* log. Closed chunks must have the invalid data invalidated at P2-BLK.

By following the steps above we guarantee that committed transactions are always recovered at P7-REC and that failed transactions are always discarded. Our transactional SSD allows us to safely remove the checkpoint from LLAMA and use OX-Block checkpoint instead. By mirroring the BwTree mapping in $OX_{MT}$, we safely remove file offsets from $BT_{MT}$ and garbage collection from LLAMA, we then entrust LSS cleaning and chunk recycling to OX-Block garbage collection.

## 7.4 Experimentation

We run all experiments in the same setup used for the OX-Block experiments (section 6.2). A single Dragon Fire Card equipped with OX Controller was connected to a host machine via 2x10 Gbit Ethernet interfaces. The host was an Intel® Xeon® Silver 4109T Core @ 2 GHz equipped with 128 GB of DRAM. All communication between host and DFC was performed via our NVMe over Fabrics implementation.

This section describes a set of experiments to measure performance of OX-ELEOS and Bw-Tree configured for 4 KB fixed-size pages. In all experiments, we compare the *original* version of BwTree running on a standard PCIe-attached NVMe SSD against the *offloaded* BwTree running on our fabrics-attached transactional SSD equipped with OX Controller. In *original*, we used a Samsung 970 EVO Pro 512 GB rated at around 1.2 GB/s of random 4KB reads and 2 GB/s for sequential writes. In *offloaded*, we used a CNEX open-channel SSD rated at around 1.6 GB/s for random 4KB reads and 2.4 GB/s for sequential writes. The OCSSD was PCIe-attached to the DFC which was fabrics-attached to the host. Comparing the fabrics against a PCIe-attached SSD is a challenge in our experiments due to additional latency added by the network layer.

### 7.4.1 YCSB Benchmark

The YCSB benchmark [23] is widely used to measure performance of key-value stores. The benchmark generates a synthetic dataset composed of keys and values of custom sizes. Alongside the dataset, it generates a synthetic workload composed of updates and reads, the size of updates is also customizable. We calculate the dataset size by tuning the key and value sizes to a fixed value, then we define the

number of records to be inserted. YCSB comes with a set of predefined workloads that can be modified. An important parameter is the workload type that can be uniform (random reads and updates) or zipfian (a set of records are updated/read more often to simulate hot and cold data) with a default skew of 20% of hot data.

In our experiments, we vary the YCSB parameters to generate different workloads and dataset sizes. We run the benchmark via two BwTree binaries, one for the original BwTree and the other was modified to support the ELEOS host libraries. The binaries accept a wide range of parameters including YCSB integrations and setup for checkpoint and garbage collection.

### 7.4.2 Cache Size

In this experiment, we want to measure (i) the impact of BwTree cache size on performance, and (ii) the impact of checkpoint and garbage collection. We then compare the performance of *offloaded* against the performance of *original*. We used YCSB to create a dataset with 5 million records, 8-byte keys, and 1KB values, the total dataset size was around 5 GB. We selected a read-mostly workload with 10 million operations of 25% updates and 75% reads, each update was 1 KB in size. The total of updated data was around 7.15 GB, enough for triggering garbage collection and checkpoint several times during the experiment. To minimize the difference between the PCIe and fabrics SSDs we limited BwTree to run on a single CPU core, thus, the experiment is never I/O bound but CPU cycles bound. Figure 7.3 shows the performance of several cache sizes.

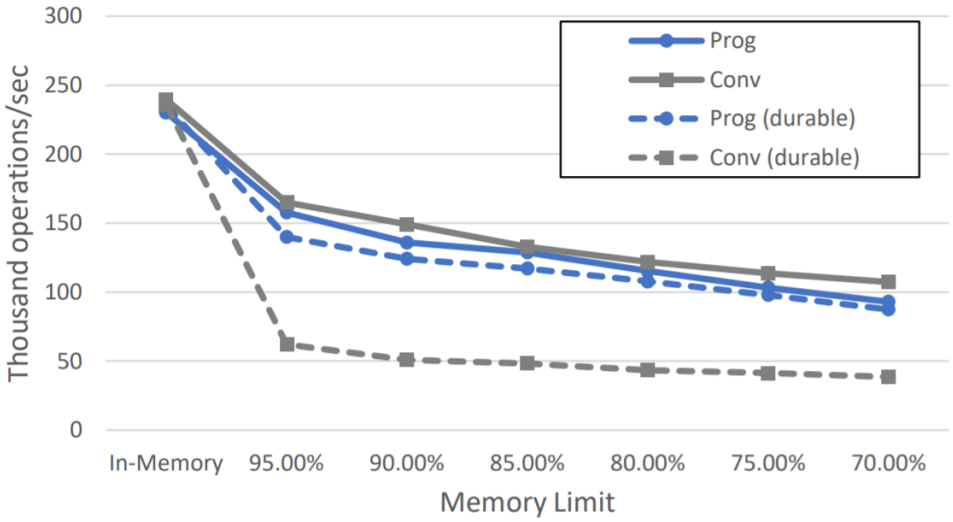



FIGURE 7.3: Impact of Cache Size on OX-ELEOS Performance [21]

We run the same workload several times starting from a cache size where all data fitted in memory, then we limited the memory until we reached 70% of the dataset size. We first compare the impact of checkpoint and garbage collection. In *original* (Conv) we see a drastic hurt on performance when checkpoint and garbage

collection are enabled (durable). BwTree requires a considerable amount of CPU cycles at the single core to guarantee durability and garbage collection on the host. Differently, in *offloaded*, the performance difference is not much due to checkpoint and GC being offloaded to OX Controller. In both *original* and *offloaded* durable experiments, we set the checkpoint interval to 10 seconds and triggered GC during the entire experiment. In *offloaded*, we set OX-Block GC *tr* parameter to 50% and *gc_slots* to 8.

The benefits of offloading log structuring functionalities from hosts to the SSD controller are evident. We measured the overhead implied on CPU utilization in a single core, the next experiment shows the performance while we scale up to multiple CPU cores.

### 7.4.3 Scalability

In this experiment we vary the number of cores and threads on BwTree. We used a 5 GB dataset with zipfian distribution where 20% of the data was hot. We experimented with two different workloads, (i) 5% of updates and 95% of reads, and (ii) 25% of updates and 75% of reads. We increased the number of benchmark threads in each experiment from 1 to 8, in each run we issued 10 million operations. By looking at figures 3.4 and 3.5 (Chapter 3), CPU utilization on the DFC is a concern, we decided to collect CPU utilization in OX Controller and see the impact of scalability. As the number of cores increases, more I/Os and higher network traffic is expected, thus we also expect higher load on the DFC CPU. Figure 7.4 and 7.5 show the results for the workloads (i) and (ii), respectively. The CPU usage shown in the figures are collected from the DFC ARM cores.

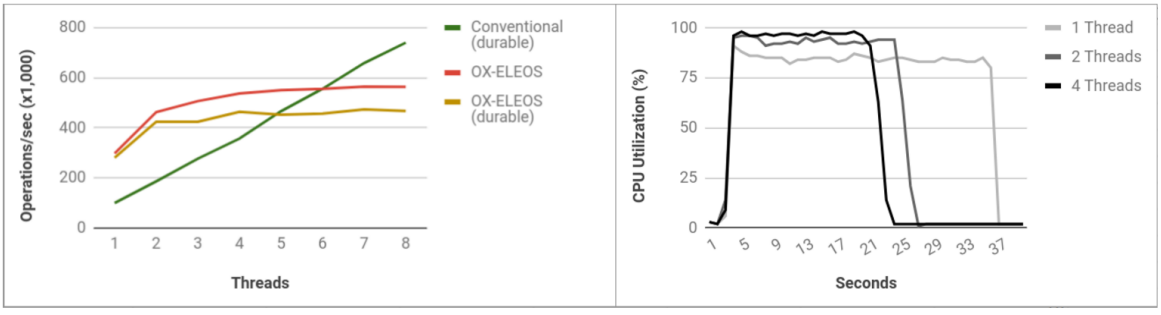



FIGURE 7.4: OX-ELEOS Scalability - 95/5% Reads/Updates

As expected, the CPU utilization increases as we increase the threads. However, surprisingly, the load on the CPU is already very high (above 80%) for a single benchmark thread. We believe that copying data from the network buffers to OX Controller memory then copying again from OX to the open-channel SSD is the major overhead. Our NVMe over fabrics does not use any form of RDMA but standard TCP-based sockets which require memory copies. From 80% of CPU load in a single benchmark thread, it is not a surprise that OX-ELEOS cannot scale above a few

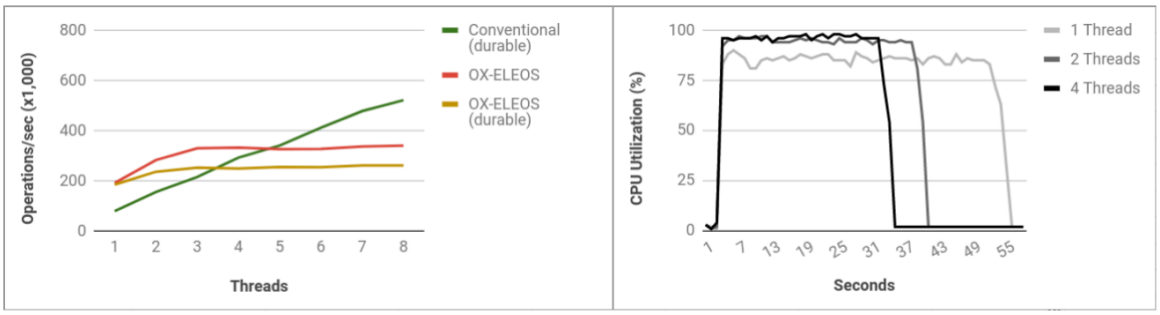



FIGURE 7.5: OX-ELEOS Scalability - 75/25% Reads/Updates

benchmark threads. For the first time, we see the overhead of having a full CPU-based SSD controller moving data via standard network protocols. We now make a few commentaries and give a few insights on how to proceed.

- **Programmable Platform**: Before the CPUs get to its limits, the performance gain of OX-ELEOS (durable) is noticeable. Unfortunately, the ARM cores reach 100% with a few benchmark threads, we cannot predict if the scalability would be linear if stronger CPUs were used. We have an idea of scalability by looking at figure 3.4, OX-MQ performs a lot better in platforms other than the DFC cores. A question arises from these arguments. Which platform is the best fit for programmable SSD controllers? Platforms other than CPUs are also an option, for instance, Samsung has recently introduced the SmartSSD as an FPGA-based programmable controller.

- **Standard versus Specific Protocols:** This thesis is based on the argument made by Jim Gray that active disks would rely on standard network protocols. If RDMA is used instead of sockets, we expect performance gains due to avoiding memory copies on the network stack, however, hardware specialization would be introduced. We still believe that standard protocols can be improved to support active disks. For instance, efforts on XDP[1] socket are led by Jesper D. Brouer at Red Hat. XDP implements techniques to avoid memory copies in the network stack, as does RDMA.

## 7.5 Related Work

Wang et al. at CMU implement OpenBw-Tree [84], an open-source version based on Microsoft's BwTree. However, OpenBw-Tree does not support eviction of pages to persistent storage, as done in BwTree via LLAMA [56]. Other in-memory indexes used in databases include SkipList [72], modified versions of B+Tree [87],

[1]XDP socket: https://www.kernel.org/doc/html/v4.18/networking/af_xdp.html

and adaptive radix tree (ART) [53]. In-memory indexes may adopt lock-free mechanisms as done in BwTree with CAS [60]. For instance, B+Tree was originally designed for disk-oriented database systems, but in [87] implementation, the B+Tree uses optimistic lock coupling (OLC) [52]. Other in-memory systems with latch-free techniques include BzTree [4], a high-performance latch-free index for non-volatile memories, and Hyper [43] which uses adaptive radix trees with OLC techniques. Opportunities for computational storage raises when these systems need to persist its data. In our work [70], we raise the question of whether common languages should be defined for programming storage controllers. Such languages could be used to support larger than main memory workloads in systems such as OpenBw-Tree or other types of in-memory only databases [41, 43, 19].

Alexander van Renen et al. and his team proposes a 3-tier storage hierarchy [74] by adding persistent memory in between DRAM and flash. Exploring the design of systems built for persistent memory in the context of computational storage is future work. For instance, persistent memory could be used to store our recovery log and improve the latency of OX transactions. Other work in persistent memory was done in FOEDUS [47], a transactional engine built for NVRAM and large NUMA machines. Scalability in multi and many-cores architectures was studied by Ailamaki in her book [2]. In our experiments with OX-MQ in section 3.5 we see the impact of multi-core NUMA machines on scalability of our queues.

## 7.6 Conclusions and Future Work

OX-ELEOS is our first design of an application-specific FTL with the OX framework. We showed that an application-specific FTL can be defined in the context of the modular OX architecture. We adopted a white box approach in order to minimize overhead on the read path. Experimental results show the potential of our approach for reducing load on the host CPU. Our implementation of recovery also illustrates the benefits of an application-specific FTL, where conditions enforced on the host (concurrency control enforced above LLAMA) make it possible to relax constraints and improve performance (dropping the requirement of shadow paging across PUs to enforce before-or-after atomicity in OX-ELEOS).

OX-ELEOS provides the abstraction of a log-structured SSD accessed from LLAMA on the host side. OX-ELEOS implements an application specific FTL defined for the log structured abstraction. We are now considering application-specific FTL for other abstractions. More specifically, we are exploring an FTL specialized for LSM-tree management. LSM-trees constitute ideal candidates for collapsing layers on the storage stack. Indeed, LSM-trees and flash translation layers both rely on immutable storage structures associated with a form of garbage collection. Initial work by Baidu [83] and CNEX Labs [26] focused on supporting an LSM-based key-value

store on open-channel SSDs by implementing an LSM-Tree specific FTL on the host. These solutions are defined in the context of traditional host-SSD architectures. On the other hand, recently, Samsung announced KV-SSD, an SSD equipped with an LSM-tree specific FTL [76]. Our assumption is that these solutions can be applied in the context of computational storage, and thus be integrated into the OX architecture via programmable FTL components. Implementing LSM management on programmable controllers, with OX, is a work in progress [68].

# 8 Conclusion

## 8.1 Summary of Results

We started the thesis with a deep evaluation of open-channel SSDs. The goal was acquiring a solid understanding of the underlying media. We built the first generation of OX based on the FPGA storage board and MLC NAND as underlying media, OX was designed to expose the open-channel SSD interface to hosts via PCIe. We conducted a set of experiments to evaluate the open-channel SSD. Since no benchmark tool for open-channel SSDs were available, we built FOX. We architected and implemented the $\mu$FLIP-OC benchmark in FOX. The results on the DFC equipped with the FPGA showed the impact of parallelism, isolation, and wear on performance. Then, we applied $\mu$FLIP-OC on an industry-grade open-channel SSD built by CNEX Labs, the behavior of the underlying media was similar to the FPGA but the overall performance in terms of latency and throughput was higher. We decided to use the CNEX open-channel SSD for further experiments in the thesis.

The next step was deconstructing the FTL based on the lessons learned from $\mu$FLIP-OC. We explored the design of flash translation layers on top of open-channel SSDs, the goal was identifying common components that could be programmed independently. Inspired by FTL designs such as pblk, we encountered nine components that are essential in FTLs. As a result, we architected and built OX-App, a framework for FTL development based on the nine components. OX-App was introduced in OX second generation together with NVMe over Fabrics support. For the fabrics, we used standard network protocols (TCP/IP) in our design. We did not implement RDMA techniques, which is a subject of future work.

In the next part of the thesis, we used OX-App to materialize FTL designs. First, we built OX-Block, a page-level FTL with generic components. The goal of OX-Block was twofold, we wanted to (i) validate OX-App by building and evaluating a full-fledged FTL, and (ii) develop generic FTL components that could be reused for application-specific FTLs. To achieve these goals, we relied on state-of-the-art techniques to orchestrate the FTL in terms of metadata management, parallelism, durability guarantees, and garbage collection. The bulk of the work on OX-Block was done during my stay abroad at Tsinghua University.

The last part of the thesis was a collaboration with Microsoft Research. I spent a summer in Redmond and did further work at ITU to materialize OX-ELEOS, an

application-specific FTL for log structured stores. OX-ELEOS improves performance of log structured stores by collapsing layers that are duplicated in the database and storage controller. We experimented with Microsoft's BwTree and offloaded its checkpoint and garbage collection processes to our OX Controller equipped with OX-ELEOS FTL. We conducted a set of experiments that exhibit the benefits of collapsing layers in the storage stack. The results also show the overhead of the SoC in terms of high performance packet processing for NVMe over Fabrics, and the impact of standard network protocols on the deployment of computational storage in disaggregated setups.

We thus progressively built a system, the OX framework, implementing a modular FTL architecture, OX-App, that made it possible for us to test our hypothesis thanks to its instantiation as OX-ELEOS, an application-specific FTL. We could reuse the major part of our generic FTL implementation to build OX-ELEOS. We could also adopt a white-box approach that made it possible to bypass modules and minimize overhead for OX-ELEOS on the read path. Much more work is needed to fully explore the benefits and drawbacks of application-specific FTLs. We have, however, established that OX is a viable framework to conduct such an exploration. The lessons we have learned should benefit researchers conducting future work as well as practitioners considering computational storage.

## 8.2 Lessons Learned

Throughout this thesis, we have learned lessons about computational storage, the design of an FTL and the design of application-specific FTLs.

**Computational Storage**

- The storage controller is a bottleneck when moving data back and forth between a host and the storage media.
  - It is necessary to bypass the CPU whenever possible when transferring data. Techniques to bypass the CPU include hardware acceleration (e.g., hardware support for RDMA), zero-copy abstractions (e.g., AF_XDP sockets) and a white box approach to software controller software that makes it possible to bypass unnecessary functions on the data path.
  - When choosing a computational storage platform, emphasis should be laid on the presence of hardware accelerators and on the performance of RAM accesses from the storage controller.
- NVMe is an appropriate protocol for accessing computational storage. As we saw in Chapter 3, transferring data with NVMe over fabrics puts a high burden on the storage controller. More verbose and less streamlined protocols would be worse. For example, we considered traditional RPC as a potential protocol

for accessing computational storage SSDs. Our experience with NVMe shows that this would introduce significant limitations in terms of performance.

**FTL Design**

- The solutions we implemented for garbage collection and wear-leveling are straightforward. Even if there exist more advanced techniques to improve performance (e.g., grouping cold and hot pages to minimize the garbage collection overhead and improve resource utilization), it is not clear that those techniques would bring very significant performance advantages. At least, the overhead of the straightforward solutions is not a significant problem.
- The multi-level mapping table that we propose is an elegant way to reduce the memory footprint of the mapping table for fixed size pages and to organize checkpoints. It remains to be seen how it can be adapted for variable size pages.

**Application-Specific FTLs**

- Cross-layer optimization makes it possible to relax the transactional properties provided by ELEOS (there is no shadow paging mechanism, and it is thus possible for a reader to access old and new versions of an updated multi-page transaction) because those properties are enforced in the upper layers of the system, on the host. This is a crucial advantage of an application-specific FTL. There is a need to quantify this advantage by comparing computational storage with application-specific FTL and computational storage with generic FTL. This is a topic for future work.

## 8.3 Future Work

In the short term, future work includes further work on ELEOS, work on other computational storage platforms, and work on other forms of application-specific FTLs. The overall goal is to explore further the benefits and drawbacks of application-specific FTLs for computational storage.

ELEOS, for now, offers a log-structured interface for fixed sized objects. The next step is to define a log-structured SSD for variable-size multipart objects (composed of a base page of fixed size and variable size delta modifications) as proposed in the original LLAMA paper. Variable-size objects are a key challenge for the mapping component. Indeed, objects might be smaller than the unit of access to the physical address space. Additional bits are necessary to represent such mappings and it is necessary to manage a much larger mapping table. Multipart object embedded within a large write operation introduces the need to consider a form of nested transaction.

Immediate future work also includes experimenting with a SST-100 SoC from Broadcom, rather than the DFC, in order to benefit from hardware accelerated RDMA and better RAM access performance from the ARM SoC. We expect a significant improvement in the performance of ELEOS on OX when multiple threads are used on a multicore host.

Finally, we have already started work on another form of application-specific FTL. We are developing an application-specific FTL on computational storage for RocksDB. This is an area where we will be able to compare a KV-SSD (a product from Samsung with a specialized FTL), a version of SmartSSD (a computational storage SSD from Samsung with an FPGA on top of a generic FTL), and an OX-based LSM-specific FTL on a SoC on top of open-channel SSDs. This is a great topic for future work.

# Bibliography

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy. "Design Tradeoffs for SSD Performance". In: *USENIX 2008 Annual Technical Conference*. ATC'08. Boston, Massachusetts: USENIX Association, 2008, pp. 57–70.

[2] A. Ailamaki, E. Liarou, P. Tözün, D. Porobic, and I. Psaroudakis. *Databases on Modern Hardware: How to Stop Underutilization and Love Multicores*. 2017.

[3] M. Andreessen. "Why Software Is Eating The World". In: *The Wall Street Journal* (Aug. 2011).

[4] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. "Bztree: A High-performance Latch-free Range Index for Non-volatile Memory". In: *Proc. VLDB Endow.* 11.5 (Jan. 2018), pp. 553–565.

[5] S. Balakrishnan, R. Black, A. Donnelly, P. England, A. Glass, D. Harper, S. Legtchenko, A. Ogus, E. Peterson, and A. Rowstron. "Pelican: A Building Block for Exascale Cold Data Storage". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 351–365.

[6] M. Bjørling, J. González, and P. Bonnet. "LightNVM: The Linux Open-channel SSD Subsystem". In: *Proceedings of the 15th Usenix Conference on File and Storage Technologies*. FAST'17. Santa clara, CA, USA: USENIX Association, 2017, pp. 359–373.

[7] M. Bjørling, M. Wei, J. Madsen, J. González, S. Swanson, and P. Bonnet. "AppNVM: A software-defined, application-driven SSD". In: *Non-Volatile Memory Workshop (NVMW '15)* (Mar. 2015).

[8] L. Bouganim and P. Bonnet. "Flash Device Support for Database Management". In: *5th Biennal Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California, United States, Jan. 2011, pp. 1–8.

[9] L. Bouganim, B. Þ. Jónsson, and P. Bonnet. "uFLIP: Understanding Flash IO Patterns". In: *4th Biennial Conference on Innovative Data Systems Research (CIDR)* (2009).

[10] Y. Cai, E. F. Haratsch, O. Mutlu, and K. Mai. "Error Patterns in MLC NAND Flash Memory: Measurement, Characterization, and Analysis". In: *Proceedings -Design, Automation and Test in Europe, DATE* (Mar. 2012).

[11] L.-P. Chang and T.-W. Kuo. "An Adaptive Striping Architecture for Flash Memory Storage Systems of Embedded Systems". In: *Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'02)*. RTAS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 187–.

[12] F. Chen, R. Lee, and X. Zhang. "Essential Roles of Exploiting Internal Parallelism of Flash Memory Based Solid State Drives in High-speed Data Processing". In: *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture*. HPCA '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 266–277.

[13] F. Chen, T. Luo, and X. Zhang. "CAFTL: A Content-aware Flash Translation Layer Enhancing the Lifespan of Flash Memory Based Solid State Drives". In: *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*. FAST'11. San Jose, California: USENIX Association, 2011, pp. 6–6.

[14] J.-Y. Choi, E. H. Nam, Y. J. Seong, J. H. Yoon, S. Lee, H. S. Kim, J. Park, Y.-J. Woo, S. Lee, and S. L. Min. "HIL: A Framework for Compositional FTL Development and Provably-Correct Crash Recovery". In: *ACM Trans. Storage* 14.4 (Dec. 2018), 36:1–36:29.

[15] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song. "A Survey of Flash Translation Layer". In: *J. Syst. Archit.* 55.5-6 (May 2009), pp. 332–343.

[16] N. Dayan, M. Athanassoulis, and S. Idreos. "Monkey: Optimal Navigable Key-Value Store". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: ACM, 2017, pp. 79–94.

[17] N. Dayan and S. Idreos. "Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 505–520.

[18] J. Dean and L. A. Barroso. "The Tail at Scale". In: *Communications of the ACM* 56 (2013), pp. 74–80.

[19] C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. "Hekaton: SQL Server's Memory-optimized OLTP Engine". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: ACM, 2013, pp. 1243–1254.

[20] J. Do, D. Lomet, and I. L. Picoli. "High IOPS via Log Structuring in an SSD Controller". In: *Non-Volatile Memory Workshop (NVMW '19)* (Mar. 2019).

[21] J. Do, D. Lomet, and I. L. Picoli. "Improving SSD I/O Performance via Controller FTL Support for Batched Writes". In: *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN '19* (July 2019).

[22] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. "Optimizing Space Amplification in RocksDB". In: *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. 2017.

[23] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC '10*. Sept. 2010, pp. 143–154.

[24] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker. "Network Requirements for Resource Disaggregation". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, 2016, pp. 249–264.

[25] J. González. *Towards Application Driven Storage*. Talk at LinuxCon Europe 2015.

[26] J. González, M. Bjørling, S. Lee, C. Dong, and Y. Ronnie Huang. "Application-Driven Flash Translation Layers on Open-Channel SSDs". In: *Non-Volatile Memory Workshop (NVMW '16)* (Mar. 2016).

[27] J. Gray. *Put Everything in the Disk Controller*. Talk at NASD workshop (1998).

[28] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992.

[29] L. Grupp, J. Davis, and S. Swanson. "The Bleak Future of NAND Flash Memory". In: *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. May 2012.

[30] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn. "RDMA over Commodity Ethernet at Scale". In: *Proceedings of the 2016 ACM SIGCOMM Conference*. SIGCOMM '16. Florianopolis, Brazil: ACM, 2016, pp. 202–215.

[31] J. Guo, Y. Hu, B. Mao, and S. Wu. "Parallelism and Garbage Collection Aware I/O Scheduler with Improved SSD Performance". In: *IEEE International Parallel and Distributed Processing Symposium (IPDPS '17)*. May 2017.

[32] A. Gupta, Y. Kim, and B. Urgaonkar. "DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings". In: *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XIV. Washington, DC, USA: ACM, 2009, pp. 229–240.

[33] Z. Guz, H. ( Li, A. Shayesteh, and V. Balakrishnan. "NVMe-over-fabrics Performance Characterization and the Path to Low-overhead Flash Disaggregation". In: *Proceedings of the 10th ACM International Systems and Storage Conference*. SYSTOR '17. Haifa, Israel: ACM, 2017, 16:1–16:9.

[34] S. S. Hahn, J. Kim, and S. Lee. "To Collect or Not to Collect: Just-in-time Garbage Collection for High-performance SSDs with Long Lifetimes". In: *Proceedings of the 52Nd Annual Design Automation Conference*. DAC '15. San Francisco, California: ACM, 2015, 191:1–191:6.

[35] M. Hao, G. Soundararajan, D. Kenchammana-Hosekote, A. A. Chien, and H. S. Gunawi. "The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments". In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 263–276.

[36] J. He, S. Kannan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. "The Unwritten Contract of Solid State Drives". In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys '17. Belgrade, Serbia: ACM, 2017, pp. 127–144.

[37] Y. Jae Seong, E. Nam, J. Yoon, H. Kim, J.-y. Choi, S. Lee, Y. Hyun Bae, J. Lee, Y. Cho, and S. Min. "Hydra: A Block-Mapped Parallel Flash Memory Solid-State Disk Architecture". In: *IEEE Trans. Computers* 59 (July 2010), pp. 905–921.

[38] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson. "KAML: A Flexible, High-Performance Key-Value SSD". In: *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (2017), pp. 373–384.

[39] K. Joshi, P. Choudhary, and K. Yadav. "Enabling NVMe WRR Support in Linux Block Layer". In: *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*. HotStorage'17. Santa Clara, CA: USENIX Association, 2017, pp. 22–22.

[40] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. "BlueDBM: An Appliance for Big Data Analytics". In: *SIGARCH Comput. Archit. News* 43.3 (June 2015), pp. 1–13.

[41] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. "H-Store: A high-performance, distributed main memory transaction processing system". In: *Proc. VLDB Endow.* 1 (Aug. 2008), pp. 1496–1499.

[42] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho. "The Multi-streamed Solid-State Drive". In: *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*. Philadelphia, PA: USENIX Association, 2014.

[43] A. Kemper and T. Neumann. "HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots". In: *Proceedings of the*

*2011 IEEE 27th International Conference on Data Engineering*. ICDE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 195–206.

[44] H. S. Kim, E. H. Nam, J. H. Yun, S. Lee, and S. L. Min. "P-BMS: A Bad Block Management Scheme in Parallelized Flash Memory Storage Devices". In: *ACM Trans. Embed. Comput. Syst.* 16.5s (Sept. 2017), 140:1–140:19.

[45] J. Kim, D. Lee, and S. H. Noh. "Towards SLO Complying SSDs Through OPS Isolation". In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. FAST'15. Santa Clara, CA: USENIX Association, 2015, pp. 183–189.

[46] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. "NVWAL: Exploiting NVRAM in Write-Ahead Logging". In: *SIGOPS Oper. Syst. Rev.* 50.2 (Mar. 2016), pp. 385–398.

[47] H. Kimura. "FOEDUS: OLTP Engine for a Thousand Cores and NVRAM". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. Melbourne, Victoria, Australia: ACM, 2015, pp. 691–706.

[48] J. Lee, M. Muehle, N. May, F. Färber, V. Sikka, H. Plattner, and J. Krueger. "High-Performance Transaction Processing in SAP HANA". In: *IEEE Data Engineering Bulletin* 36 (Jan. 2013), pp. 28–33.

[49] M. Lee, D. H. Kang, M. Lee, and Y. I. Eom. "Improving Read Performance by Isolating Multiple Queues in NVMe SSDs". In: *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*. IMCOM '17. Beppu, Japan: ACM, 2017, 36:1–36:6.

[50] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and A. Arvind. "Application-managed Flash". In: *Proceedings of the 14th Usenix Conference on File and Storage Technologies*. FAST'16. Santa Clara, CA: USENIX Association, 2016, pp. 339–353.

[51] V. Leis, M. Haubenschild, A. Kemper, and T. Neumann. "LeanStore: In-Memory Data Management beyond Main Memory". In: *Proceedings of the 2018 IEEE 34th International Conference on Data Engineering*. ICDE '18. Apr. 2018, pp. 185–196.

[52] V. Leis, M. Haubenschild, and T. Neumann. "Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method". In: *Bulletin of the IEEE Computer Society Technical Commitee on Data Engineering* (2019).

[53] V. Leis, A. Kemper, and T. Neumann. "The Adaptive Radix Tree: ARTful Indexing for Main-memory Databases". In: *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*. ICDE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 38–49.

[54] V. Leis, F. Scheibner, A. Kemper, and T. Neumann. "The ART of Practical Synchronization". In: *Proceedings of the 12th International Workshop on Data Management on New Hardware*. DaMoN '16. San Francisco, California: ACM, 2016, 3:1–3:8.

[55] J. J. Levandoski, D. B. Lomet, and S. Sengupta. "The Bw-Tree: A B-tree for New Hardware Platforms". In: *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*. ICDE '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 302–313.

[56] J. Levandoski, D. Lomet, and S. Sengupta. "LLAMA: A Cache/Storage Subsystem for Modern Hardware". In: *Proc. VLDB Endow.* 6.10 (Aug. 2013), pp. 877–888.

[57] D. Lomet. "Cost/Performance in Modern Data Stores: How Data Caching Systems Succeed". In: *Proceedings of the 14th International Workshop on Data Management on New Hardware*. DAMON '18. Houston, Texas: ACM, 2018, 9:1–9:10.

[58] Y. Lu, J. Shu, and W. Zheng. "Extending the Lifetime of Flash-based Storage Through Reducing Write Amplification from File Systems". In: *Proceedings of the 11th USENIX Conference on File and Storage Technologies*. FAST'13. San Jose, CA: USENIX Association, 2013, pp. 257–270.

[59] S. A. F. Lund. *LibLightNVM*. https://github.com/OpenChannelSSD/liblightnvm.

[60] D. Makreshanski, J. Levandoski, and R. Stutsman. "To Lock, Swap, or Elide: On the Interplay of Hardware Transactional Memory and Lock-free Indexing". In: *Proc. VLDB Endow.* 8.11 (July 2015), pp. 1298–1309.

[61] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. "A Large-Scale Study of Flash Memory Failures in the Field". In: *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '15. Portland, Oregon, USA: ACM, 2015, pp. 177–190.

[62] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. "ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging". In: *ACM Trans. Database Syst.* 17.1 (Mar. 1992), pp. 94–162.

[63] R. Mueller, J. Teubner, and G. Alonso. "Data Processing on FPGAs". In: *Proc. VLDB Endow.* 2.1 (Aug. 2009), pp. 910–921.

[64] *NVMe Specifications*. https://nvmexpress.org/resources/specifications/.

[65] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang. "SDF: Software-defined Flash for Web-scale Internet Storage Systems". In: *SIGARCH Comput. Archit. News* 42.1 (Feb. 2014), pp. 471–484.

[66] C. Park, W. Cheon, J. Kang, K. Roh, W. Cho, and J.-S. Kim. "A Reconfigurable FTL (Flash Translation Layer) Architecture for NAND Flash-based Applications". In: *ACM Trans. Embed. Comput. Syst.* 7.4 (Aug. 2008), 38:1–38:23.

[67] K. Patiejunas. *Freezing Exabytes of Data at Facebook's Cold Storage*. Talk at Facebook 2014.

[68] I. L. Picoli, P. Bonnet, and P. Tözün. "LSM Management on Computational Storage". In: *Proceedings of the 15th International Workshop on Data Management on New Hardware, DaMoN '19* (July 2019).

[69] I. L. Picoli, C. V. Pasco, and P. Bonnet. "Beyond Open-Channel SSDs". In: *Non-Volatile Memory Workshop (NVMW '17)* (Mar. 2017).

[70] I. L. Picoli, P. Tözün, A. Wasowski, and P. Bonnet. "Programming Storage Controllers with OX". In: *Non-Volatile Memory Workshop (NVMW '19)* (Mar. 2019).

[71] I. L. Picoli, C. V. Pasco, B. Þ. Jónsson, L. Bouganim, and P. Bonnet. "uFLIP-OC: Understanding Flash I/O Patterns on Open-Channel Solid-State Drives". In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. Mumbai, India: ACM, 2017, 20:1–20:7.

[72] W. Pugh. "Skip Lists: A Probabilistic Alternative to Balanced Trees". In: *Commun. ACM* 33.6 (June 1990), pp. 668–676.

[73] J. Ren, Q. Hu, S. Khan, and T. Moscibroda. "Programming for Non-Volatile Main Memory Is Hard". In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. APSys '17. Mumbai, India: ACM, 2017, 13:1–13:8.

[74] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. "Managing Non-Volatile Memory in Database Systems". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 1541–1555.

[75] M. Rosenblum and J. K. Ousterhout. "The Design and Implementation of a Log-structured File System". In: *ACM Trans. Comput. Syst.* 10.1 (Feb. 1992), pp. 26–52.

[76] Samsung. *KVSSD*. https://github.com/OpenMPDK/KVSSD.

[77] C. Sauer, G. Graefe, and T. Härder. "Instant Restore After a Media Failure". In: *Advances in Databases and Information Systems. (ADBIS '17)*. Feb. 2017, pp. 311–325.

[78] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. "Willow: A User-Programmable SSD". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, 2014, pp. 67–80.

[79] J.-Y. Shin, Z.-L. Xia, N.-Y. Xu, R. Gao, X.-F. Cai, S. Maeng, and F.-H. Hsu. "FTL Design Exploration in Reconfigurable High-performance SSD for Server Applications". In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS '09. Yorktown Heights, NY, USA: ACM, 2009, pp. 338–349.

[80] Y. H. Song. *Cosmos+OpenSSD: A NVMe-based Open Source SSD Platform*. Flash Memory Summit (2016), Santa Clara, CA, USA.

[81] B. Stephens, A. Singhvi, A. Akella, and M. Swift. "Titan: Fair Packet Scheduling for Commodity Multiqueue NICs". In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '17. Santa Clara, CA, USA: USENIX Association, 2017, pp. 431–444.

[82] M. Stonebraker and A. Weisberg. "The voltdb main memory dbms". In: *IEEE Data Eng. Bull.* 36 (Jan. 2013), pp. 21–27.

[83] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. Cong. "An Efficient Design and Implementation of LSM-tree Based Key-value Store on Open-channel SSD". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: ACM, 2014, 16:1–16:14.

[84] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. "Building a Bw-Tree Takes More Than Just Buzz Words". In: *Proceedings of the 2018 International Conference on Management of Data*. SIGMOD '18. Houston, TX, USA: ACM, 2018, pp. 473–488.

[85] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002.

[86] J. Yang, N. Plasson, G. Gillis, N. Talagala, and S. Sundararaman. "Don't Stack Your Log On My Log". In: *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14)*. Broomfield, CO: USENIX Association, 2014.

[87] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen. "Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes". In: *Proceedings of the 2016 International Conference on Management of Data*. SIGMOD '16. San Francisco, California, USA: ACM, 2016, pp. 1567–1581.

[88] F. Zhu. *Toward the Large Deployment of Open Channel SSD*. Flash Memory Summit (2019), Santa Clara, CA, USA.

[89] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang. "Congestion Control for Large-Scale RDMA Deployments". In: *SIGCOMM Comput. Commun. Rev.* 45.4 (Aug. 2015), pp. 523–536.