Operating System Support for High-Performance Solid State Drives

Matias Bjørling

IT University
of Copenhagen

A dissertation submitted to the PhD school at the IT University of Copenhagen for the degree of Doctor of Philisophy.

Submitted: May 2015, Copenhagen
Defended: August 24th 2015, Copenhagen
Final version: January 24th 2016, Copenhagen

Advisors:
Philippe Bonnet, IT University of Copenhagen, Denmark
Luc Bouganim, INRIA Paris-Rocquencourt, France

Evaluation committee:
Peter Sestoft, IT University of Copenhagen, Denmark
Gustavo Alonso, ETH Zürich, Switzerland
Alejandro Buchmann, TU Darmstadt, Germany

Abstract

The performance of Solid State Drives (SSD) has evolved from hundreds to millions of I/Os per second in the past three years. Such a radical evolution is transforming both the storage and the software industries. Indeed, software designed based on the assumption of slow IOs has become the bottleneck in the era of high-performance SSDs.

In the context of the CLyDE project, a collaboration between the IT University of Copenhagen and INRIA Rocquencourt, we recognized the mismatch between software designed for hard drives and high- performance SSDs. This thesis focuses on the role of the operating system in reducing the gap, and enabling new forms of communication and even co-design between applications and high-performance SSDs.

More specifically, we studied the storage layers within the Linux kernel. We explore the following issues: (i) what are the limitations of the legacy block interface and its cost in the context of high-performance SSDs? (ii) Can the Linux kernel block layer cope with high-performance SSDs? (iii) What is an alternative to the legacy block interface? How to explore the design space? (iv) How about exposing SSD characteristics to the host in order to enable a form of application-SSD co-design? What are the impacts on operating system design? (v) What would it take to provide quality of service for applications requiring millions of I/O per second?

The dissertation consists of six publications covering these issues. Two of the main contributions of this thesis (the multi-queue block layer and LightNVM) are now an integral part of the Linux kernel.

Acknowledgments

The following dissertation concludes my studies at IT University of Copenhagen. It is a synopsis of six papers: (i) Three papers published and one under review. (ii) Two are workshop papers; one born out of the need of simulating SSDs, and (iii) one while Michael Wei visiting ITU research group December 2014. The years have gone by and it is incredible how fast the time have gone. It has been a pleasure to dive deep into a topic, meet very talented researchers and explore a subject at my own pace. For all this:

Thanks to my adviser, Philippe Bonnet, for guiding me through university life. We have been working together since his first database course at Copenhagen University and I did not think for a second that this would be where we ended up ten years later. It has been a great ride.

Thanks to Luc Bouganim from INRIA and Steven Swanson from UCSD. The former for letting me visit INRIA and long discussions about SSDs. The latter for hosting my visit at UCSD. It was great to join his lab for four months. His students were very welcoming, and I enjoyed my stay immensely.

Thanks to Jens Axboe for guiding me through the industry and opening a lot of opportunities that else would not have been possible.

Freja Krob Koed Eriksen and Christina Rasmussen from the ITU Ph.D. School. They have been a tremendous help while doing the Ph.D. I am grateful for having such wonderful people taking care of us *kids*.

Thanks to my wife, Trine, for allowing me to pursue my dreams and letting me spend the necessary time to go deep into a particular topic. Thanks for all the great days and nights we spent together.

At last, thanks to my fellow lab mates; Javier González, Jonathan Fürst, Aslak Johansen, and Niv Dayan from IT University of Copenhagen for being part of a great research environment. Special thanks to Michael Wei at UCSD, that made the stay at UCSD so much more fun. I never have had so much fun discussing new ideas as much as with him. Wish you all the best of luck.

Contents

Chapter 1

Introduction

1.1 Context

Millions of petabytes of data are generated each year and must be stored and accessed efficiently for business, entertainment, government or science. From the 80s until a couple of years ago, database systems have relied on magnetic disks as secondary storage.

Successive generations of magnetic disks have complied with two simple axioms: (1) locality in the logical address space is preserved in the physical address space; (2) sequential access is much faster than random access. The former axiom reflects the simple logic embedded in magnetic disk controllers. The latter axiom is shaped by the physical characteristics of rotational devices: startup costs involving disk head movement are high while running costs involving rotation of the disk platter are low: Random accesses are slow, while sequential accesses are fast.

As long as magnetic disks remained the sole medium for secondary storage, the block device interface proved to be a very robust operating system abstraction that allowed the kernel to hide the complexity of I/O management without sacrificing performance. The block device interface is a simple memory abstraction based on read and write primitives and a flat logical address space (i.e., an array of sectors). Since the advent of Unix, the stability of disks characteristics and interface have guaranteed the timelessness of major database system design decisions, i.e., pages are the unit of I/O with an identical representation of data on-disk and in-memory; random accesses are avoided (e.g., query processing algorithms) while sequential accesses are favored (e.g., extent-based allocation, clustering).

Today, the advent of solid state disks (SSD) based on NAND flash memory has transformed the performance characteristics of secondary storage. SSDs rely on tens of flash chips wired in parallel to deliver high throughput (hundreds of MB/s to GB/s) and latency in the microsecond range. Over the last five years, the transition has been very fast from magnetic disks providing throughput of 100s of random I/O per second (IOPS), to SSDs providing hundreds of thousands of IOPS (OCZ Vertex 3) and beyond a million IOPS (Fusion-io ioDrive Octal) for a single device. Similarly, sequential throughput for a single device has increased from hundred of MB/s to GB/s.

The adoption of flash-based SSDs has created a mismatch between the simple disk model that underlies the design of today's database systems and the complex flash devices of today's computers. The CLyDE (Cross Layer Database Engine for flash-based SSDs) research project, a collaboration between the IT University of Copenhagen and the SMIS Project at INRIA Paris-Rocquencourt, was funded in 2011 by the Danish Independent Research Council to tackle this issue.

CLyDE is based on the insight that the strict layering that has been so successful for designing database systems on top of magnetic disks is no longer applicable with SSDs. In other words, the complexity of SSDs cannot and should not be abstracted away if it results in unpredictable and suboptimal performance. This PhD project has been funded by the CLyDE project. This thesis, with the software I produced and the papers collected in this manuscript, constitute a significant part of its contribution.

1.2 Solid State Drive Basics

Before I proceed with the focus of my thesis, I need to introduce the basics of SSD design. Note that SSD basics are repeatedly covered throughout the papers that constitute this manuscript. I describe SSD internals in details in Chapter 2, where I also discuss misconceptions about their performance characteristics.

The non-volatile storage in an SSD is NAND flash chips. They provide high performance, at low energy consumption. Unfortunately, flash chips have severe constraints [1, 4]: (C1) Write granularity. Writes must be performed at a page granularity (4KB-16KB); (C2) Erase before write. A costly erase operation must be performed before overwriting a flash page. Even worse, erase operations are only performed at the granularity of a flash block (typically 64-256 flash pages); (C3) Sequential writes within a block. Writes must be performed sequentially within a flash block in order to minimize write errors resulting from the electrical side effects of writing a series of cells; and (C4) Limited lifetime. Flash chips can support thousands erase operations per block.

The software embedded into flash devices is called Flash Translation Layer (FTL). Its goal is to hide flash chip constraints (erase-before-write, limited number of erase-write cycles, sequential page-writes within a flash block) so that flash devices can support the block device interface. An FTL provides address translation, wear leveling and strives to hide the impact of updates and random writes based on observed update frequencies, access patterns, or temporal locality.

1.3 Problem

With magnetic disks, hardware was the bottleneck in the I/O stack. Software should find interesting work to do while waiting for I/O to complete. With SSDs however, it is software that has become the bottleneck. The software that composes the I/O stack, including FTL, device driver, operating system block layer to applications, must be streamlined and possibly reorganized in order to keep up with hardware performance.

For years, the transport interface has bounded the performance of the underlying storage hardware. The advent of SSDs quickly exposed the relatively slow transfer speeds and latencies in these legacy protocols. For example, SATA/SAS limited the throughput to 300-600 MB/s.

In 2007, Intel introduced the NVM Express [13] (NVMe) protocol to fully leverage the PCI-e transport interface for storage devices. The PCI-e bus, with its 1GB/s per lane in PCI gen3 standardilization enables SSDs to scale their throughput. In terms of latency, PCI-e roundtrip is around 1us for a 4K I/O request [3].

NVMe exposes SSD parallelism so that it can be fully utilized by host's hardware and software. The first commercially available controllers supporting NVMe were released in 2012. Today, NVMe has become the standard interface for high-performance SSD on top of PCI-e.

I have focused my work on the consequences of the introduction of high speed interfaces on system design. My core assumption is that the operating system should be redesigned to fully utilize the parallelism of SSDs exposed via NVMe. My thesis is that the operating system should remain a mediator between diverse applications and a variety of high-performance SSDs. The fundamental question is which form of mediation should the operating system perform. More specifically, I have studied the following questions:

1. *Is the block device interface still relevant as a secondary storage abstraction? What are the alternatives?* The block device interface hides the complexity of the underlying storage devices behind a very simple memory abstraction. System design has been based on the assumption of a stable performance contract exists across the block device interface. Such a contract ((i) contiguity in the logical address space is reflected on disk, and (ii) sequential access is much faster than random access) has driven the design of system software on both sides of the block device interface.

   For magnetic drives, rigorous performance models have been extensively studied [20, 1] based on the mechanical characteristics of hard-drives (seek time, number of spindles, rotations per minute). For SSDs however, my previous work has contributed to showing the limits of SSD performance modeling. Together with Luc Bouganim and Philippe Bonnet, I showed that the performance characteristics and energy profiles vary significantly across SSDs, and that performance varies in time based on the history of accesses.

   SSD performance fluctuations are a product of the characteristics of flash chips and of the complexity of the FTL that handles the mismatch that exists between the interface that the SSD exposes (read/write) and the internal interface for flash chips (read, write, erase).

   Back in 2011, Bill Nesheim, then Oracle VP for Solaris Platform Engineering, illustrated the consequences of SSD performance fluctuations during his keynote at the Flash Memory Summit [17]: “An average latency of about 1ms is irrelevant when the application is waiting for the 100+ ms outlier to complete". In 2015, this is still true and poses a significant challenge for system design.

2. *What are the bottlenecks in the operating system I/O stack? How to remove them?* In the CLyDE project, the initial focus lied on cross-layer optimizations between database system and SSD. However, we quickly made two observations. First, it became apparent that we could not oversee the crucial role of the operating system in the I/O stack. Indeed, as SSD latency decreases, the overhead of the operating system software stack becomes a significant part of the time it takes to process I/Os [2]. Second, we realized that key-value stores and other types of data management systems had become important for increasingly large classes of applications and that they should also benefit from SSD performance. Here again, the operating system plays a central role in supporting various types of applications. When I started my PhD, several initiatives had been taken to improve the communication between operating system and SSD: (a) the Trim command [23, 14] had been introduced to notify SSDs of invalid sectors, (b) file-systems and applications had the option to know whether an SSD was used as secondary storage and (c) careful optimization of hot code paths for each I/O had been applied. These initiatives increased I/O throughput and lowered I/O latency. However, the Linux operating software stack still exhibited a significant overhead as it had been designed for a single core and was heavily optimized for traditional hard-drives. I thus chose to focus on the following question: How to get Linux to scale to millions IOPS and microsecond I/O latency?

1.4 Approach

The approach I have taken to study these problems is rooted in experimental computer science [12]. First, the core of this thesis is based on the design, implementation and experimental evaluation of operating system components. Second, throughout this thesis, I have formulated hypotheses, and then directed effort at trying to confirm or falsify them qualitatively, or quantitatively when appropriate.

Another important aspect of the approach I have taken to my thesis concerns my contribution to open source software in general and to the Linux kernel in particular. Early on in my thesis, I realized that I was fortunate to work in an area where deep changes were taking places, and that I might have an opportunity to contribute to some of these changes. In order to maximize the impact of my work I wanted to engage the industry as well as the research community. This is why I spent 4 months as an intern at Fusion-io in the summer 2012.

I had the privilege to work with Jens Axboe, previously at Fusion-io, now at Facebook. Jens has been the maintainer of the Linux block layer for many years. This internship and the collaboration with Jens enabled me to establish a pattern that I followed throughout the thesis based on (1) contributions to the Linux kernel and (2) a principled design and experimental evaluation that could lead to scientific publication.

1.5 Contributions

This manuscript is composed of 6 publications I co-authored throughout my thesis. Together, they form a contribution that can be represented as a bottom up sweep of the operating system's I/O stack in six steps.

Step 1: We observe that the interface between SSD and the host is restricted to a narrow block interface that exposes a simple memory abstraction (read/write) on a flat address space. This interfaces leaves no space for communicating the intent of the applications that run on the host. There is thus a complete disconnect between what the disk knows and what the host knows. For magnetic disks, this disconnect was bridged by an implicit performance contract. For SSD, there is no stable performance contract.

In the first paper, published at CIDR in 2013, we debunk some of the common myths about SSD performance. We argue that the block device interface has become an obstacle to principled system design due to the non existence of a generic performance contract. We conclude this paper by giving a vision of how database systems can efficiently interact with SSDs and we argue that a deep redesign of the entire I/O stack, including the operating system, is needed.

Step 2: Having laid the foundation for the upcoming research, we tackle the block layer within the operating system. Research papers [2, 9, 3] have shown significant overhead per I/O within the operating system. We expect that hardware performance will continue to outgrow software performance for the foreseeable future. Moving our attention to the OS, there are several parts that can be optimized.

The block I/O interface within a kernel is usually divided into several parts: (a) Device driver that handle communication to and from the block device, (b) the block layer that handle fairness, scheduling, etc. of an I/O, and (c) libraries that are exported to the user-space through asynchronous and synchronous mechanisms. The device drivers are device specific and we leave any research on optimizations to the device vendors. We instead focus on the block layer and submission/completion libraries.

This work was published at SYSTOR in 2013 and is a direct result of my internship at Fusion-io. The paper argues that the current block layer within the Linux kernel is inadequate for current and future SSDs. We present the design of a new block layer, that allows the number of I/O per seconds available to scale with the number of cores in a system and with the device itself. Additionally, the engineering work reported in the paper led to significantly reduce the time spent by I/Os traversing the block layer. The work was integrated in the Linux kernel and upstreamed with Linux Kernel 3.13.

Step 3: In order to continue our endeavor, we needed to find new ways to explore the design space of cross layer optimizations involving operating system and SSDs. We considered simulation or custom-made hardware platforms. Both allow us to test our ideas. SSD simulators, such as DiskSim [1], FlashSim [3] and EagleTree [11], that Niv Dayan built as an extension of the simulator I built during my MSc thesis, allow us to prototype ideas quickly and understand their implications. Unfortunately, all the simulators are defined in user-space. None of them allow us prototype operating system kernel changes without significant latency overhead. Conversely, hardware platforms, such as OpenSSD [13, 16, 22] and Moneta [2], let us keep the code paths through the software stack without breaking compatibility, but requires significant plumping to yield a result. Moneta also assumes that applications access a SSD directly, without mediation from the operating system (an approach which is not a good match for our core thesis).

As a result, we decided to implement our own simulator, that performs I/O scheduling at run-time and allows users to directly measure the performance impact of any software and hardware design decision. The paper included in this thesis was presented at the Non-Volatile Memory Workshop (NVMW) in 2014.

Step 4: Moving back to the original research question around the narrow block interface for SSDs, we extend the ideas that underlie the design of our simulator into an extension of the Linux block layer. A lot of interesting research towards cross-layer stack ideas have been proposed [10, 9] as well as extensions of block I/O interface [2, 18]. These approaches look at the overhead and guarantees of the OS, and moves parts or all OS logic into the SSD or the application. We take the dual approach. We wish to explore how far can we move SSD processing into the host software stack in order to improve performance.

We extend the narrow block interface, with additional commands so that the SSD exposes its physical address space and flash commands directly to the host. We denote such SSDs, open-channel SSDs. Such SSDs were envisaged by Philippe Bonnet and Luc Bouganim back in 2011 [19]. At the time it was science fiction. Today, all major Internet service and public cloud providers rely on their own custom-made open channel SSDs. SSD controller vendors are competing to define products adapted for such SSDs. Memblaze already commercializes a version of its PBlaze3 SSD as open channel SSDs.

We defined the LightNVM specification[1] , that describes a command set to communicate the characteristics of SSDs, and allows the host to implement a generic layer, which can either be used to implement a host-side FTL, similarly to the Virtual Storage Layer provided by Fusion-io, or move FTLs (we call these targets) into the applications itself. LightNVM allows applications to work together on the same address space, and thus maintain access to storage, without going though the kernel (if that is the wish), but still consult the kernel for chunk accesses (1-16MB). This provides a less granular split of accesses and thus allows the CPU overhead to scale with the speed of the storage device. Allowing applications to directly optimize their data placement on top of SSDs, removes the need for intermediate layers, such as file-systems. LightNVM[2] is currently being prepared for the Linux kernel and will soon hit upstream, with several SSD vendors implementing the command set to bringing

[1] LightNVM Specification Draft: `http://goo.gl/BYTjLI`

[2] Website: `http://github.com/OpenChannelSSD`

application-defined storage to the next-generation storage systems. The paper included in this thesis will be submitted to FAST 2016.

Step 5: Looking forward, we consider the problem of processing I/Os with future SSDs whose latency lies in the microsecond range with throughput of millions of I/Os per seconds. Our work was inspired by the FAST 2012 paper [4] introducing pull versus poll. For SSDs that whose latency lied in the 5-10us range, it was not very clear if a pull or poll model should be used. By introducing speculation into the mix, we enable applications to keep on working, without waiting for I/O completion. The paper included in this thesis is a collaboration with Michael Wei from UCSD. The paper was published at the Usenix Annual Technical Conference in 2014.

Step 6: For future high-performance SSDs, we looked at what kind of processing could be moved back to the SSD, and possibly encoded in hardware, in order to basically get the operating system out of the data path, with quality of service guarantees enforced on the control path. Today, performing fairness is fully disregarded for SSDs (CFQ and other I/O schedulers are very often disabled for SSDs). We argue that a rule-based approach, which could be efficiently supported in hardware, would be a significant improvement of the state of the art. The position paper included in this thesis was presented at the Non-Volatile Memory Workshop in 2015 and constitutes the basis for a patent application.

In summary, this thesis has the following four major contributions:

- Questioning whether the narrow block interface is the right interface for the future (Paper 1, 6);
- Defining a new block layer within the Linux Kernel to add scalability for the storage stack in the Linux kernel (Paper 2);
- Defining a new software framework for controlling SSDs by moving data placement decisions into the host (Paper 4);
- At last, opportunities for enabling faster storage accesses and intelligent I/O processing on SSDs with microsecond latency (Paper 3, 5).

1.6 Structure of the Thesis

The thesis is structured around these 6 contributions. Paper 1 “Necessary Death of the Block Device Interface” [4] presented at CIDR 2013. Paper 2, together with Jens Axboe, “Linux Block IO: Introducting Multi-queue SSD Access on Multi-core Systems” [3] presented at SYSTOR 2013. Paper 3, together with Michael Wei from UCSD, “I/O Speculation for the Microsecond Era” [25] presented at USENIX ATC 2014. Paper 4, “LightNVM: Lightning Fast Evaluation Platform for Non-Volatile Memories” [1] presented at the Non-Volatile Memory Workshop. Paper 5, “Open-Channel Solid State Drives” short paper presented at the Non-Volatile Memory Workshop and Linux Software Foundation Vault 2015. Long version to be submitted to FAST 2016. At last Paper 6, “AppNVM: A software-defined, application-driven SSD” [6] presented at the Non-Volatile Memory Workshop 2015. Finally, the manuscript offers concluding remarks, open issues and directions for future work.

Literature

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, J.D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, number June, pages 57–70. USENIX Association, 2008.

[2] AC Arpaci-Dusseau and RH Arpaci-Dusseau. Removing the costs of indirection in flash-based SSDs with nameless writes. *HotStorage'10 Proceedings of the 2nd USENIX conference on Hot topics in storage and file systems*, 2010.

[3] Matias Bjø rling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux Block IO : Introducing Multi-queue SSD Access on Multi-core Systems. *Proceedings of the 6th International Systems and Storage Conference*, 2013.

[4] Matias Bjø rling, Philippe Bonnet, Luc Bouganim, and Niv Dayan. The Necessary Death of the Block Device Interface. In *6th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2013.

[5] Matias Bjø rling, Jesper Madsen, Philippe Bonnet, Aviad Zuck, Zvonimir Bandic, and Qingbo Wang. LightNVM: Lightning Fast Evaluation Platform for Non-Volatile Memories. In *Non-Volatile Memories Workshop*, NVMWorkshop'14, 2014.

[6] Matias Bjø rling, Michael Wei, Javier González, Jesper Madsen, and Philippe Bonnet. AppNVM: A software-defined, application-driven SSD. In *Non-Volatile Memories Workshop*, 2015.

[7] John S Bucy, Jiri Schindler, Steven W Schlosser, and Gregory R

Ganger. The DiskSim Simulation Environment. *Parallel Data Laboratory*, 2008.

[8] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, December 2010.

[9] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 387, 2012.

[10] Joel Coburn, Adrian M Caulfield, Laura M Grupp, Rajesh K Gupta, and Steven Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation , Non-Volatile Memories. In *ACM SIGARCH Computer Architecture News*, 2011.

[11] Niv Dayan, MK Svendsen, and Matias Bjø rling. EagleTree: exploring the design space of SSD-based algorithms. *Proceedings of the VLDB Endowment*, 2013.

[12] Peter J Denning. ACM President's Letter: What is experimental computer science? *Communications of the ACM*, 23(10), 1980.

[13] Amber Huffman. NVM Express Specification 1.1. 2012.

[14] Kim Joohyun, Haesung Kim, Seongjin Lee, and Youjip Won. FTL Design for TRIM Command. *The Fifth International Workshop on Software Support for Portable Storage*, pages 7–12, 2010.

[15] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. FlashSim: A Simulator for NAND Flash-Based Solid-State Drives. *2009 First International Conference on Advances in System Simulation*, pages 125–131, September 2009.

[16] Sang-won Lee and Jin-soo Kim. Understanding SSDs with the OpenSSD Platform. *Flash Memory Summit*, 2011.

[17] Oracle Nesheim, Bill. Mythbusting Flash Performance. *Flash Memory Summit*, 2011.

[18] Xiangyong Ouyang, David Nellans, Robert Wipfel, David Flynn, and D.K. Panda. Beyond block I/O: Rethinking traditional storage primitives. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 301–311. IEEE, 2011.

[19] Luc Bouganim Philippe Bonnet. Flash Device Support for Database Management. *5th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2011.

[20] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.

[21] Mohit Saxena, Yiying Zhang, Michael M Swift, Andrea Arpaci-Dusseau C, and Remzi H Arpaci-Dusseau. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013.

[22] Yong Ho Song, Sanghyuk Jung, Sang-won Lee, and Jin-soo Kim. Cosmos OpenSSD : A PCIe-based Open Source SSD Platform OpenSSD Introduction. *Flash Memory Summit*, pages 1–30, 2014.

[23] Curtis E Stevens and Dan Colgrove. Technical Comittee TI13 - ATA / ATAPI Command Set - 2. 2010.

[24] Dejan Vučinić, Qingbo Wang, Cyril Guyot, Robert Mateescu, Filip Blagojević, Luiz Franca-neto, Damien Le Moal, Hgst San, Trevor Bunker, Jian Xu, Steven Swanson, San Diego, Santa Clara, and Zvonimir Bandi. DC Express : Shortest Latency Protocol for Reading Phase Change Memory over PCI Express. *12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[25] Michael Wei, Matias Bjø rling, Philippe Bonnet, and Steven Swanson. I/O Speculation for the Microsecond Era. In *USENIX Annual Technical Conference*, 2014.

[26] Jisoo Yang, DB Minturn, and Frank Hady. When poll is better than

interrupt. *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, pages 1–7, 2012.

Chapter 2

The Necessary Death of the Block Device Interface

The premise of the CLyDE project is that the strict layering that has been so successful for designing database systems on top of magnetic disks is no longer applicable with SSDs based on existing observation that SSD performance varies a lot and in unpredictable ways [7, 8, 6, 9, 3, 14, 5, 12].

To start the thesis, I spent time reviewing the literature and surveying existing assumptions about SSD performance model. I found that most papers made a set of assumptions, most often implicitly about SSD performance. The original plan was to evaluate these hypotheses experimentally. It turned out that an analysis based on a clear understanding of SSD internals was sufficient to show that SSD complexity was not captured by existing performance models. The first part of the following paper thus makes the case against the narrow block device interface.

However, what then? If the block device interface is not appropriate, how should secondary storage be abstracted? Moreover, how would hardware evolution, in particular the advent of byte addressable persistent memory, impact such an abstraction? We propose a vision that addresses these questions in the second part of this paper.

This vision paper, called "The Necessary Death of the Block Device Interface", was published at CIDR in 2013.

After the paper was published, it has been discussed at workshops and presentations in the industry [1, 2]. Interfaces have been specifically designed to change the way the block device interface is used in large cloud data-centers [11], moving from a block device abstraction per SSD to a block device abstraction per flash chip in an SSD. Similarly, NVMKV [10], NVMFS [2, 13] propose special purpose instantiations of the storage block interface.

What was a slightly unconventional take on SSD performance characterization has now become a mainstream view. For example, it was the opinion adopted by all industrial presenters at the EMC University Flash Forum in September 2014. Note though that some industry players are still very much attached to the block device interface to keep the status quo and a very profitable business as long as possible. So the block device interface is not going to disappear anytime soon. The point I make in the paper though is that it has become a legacy that should no longer constraint system research.

Literature

[1] EMC University Flash Forum, September. 2014.

[2] Memblaze Flash Forum, May. 2015.

[3] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. *ACM SIGPLAN Notices*, 44(3):229, February 2009.

[4] William K. Josephson, Lars a. Bongo, Kai Li, and David Flynn. DFS: A File System for Virtualized Flash Storage. *ACM Transactions on Storage*, 6(3):1–25, September 2010.

[5] Dawoon Jung, Jeong-UK Kang, Heeseung Jo, Jin-Soo Kim, and Joonwon Lee. Superblock FTL: A Superblock-Based Flash Translation Layer with a Hybrid Address Translation Scheme. *ACM Transactions on Embedded Computing Systems*, 9(4):1–41, March 2010.

[6] J KANG, J KIM, C PARK, H PARK, and J LEE. A multi-channel architecture for high-performance NAND flash-based storage system. *Journal of Systems Architecture*, 53(9):644–658, September 2007.

[7] Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, May 2002.

[8] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee,

Sangwon Park, and Ha-Joo Song. FAST: A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems*, 6(3), July 2007.

[9] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 42(6):36–42, October 2008.

[10] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. *6th USENIX Workshop on Hot Topics in Storage and File Systems*, 2014.

[11] Jian Ouyang, Shiding Lin, S Jiang, and Z Hou. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014.

[12] Dongchul Park, Biplob Debnath, and David H.C. Du. A Workload-Aware Adaptive Hybrid Flash Translation Layer with an Efficient Caching Strategy. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 248–255. IEEE, July 2011.

[13] Mohit Saxena, Yiying Zhang, Michael M Swift, Andrea Arpaci-Dusseau C, and Remzi H Arpaci-Dusseau. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST)*, 2013.

[14] Qingsong Wei, Bozhao Gong, Suraj Pathak, Bharadwaj Veeravalli, Lingfang Zeng, and Kanzo Okada. WAFTL : A Workload Adaptive Flash Translation Layer with Data Partition. *Symposium on Massive Storage Systems and Technologies and Co-located Events*, 2011.

The Necessary Death of the Block Device Interface

Matias Bjørling[1], Philippe Bonnet[1], Luc Bouganim[2,3], Niv Dayan[1]

[1] IT University of Copenhagen
Copenhagen, Denmark
{mabj,phbo,nday}@itu.dk

[2] INRIA Paris-Rocquencourt
Le Chesnay, France
Luc.Bouganim@inria.fr

[3] PRISM Laboratory
Univ. of Versailles, France
Luc.Bouganim@prism.uvsq.fr

ABSTRACT

Solid State Drives (SSDs) are replacing magnetic disks as secondary storage for database management as they offer orders of magnitude improvement in terms of bandwidth and latency. In terms of system design, the advent of SSDs raises considerable challenges. First, the storage chips, which are the basic component of a SSD, have widely different characteristics – e.g., copy-on-write, erase-before-write and page-addressability for flash chips vs. in-place update and byte-addressability for PCM chips. Second, SSDs are no longer a bottleneck in terms of IO latency forcing streamlined execution throughout the I/O stack to minimize CPU overhead. Finally, SSDs provide a high degree of parallelism that must be leveraged to reach nominal bandwidth. This evolution puts database system researchers at a crossroad. The first option is to hang on to the current architecture where secondary storage is encapsulated behind a block device interface. This is the mainstream option both in industry and academia. This leaves the storage and OS communities with the responsibility to deal with the complexity introduced by SSDs in the hope that they will provide us with a robust, yet simple, performance model. In this paper, we show that this option amounts to building on quicksand. We illustrate our point by debunking some popular myths about flash devices and pointing out mistakes in the papers we have published throughout the years. The second option is to abandon the simple abstraction of the block device interface and reconsider how database storage manager, operating system drivers and SSD controllers interact. We give our vision of how modern database systems should interact with secondary storage. This approach requires a deep cross-layer understanding and a deep re-design of the database system architecture, which is the the only viable option for database system researchers to avoid becoming irrelevant.

1. INTRODUCTION

For the last thirty years, database systems have relied on magnetic disks as secondary storage [18]. Today, the growing performance gap between processors and magnetic disk is pushing solid-state drives (SSDs) as replacements for disks [10]. SSDs are based on non-volatile memories such as flash and PCM (Phase-Change memory). They offer great performance at an ever-decreasing cost. Today, tens of flash chips wired in parallel behind a safe cache deliver hundreds of thousands accesses per second at a latency of tens of microseconds. Compared to modern hard disks, this is a hundredfold improvement in terms of bandwidth and latency, at ten-times the cost. New SSD technologies, such as PCM, promise to keep on improving performance at a fraction of the cost.

It has now been six years since Jim Gray pointed out the significance of flash-based SSDs. Has a new generation of database systems emerged to accommodate those profound changes? No. Is a new generation of database systems actually needed? Well, the jury is up. There are two schools of thoughts:

- The conservative approach, taken by all database constructors, and many in the research community, is to consider that the advent of SSDs does not require any significant re-design. The fact that SSDs offer the same block device interface as magnetic disks allows preserving existing database systems, running them unchanged on SSDs (slight adaptations being sold as SSD-optimizations). A fraction of radical conservatives ignore SSDs and keep on writing articles and grant proposals based on disks, as if we were in the 90s (How will they teach about database systems in five years, when none of their bachelor students has ever seen a disk?) More moderate conservatives, focusing on storage management, consider that database systems have to be redesigned on top of the block device interface, based on the new performance characteristics of SSDs. The hope is that the storage and operating system communities provide a robust, yet simple, performance model for the new generation of storage devices.
- The progressive approach is to consider that the advent of SSDs, and non-volatile memories more generally, requires a complete re-thinking of the interactions between database system, operating system and storage devices. The argument is that SSDs challenge the strict layering established between these components on the basis of a simple performance contract, e.g., sequential access is no longer orders of magnitude faster than random access, SSDs are no longer the bottleneck in terms of latency, SSDs require a high-level of parallelism, SSDs do not constitute a homogeneous class of devices (as opposed to disks). This approach, that requires a deep cross layer understanding, is mainstream in the operating system and storage research communities [7,9,13]; not yet in the database systems research community.

The premise of the conservative approach is that the block device interface should be conserved as a robust abstraction that allows the operating system to hide the complexity of I/O management without sacrificing performance. We show, in Section 2, that this assumption does not hold; neither for flash-based nor for PCM-based devices. Worse, we show that it leads to brittle research based on myths rather than sound results. We debunk a few of these myths, illustrating our points with mistakes published in the articles we have written throughout the years.

In Section 3, we present the challenges that SSDs and non-volatile memories raise in terms of system design and discuss how they impact database systems. We present our vision of the necessary collaboration between database storage manager and operating system.

Note that we do not dispute that the conservative approach is economically smart. Neither do we ignore the fact that disks still largely dominate the storage market or that the block device interface will live on as a legacy for years. Our point is that the advent of SSDs and non-volatile memories has a deep impact on system design, and that we, as database systems researchers, must re-visit some grand old design decisions and engage with the operating system and storage communities in order to remain relevant.

2. THE CASE AGAINST BLOCK DEVICES

2.1 SSD MYTHS

Even if the block device interface has been challenged for some years [17], these critics have had, so far, a limited impact. For instance, all research papers published in the database community, proposing new storage models, indexing methods or query execution strategies for flash devices still build on the premise of SSDs encapsulated behind a block device interface [5]. All of these approaches assume, more or less explicitly, a simple performance model for the underlying SSDs. The most popular assumptions are the following:

- *SSDs behave as to the non-volatile memory they contain*: Before flash-based SSDs became widely available, there was a significant confusion between flash memory and flash devices. Today, we see a similar confusion with PCM.
- *On flash-based SSDs random writes are extremely costly, and should be avoided*: This was actually always true for flash devices on the market before 2009. Moreover, this rule makes sense after a quick look at flash constraints and SSD architecture. Many thus propose to avoid random writes using buffering and log-based strategies
- *On flash-based SSDs, reads are cheaper than writes*: Again this seems to make sense because, (1) reads on flash chips are much cheaper than writes (the so-called program operations); (2) flash chip constraints impact write operations (need for copy-on-write as in-place updates are forbidden on a flash chip); (3) flash devices have no mechanical parts. Some proposals are built on this rule, making aggressive use of random read IOs.

We will show in Section 2.3 that these assumptions about (flash-based) SSDs are plain wrong, but first, let us review the internals of a flash-based IO stack -- from flash chips to the OS block layer.

2.2 I/O STACK INTERNALS

A point that we would like to carry across is that we, as database researchers, can no longer consider storage devices as black boxes that respect a simple performance contract. We have to dig into their internals in order to understand the impact of these devices on system design. Here is a bottom up review of the IO stack with flash-based SSDs. We discuss PCM in Section 2.4.

Flash chip: A flash chip is a complex assembly of a huge number of flash cells[1], organized by pages (512 to 4096 bytes per page), blocks (64 to 256 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 (today 20nm), (ii) in the number of bits per flash cells, (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. For now, this lower performance can be compensated by increased parallelism within and across chips. At some point though, it will be impossible to further reduce the size of a flash cell. At that point, PCM might be able to take over and still provide exponential growth in terms of density.

Flash SSD: A flash-based SSD contains tens of flash chips wired in parallel to the SSD controller though multiple channels. Flash chips are decomposed into logical units (LUN). LUNs are the unit of operation interleaving, i.e., operations on distinct LUNs can be executed in parallel, while operations on a same LUN are executed serially. We consider that SSD performance is *channel-bound* if channels are the bottleneck and IOs wait for a channel to be available before they can be executed. SSD performance is *chip-bound* if chip operations are the bottleneck and IOs wait for a chip operation to terminate before they can be executed. Figure 1 illustrates these notions on an example.

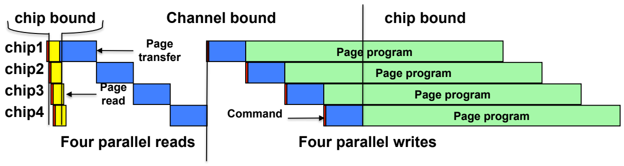

Figure 1: Example of channel transfer and chip operations on four chips (we assume 1 LUN per chip) attached to the same channel.

SSD controller: The SSD controller embeds the so-called Flash Translation Layer (FTL) that maps incoming application IOs –a read, a write or a trim[2] on a logical block address (LBA)– into appropriate chip operations. As illustrated on Figure 2, FTL is responsible for:

- *Scheduling & Mapping:* The FTL provides a virtualization of the physical address space into a logical address space. This mapping is done at the page (and possibly block) level. The FTL implements out-of-place updates (copy-on-write) to handle C2 and C3. It also handles chip errors and deals with parallelism across flash chips. While each read (resp. trim) operation is mapped onto a specific chip, each write operation can be scheduled on an appropriate chip.
- *Garbage Collection:* Each update leaves an obsolete flash page (that contains the before image). Over time such obsolete flash pages accumulate, and must be reclaimed through garbage collection.
- *Wear Leveling:* The FTL relies on wear-leveling to address C4--distributing the erase counts across flash blocks and masking bad blocks.

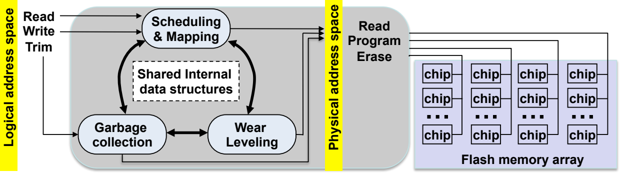

Figure 2: Internal architecture of a SSD controller

[1] See [5] for a discussion of flash cells internals.

[2] The Trim command has been introduced in the ATA interface standard to communicate to a flash device that a range of LBAs are no longer used by an application

Note that both the garbage collection and wear leveling modules read live pages from a victim block and write those pages (at a location picked by the scheduler), before the block is erased. The garbage collection and wear leveling operations thus interfere with the IOs submitted by the applications.

OS Driver: SSDs are not directly accessible from the CPU; the operating system provides a driver that manages communications to and from the device. Most SSDs implement a SATA interface and are accessed via the generic SATA driver. Some high-end SSDs (e.g., ioDrive from FusionIO) are directly plugged on the PCI bus. They provide a specific driver, which implements part of the SSD controller functionalities (leveraging CPU and RAM on the server to implement part of the FTL).

Block Layer: The block layer provides a simple memory abstraction. It exposes a flat address space, quantized in logical blocks of fixed size, on which I/O (read and write) requests are submitted. I/O requests are asynchronous. When an I/O request is submitted, it is associated to a completion queue. A worker thread then sends a page request to the disk scheduler. When the page request completes, an interrupt is raised (within the device driver), and the I/O request completes. In the last few years, the Linux block layer has been upgraded to accommodate SSDs and multi-cores. CPU overhead has been reduced– it was acceptable on disk to reduce seeks –, lock contention has been reduced, completions are dispatched on the core that submitted the request, and currently, the management of multiple IO queues for each device is under implementation.

Is it still reasonable to hide all this complexity behind a simple memory abstraction? Let us now revisit the performance assumptions popular in the database community.

2.3 DEBUNKING SSD MYTHS

(1) *SSDs behave as to the non-volatile memory they contain.*

Hopefully, the previous section will have made it very clear that this statement is not true. We pointed out this confusion in [6]. Still, two years later, we proposed a bimodal FTL that exposed to applications the constraints of a single flash chip [4]. We ignored the intrinsic parallelism of SSDs and the necessary error management that should take place within a device controller. Exposing flash chip constraints through the block layer, as we proposed, would in effect suppress the virtualization of the physical flash storage. This would limit the controller's ability to perform garbage collection and wear leveling (as it could not redirect the live pages of a victim block onto other chips) and its ability to deal with partial chip failures. It would also put a huge burden on the OS block layer if the application aimed at efficiently leveraging SSD parallelism by scheduling writes on multiple chips. Today, papers are published that attribute the characteristics of a phase-change memory chip to a SSD, thus ignoring that parallelism and error management must be managed at the SSD level.

(2*) On flash-based SSDs, random writes are very costly and should be avoided***.**

While this statement was true on early flash-based SSDs, it is no longer the case [2,3,5]. There are essentially two reasons why a flash-based SSD might provide random writes which are as fast as, or even faster than sequential writes. First, high-end SSDs now include safe RAM buffers (with batteries), which are designed for buffering write operations. Such SSDs provide a form of write-back mechanism where a write I/O request completes as soon as it hits the cache. Second, modern SSD can rely on page mapping, either because mapping is stored in the driver (without much RAM constraints), or because the controller supports some form of efficient page mapping cache (e.g., DFTL [10]). With page mapping, there are no constraints on the placement of any write – regardless of whether they are sequential or random. Thus a controller can fully benefit from SSD parallelism when flushing the buffer regardless of the write pattern! An interesting note is that random writes have a negative impact on garbage collection, as locality is impossible to detect for the FTL. As a result, pages that are to be reclaimed together tend to be spread over many blocks (as opposed to sequential writes where locality is easy to detect). Quantifying these effects is a topic for future work. To sum up, the difference between random writes and sequential writes on flash-based SSDs is rather indirect. We completely missed that point in [4], where we ventured design hints for SSD-based system design.

(3) *On flash-based SSDs, reads are cheaper than writes*.

While at the chip level reads are much faster than writes, at the SSD level this statement is not necessarily true. First, for reads, any latency or delay in the execution leads to visible latency in the application. It is not possible to hide this latency behind a safe cache, as it is the case for writes. So if subsequent reads are directed to a same LUN and, if that LUN or the associated channel is busy, then the read operation must wait (e.g., wait 3ms for the completion of an erase operation on that LUN)! Third, reads will benefit from parallelism only if the corresponding writes have been directed to different LUNs (on different channels). As we have seen above, there is no guarantee for this. Fourth, reads tend to be channel-bound --while writes tend to be chip-bound --, and channel parallelism is much more limited than chip parallelism.

2.4 DISCUSSION

It is unlikely that the complexity of flash-based SSDs can be tamed into a simple performance model behind the block device interface. So what should we do? An option is to wait for the OS and storage communities to define such a model. In the meantime, we should stop publishing articles based on incorrect assumptions. Another option is to skip the complexity of flash-based SSDs and wait for PCM to take over, as the characteristics of PCM promise to significantly reduce complexity (in-place updates, no erases, on-chip error detection, no need for garbage collection). First, there is a large consensus that PCM chips should be directly plugged onto the memory bus (because PCM is byte addressable and exhibits low latency) [7,15]. The capacity of each PCM chip is unlikely to be much larger than RAM chips. That still leaves us with the problem of secondary storage. Second, PCM is likely to be integrated into flash-based SSDs, i.e., to expand buffer capacity and performance. As a result, flash-based SSDs are unlikely to disappear any time soon. Third, even if we contemplate pure PCM-based SSDs [1], the issues of parallelism, wear leveling and error management will likely introduce significant complexity. Also, PCM-based SSDs will not make the issues of low latency and high-parallelism disappear. More generally, PCM and flash mark a significant evolution of the nature of the interactions between CPU, memory (volatile as well as non-volatile) and secondary storage. This is an excellent opportunity to revisit how database systems interact with secondary storage.

3. SECONDARY STORAGE REVISITED

For years, we have assumed that persistence was to be achieved through secondary storage, via a memory abstraction embodied by the block device interface. The advent of SSDs force us to reconsider this assumption:

1. We can now achieve persistence through PCM-based SSDs plugged on the memory bus and directly addressable by the CPU [6], in addition to secondary storage, composed of flash-based SSDs.
2. Flash-based SSDs are no longer accessed via a strict memory abstraction. The TRIM command has been added to read and write to make it possible to applications to communicate to a SSD that a range of logical addresses were no longer used and could thus be un-mapped by the FTL. SSD constructors are now proposing to expose new commands, e.g., atomic writes [17], at the driver's interface. More radically, FusionIO is now proposing direct access to its driver, entirely bypassing the block layer (ioMemory SDK). The point here is that the block device interface provides too much abstraction in the absence of a simple performance model.

This evolution forces us to re-visit the nature of persistence in database systems. We see three fundamental principles:

- *We should keep synchronous and asynchronous patterns separated*, as Mohan suggested [16]. Until now, database storage managers have implemented conservative asynchronous I/O submission policies to account for occasional synchronous I/Os [13]. Instead synchronous patterns (log writes, buffer steals under memory pressure) should be directed to PCM-based SSDs via non-volatile memory accesses from the CPU, while asynchronous patterns (lazy writes, prefetching, reads) should be directed to flash-based SSDs via I/O requests.
- *We should abandon the memory abstraction in favor of a communication abstraction to manage secondary storage*, as we suggested in [4]. The consequence is that (a) the database system is no longer the master and secondary storage a slave (they are communicating peers), and (b) the granularity of interactions is not limited to blocks. This has far reaching consequences on space allocation and naming (extent-based allocation is irrelevant, nameless writes are interesting), the management of log-structured files (which is today handled both at the database level and within the FTL), the management of interferences between I/Os, garbage collection and wear leveling. Interestingly, Jim Gray noted in [11] that RAM locality is king. An extended secondary storage interface would allow us to efficiently manage locality throughout the I/O stack.
- *We should seek inspiration in the low-latency networking literature.* Secondary storage is no longer a bottleneck in terms of latency, and it requires parallelism to reach nominal bandwidth. A similar evolution has been witnessed for some years in the networking community, where the developments of network cards, and the advent of 10/40/100 GB Ethernet, forced them to tackle the problems caused by low-latency. The solutions they explored including cross-layer design, shared memory, use of FPGA, out-of-band signaling are very much relevant in the context of a re-designed I/O stack, all the way to a database system. Major differences include the need to manage state for I/O completion and the need to handle small requests.

Note that any evolution of the role of secondary storage will take place in the context of multi-core CPUs. So, the staging architecture [12], based on the assumption that all data is in-memory, should be the starting point for our reflection.

Why don't we let the OS community redefine the IO stack? Well, they are not waiting for us. Proposals are flourishing for PCM-based [9,7], flash-based [14] and even database storage [15] systems. Note that these approaches are based on actual storage hardware and complete system design. We argue that it is time for database system researchers to engage other systems communities to contribute to the on-going re-design of the I/O stack and re-think the role of persistence in database systems.

4. CONCLUSION

In this paper, we established that the database systems research community has a flash problem. We argued that the high-level of abstraction provided by the block device interface is a significant part of the problem. We joined the choir of those who preach a re-design of the architecture of (single-site) database systems. We argued that we ignore the evolution of secondary storage at our own peril. First, because some of the assumptions we are making are myths rather than sound results. Second, because the on-going re-design of the I/O stack is an opportunity for intriguing research.

5. REFERENCES

[1] A. Akel, A. Caulfield, T.Mollov, R.Gupta, S. Swanson. Onyx: A Prototype Phase Change Memory Storage Array. HotStorage 2011.

[2] M. Bjørling, P. Bonnet, L. Bouganim, and B. T. Jònsson. Understanding the energy consumption of flash devices with uFLIP. IEEE Data Eng. Bull., December, 2010.

[3] M. Bjørling, L. L. Folgoc, A. Mseddi, P. Bonnet, L. Bouganim, and B. Jònsson. Performing sound flash device measurements: some lessons from uFLIP. SIGMOD Conference, 2010.

[4] P. Bonnet and L. Bouganim. Flash Device Support for Database Management. CIDR, 2011.

[5] P. Bonnet, L. Bouganim, I. Koltsidas, S. Viglas. System Co-Design and Data Management for Flash Devices. VLDB 2011.

[6] L. Bouganim, B. T. Jònsson, and P. Bonnet. uFLIP: Understanding flash I/O patterns. CIDR, 2009.

[7] A. Caulfield, T. Mollov, L. Eisner, A. De, J. Coburn, S. Swanson: Providing safe, user space access to fast, solid state disks. ASPLOS 2012.

[8] S. Chen, P. Gibbons, S. Nath: Rethinking Database Algorithms for Phase Change Memory. CIDR 2011.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. C. Lee, D. Burger, D. Coetzee: Better I/O through byte-addressable, persistent memory. SOSP 2009.

[10] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In ASPLOS, 2009.

[11] J. Gray. Tape is dead, disk is tape, flash is disk, RAM locality is king. Pres. at the CIDR Gong Show, Asilomar, CA, USA, 2007.

[12] Stavros Harizopoulos, Anastassia Ailamaki: A Case for Staged Database Systems. CIDR 2003

[13] Christoffer Hall, Philippe Bonnet: Getting Priorities Straight: Improving Linux Support for Database I/O. VLDB 2005.

[14] H. Lim, B. Fan, D. Andersen, M. Kaminsky: SILT: a memory-efficient, high-performance key-value store. SOSP 2011:1-13

[15] M. Mammarella, S. Hovsepian, E. Kohler: Modular data storage with Anvil. SOSP 2009:147-160

[16] C.Mohan, S.Bhattacharya. Implications of Storage Class Memories on Software Architectures. HPCA Workshop on the Use of Emerging Storage and Memory Technologies. 2010

[17] X. Ouyang, D. W. Nellans, R. Wipfel, D. Flynn, D. K. Panda: Beyond block I/O: Rethinking traditional storage primitives. HPCA 2011.

[18] S. W. Schlosser and G. R. Ganger. MEMS-based Storage Devices and Standard Disk Interfaces: A Square Peg in a Round Hole? USENIX FAST, 2004.

[19] M. Stonebraker. Operating system support for database management. Commun. ACM, 24(7):412–418, 1981.

Chapter 3

Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems

In the previous paper, we made the case agains the block device interface. The following paper proposes a new design for the Linux block layer. How is that consistent? Should we get rid of the block layer or should we improve it? There is a subtle difference between the block device interface (which defines the logical address space and read/write operations on secondary storage), and the operating system block layer, which is responsible for IO submission and completion. In principle, the block layer can support any form of storage abstraction. The focus of this paper is thus on adapting IO handling to modern SSDs and multi-core hosts (regardless of how secondary storage is actually abstracted).

For decades IO handling has been optimized for traditional hard-drives. The software stack was specifically optimized for spinning hard-drives, i.e. large block I/Os and sequential accesses. When SSDs were introduced the read and write block interface typical of traditional hard-drives was preserved. This design decision guaranteed adoption, but quickly revealed significant bottlenecks in the storage stack.

In response, applications incorporate lighter and faster data structures and algorithms [3, 1, 4] and particular NoSQL databases, such as MongoDB, LevelDB, Aerospike, etc. takes advantage of the new storage media. Applications specifically optimized for SSDs rapidly revealed that the operating system was the bottleneck and that it had to be improved in order to increase application performance.

The kernel storage stack is composed by four core layers. From top to bottom: (i) I/Os arrive through system calls with a file descriptor (pread-/pwrite, ioctl, libaio, etc.). On submission, (ii) the I/O is submitted to the file-system. The file-system maps the file I/O to its logical device sector and issue it to the appropriate storage device through the (iii) block layer. The block layer provides opportunity for merging I/Os, plugging, fairness between processes, etc. and at last issues one or multiple requests to a device driver. Finally, (iv) The device driver transforms the request to the corresponding hardware commands and issues it to the hardware device. The device completes the I/O and the host then complete the I/O back up through the storage stack layers.

Bottlenecks occurs in all of these layers in one way or another. We specifically target the block layer, as data-intensive applications often uses little of file-system features and thus expose bottlenecks in the block layer and device drivers. The block layer has traditionally optimized for hard-drives and its design choices actively prevents utilization of high performance SSDs.

Our contribution is a scalable block layer for high-performance SSDs. The work is desbrived in "Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems". It was presented at the SYSTOR 2013 conference. The contribution had three insights:

- The Linux kernel block layer is inadequate for high-performance SSDs. Benchmarking the block layer and heavily optimize for throughput showsthat the block layer is incapable to process more than 800-1000K IOPS.

- SSDs that are capable of hundreds of thousands of IOPS, quickly overflow the attached CPU with interrupts. At high interrupt rates, a single CPU unit is dedicated to process I/O for other CPUs in the system. This increases latency, work and overhead for the complete system. To mitegate this, the new block layer allows I/Os to be routed using IRQ steering. IRQ steering enables the I/O system to send completion interrupt to the CPU that issued the I/O request, without interrupting the CPU of which the SSD is attached to.

- A core design of the original block layer is its single queue design. For each hard-drive in the system, there is a request queue allocated. Every time this queue is manipulated, its locks must be acquired and released. For each request, this can be multiple times. As merged, reordering and similar operatings are performed before its sent to the hard-drive for execution. This performs adequate on single CPU systems, but significantly exacerbated in multi-core and multi-nodes systems, preventing a scalable I/O stack.

These three insights underlie the design of the multi-queue block layer (blk-mq). The new block layer introduces two sets of queues. A set of per-core queues and another set of hardware dispatch queues. The per-core queues are used for I/O merging, reordering, etc. of incoming I/O on a specific core. Eliminating any need for coordination with other queues in the set. The hardware dispatch queues matches the corresponding hardware capabilities. For example the AHCI standard for hard-drives supports a queue depth of 32 outstanding I/Os. A hardware dispatch queue is initialized with the same depth. When I/Os are submitted to the device driver, they are first added to the hardware dispatch queue and then submitted to the device driver one at a time. If the hardware has enough hardware queues to support all cores in a system. The submission of I/Os are performed directly on a single core without any synchronization across cores. The two queue set allows I/Os to be manipulated efficiently in a per-core data structures, and later when I/Os are pushed to the device driver, and not manipulated again, they are moved to the hardware dispatch queue for a specific hardware queue.

Note that, at the time the paper was published, there was no commercial SSD equipped with several hardware queues. The open channel SSDs that we discuss in Chapter 6 do provide several hardware queues. In fact all commercial SSDs based on NVMe now provide several hardware queues.

The new block layer scales from 800-1000K to 15M IOPS performing 512 bytes I/Os. Additionally, latency timings means are reduced both for low and high contention workloads. The best case reported in an 8-node setup. The latency was improved from 7.8x at full CPU utilization and 36x with half CPU utilization.

After the paper was published, blk-mq was accepted upstream in Linux kernel version 3.13. It has been improved significantly since its introduction with for example Scalable per-request data structures[1], additional removal of locks in the hot path[2], prototyping of polling [4][3], and converting significant amount of device drivers to blk-mq, including Micron mtip driver[4], Intel NVMe driver[5], ATA/SCSI stack[6], device mapper[7], virtio [2], and several others.

[1] https://lkml.org/lkml/2014/6/2/329

[2] https://lkml.org/lkml/2015/5/14/713

[3] https://lkml.org/lkml/2013/6/20/591

[4] https://lkml.org/lkml/2014/6/2/575

[5] http://lists.infradead.org/pipermail/linux-nvme/2014-October/001187.html

[6] http://lwn.net/Articles/602159/

[7] https://www.redhat.com/archives/dm-devel/2015-March/msg00066.html

Literature

[1] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory ssd in enterprise database applications. *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, page 1075, 2008.

[2] Ming Lei. Virtio-blk Multi-queue Conversion and QEMU Optimization KVM Forum 2014 Virtio-blk Linux driver evolution. 2014.

[3] Patrick O Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):1–32, 1996.

[4] Dimitris Tsirogiannis and Stavros Harizopoulos. Query Processing Techniques for Solid State Drives Categories and Subject Descriptors. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, 2009.

[5] Jisoo Yang, DB Minturn, and Frank Hady. When poll is better than interrupt. *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, pages 1–7, 2012.

Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems

Matias Bjørling*† Jens Axboe† David Nellans† Philippe Bonnet*

*IT University of Copenhagen
{mabj,phbo}@itu.dk

†Fusion-io
{jaxboe,dnellans}@fusionio.com

ABSTRACT

The IO performance of storage devices has accelerated from hundreds of IOPS five years ago, to hundreds of thousands of IOPS today, and tens of millions of IOPS projected in five years. This sharp evolution is primarily due to the introduction of NAND-flash devices and their data parallel design. In this work, we demonstrate that the block layer within the operating system, originally designed to handle thousands of IOPS, has become a bottleneck to overall storage system performance, specially on the high NUMA-factor processors systems that are becoming commonplace. We describe the design of a next generation block layer that is capable of handling tens of millions of IOPS on a multi-core system equipped with a single storage device. Our experiments show that our design scales graciously with the number of cores, even on NUMA systems with multiple sockets.

Categories and Subject Descriptors

D.4.2 [**Operating System**]: Storage Management—*Secondary storage*; D.4.8 [**Operating System**]: Performance—*measurements*

General Terms

Design, Experimentation, Measurement, Performance.

Keywords

Linux, Block Layer, Solid State Drives, Non-volatile Memory, Latency, Throughput.

1 Introduction

As long as secondary storage has been synonymous with hard disk drives (HDD), IO latency and throughput have been shaped by the physical characteristics of rotational devices: Random accesses that require disk head movement are slow and sequential accesses that only require rotation of the disk platter are fast. Generations of IO intensive algorithms and systems have been designed based on these two fundamental characteristics. Today, the advent of solid state disks (SSD) based on non-volatile memories (NVM) (e.g., flash or phase-change memory [11, 6]) is transforming the performance characteristics of secondary storage. SSDs often exhibit little latency difference between sequential and random IOs [16]. IO latency for SSDs is in the order of tens of microseconds as opposed to tens of milliseconds for HDDs. Large internal data parallelism in SSDs disks enables many concurrent IO operations which, in turn, allows single devices to achieve close to a million IOs per second (IOPS) for random accesses, as opposed to just hundreds on traditional magnetic hard drives. In Figure 1, we illustrate the evolution of SSD performance over the last couple of years.

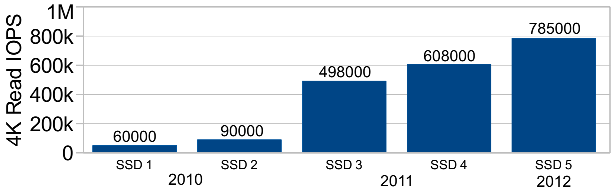

Figure 1: IOPS for 4K random read for five SSD devices.

A similar, albeit slower, performance transformation has already been witnessed for network systems. Ethernet speed evolved steadily from 10 Mb/s in the early 1990s to 100 Gb/s in 2010. Such a regular evolution over a 20 years period has allowed for a smooth transition between lab prototypes and mainstream deployments over time. For storage, the rate of change is much faster. We have seen a 10,000x improvement over just a few years. The throughput of modern storage devices is now often limited by their hardware (i.e., SATA/SAS or PCI-E) and software interfaces [28, 26]. Such rapid leaps in hardware performance have exposed previously unnoticed bottlenecks at the software level, both in the operating system and application layers. Today, with Linux, a single CPU core can sustain an IO submission rate of around 800 thousand IOPS. Regardless of how many cores are used to submit IOs, the operating system block layer can not scale up to over one million IOPS. This may be fast enough for today's SSDs - but not for tomorrow's.

We can expect that **(a)** SSDs are going to get faster, by increasing their internal parallelism[1] [9, 8] and **(b)** CPU

SYSTOR '13, June 30 - July 02 2013, Haifa, Israel

[1] If we look at the performance of NAND-flash chips, access times are getting slower, not faster, in timings [17]. Access time, for individual flash chips, increases with shrinking feature size, and increasing number of dies per package. The decrease in individual chip performance is compensated by improved parallelism within and across chips.

performance will improve largely due to the addition of more cores, whose performance may largely remain stable [24, 27].

If we consider a SSD that can provide 2 million IOPS, applications will no longer be able to fully utilize a single storage device, regardless of the number of threads and CPUs it is parallelized across due to current limitations within the operating system.

Because of the performance bottleneck that exists today within the operating system, some applications and device drivers are already choosing to bypass the Linux block layer in order to improve performance [8]. This choice increases complexity in both driver and hardware implementations. More specifically, it increases duplicate code across error-prone driver implementations, and removes generic features such as IO scheduling and quality of service traffic shaping that are provided by a common OS storage layer.

Rather than discarding the block layer to keep up with improving storage performance, we propose a new design that fixes the scaling issues of the existing block layer, while preserving its best features. More specifically, our contributions are the following:

1. We recognize that the Linux block layer has become a bottleneck (we detail our analysis in Section 2). The current design employs a single coarse lock design for protecting the request queue, which becomes the main bottleneck to overall storage performance as device performance approaches 800 thousand IOPS. This single lock design is especially painful on parallel CPUs, as all cores must agree on the state of the request queue lock, which quickly results in significant performance degradation.

2. We propose a new design for IO management within the block layer. Our design relies on multiple IO submission/completion queues to minimize cache coherence across CPU cores. The main idea of our design is to introduce two levels of queues within the block layer: (i) software queues that manage the IOs submitted from a given CPU core (e.g., the block layer running on a CPU with 8 cores will be equipped with 8 software queues), and (ii) hardware queues mapped on the underlying SSD driver submission queue.

3. We evaluate our multi-queue design based on a functional implementation within the Linux kernel. We implement a new no-op block driver that allows developers to investigate OS block layer improvements. We then compare our new block layer to the existing one on top of the noop driver (thus focusing purely on the block layer performance). We show that a two-level locking design reduces the number of cache and pipeline flushes compared to a single level design, scales gracefully in high NUMA-factor architectures, and can scale up to 10 million IOPS to meet the demand of future storage products.

The rest of the paper is organized as follows: In Section 2 we review the current implementation of the Linux block layer and its performance limitations. In Section 3 we propose a new multi-queue design for the Linux block layer. In Section 4 we describe our experimental framework, and in Section 5, we discuss the performance impact of our multi-queue design. We discuss related work in Section 6, before drawing our conclusions in Section 7.

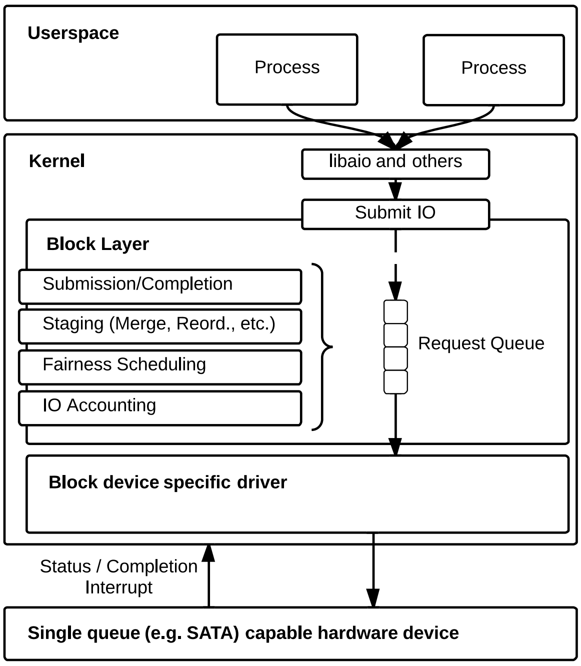
Userspace
Process
Process
Kernel
libaio and others
Submit IO
Block Layer
Submission/Completion
Staging (Merge, Reord., etc.)
Fairness Scheduling
IO Accounting
Request Queue
Block device specific driver
Status / Completion Interrupt
Single queue (e.g. SATA) capable hardware device

Figure 2: Current single queue Linux block layer design.

2 OS Block Layer

Simply put, the operating system block layer is responsible for shepherding IO requests from applications to storage devices [2]. The block layer is a glue that, on the one hand, allows applications to access diverse storage devices in a uniform way, and on the other hand, provides storage devices and drivers with a single point of entry from all applications. It is a convenience library to hide the complexity and diversity of storage devices from the application while providing common services that are valuable to applications. In addition, the block layer implements IO-fairness, IO-error handling, IO-statistics, and IO-scheduling that improve performance and help protect end-users from poor or malicious implementations of other applications or device drivers.

2.1 Architecture

Figure 2 illustrates the architecture of the current Linux block layer. Applications submit IOs via a kernel system call, that converts them into a data structure called a block IO. Each block IO contains information such as IO address, IO size, IO modality (read or write) or IO type (synchronous/ asynchronous)[2]. It is then transferred to either libaio for asynchronous IOs or directly to the block layer for synchronous IO that submit it to the block layer. Once an IO request is submitted, the corresponding block IO is buffered in the staging area, which is implemented as a queue, denoted the *request queue*.

Once a request is in the staging area, the block layer may perform IO scheduling and adjust accounting information before scheduling IO submissions to the appropriate storage

[2] See *include/linux/blk_types.h* in the Linux kernel (kernel.org) for a complete description of the Block IO data structure.

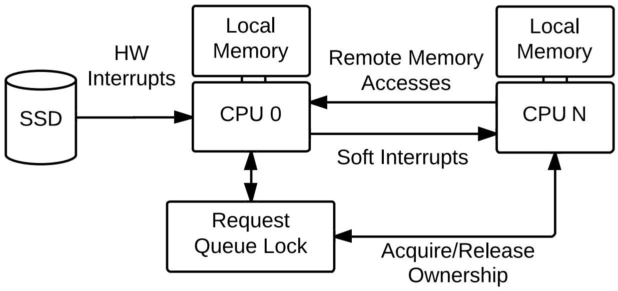

Figure 3: Simplified overview of bottlenecks in the block layer on a system equipped with two cores and a SSD.

device driver. Note that the Linux block layer supports pluggable IO schedulers: noop (no scheduling), deadline-based scheduling [12], and CFQ [10] that can all operate on IO within this staging area. The block layer also provides a mechanism for dealing with IO completions: each time an IO completes within the device driver, this driver calls up the stack to the generic completion function in the block layer. In turn the block layer then calls up to an IO completion function in the libaio library, or returns from the synchronous read or write system call, which provides the IO completion signal to the application.

With the current block layer, the staging area is represented by a request queue structure. One such queue is instantiated per block device. Access is uniform across all block devices and an application need not know what the control flow pattern is within the block layer. A consequence of this single queue per device design however is that the block layer cannot support IO scheduling across devices.

2.2 Scalability

We analyzed the Linux kernel to evaluate the performance of the current block layer on high performance computing systems equipped with high-factor NUMA multi-core processors and high IOPS NAND-flash SSDs. We found that the block layer had a considerable overhead for each IO; Specifically, we identified three main problems, illustrated in Figure 3:

1. *Request Queue Locking*: The block layer fundamentally synchronizes shared accesses to an exclusive resource: the IO request queue. (i) Whenever a block IO is inserted or removed from the request queue, this lock must be acquired. (ii) Whenever the request queue is manipulated via IO submission, this lock must be acquired. (iii) As IOs are submitted, the block layer proceeds to optimizations such as plugging (letting IOs accumulate before issuing them to hardware to improve cache efficiency), (iv) IO reordering, and (v) fairness scheduling. Before any of these operations can proceed, the request queue lock must be acquired. This is a major source of contention.

2. *Hardware Interrupts*: The high number of IOPS causes a proportionally high number of interrupts. Most of today's storage devices are designed such that one core (within CPU 0 on Figure 3) is responsible for handling all hardware interrupts and forwarding them to other cores as soft interrupts regardless of the CPU issuing and completing the IO. As a result, a single core may spend considerable time in handling these interrupts, context switching, and polluting L1 and L2 caches that applications could rely on for data locality [31]. The other cores (within CPU N on Figure 3) then also must take an IPI to perform the IO completion routine. As a result, in many cases two interrupts and context switches are required to complete just a single IO.

3. *Remote Memory Accesses*: Request queue lock contention is exacerbated when it forces remote memory accesses across CPU cores (or across sockets in a NUMA architecture). Such remote memory accesses are needed whenever an IO completes on a different core from the one on which it was issued. In such cases, acquiring a lock on the request queue to remove the block IO from the request queue incurs a remote memory access to the lock state stored in the cache of the core where that lock was last acquired, the cache line is then marked shared on both cores. When updated, the copy is explicitly invalidated from the remote cache. If more than one core is actively issuing IO and thus competing for this lock, then the cache line associated with this lock is continuously bounced between those cores.

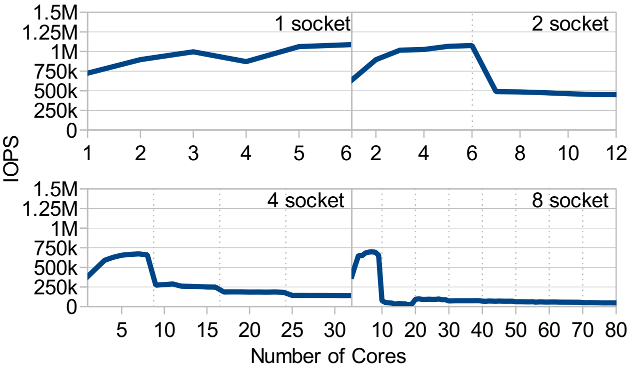

Figure 4: IOPS throughput of Linux block layer as a function of number of CPU's issuing IO. Divided into 1, 2, 4 and 8 socket systems. Note: Dotted line show socket divisions.

Figure 4 shows 512 bytes IOs being submitted to the kernel as fast as possible; IOPS throughput is depicted as a function of the number of CPU's that are submitting and completing IOs to a single device simultaneously. We observe that when the number of processes is lower than the number cores on a single socket (i.e., 6), throughput improves, or is at least maintained, as multiple CPU's issue IOs. For 2, 4, and 8-socket architectures which have largely supplanted single socket machines in the HPC space, when IOs are issued from a CPU that is located on a remote socket (and typically NUMA node), absolute performance drops substantially regardless the absolute number of sockets in the system.

Remote cacheline invalidation of the request queue lock is significantly more costly on complex four and eight socket systems where the NUMA-factor is high and large cache directory structures are expensive to access. On four and eight socket architectures, the request queue lock contention is so high that multiple sockets issuing IOs reduces the throughput of the Linux block layer to just about 125 thousand IOPS even though there have been high end solid state devices on the market for several years able to achieve higher IOPS than this. The scalability of the Linux block layer is not an issue that we might encounter in the future, it is a significant problem being faced by HPC in practice today.

3 Multi-Queue Block Layer

As we have seen in Section 2.2, reducing lock contention and remote memory accesses are key challenges when redesigning the block layer to scale on high NUMA-factor architectures. Dealing efficiently with the high number of hardware interrupts is beyond the control of the block layer (more on this below) as the block layer cannot dictate how a device driver interacts with its hardware. In this Section, we propose a two-level multi-queue design for the Linux block layer and discuss its key differences and advantages over the current single queue block layer implementation. Before we detail our design, we summarize the general block layer requirements.

3.1 Requirements

Based on our analysis of the Linux block layer, we identify three major requirements for a block layer:

- **Single Device Fairness**

  Many application processes may use the same device. It is important to enforce that a single process should not be able to starve all others. This is a task for the block layer. Traditionally, techniques such as CFQ or deadline scheduling have been used to enforce fairness in the block layer. Without a centralized arbiter of device access, applications must either coordinate among themselves for fairness or rely on the fairness policies implemented in device drivers (which rarely exist).

- **Single and Multiple Device Accounting**

  The block layer should make it easy for system administrators to debug or simply monitor accesses to storage devices. Having a uniform interface for system performance monitoring and accounting enables applications and other operating system components to make intelligent decisions about application scheduling, load balancing, and performance. If these were maintained directly by device drivers, it would be nearly impossible to enforce the convenience of consistency application writers have become accustom to.

- **Single Device IO Staging Area**

  To improve performance and enforce fairness, the block layer must be able to perform some form of IO scheduling. To do this, the block layer requires a staging area, where IOs may be buffered before they are sent down into the device driver. Using a staging area, the block layer can reorder IOs, typically to promote sequential accesses over random ones, or it can group IOs, to submit larger IOs to the underlying device. In addition, the staging area allows the block layer to adjust its submission rate for quality of service or due to device back-pressure indicating the OS should not send down additional IO or risk overflowing the device's buffering capability.

3.2 Our Architecture

The key insight to improved scalability in our multi-queue design is to distribute the lock contention on the single request queue lock to multiple queues through the use of two levels of queues with distinct functionally as shown in Figure 5:

- *Software Staging Queues.* Rather than staging IO for dispatch in a single software queue, block IO requests are now maintained in a collection of one or more request queues. These staging queues can be configured such that there is one such queue per socket, or per core, on the system. So, on a NUMA system with 4 sockets and 6 cores per socket, the staging area may contain as few as 4 and as many as 24 queues. The variable nature of the request queues decreases the proliferation of locks if contention on a single queue is not a bottleneck. With many CPU architectures offering a large shared L3 cache per socket (typically a NUMA node as well), having just a single queue per processor socket offers a good trade-off between duplicated data structures which are cache unfriendly and lock contention.

- *Hardware Dispatch Queues.* After IO has entered the staging queues, we introduce a new intermediate queuing layer known as the hardware dispatch queues. Using these queues block IOs scheduled for dispatch are not sent directly to the device driver, they are instead sent to the hardware dispatch queue. The number of hardware dispatch queues will typically match the number of hardware contexts supported by the device driver. Device drivers may choose to support anywhere from one to 2048 queues as supported by the message signaled interrupts standard MSI-X [25]. Because IO ordering is not supported within the block layer any software queue may feed any hardware queue without needing to maintain a global ordering. This allows hardware to implement one or more queues that map onto NUMA nodes or CPU's directly and provide a fast IO path from application to hardware that never has to access remote memory on any other node.

This two level design explicitly separates the two buffering functions of the staging area that was previously merged into a single queue in the Linux block layer: (i) support for IO scheduling (software level) and (ii) means to adjust the submission rate (hardware level) to prevent device buffer over run.

The number of entries in the software level queue can dynamically grow and shrink as needed to support the outstanding queue depth maintained by the application, though queue expansion and contraction is a relatively costly operation compared to the memory overhead of maintaining enough free IO slots to support most application use. Conversely, the size of the hardware dispatch queue is bounded and correspond to the maximum queue depth that is supported by the device driver and hardware. Today many SSD's that support native command queuing support a queue depth of just 32, though high-end SSD storage devices may have much deeper queue support to make use of the high internal parallelism of their flash architecture. The 32 in-flight request limit found on many consumer SSD's is likely to increase substantially to support increased IOPS rates as

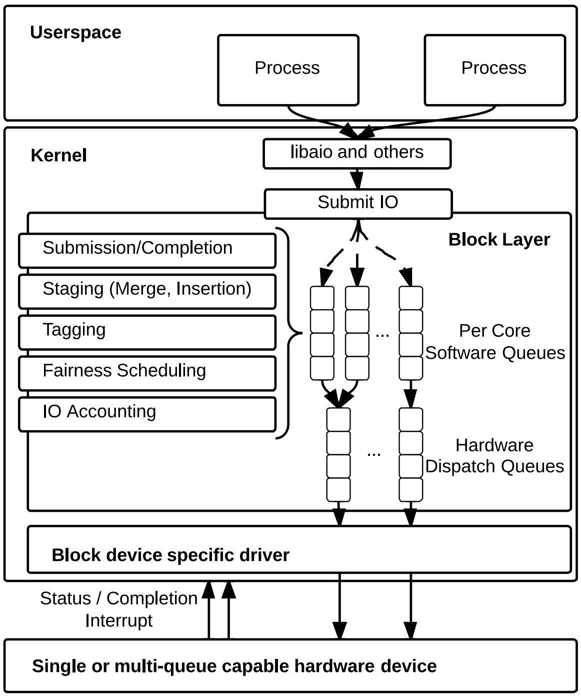

Figure 5: Proposed two level Linux block layer design.

a 1 million IOPS capable device will cause 31 thousand context switches per second simply to process IO in batches of 32. The CPU overhead of issuing IO to devices is inversely proportional to the amount of IO that is batched in each submission event.

3.2.1 IO-Scheduling

Within the software queues, IOs can be shaped by per CPU or NUMA node policies that need not access local memory. Alternatively, policies may be implemented across software queues to maintain global QoS metrics on IO, though at a performance penalty. Once the IO has entered the hardware dispatch queues, reordering i/Nums no longer possible. We eliminate this possibility so that the only contenders for the hardware dispatch queue are inserted to the head of the queue and removed from the tail by the device driver, thus eliminating lock acquisitions for accounting or IO-scheduling. This improves the fast path cache locality when issuing IO's in bulk to the device drivers.

Our design has significant consequences on how IO may be issued to devices. Instead of inserting requests in the hardware queue in sorted order to leverage sequential accesses (which was a main issue for hard drives), we simply follow a FIFO policy: we insert the incoming block IO submitted by core i at the top of the request queue attached to core i or the NUMA socket this core resides on. Traditional IO-schedulers have worked hard to turn random into sequential access to optimize performance on traditional hard drives. Our two level queuing strategy relies on the fact that modern SSD's have random read and write latency that is as fast as their sequential access. Thus interleaving IOs from multiple software dispatch queues into a single hardware dispatch queue does not hurt device performance. Also, by inserting requests into the local software request queue, our design respects thread locality for IOs and their completion.

While global sequential re-ordering is still possible across the multiple software queues, it is only necessary for HDD based devices, where the additional latency and locking overhead required to achieve total ordering does not hurt IOPS performance. It can be argued that, for many users, it is no longer necessary to employ advanced fairness scheduling as the speed of the devices are often exceeding the ability of even multiple applications to saturate their performance. If fairness is essential, it is possible to design a scheduler that exploits the characteristics of SSDs at coarser granularity to achieve lower performance overhead [23, 13, 19]. Whether the scheduler should reside in the block layer or on the SSD controller is an open issue. If the SSD is responsible for fair IO scheduling, it can leverage internal device parallelism, and lower latency, at the cost of additional interface complexity between disk and OS [8, 4].

3.2.2 Number of Hardware Queues

Today, most SATA, SAS and PCI-E based SSDs, support just a single hardware dispatch queue and a single completion queue using a single interrupt to signal completions. One exception is the upcoming NVM Express (NVMe) [18] interface which supports a flexible number of submission queues and completion queues. For devices to scale IOPS performance up, a single dispatch queue will result in cross CPU locking on the dispatch queue lock, much like the previous request queue lock. Providing multiple dispatch queues such that there is a local queue per NUMA node or CPU will allow NUMA local IO path between applications and hardware, decreasing the need for remote memory access across all subsections of the block layer. In our design we have moved IO-scheduling functionality into the software queues only, thus even legacy devices that implement just a single dispatch queue see improved scaling from the new multiqueue block layer.

3.2.3 Tagged IO and IO Accounting

In addition to introducing a two-level queue based model, our design incoporates several other implementation improvements. First, we introduce tag-based completions within the block layer. Device command tagging was first introduced with hardware supporting native command queuing. A tag is an integer value that uniquely identifies the position of the block IO in the driver submission queue, so when completed the tag is passed back from the device indicating which IO has been completed. This eliminates the need to perform a linear search of the in-flight window to determine which IO has completed.

In our design, we build upon this tagging notion by allowing the block layer to generate a unique tag associated with an IO that is inserted into the hardware dispatch queue (between size 0 and the max dispatch queue size). This tag is then re-used by the device driver (rather than generating a new one, as with NCQ). Upon completion this same tag can then be used by both the device driver and the block layer to identify completions without the need for redundant tagging. While the MQ implementation could maintain a traditional in-flight list for legacy drivers, high IOPS drivers will likely need to make use of tagged IO to scale well.

Second, to support fine grained IO accounting we have modified the internal Linux accounting library to provide statistics for the states of both the software queues and dispatch queues. We have also modified the existing tracing and profiling mechanisms in blktrace, to support IO tracing for future devices that are multi-queue aware. This will allow future device manufacturers to optimize their implementations and provide uniform global statistics to HPC customers whose application performance is increasingly dominated by the performance of the IO-subsystem.

3.3 Multiqueue Impact on Device Manufacturers

One drawback of the our design is that it will require some extensions to the bottom edge device driver interface to achieve optimal performance. While the basic mechanisms for driver registration and IO submission/completion remain unchanged, our design introduces these following requirements:

- *HW dispatch queue registration*: The device driver must export the number of submission queues that it supports as well as the size of these queues, so that the block layer can allocate the matching hardware dispatch queues.
- *HW submission queue mapping function*: The device driver must export a function that returns a mapping between a given software level queue (associated to core i or NUMA node i), and the appropriate hardware dispatch queue.
- *IO tag handling*: The device driver tag management mechanism must be revised so that it accepts tags generated by the block layer. While not strictly required, using a single data tag will result in optimal CPU usage between the device driver and block layer.

These changes are minimal and can be implemented in the software driver, typically requiring no changes to existing hardware or software. While optimal performance will come from maintaining multiple hardware submission and completion queues, legacy devices with only a single queue can continue to operate under our new Linux block layer implementation.

4 Experimental Methodology

In the remaining of the paper, we denote the existing block layer as single queue design (SQ), our design as the multiqueue design (MQ), and a driver which bypasses the Linux block layer as Raw. We implemented the MQ block layer as a patch to the Linux kernel 3.10[3].

4.1 Hardware Platforms

To conduct these comparisons, we rely on a *null* device driver, i.e., a driver that is not connected to an underlying storage device. This null driver simply receives IOs as fast as possible and acknowledges completion immediately. This pseudo block device can acknowledge IO requests faster than even a DRAM backed physical device, making the null block device an ideal candidate for establishing an optimal baseline for scalability and implementation efficiency.

Using the null block device, we experiment with 1, 2, 4 and 8 sockets systems, i.e., Sandy Bridge-E, Westmere-EP, Nehalem-EX and Westmere-EX Intel platforms. Table 1 summarizes the characteristics of these four platforms. The 1, 2 and 4-sockets systems use direct QPI links as interconnect between sockets, while the 8-nodes system has a lower and upper CPU board (with 4 sockets each) and an interconnect board for communication. We disabled the turbo boost and hyper-threading CPU features as well as any ACPI C and P-state throttling on our systems to decrease the variance in our measurements that would be caused by power savings features.

Platform/Intel	Sandy Bridge-E	Westmere-EP	Nehalem-EX	Westmere-EX
Processor	i7-3930K	X5690	X7560	E7-2870
Num. of Cores	6	12	32	80
Speed (Ghz)	3.2	3.46	2.66	2.4
L3 Cache (MB)	12	12	24	30
NUMA nodes	1	2	4	8

Table 1: Architecture of Evaluation Systems

4.2 IO Load Generation

We focus our evaluations on latency and throughput. We experiment with latency by issuing a single IO per participating core at a time using the pread/pwrite interface of the Linux kernel. We experiment with throughput by overlapping the submission of asynchronous IOs. In the throughput experiment we sustain 32 outstanding IOs per participating core, i.e., if 8 cores are issuing IOs, then we maintain 256 outstanding IOs. We use 32 IOs per process context because it matches the requirements of today's SSD devices. Our IO-load is generated using the flexible io generator (fio) [14] that allows us to carefully control the queue-depth, type, and distribution of IO onto a the LBA space of the block device. In all experiments we use 512 bytes read IO's, though the type of IO is largely irrelevant since the null block driver does not perform any computation or data transfer, it simply acknowledges all requests immediately.

4.3 Performance Metrics

The primary metrics for our experiments are absolute throughput (IOPS) and latency (μ-seconds) of the block layer.

5 Results

In a first phase, we compare our new block layer design (MQ) with the existing Linux block layer (SQ), and the optimal baseline (Raw). In a second phase, we investigate how our design allows the block layer to scale as the number of available cores in the system increases. We leave a performance tuning study of MQ (e.g., quality of the performance optimizations within the block layer) as a topic for future work.

For each system configuration, we create as many fio processes as there are cores and we ensure that all cores are utilized 100%. For the 1 socket system, the maximum number of cores is 6. For the 2 (resp., 4 and 8) sockets system, the maximum number of core is 12 (resp., 32 and 80), and we mark the separation between both 6 (resp., 8 and 10) cores sockets with a vertical dotted line. Unless otherwise noted, for MQ, a software queue is associated to each core and a hardware dispatch queue is associated to each socket.

[3] Our implementation is available online at `http://git.kernel.dk/?p=linux-block.git;a=shortlog;h=refs/heads/new-queue`

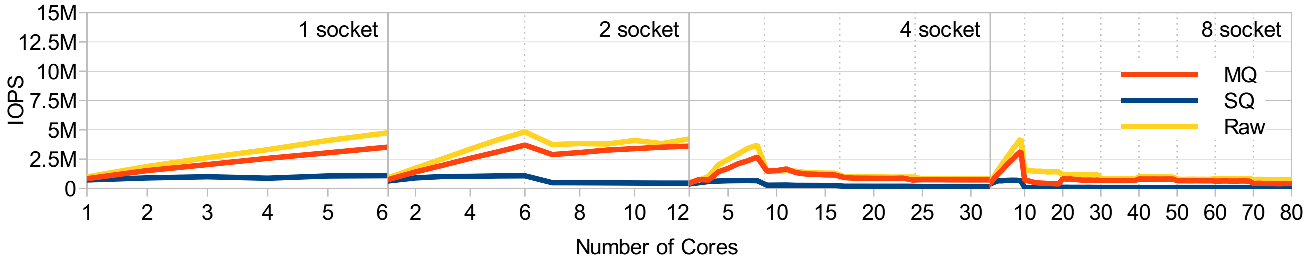

Figure 6: IOPS for single/multi-queue and raw on the 1, 2, 4 and 8-nodes systems.

5.1 Comparing MQ, SQ and Raw

Figure 6 presents throughput (in IOPS) for SQ, MQ and Raw as a function of the number of cores available in the 1 socket, 2-sockets, 4-sockets, and 8-sockets systems respectively. Overall, we can make the following observations. First, with the single queue block layer implementation, throughput is limited below 1 million IOPS regardless of the number of CPUs issuing IO or of the number of sockets in the system. *The current Linux block layer implementation can not sustain more than 1 million IOPS on a single block device.*

Second, our new two layer multi-queue implementation improves performance significantly. The system can sustain up to 3.5 million IOPS on a single socket system. However, in multi-socket systems scaling does not continue at nearly the same rate.

Let us analyze those results in more details:

1. The scalability problems of SQ are evident as soon as more than one core is used in the system. Additional cores spend most of their cycles acquiring and releasing spin locks for the single request queue and as such do not contribute to improving throughput. This problem gets even worse on multi-socket systems, because their inter-connects and the need to maintain cache-coherence.

2. MQ performance is similar to SQ performance on a single core. This shows that the overhead of introducing multiple queues is minimal.

3. MQ scales linearly within one socket. This is because we removed the need for the block layer to rely on synchronization between cores when block IOs are manipulated inside the software level queues.

4. For all systems, MQ follows the performance of Raw closely. MQ is in fact a constant factor away from the raw measurements, respectively 22%, 19%, 13% and 32% for the 1, 2, 4, 8-sockets systems. This overhead might seem large, but the raw baseline does not implement logic that is required in a real device driver. For good scalability, the MQ performance just needs to follow the trend of the baseline.

5. The scalability of MQ and raw exhibits a sharp dip when the number of sockets is higher than 1. We see that throughput reaches 5 million IOPS (resp., 3.8 and 4) for 6 cores (resp., 7 and 9) on a 2 sockets system (resp., 4 and 8 sockets system). This is far from the 10 million IOPS that we could have hoped for. Interestingly, MQ follows roughly the raw baseline. *There is thus a problem of scalability, whose root lies outside the block layer, that has a significant impact on performance. We focus on this problem in the next Section.*

	1 socket	2 sockets	4 sockets	8 sockets
SQ	50 ms	50 ms	250 ms	750 ms
MQ	50 ms	50 ms	50 ms	250 ms
Raw	50 ms	50 ms	50 ms	250 ms

Table 2: Maximum latency for each of the systems.

Let us now turn our attention to latency. As we explained in the previous section, latency is measured through synchronous IOs (with a single outstanding IO per participating core). The latency is measured as the time it takes to go from the application, through the kernel system call, into the block layer and driver and back again. Figure 7 shows average latency (in μ-seconds) as a function of the number of cores available for the four systems that we study.

Ideally, latency remains low regardless of the number of cores. In fact, remote memory accesses contribute to increase latency on multi-sockets systems. For SQ, we observe that latency increases linearly with the number of cores, slowly within one socket, and sharply when more than one socket is active. For MQ, latency remains an order of magnitude lower than for SQ. This is because, for MQ, the only remote memory accesses that are needed are those concerning the hardware dispatch queue (there is no remote memory accesses for synchronizing the software level queues). Note that, on 8 sockets system, the SQ graph illustrates the performance penalty which is incurred when crossing the interconnect board (whenever 2, 4, 6 and 8 sockets are involved).

Table 2 shows the maximum latency across all experiments. With SQ, the maximum latency reaches 250 milliseconds in the 4 sockets system and 750 milliseconds on the 8 sockets system. Interestingly, with SQ on a 8 sockets systems, 20% of the IO requests take more than 1 millisecond to complete. This is a very significant source of variability for IO performance. In contrast, with MQ, the number of IOs which take more than 1ms to complete only reaches 0.15% for an 8 socket system, while it is below 0.01% for the other systems. Note Raw exhibits minimal, but stable, variation across all systems with around 0.02% of the IOs that take more than 1ms to complete.

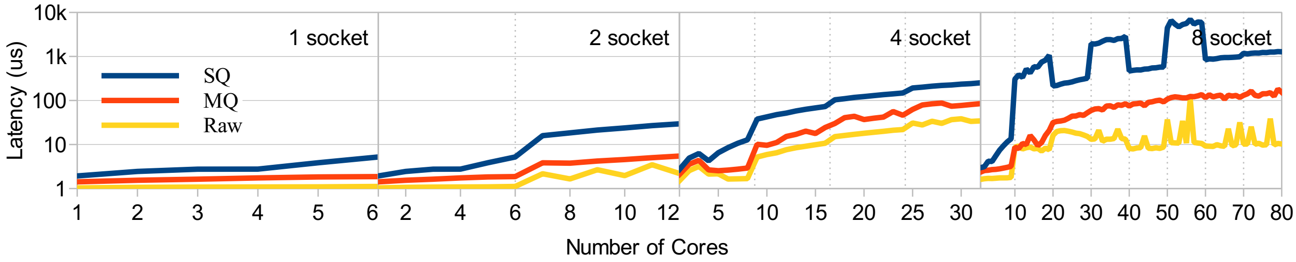

Figure 7: Latency on the 1, 2, 4 and 8 node system using the null device.

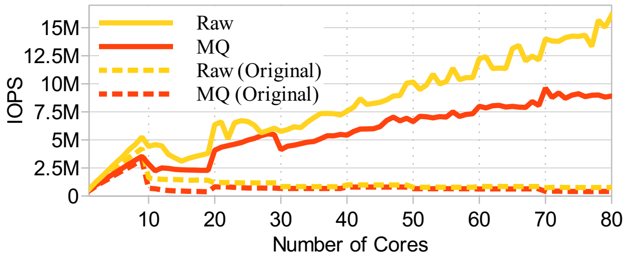

Figure 8: IOPS for MQ and raw with libaio fixes applied on the 8-nodes systems.

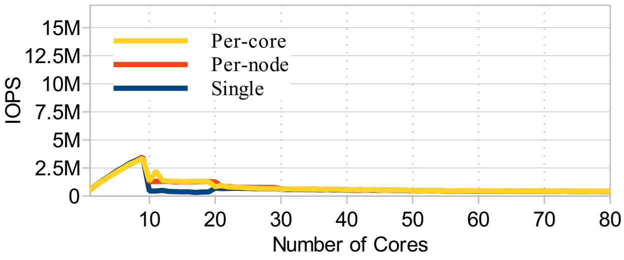

Figure 9: IOPS for a single software queue with varied number of mapped hardware dispatch queues on the 8 socket system.

5.2 Improving Application Level IO Submission

The throughput graphs from the previous Section exposed scalability problem within the Linux stack, on top of the block layer. Through profiling we were able to determine that the asynchronous (libaio) and direct IO layers, used within the kernel to transfer block IOs from userspace into to the block layer, have several bottlenecks that have are first being exposed with the new MQ block layer implementation. These bottlenecks are: (i) a context list lock is issued for each request, (ii) a completion ring in libaio used to manage sleep/wakeup cycles and (iii) a number of shared variables are being updated throughout the library. We removed these bottlenecks through a series of implementation improvements.

First, we replaced the context list lock with a lockless list, which instead of using mutexes to update a variable used the compare-and-swap instruction of the processor to perform the update. Second, we eliminated the use of the completion ring as it caused an extra lock access when updating the number of elements in the completion list, which in the worst case, could put the application process to sleep. Third, we used atomic compare-and-swap instructions to manipulate the shared counters for internal data structures (e.g. the number of users of AIO context) instead of the native mutex structures.

Figure 8 demonstrates the IOPS of the raw and MQ designs using this new userspace IO submission library, on the 8-socket system, which has the hardest time maintaining IO scalability. We observe that both the MQ and Raw implementations, while still losing efficiency when moving to a second socket, are able to scale IOPS near linearly up to the maximum number of cores within the system. The multiqueue design proposed here allows the block layer to scale up to 10 million IOPS utilizing 70s cores on an 8 socket NUMA system while maintaining the conveniences of the block layer implementation for application compatibility. We recognize that the efficiency of the MQ (and Raw) implementations drops significantly when moving from one socket onto a second. This indicates that there are further bottlenecks to be improved upon in Linux that lay outside the block layer. Possible candidates are interrupt handling, context switching improvements, and other core OS functions that we leave for future work.

5.3 Comparing Allocation of Software and Hardware Dispatch Queues

As our design introduces two levels of queues (the software and hardware dispatch queues), we must investigate how number of queues defined for each level impacts performance. We proceed in two steps. First, we fix the number of software level queues to one and we vary the number of hardware dispatch queues. Second, we fix the number of software level queues to one per core and we vary the number of hardware dispatch queues. In both experiments, the number of hardware dispatch queues is either one, one per core (denoted per-core) or one per socket (denoted per-socket). All experiments with the MQ block layer on on the 8-socket system.

Figure 9 presents throughput using a single software queue. We observe that all configurations show a sharp performance dip when the second socket is introduced. Furthermore, we observe that a single software and hardware dispatch queue perform significantly worse on the second socket, but follow each other when entering the third socket. Overall, this experiment shows that a single software queue does not allow the block layer to scape gracefully.

We show in Figure 10 the results of our experiments with

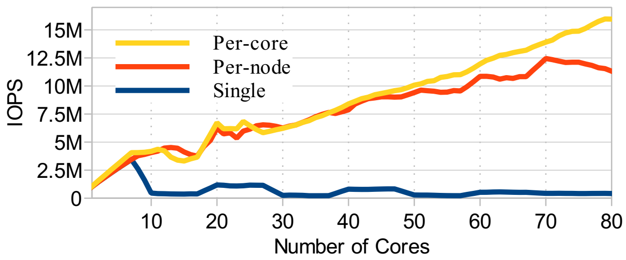

Figure 10: IOPS for per-core software queue with a different number of hardware dispatch queues.

a software queue per core. We see that combining multiple software and hardware dispatch queues enable high performance, reaching more than 15 million IOPS using the least contended per-core/per-core queue mapping. The per-node hardware dispatch queue configuration also scales well up to the fifth socket, but then the per-node slowly decrease in throughput. This occur when the socket interconnect becomes the bottleneck. Further performance is possible as more processes are added, but they slightly suffer from less available bandwidth. To achieve the highest throughput, per-core queues for both software and hardware dispatch queues are advised. This is easily implemented on the software queue side, while hardware queues must be implemented by the device itself. Hardware vendors can restrict the number of hardware queues to the system sockets available and still provide scalable performance.

6 Related Work

Our redesign of the block layer touch on network, hardware interfaces and NUMA systems. Below we describe related work in each of these fields.

6.1 Network

The scalability of operating system network stacks has been addressed in the last 10 years by incorporating multiple sender and receiver queues within a single card to allow a proliferation of network ports [1, 21]. This allows a single driver to manage multiple hardware devices and reduce code and data structure duplication for common functionality. Our work builds upon the foundation of networking multi-queue designs by allowing a single interface point, the block device, with multiple queues within the software stack.

Optimization of the kernel stack itself, with the purpose of removing IO bottlenecks, has been studied using the Moneta platform [7]. Caulfield et al. propose to bypass the block layer and implement their own driver and single queue mechanism to increase performance. By bypassing the block layer, each thread issues an IO and deals with its completion. Our approach is different, as we propose to redesign the block layer thus improving performance across all devices, for all applications.

6.2 Hardware Interface

The NVMe interface [18] attempts to address many of the scalability problems within the block layer implementation. NVMe however proposes a new dynamic interface to accommodate the increased parallelism in NVM storage on which each process has its own submission and completion queue to a NVM storage device. While this is excellent for scalability, it requires application modification and pushes much of the complexity of maintaining storage synchronization out of the operating system into the application. This also exposes security risks to the application such as denial of service without a central trusted arbitrar of device access.

6.3 NUMA

The affect of NUMA designs on parallel applications has been studied heavily in the HPC space [22, 30, 15, 20] and big data communities [29, 3, 5]. We find that many of these observations, disruptive interrupts, cache locality, and lock-contention have the same negative performance penalty within the operating system and block layer. Unfortunately, as a common implementation for all applications, some techniques to avoid lock contention such as message passing simply are in-feasible to retrofit into a production operating system built around shared memory semantics.

One approach to improving IO performance is to access devices via memory-mapped IO. While this does save some system call overhead, this does not fundamentally change or improve the scalability of the operating system block layer. Additionally, it introduces a non-powercut safe fault domain (the DRAM page-cache) that applications may be un-aware of while simultaneously requiring a large application re-write to take leverage.

7 Conclusions and Future Work

In this paper, we have established that the current design of the Linux block layer does not scale beyond one million IOPS per device. This is sufficient for today's SSD, but not for tomorrow's. We proposed a new design for the Linux block layer. This design is based on two levels of queues in order to reduce contention and promote thread locality. Our experiments have shown the superiority of our design and its scalability on multi-socket systems. Our multiqueue design leverages the new capabilities of NVM-Express or high-end PCI-E devices, while still providing the common interface and convenience features of the block layer.

We exposed limitations of the Linux IO stack beyond the block layer. Locating and removing those additional bottlenecks is a topic for future work. Future work also includes performance tuning with multiple hardware queues, and experiments with multiqueue capable hardware prototypes. As one bottleneck is removed, a new choke point is quickly created, creating an application through device NUMA-local IO-stack is an on-going process. We intend to work with device manufacturers and standards bodies to ratify the inclusion of hardware capabilities that will encourage adoption of the multiqueue interface and finalize this new block layer implementation for possible inclusion in the mainline Linux kernel.

8 References

[1] Improving network performance in multi-core systems. *Intel Corporation*, 2007.

[2] J. Axboe. Linux Block IO present and future. *Ottawa Linux Symposium*, 2004.

[3] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and S. Akhilesh. The multikernel: a new OS architecture for scalable multicore systems. *Symposium on Operating Systems Principles*, 2009.

[4] M. Bjørling, P. Bonnet, L. Bouganim, and N. Dayan. The necessary death of the block device interface. In *Conference on Innovative Data Systems Research*, 2013.

[5] S. Boyd-wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. *Operating Systems Design and Implementation*, 2010.

[6] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, C. Lam, and A. Luis. Phase change memory technology. *Journal of Vacuum Science and Technology B*, 28(2):223–262, 2010.

[7] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of The 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010.

[8] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. *SIGARCH Comput. Archit. News*, 40(1):387–400, Mar. 2012.

[9] S. Cho, C. Park, H. Oh, S. Kim, Y. Y. Yi, and G. Ganger. Active Disk Meets Flash: A Case for Intelligent SSDs. Technical Report CMU-PDL-11-115, 2011.

[10] Completely Fair Queueing (CFQ) Scheduler. `http://en.wikipedia.org/wiki/CFQ`.

[11] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. *Symposium on Operating Systems Principles*, page 133, 2009.

[12] Deadline IO Scheduler. `http://en.wikipedia.org/wiki/Deadline_scheduler`.

[13] M. Dunn and A. L. N. Reddy. A new I/O scheduler for solid state devices. *Texas A&M University*, 2010.

[14] fio. `http://freecode.com/projects/fio`.

[15] P. Foglia, C. A. Prete, M. Solinas, and F. Panicucci. Investigating design tradeoffs in S-NUCA based CMP systems. *UCAS*, 2009.

[16] Fusion-io ioDrive2. `http://www.fusionio.com/`.

[17] L. M. Grupp, J. D. David, and S. Swanson. The Bleak Future of NAND Flash Memory. *USENIX Conference on File and Storage Technologies*, 2012.

[18] A. Huffman. NVM Express, Revision 1.0c. *Intel Corporation*, 2012.

[19] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S. H. Noh. Disk Schedulers for Solid State Drives. In *EMSOFTâĂŹ09: 7th ACM Conf. on Embedded Software*, pages 295–304, 2009.

[20] F. Liu, X. Jiang, and Y. Solihin. Understanding How Off-Chip Memory Bandwidth Partitioning in Chip Multiprocessors Affects System Performance. *High Performance Computer Architecture*, 2009.

[21] S. Mangold, S. Choi, P. May, O. Klein, G. Hiertz, and L. Stibor. 802.11e Wireless LAN for Quality of Service. *IEEE*, 2012.

[22] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A Non-Uniform-Memory-Access Programming Model For High-Performance Computers. *The Journal of Supercomputing*, 1996.

[23] S. Park and K. Shen. FIOS: A Fair, Efficient Flash I/O Scheduler. In *USENIX Conference on File and Storage Technologies*, 2010.

[24] J. Parkhurst, J. Darringer, and B. Grundmann. From single core to multi-core: preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, 2006.

[25] PCI-SIG. PCI Express Specification Revision 3.0. Technical report, 2012.

[26] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[27] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[28] V. Vasudevan, M. Kaminsky, and D. G. Andersen. Using vector interfaces to deliver millions of iops from a networked key-value storage server. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[29] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1996.

[30] J. Weinberg. *Quantifying Locality In The Memory Access Patterns of HPC Applications*. PhD thesis, 2005.

[31] J. Yang, D. B. Minturn, and F. Hady. When Poll is Better than Interrupt. In *USENIX Conference on File and Storage Technologies*, 2012.

Chapter 4

I/O Speculation for the Microsecond Era

Microsecond latencies and access times will soon dominate most datacenter I/O workloads. With the introduction of enterprise SSDs that employs advance buffering techniques and new types of persistent memories, such as PCM [2], SSD write latency is approaching low tens microseconds [3][1].

A core principle for system design is to aim for high resource utilization. For example, a process issuing an IO to a slow storage device should yield the processor just after IO submission to let other processes do useful work. The resulting context switch requires 10-20us of in-kernel processing time. If I/Os complete with hundreds of millisecond latency, this overhead is negligible. If now I/Os complete with sub-millisecond latency, the context switch becomes significant. In that case, it can be beneficial to poll for I/O completion instead of yield [4]. However, for devices with low tens of microsecond latency, polling for 10-20us is a waste of resources.

Similarly, Asynchronous I/Os [1] allow applications to submit I/O without yield. Instead checks for completion are required at a later point. At that point, the processor could be executing code in another thread or on an interrupt context. Thus, an I/O completion causes rescheduling of process no matter what. We argue for a third method: Speculation. I/O speculation allows processes to continue execution, after submitting

[1] Intel P3700 SSD reports a 20us write latency

a write, without checking for I/O completion. In case, the I/O completes successfully, all is fine. Now, if the I/O fails to complete, some form of reparation or compensation must be enforced.

Several approaches to speculation are evaluated in our contribution "I/O Speculation in the Microsecond Era" published at the Usenix ATC 2014 conference.

The insight of speculation is that I/Os rarely fail. An application therefore can continue execution directly after submission of a synchronous I/O. Contrary to asynchronous I/O, there is no need to check for the following completion. Execution continues until there is a visible external side-effects. Such a side-effect would for example be additional I/O or irreversible system calls.

To understand which application types would benefit from I/O speculation, the paper evaluates three types of application I/O patterns: pure I/O, I/O intensive and compute intensive. For each type, the number of instructions is measured between visible side-effects. For a subset of applications, this amount to tens of thousands of instructions and those may benefit from speculation.

The paper evaluates several techniques to enable speculation. It is up to the technique to rollback to the previous execution point if an I/O fails. Four approaches were evaluated. Neither were optimal.

Two approaches use execution check-points. The first copies the process address space after I/O submission using copy-on-write (COW). On failure, it rolls back to previous check-point and reenter with the failed I/O logic. The COW creation was measure to evaluate this. Already at small process sizes it took more time than actually poll for I/O completion. The second study if the Intel TSX [5] transaction extension can be applied. This was quickly dismissed because of early abort and the scope of transactions.

Check-point free speculation was evaluated as well. In that case the application roll-back itself by keeping extra structures, or I/Os was buffered in the kernel. However, this similarly did not solve the problem, as I/Os must be buffered in kernel space and thereby copied, which also is a significant overhead for microsecond storage.

The contribution of this paper is the evaluation of current state-of-art in techniques for providing speculation. Future research can benefit from our work and avoid the common pitfalls.

Literature

[1] Janet Bhattacharya, Suparna and Pratt, Steven and Pulavarty, Badari and Morgan. Asynchronous I/O support in Linux 2.5. *Proceedings of the Linux Symposium*, 2003.

[2] HGST. HGST Research Demonstrates World's Fastest SSD. Technical report, Flash Memory Summit (FMS), 2014.

[3] Dejan Vučinić, Qingbo Wang, Cyril Guyot, Robert Mateescu, Filip Blagojević, Luiz Franca-neto, Damien Le Moal, Hgst San, Trevor Bunker, Jian Xu, Steven Swanson, San Diego, Santa Clara, and Zvonimir Bandi. DC Express : Shortest Latency Protocol for Reading Phase Change Memory over PCI Express. *12th USENIX Conference on File and Storage Technologies (FAST)*, 2014.

[4] Jisoo Yang, DB Minturn, and Frank Hady. When poll is better than interrupt. *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST)*, pages 1–7, 2012.

[5] Ravi Yoo, Richard M and Hughes, Christopher J and Lai, Konrad and Rajwar. Performance evaluation of Intel transactional synchronization extensions for high-performance computing. *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013.

I/O Speculation for the Microsecond Era

Michael Wei[†], Matias Bjørling[‡], Philippe Bonnet[‡], Steven Swanson[†]
[†]*University of California, San Diego* [‡]*IT University of Copenhagen*

Abstract

Microsecond latencies and access times will soon dominate most datacenter I/O workloads, thanks to improvements in both storage and networking technologies. Current techniques for dealing with I/O latency are targeted for either very fast (nanosecond) or slow (millisecond) devices. These techniques are suboptimal for microsecond devices - they either block the processor for tens of microseconds or yield the processor only to be ready again microseconds later. Speculation is an alternative technique that resolves the issues of yielding and blocking by enabling an application to continue running until the application produces an externally visible side effect. State-of-the-art techniques for speculating on I/O requests involve checkpointing, which can take up to a millisecond, squandering any of the performance benefits microsecond scale devices have to offer. In this paper, we survey how speculation can address the challenges that microsecond scale devices will bring. We measure applications for the potential benefit to be gained from speculation and examine several classes of speculation techniques. In addition, we propose two new techniques, hardware checkpoint and checkpoint-free speculation. Our exploration suggests that speculation will enable systems to extract the maximum performance of I/O devices in the microsecond era.

1 Introduction

We are at the dawn of the *microsecond era*: current state-of-the-art NAND-based Solid State Disks (SSDs) offer latencies in the sub-100μs range at reasonable cost [16, 14]. At the same time, improvements in network software and hardware have brought network latencies closer to their physical limits, enabling sub-100μs communication latencies. The net result of these developments is that the datacenter will soon be dominated by microsecond-scale I/O requests.

Device	Read	Write
Millisecond Scale		
10G Intercontinental RPC	100 ms	100 ms
10G Intracontinental RPC	20 ms	20 ms
Hard Disk	10 ms	10 ms
10G Interregional RPC	1 ms	1 ms
Microsecond Scale		
10G Intraregional RPC	300 μs	300 μs
SATA NAND SSD	200 μs	50 μs
PCIe/NVMe NAND SSD	60 μs	15 μs
10Ge Inter-Datacenter RPC	10 μs	10 μs
40Ge Inter-Datacenter RPC	5 μs	5 μs
PCM SSD	5 μs	5 μs
Nanosecond Scale		
40 Gb Intra-Rack RPC	100 ns	100 ns
DRAM	10 ns	10 ns
STT-RAM	<10 ns	<10 ns

Table 1: **I/O device latencies.** Typical random read and write latencies for a variety of I/O devices. The majority of I/Os in the datacenter will be in the microsecond range.

Today, an operating system uses one of two options when an application makes an I/O request: either it can block and poll for the I/O to complete, or it can complete the I/O asynchronously by placing the request in a queue and yielding the processor to another thread or application until the I/O completes. Polling is an effective strategy for devices with submicrosecond latency [2, 20], while programmers have used yielding and asynchronous I/O completion for decades on devices with millisecond latencies, such as disk. Neither of these strategies, however, is a perfect fit for microsecond-scale I/O requests: blocking will prevent the processor from doing work for tens of microseconds, while yielding may reduce performance by increasing the overhead of each I/O operation.

A third option exists as a solution for dispatching I/O requests, *speculation*. Under the speculation strategy, the operating system completes I/O operations speculatively, returning control to the application without yielding. The operating system monitors the application: in the case of a write operation, the operating system blocks the application if it makes a side-effect, and in the case of a read operation, the operating system blocks the application if it attempts to use data that the OS has not read yet. In addition, the operating system may have a mechanism to rollback if the I/O operation does not complete successfully. By speculating, an application can continue to do useful work even if the I/O has not completed. In the context of microsecond-scale I/O, speculation can be extremely valuable since, as we discuss in the next section, there is often enough work available to hide microsecond latencies. We expect that storage class memories, such as phase-change memory (PCM), will especially benefit from speculation since their access latencies are unpredictable and variable [12].

Any performance benefit to be gained from speculation is dependent upon the performance overhead of speculating. Previous work in I/O speculation [9, 10] has relied on checkpointing to enable rollback in case of write failure. Even lightweight checkpointing, which utilizes copy-on-write techniques, has a significant overhead which can exceed the access latency of microsecond devices.

In this paper, we survey speculation in the context of microsecond-scale I/O devices, and attempt to quantify the performance gains that speculation has to offer. We then explore several techniques for speculation, which includes exploring existing software-based checkpointing techniques. We also propose new techniques which exploit the semantics of the traditional I/O interface. We find that while speculation could allow us to maximize the performance of microsecond scale devices, current techniques for speculation cannot deliver the performance which microsecond scale devices require.

2 Background

Past research has shown that current systems have built-in the assumption that I/O is dominated by millisecond scale requests [17, 2]. These assumptions have impacted the core design of the applications and operating systems we use today, and may not be valid in a world where I/O is an order of magnitude faster. In this Section, we discuss the two major strategies for handling I/O and show that they do not adequately address the needs of microsecond-scale devices, and we give an overview of I/O speculation.

2.1 Interfaces versus Strategies

When an application issues a request for an I/O, it uses an *interface* to make that request. A common example is the POSIX `write/read` interface, where applications make I/O requests by issuing blocking calls. Another example is the POSIX asynchronous I/O interface, in which applications enqueue requests to complete asynchronously and retrieve the status of their completion at some later time.

Contrast interfaces with *strategies*, which refers to how the operating system actually handles I/O requests. For example, even though the `write` interface is blocking, the operating system may choose to handle the I/O asynchronously, yielding the processor to some other thread.

In this work, we primarily discuss operating system strategies for handling I/O requests, and assume that application developers are free to choose interfaces.

2.2 Asynchronous I/O - Yielding

Yielding, or the asynchronous I/O strategy, follows the traditional pattern for handling I/O requests within the operating system: when a userspace thread issues an I/O request, the I/O subsystem issues the request and the scheduler places the thread in an I/O wait state. Once the I/O device completes the request, it informs the operating system, usually by means of a hardware interrupt, and the operating system then places the thread into a ready state, which enables the thread to resume when it is rescheduled.

Yielding has the advantage of allowing other tasks to utilize the CPU while the I/O is being processed. However, yielding introduces significant overhead, which is particularly relevant for fast I/O devices [2, 20]. For example, yielding introduces contexts switches, cache and TLB pollution as well as interrupt overhead that may exceed the cost of doing the I/O itself. These overheads are typically in the microsecond range, which makes the cost of yielding minimal when dealing with millisecond latencies as with disks and slow WANs, but high when dealing with nanosecond devices, such as fast NVMs.

2.3 Synchronous I/O - Blocking

Blocking, or the synchronous I/O strategy, is a solution for dealing with devices like fast NVMs. Instead of yielding the CPU in order for I/O to complete, blocking prevents unnecessary context switches by having the application poll for I/O completions, keeping the entire context of execution within the executing thread. Typically, the application stays in a spin-wait loop until the I/O completes, and resumes execution once the device flags the I/O as complete.

Blocking prevents the CPU from incurring the cost of context switches, cache and TLB pollution as well as interrupt overhead that the yielding strategy incurs. However, the CPU is stalled for the amount of time the I/O takes to complete. If the I/O is fast, then this strategy is optimal since the amount of time spent waiting is much shorter than the amount of CPU time lost due to software overheads. However, if the I/O is in the milliseconds range, this strategy wastes many CPU cycles in the spin-wait loop.

2.4 Addressing Microsecond-Scale Devices

Microsecond-scale devices do not fit perfectly into either strategy: blocking may cause the processor to block for a significant amount of time, preventing useful work from being done, and yielding may introduce overheads that may not have been significant with millisecond-scale devices, but may exceed the time to access a microsecond scale device. Current literature [2, 20] typically recommends that devices with microsecond ($\geq 5\mu$s) latencies use the yielding strategy.

2.5 I/O Speculation

Speculation is a widely employed technique in which a execution occurs before it is known whether it is needed or correct. Most modern processors use speculation: for example, branch predictors resolve branches before the branch path has been calculated [15]. Optimistic concurrency control in database systems enables multiple transactions to proceed before conflicts are resolved [5]. Prefetching systems attempt to make data available before it is known to be needed [8]. In all these speculative systems, speculation has no effect on correctness – if a misspeculation occurs either it has no effect on correctness or the system can rollback state as if no speculation had occurred in the first place.

I/O requests are a good speculation candidate for several reasons. The results of an I/O request are simple and predictable. In the case of a write, the write either succeeds or fails. For a read, the request usually returns success or failure immediately, and a buffer with the requested data is filled. In the common case, I/Os typically succeed – failures such as a disk error or an unreachable host are usually exceptional conditions that do not occur in a typical application run.

We depict the basic process of speculation in Figure 1. In order to speculate, a speculative context is created first. Creating a speculative context incurs a performance penalty ($t_{speculate}$), but once the context is created, the task can speculatively execute for some time ($t_{spec\_execute}$), doing useful work until it is no longer safe to speculate (t_{wait}). In the meantime, the kernel can dispatch the I/O request asynchronously (t_{io_start}). Once the I/O request completes (t_{io_comp}), the kernel commits the speculative execution if the request is successful, or aborts the speculative execution if it is not.

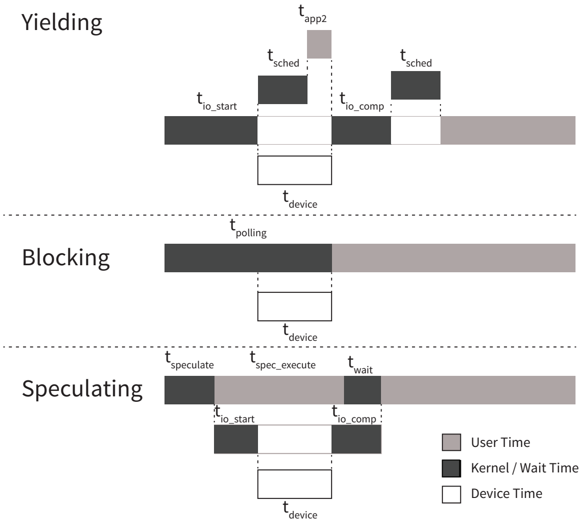

Figure 1: **Cost breakdown by strategy.** These diagrams show the relative costs for the yielding, blocking and speculating strategies.

In contrast to the blocking strategy, where the application cannot do useful work while the kernel is polling for the device to complete, speculation allows the application to perform useful work while the I/O is being dispatched. Compared to the yielding strategy, speculation avoids the overhead incurred by context switches.

This breakdown indicates that the perofrmance benefits from speculation hinges upon the time to create a speculative context and the amount of the system can safely speculate. If the cost is zero, and the device time (t_{device}) is short, then it is almost always better to speculate because the CPU can do useful work while the I/O is in progress, instead of spinning or paying the overhead of context switches. However, when t$_{device}$ is long compared to the time the system can safely speculate (t$_{spec_{e}xecute}$), then yielding will perform better, since it can at least allow another application to do useful work where the speculation strategy would have to block. When the time to create a context (t$_{speculate}$) is high compared to t_{device}, then the blocking strategy would be better since it does not waste cycles creating a speculative context which will be commited before any work is done.

For millisecond devices, yielding is optimal because t_{device} is long, so the costs of scheduling and context switches are minimal compared to the time it takes to dispatch the I/O. For nanosecond devices, blocking is optimal since t_{device} is short, so overhead incurred by either speculation or yielding will be wasteful. For microsecond devices, we believe speculation could be optimal if

Application	Description
bzip2	bzip2 on the Linux kernel source.
dc	NPB Arithmetic data cube.
dd	The Unix dd utility.
git clone	Clone of the Linux git repository.
make	Build of the Linux 3.11.1 kernel.
mongoDB	A 50% read, 50% write workload.
OLTP	An OLTP benchmark using MySQL.
postmark	E-mail server simulation benchmark.
tar	Tarball of the Linux kernel.
TPCC-Uva	TPC-C running on postgresql.

Table 3: **Applications.** A wide array of applications which we analyzed for speculation potential.

there are microseconds of work to speculate across, and the cost of speculating is low.

3 The Potential for Speculation

In order for speculation to be worthwhile, $t_{spec\_execute}$ must be significantly large compared to the cost of speculation and the device time. In order to measure this potential, we instrumented applications with Pin [13], a dynamic binary instrumentation tool, to measure the number of instructions between I/O requests and the point speculation must block. For writes, we measured the number of instructions between a write system call and side effects-causing system calls (for example, `kill(2)` but not `getpid(2)`), as well as writes to shared memory. For reads, we measure the number of instructions between a read system call and the actual use of the resulting buffer (for example, as a result of a read instruction to the buffer), or other system call, as with a write. Our estimate of the opportunity for speculation is an extremely conservative one: we expect that we will have to block on a large number of system calls that many systems we discuss in section 4 speculate through. However, by limiting the scope of speculation, our findings reflect targets for speculation that produce a minimal amount of speculative state.

We instrumented a wide variety of applications (Table 3), and summarize the results in Table 2. In general, we found applications fell into one of three categories: pure I/O applications, I/O intensive applications, and compute intensive applications. We briefly discuss each class below:

Pure I/O applications such as `dd` and `postmark` performed very little work between side-effects. For example, `dd` performs a read on the input file to a buffer, followed by write to the output file repeatedly. On average, these applications perform on the order of 100 instructions between I/O requests and side effects.

We also looked at database applications including `TPCC-Uva`, `MongoDB` and `OLTP`. These applications are I/O intensive, but perform a significant amount of compute between side effects. On average, we found that these applications perform on the order of 10,000 instructions between read and write requests. These workloads provide an ample instruction load for microsecond devices to speculate through.

Compute intensive applications such as `bzip2` and `dc` performed hundreds of thousands to millions of instructions between side-effects. However, these applications made I/O calls less often than other application types, potentially minimizing the benefit to be had from speculation.

Many of the applications we tested used the buffer following a `read` system call immediately: most applications waited less than 100 instructions before using a buffer that was read. For many applications, this was due to buffering inside libc, and for many other applications, internal buffering (especially for the database workloads, which often employ their own buffer cache) may have been a factor.

4 Speculation Techniques

In the next section, we examine several techniques for speculation in the context of the microsecond era. We review past work and propose new design directions in the context of microsecond scale I/O.

4.1 Asynchronous I/O Interfaces

While asynchronous I/O interfaces [1] are not strictly a speculation technique, we mention asynchronous I/O since it provides similar speedups as speculation. Indeed, just as in speculation, program execution will continue without waiting for the I/O to complete. However, asynchronous I/O requires the programmer to explicitly use and reason about asynchrony, which increases program complexity. In practice, while modern Linux kernels support asynchronous I/O, applications use synchronous I/O unless they require high performance.

4.2 Software Checkpoint Speculation

In order to perform speculation, software checkpointing techniques generate a *checkpoint*, which is a copy of an application's state. To generate a checkpoint, we call `clone(2)`, which creates a copy-on-write clone of the calling process. After the checkpoint has been created, the system may allow an application to speculatively continue through a synchronous I/O call before it completes (by returning control to the application as if the I/O had completed successfully). The system then monitors the application and stops it if it performs an action which produces an external output (for example, writing

	Writes			Reads		
Application	Instructions	Calls/s	Stop Reason	Instructions	Calls/s	Stop Reason
Pure I/O Applications						
`postmark`	74 ± 107	518	`close`	15 ± 11	123	buffer
`make (ld)`	115 ± 6	55	`lseek`	8,790 ± 73,087	180	`lseek` (31%) buffer (68%)
`dd`	161 ± 552	697	`write`	69 ± 20	698	`write`
`tar`	248 ± 1,090	1,001	`write` (90%) `close` (9%)	144 ± 11	1,141	`write`
`git clone`	1,940 ± 11,033	2,833	`write` (73%) `close` (26%)	14 ± 10	1,820	buffer
I/O Intensive Applications						
`MongoDB`	10,112 ± 662,117	13,155	`pwrite` (94%)	62 ± 196	<1	buffer
`TPCC-Uva`	11,390 ± 256,018	115	`write` (49%) `sendto` (22%)	37 ± 8	22	buffer
`OLTP`	22,641 ± 342,110	141	`pwrite` (79%) `sendto` (7%)	31 ± 21	19	buffer
Compute Intensive Applications						
`dc`	1,216,281 ± 13,604,751	225	`write`	8,677 ± 66,273	156	buffer
`make (cc1)`	1,649,322 ± 819,258	12	`write`	165 ± 21	431	buffer
`bzip2`	43,492,452 ± 155,858,431	7	`write`	1,472 ± 345,827	18	buffer

Table 2: **Speculation Potential.** Potential for speculation in the read/write path of profiled applications. We list only the stop reasons that occur in >5% of all calls. Error numbers are standard deviations, and "buffer" indicates that speculation was stopped due to a read from the read buffer.

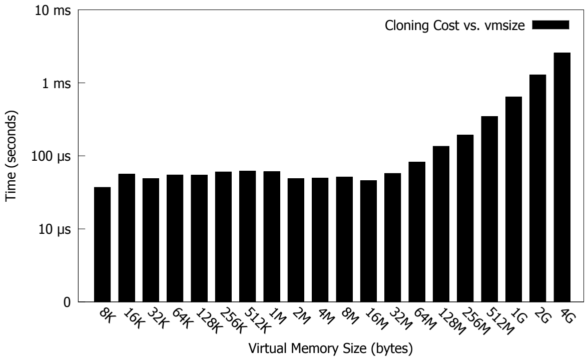

Figure 2: **Cloning Cost.** The cost of copy-on-write cloning for applications of various virtual memory sizes. Note that the axes are in log scale.

a message to a screen or sending a network packet) and waits for the speculated I/O to complete. If the I/O fails, the system uses the checkpoint created earlier to restore the application state, which allows the application to continue as if the (mis)speculation never occurred in the first place.

Software-based checkpointing techniques are at the heart of a number of systems which employ speculation, such as Speculator [9] and Xsyncfs [10]. These systems enabled speculative execution in both disks and over distributed file systems. These systems are particularly attractive because they offer increased performance without sacrificing correctness. Checkpoint-based speculation techniques hide misspeculations from the programmer, enabling applications to run on these systems unmodified.

However, providing the illusion of synchrony using checkpoints has a cost. We examined the cost of the clone operation, which is used for checkpointing (Figure 2). We found that for small applications, the cost was about 50μs, but this cost increased significantly as the virtual memory (vm) size of the application grew. As the application approached a vm size of 1GB, the cloning cost approached 1ms. While these cloning latencies may have been a small price to pay for slower storage technologies, such as disk and wide area networks, the cost of cloning even the smallest application can quickly eclipse the latency of a microsecond era device. In order for checkpoint-based speculation to be effective, the cost of taking a checkpoint must be minimized.

4.3 Hardware Checkpoint Speculation

Since we found checkpointing to be an attractive technique for enabling speculation given its correctness properties, creating checkpoints via hardware appeared to be a reasonable approach to accelerating checkpointing. Intel's transactional memory instructions, introduced with the Haswell microarchitecture [21] seemed to be a good match. Hardware transactional memory support has the potential of significantly reducing the cost of speculation, since speculative execution is similar to transactions. We can wrap speculative contexts into transactions which are committed only when the I/O succeeds. Checkpoints would then be automatically created and tracked by hardware, which buffers away modifications until they are ready to be committed.

We examined the performance of TSX and found that the cost of entering a transactional section is very low (<20 ns). Recent work [18, 21] suggests that TSX transaction working sets can write up to 16KB and <1 ms with low abort rates (<10%). While TSX shows much promise in enabling fast, hardware-assisted checkpointing, many operations including some I/O operations, cause a TSX transaction to abort. If an abort happens for any reason, all the work must be repeated again, significantly hampering performance. While hardware checkpoint speculation is promising, finer-grained software control is necessary. For example, allowing software to control which conditions cause an abort as well as what happens after an abort would enable speculation with TSX.

4.4 Checkpoint-Free Speculation

During our exploration of checkpoint-based speculation, we observed that the created checkpoints were rarely, if ever used. Checkpoints are only used to ensure correctness when a write fails. In a system with local I/O, a write failure is a rare event. Typically, such as in the case of a disk failure, there is little the application developer will do to recover from the failure other than reporting it to the user. Checkpoint-free speculation makes the observation that taking the performance overhead of checkpointing to protect against a rare event is inefficient. Instead of checkpointing, checkpoint-free speculation makes the assumption that every I/O will succeed, and that only external effects need to be prevented from appearing before the I/O completes. If a failure does occur, then the application is interrupted via a signal (instead of being rolled back) to do any final error handling before exiting.

Unfortunately, by deferring synchronous write I/Os to after a system call, the kernel must buffer the I/Os until they are written to disk. This increases memory pressure and requires an expensive memory copy for each I/O. We continue to believe that checkpoint-free speculation, if implemented together with kernel and user-space processes to allow omitting the memory copy, will result in a significant performance increase for microsecond-scale devices.

4.5 Prefetching

While the previous techniques are targeted towards speculating writes, prefetching is a technique for speculating across reads. In our characterization of speculation potential, we found that speculating across read calls would be ineffective because applications are likely to immediately use the results of that read. This result suggests that prefetching would be an excellent technique for microsecond devices since the latency of fetching data early is much lower with microsecond era devices, reducing the window of time that a prefetcher needs to account for. We note that the profitability of prefetching also decreases with latency - it is much more profitable to prefetch from a microsecond device that a nanosecond device.

Prefetching already exists in many storage systems. For example, the Linux buffer cache can prefetch data sequentially in a file. However, we believe that more aggressive forms of prefetching are worth revisiting for microsecond scale devices. For example, SpecHint and TIP [3, 11] used a combination of static and dynamic binary translation to speculatively generate I/O hints, which they extended into the operating system [4] to improve performance. Mowry [7] proposed a similar system which inserted I/O prefetching at compile time to hide I/O latency. Since microsecond devices expose orders of magnitude more operations per second than disk, these aggressive techniques will be much more lucrative in the microsecond era.

4.6 Parallelism

Other work on speculation focuses on using speculation to extract parallelism out of serial applications. For example, Wester [19] introduced a speculative system call API which exposes speculation to programmers, and Fast Track [6] implemented a runtime environment for speculation. This work will likely be very relevant since microsecond devices expose much more parallelism than disk.

5 Discussion

As we have seen, a variety of different techniques exist for speculating on storage I/O, however, in their current state, no technique yet completely fulfills the needs of microsecond scale I/O.

Our study suggests that future work is needed in two areas. First, more work is needed to design appropriate hardware for checkpointing solutions. Second, the opportunity for checkpoint-free speculation needs to be studied in depth for both compute intensive and I/O intensive database applications.

6 Conclusion

This paper argues for the use of speculation for microsecond-scale I/O. Microsecond-scale I/O will soon dominate datacenter workloads, and current strategies are suboptimal for dealing with the I/O latencies that future devices will deliver. Speculation can serve to bridge that gap, providing a strategy that enables I/O intensive applications to perform useful work while waiting for I/O to complete. Our results show that the performance of microsecond-scale I/Os can greatly benefit from speculation, but our analysis of speculation techniques shows that the cost of speculation must be minimized in order to derive any benefit.

7 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. DGE-1144086, as well as C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[1] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan. Asynchronous I/O support in Linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, 2003.

[2] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 387–400. ACM, 2012.

[3] F. Chang and G. A. Gibson. Automatic i/o hint generation through speculative execution. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 1–14, Berkeley, CA, USA, 1999. USENIX Association.

[4] K. Faser and F. Chang. Operating System I/O Speculation: How Two Invocations Are Faster Than One. In *USENIX Annual Technical Conference, General Track*, pages 325–338, 2003.

[5] J. Huang, J. A. Stankovic, K. Ramamritham, and D. F. Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *VLDB*, volume 91, pages 35–46, 1991.

[6] K. Kelsey, T. Bai, C. Ding, and C. Zhang. Fast Track: A software system for speculative program optimization. In *Proceedings of the 7th annual IEEE/ACM International Symposium*, pages 157–168, 2009.

[7] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted i/o prefetching for out-of-core applications. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 3–17, New York, NY, USA, 1996. ACM.

[8] K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th Conference on Parallel Architectures*, pages 135–145, 2004.

[9] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 24(4):361–392, 2006.

[10] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. *ACM Transactions on Computer Systems (TOCS)*, 26(3):6, 2008.

[11] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 79–95. ACM, 1995.

[12] M. Qureshi, M. Franceschini, A. Jagmohan, and L. Lastras. Preset: Improving performance of phase change memories by exploiting asymmetry in write times. In *39th Annual International Symposium on Computer Architecture (ISCA)*, pages 380–391. IEEE, 2012.

[13] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. PIN: a binary instrumentation tool for computer architecture research and education. In *Proceedings of the 2004 workshop on Computer architecture education*, 2004.

[14] S.-H. Shin, D.-K. Shim, J.-Y. Jeong, O.-S. Kwon, S.-Y. Yoon, M.-H. Choi, T.-Y. Kim, H.-W. Park, H.-J. Yoon, Y.-S. Song, et al. A new 3-bit programming algorithm using SLC-to-TLC migration for 8MB/s high performance TLC NAND flash memory. In *VLSI Circuits (VLSIC), 2012 Symposium on*, pages 132–133. IEEE, 2012.

[15] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148, 1981.

[16] H. Tanaka, M. Kido, K. Yahashi, M. Oomura, R. Katsumata, M. Kito, Y. Fukuzumi, M. Sato, Y. Nagata, Y. Matsuoka, et al. Bit cost scalable technology with punch and plug process for ultra high density flash memory. In *IEEE Symposium on VLSI Technology*, pages 14–15. IEEE, 2007.

[17] H. Volos. Revamping the system interface to storage-class memory, 2012. PhD Thesis. University of Wisconsin at Madison.

[18] Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 26:1–26:15, New York, NY, USA, 2014. ACM.

[19] B. Wester, P. M. Chen, and J. Flinn. Operating System Support for Application-specific Speculation. In *Proceedings of the Sixth European conference on Computer systems*, pages 229–242. ACM, 2011.

[20] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012.

[21] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel(R) Transactional Synchronization Extensions for High-Performance Computing. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, page 19. ACM, 2013.

Chapter 5

LightNVM: Lightning Fast Evaluation Platform for Non-Volatile Memories

The previous papers argue for a form of cross-layer optimization involving applications, operating system and SSDs. The approach I have taken for this thesis led me to consider cross-layer optimizations in the context of actual software components. More specifically I intended to study cross layer optimizations in the context of the Linux I/O stack. None of the existing simulators enabled me to explore this design space. I thus decided to design my own.

Before my thesis, trace-based simulators, such as DiskSim with SSD extensions [1], FlashSim [3] and others [4] were the only possibilities to model and explore SSD internals.

As SSDs become more complex and employ garbage collection in the background, they become non-deterministic. Their state, FTL and NAND flash health all contributes to the timing accuracy. For example, garbage collection might be triggered based on the state of the device, and therefore interfere with incoming user I/O. A trace-based simulator therefore should capture the state of NAND flash and FTL, as well as the SSD state at run-time.

To capture the run-time state, an SSD simulator must run as a real live SSD. A simulator that supports this is VSSIM [5]. It implements an SSD simulator in a virtual machine hypervisor, that exposes a live SSD as a block device inside the guest. Accurately tracking the behavior of an SSD implementation at run-time. The approach captures the internal state of SSD, while also reporting accurate timings. However, a single caveat influence the performance of the simulator. For each I/O a hypervisor context switch between host and guest is required. That effectively limits VSSIM to thousands of IOPS. Modern SSDs execute hundred of thousand of IOPS. This limits the evaluation background garbage collection routines, etc. that are time-dependent.

I designed the LightNVM simulator in order to simulate high-performance SSDs. To accomplish this, the simulator is implemented in the host kernel, instead of being encapsulated inside a virtual machine, and similarly to VSSIM, it exposes itself as a block device. Its flash mapping strategy, garbage collection, and internal logic are implemented within the host.

By implementing the logic within the host, it creates the foundation for host-managed SSDs. The LightNVM simulator may simulate timings using real hardware. Where non-volatile memory with its characteristics is embedded or simulated [2]. The management of flash is thereby always managed by the host, similarly to how direct attach flash is managed by the host. However, our solution is different in that the flash controller continues to managed the physical flash. The host only takes control of internal data placement. Thereby getting the best of both worlds.

The simulator is described in the paper "LightNVM: Lightning Fast Evaluation Platform for Non-Volatile Memories" and presented at the Non-Volatile Memory Workshop (NVMW) 2014. It presents the evaluation of four types of SSDs. The OpenSSD platform, HGST NVMe null block PCI-e device, simulated NAND flash, and a memory backend, offering a large amount of variability.

Integration with the OpenSSD platform shows that the simulator can be applied to SATA/SAS interfaces and run in a real SSD. The HGST drives shows the overhead of faster interfaces. The simulated NAND flash allows us to experiment with various NAND flash latencies, and at last the memory backend exposes bottlenecks in the host stack.

The key contribution consists of a high-performance SSD simulator, that execute with a large range of flexibility. It is also the beginning of the Open-Channel SSD project. It allows the host FTL to control one or more SSDs and take advantage of data placement, controlled garbage collection and exposing several SSDs under a single level address space. This simplifies hardware firmware and puts detailed control from the host. I believe this is the right approach to obtain predictability and consistent performance of future SSDs.

Literature

[1] John S Bucy, Jiri Schindler, Steven W Schlosser, and Gregory R Ganger. The DiskSim Simulation Environment. *Parallel Data Laboratory*, 2008.

[2] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 385–395, December 2010.

[3] Youngjae Kim, Brendan Tauras, Aayush Gupta, and Bhuvan Urgaonkar. FlashSim: A Simulator for NAND Flash-Based Solid-State Drives. *2009 First International Conference on Advances in System Simulation*, pages 125–131, September 2009.

[4] Jongmin Lee, Eujoon Byun, Hanmook Park, Jongmoo Choi, Donghee Lee, and Sam H Noh. CPS-SIM: Configurable and Accurate Clock Precision Solid State Drive Simulator. *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 318–325, 2009.

[5] Jinsoo Yoo, Youjip Won, Joongwoo Hwang, and Sooyong Kang. Vssim: Virtual machine based ssd simulator. *Proceeding of Mass Storage Systems and Technologies (MSST)*, 2013.

LightNVM: Lightning Fast Evaluation Platform for Non-Volatile Memories

Matias Bjørling†, Jesper Madsen†, Philippe Bonnet†, Aviad Zuck‡, Zvonimir Bandic∗, Qingbo Wang∗
†IT University of Copenhagen, ‡Tel Aviv University, ∗HGST San Jose Research Center
†{mabj, jmad, phbo}@itu.dk, ‡aviadzuc@tau.ac.il, ∗{zvonimir.bandic,qingbo.wang}@hgst.com

Abstract

The IO performance of storage devices has increased by three orders of magnitude over the last five years. This is due to the emergence of solid state drives (SSDs) that wire in parallel tens of non-volatile memory chips. How can software systems keep up with this radical evolution? Commercial SSDs are black boxes, whose behavior and performance profile are largely undocumented. Today, the evaluation of SSD-based algorithms and systems is thus, mostly, trace-driven on top of simple SSD models. Only few academic groups or software companies have the resources to implement their own platform and lead a principled exploration of the design space for SSD-based systems. In order to lower the barrier for the evaluation of systems and software research on top of NVM-based SSDs, we introduce LightNVM, a new SSD evaluation platform. Our evaluation shows that LightNVM is fast, scalable and modular enough to capture the characteristics of actual SSD as well as simulated low-latency memory components.

1. Introduction

Solid State Drives, based on non volatile memories (NVM), constitute a radical departure from traditional, magnetic disk-based, secondary storage. First, they exhibit orders of magnitude improvements in terms of performance, with sub-millisecond access times, and millions of IO per seconds for a single device. Second, the complexity they introduce on each device changes the nature of the storage abstraction exported to the Operating System. As a result, designing SSD-based algorithms and systems requires challenging the assumptions and design principles that have been defined over the past thirty years with magnetic disks. But how can we explore this design space?

One option is to work with proprietary platforms, such as FusionIO ioDrives, [9, 6], or the Moneta platform designed at UC San Diego [3]. Such outstanding work is the result of multi-year efforts, so this option cannot be generalized to the systems and algorithms communities at large. Only few companies or research groups have the necessary expertise or resources to develop and maintain their own development platform. A second option, is to rely on trace-based simulators [7, 5] or memory-backed emulators. E.g. VSSIM [10]. However, generating realistic workloads for such simulators is difficult and requires abstracting away significant portions of the IO subsystem complexity. A third option is to rely on a hardware SSD evaluation platform [3, 4, 1, 8]. Such platforms are readily available and allow the evaluation of real-time workloads; but they are specific to a given technology and are quickly outdated. We summarize and compare some of these options in Table 1.

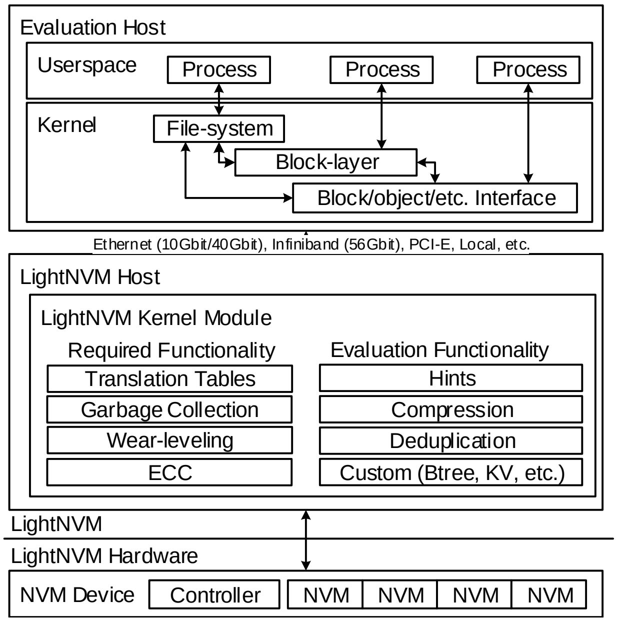

Figure 1: LightNVM architecture. Upper part is software-based, while lower part implements the LightNVM hardware interface.

In this work, we propose a new option to researchers and practitioners interested in evaluating SSD-based systems. We propose LightNVM, a new SSD research platform, that uses publicly available hardware to evaluate novel ideas, but also allows simulation whenever hardware is inadequate or unnecessary. LightNVM thus combines the robustness of hardware based platform and the flexibility of simulators and emulators.

2. LightNVM Design

The LightNVM architecture, detailed in Figure 1, consists of two layers: the hardware layer and the software layer. The LightNVM hardware subsystem can be organized in two different ways:

1. *In-memory*. In this mode, LightNVM relies on a simulated, memory-backed storage for IO operations. Waiting times are simulated to reflect the physical characteristics of the SSD and of the underlying NVM components. For simplicity, the In-memory hardware layer abstracts the SSD internals, as a collection of hardware channels, each supporting NVM operations with fixed access time. The in-memory hardware layer is responsible for (1) serializing accesses to each hardware

Platform	BlueSSD	VSSIM	OpenSSD	FRP	Moneta	LightNVM
Type	Custom HW	SW	HW	Custom HW	Custom HW	HW/SW
NVM	NAND	NAND	NAND	NAND	PCM	NAND/PCM/etc.
Interface	SATA	N/A	SATA	PCI-E/Net	PCI-E/Net	SATA/PCI-E/Net/Local
Cost	Low	N/A	Low	High	High	N/A / Low
Processing Power	Low	N/A	Low	High	Low	High

Table 1: Architecture of Evaluation Systems

channel (to reflect the limits of SSD parallelism), and (2) implementing kernel page copy to private storage (to reflect the constraints on NVM access time). This approach exhibits noticeable software overhead. It is thus less appropriate to study very high throughput workloads. However it is sufficient for less stressful workloads and for evaluating the feasibility of new SSD features, e.g., new storage hints,.

2. *With hardware*. In this mode, the LightNVM hardware layer integrates a full-fledged SSD hardware platform, e.g. the OpenSSD hardware platform. OpenSSD is a NAND flash-based SSD exposed through the SATA interface and offers a small programmable controller. Additionally the SATA, Flash, ECC functionalities are offloaded onto dedicated controllers. Management of mapping information, GC, wear-leveling can be handled within the device itself. However, to enable easy prototyping, LightNVM moves these crucial features outside of the firmware, into the host, in the context of a LightNVM kernel module coupled with a custom OpenSSD firmware. This design allows the LightNVM hardware layer to efficiently handle the control paths within the host. This full-fledged hardware layer enables experiments that use the superior computing resources of the host to perform storage actions, thus transforming the host into a device controller. This design also simplifies development, as the minimal firmware embedded on the device does not need to be changed to incorporate new developments within the hardware layer. A device may also provide only a subset of the SSD features. E.g., NVM hardware mode, where the capabilities of the hardware components are complemented by simulated components to support the LightNVM hardware layer interface.

The IOPS write performance of several hardware layer designs is shown in Figure 2. The throughput obtained for random 4KB write requests with four different hardware layer configurations is compared to the throughput results published for two existing evaluation platforms FRP [4] and Moneta-D [2]. The four hardware layer configurations are OpenSSD, NVMe device, Simulated NAND, and In-memory (Denoted as LNVM-OpenSSD, LNVM-NVMe, LNVM-SimNAND and LNVM-Mem in Figure 2). The graph shows the write IOPS for each of the platforms. The OpenSSD exposes the raw NAND operations, while the NVMe device shows the overhead of communicating with the hardware. It completes an IO as it is submitted and is currently limited by the PCI-E bus. The LNVM-SimNAND simulates an 8 channel SSD with NAND flash, and at last the LNVM-Mem shows the IOPS without simulating NVM timings.

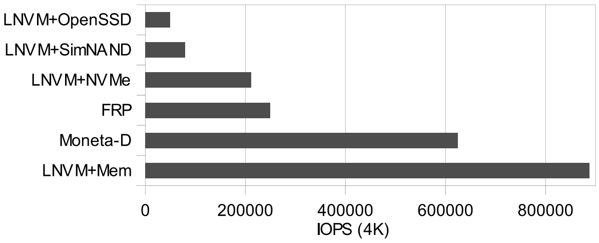

Figure 2: Write IOPS performance measured for four LightNVM hardware configurations, compared to FRP and Moneta published performance.

The software layer of LightNVM is implemented within the Linux kernel to exhibit the highest control of process scheduling, and to make NVM timings as accurate as possible. It implements logical-to-physical address translation, garbage collection, wear-leveling, etc. and allows further extensions to be easily implemented, such as hints, features and custom interfaces. The platform is specifically optimized for scalability, high IOPS, low latency and low overhead. It employs data structures such as per-core reference counting for in-flight data, per-core accounting for internal counters, and offers a streamlined design, that lets a device either be accessed through a block interface or byte-addressable interface, depending on the requirements of the controller. Currently only 4K block IOs are supported, but it is trivial to extend to a byte-addressable mapping tables. Compared to full hardware solutions, the platform exhibits overheads in the form of increased host CPU utilization and memory requirements. However, this is minor, compared to flexibility achieved. Additionally, LightNVM is being pushed toward the Linux kernel as a separate work, allowing LightNVM to be shipped in future kernel versions and lower the barrier for experimentation on SSD-based systems.

References

[1] "OpenSSD Platform." [Online]. Available: http://openssd-project.org
[2] A. M. Caulfield, "Providing safe, user space access to fast, solid state disks," in *SIGARCH*, vol. 40, no. 1. ACM, 2012, pp. 387–400.
[3] A. M. Caulfield *et al.*, "Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories," *Micro*, 2010.
[4] J. D. Davis *et al.*, "Frp: A nonvolatile memory research platform targeting nand flash," in *WISH*, 2009.
[5] B. Dayan *et al.*, "EagleTree: Exploring the Design Space of SSD-Based Algorithms ," in *VLDB (Demo)*, 2013.
[6] W. K. Josephson *et al.*, "DFS: A File System for Virtualized Flash Storage," *ACM Transactions on Storage*, vol. 6, no. 3, pp. 1–25, Sep. 2010.
[7] Y. Kim *et al.*, "FlashSim: A Simulator for NAND Flash-Based Solid-State Drives," *SIMUL*, pp. 125–131, Sep. 2009.
[8] S. Lee *et al.*, "Bluessd: an open platform for cross-layer experiments for nand flash-based ssds," in *WARP*, 2010.
[9] X. Ouyang *et al.*, "Beyond block I/O: Rethinking traditional storage primitives," in *HPCA*. IEEE, 2011, pp. 301–311.
[10] J. Yoo *et al.*, "Vssim: Virtual machine based ssd simulator," in *MSST*, 2013.

Chapter 6

Open-Channel Solid State Drives

This Section builds on the host-based simulator described in the previous Section [1], and introduces a host-managed SSD solution named *Open-Channel SSDs*. The term was first coined by the Baidu's SDF project [5, 4], which exposes individual flash units embedded on a SSD as independent block devices to the host. Thus allowing applications to directly control each flash unit (LUN) separately. This was a departure from the traditional SSDs, where SSD complexity (i.e., both hardware and software) is hidden behind the block device interface. With SDF, it is no longer the SSD but the host, more specifically each application, which is handling both SSD parallelism and NAND flash constraints. The operating system is tricked into considering a SSD as a collection of block devices.

I reuse this term and make it more general. I denote an open-channel SSD as a SSD that is host-managed, either with a full-fledged host-based FTL or using some form of hybrid approach, regardless of the nature of the storage abstraction exposed by the SSD to the host.

I propose a generic framework for (i) making open-channel SSDs accessible through Linux, and (ii) specializing part or all of the host-based FTL for a given application. Concretely, my contribution consists of an interface specification and a kernel library which application and system designers can expand to program SSDs. This work is thus the first generic framework for applications SSD co-design. Note that the name LightNVM has stuck and is now applied to both the simulator and the open channel SSD host driver.

In contrast to Baidu SDF [4] and Fusion-io VSL [3] architectures that either exposes the flash as a collection of block devices (one per LUN) or as a single block device accessed through a proprietary host-based driver, I propose a new form of storage abstraction which is available for a range of different open channel SSDs. In contrast to NVMKV [3], NVMFS [2], Baidu LSM [5] we do not require that applications take full responsibility for SSD management. Instead, we introduce operating system abstractions, as an extension of the Linux storage stack, that are responsible for SSD management.

Our definition of open-channel SSDs opens up a design space in terms of how responsibilities are split between host and SSD. At one extreme, the FTL is embedded on the SSD. At another extreme, the host is fully responsible for the FTL without any form of management embedded on the SSD. In the following paper, I consider an hybrid approach. I assume that the logic that can be kept within the SSD is kept within the SSD. Thus, the novelty of this paper is not that the SSD exposes direct control to both operating system and applications. Its core contribution is a hybrid interface, that allows host and SSDs to establish which functionalities remain on the SSD, while providing applications control of (i) data placement, (ii) garbage collection, and (iii) scheduling. Exploring this design space is crucial to enable host software to scale with high-performance SSDs.

Several industry vendors as well as academic partners are showing strong support for open channel SSDs. At the time of writing (May 2015), Memblaze, PMC Sierra, Micron, IIT Madras and several other companies have working or in progress implementation of the specification. They are the pioneers of the technology and the feedback from their implementations is incorporated into the specification. Our longer term goal is to push for the integration of our specification into standard organizations such as NVM Express and RapidIO.

The library is being upstreamed to the Linux kernel and has been described by Jonathan Corbet in Linux Weekly News (LWN)[1]. It has been presented at both the Linux Software Foundation Vault 2015 Conference and Non-Volatile Memory Workshop (NVMW) 2015.

To become a long term success, open channel SSDs must overcome the following two obstacles: (i) *Hardware support.* There are today prototype open channel SSDs from different vendors (in my office). We know that such devices have been used for some time by leading cloud service providers, but this is not documented. SSD and controller vendors

[1] `$http://lwn.net/Articles/641247/$`

tell us that they expect many commercial open channel SSDs mid 2016. (ii) *Application support.* We expect key value stores to be able to leverage open channel SSDs in a straightforward way. This is the first application I study in the following paper. The reason is that KV store can be accelerated without any re-engineering. Other data-intensive systems, e.g., relational database systems will need to be redesigned to full leverage a SSD co-design approach. This is a topic for future work.

Literature

[1] Matias Bjø rling, Jesper Madsen, Philippe Bonnet, Aviad Zuck, Zvonimir Bandic, and Qingbo Wang. LightNVM: Lightning Fast Evaluation Platform for Non-Volatile Memories. In *Non-Volatile Memories Workshop*, NVMWorkshop'14, 2014.

[2] William K. Josephson, Lars a. Bongo, Kai Li, and David Flynn. DFS: A File System for Virtualized Flash Storage. *ACM Transactions on Storage*, 6(3):1–25, September 2010.

[3] Leonardo Mármol, Swaminathan Sundararaman, Nisha Talagala, Raju Rangaswami, Sushma Devendrappa, Bharath Ramsundar, and Sriram Ganesan. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. *6th USENIX Workshop on Hot Topics in Storage and File Systems*, 2014.

[4] Jian Ouyang, Shiding Lin, S Jiang, and Z Hou. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, 2014.

[5] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, pages 1–14, 2014.

Open-Channel Solid State Drives
Technical Report

Matias Bjørling Jesper Madsen Philippe Bonnet
IT University of Copenhagen

Abstract

Solid state drives (SSDs) are based on tens of non-volatile memory chips wired in parallel to a storage controller. Most SSDs embed a flash translation layer that provides a full virtualization of the physical storage space. This way, SSDs appear to the operating system as regular block devices that can readily replace hard disk drives. While this approach has favoured SSDs' widespread adoption, it is now becoming an increasing problem in the context of data-intensive applications. Indeed, as SSD performance keeps on improving, redundancies and missed optimizations between host and SSDs, across the block device interface, lead to performance bottlenecks. To align data placement optimally on flash and control predictability, we consider Open-Channel SSDs. They are a hybrid between SSDs that expose its physical storage space directly to the host and conventional SSDs. We describe a common interface (LightNVM) for Open-Channel SSDs, which allows SSDs to describe extensions and responsibilities in addition to its physical flash characteristics. We describe the design and implementation of an open-channel SSD management library for Linux andwe evaluate our implementation with real hardware as well as a null device to identify performance bottlenecks. Our experiments show that LightNVM has low overhead and that the LightNVM library enables applications to tightly integrate with SSD data placement and management logic.

1 Introduction

The advent of high performance SSDs is exposing bottlenecks at the software level, throughout the I/O stack [2]. In the case of flash-based SSDs, complex decisions about mapping and scheduling are taken on both sides of the block device interface. Applications map their data structures onto the logical address space, which SSDs then map onto the physical address space. However, applications no longer have robust performance models to motivate their mapping, while the Flash Translation Layer (FTL) embedded on SSDs guesses I/O locality in order to avoid contention, optimize its mapping, and reduce GC overhead on NAND flash chips. This leads to waste of resources, unpredictable performance and bottlenecks [24, 20, 2, 19].

There is thus a need for a tighter form of collaboration between data-intensive applications, operating system and FTL to reconcile the complexity of SSD management with the high-performance goals of modern applications. Examples of tight integration can be found in the context of database systems, with database appliances (e.g., Exascale[1] or Netezza[2]) or custom-firmware SSD for SQL Server [**?**]. But, how about applications running in large-scale data centers? How can data-center SSD arrays implement optimized Flash Translation algorithms to cater to the (ever changing) idiosyncrasies of a given application workload? A solution is to decouple SSD management from physical storage. More concretely, a solution is to consider open-channel SSDs, i.e., SSDs that expose the physical storage space directly to the host [20, 16]. The problem then, is to organize SSD management, in a way that efficiently supports high-performance, data-intensive applications. This is the problem we tackle in this paper.

2 Requirements

In order to integrate SSDs a mismatch between SSD and host, the primary requirement is that *there should be a single level of indirection between application data structures and physical storage*. From a system design point of view, that gives us three options:

1. *Physical storage directly supports application-level data structures.* Incremental improvements in this

[1] `http://www.oracle.com/engineered-systems/exadata`
[2] `http://www-01.ibm.com/software/data/netezza/`

direction have been proposed with new storage primitives such as nameless writes [**?**], aimed to delegate data placement and allocation to the SSD, or atomic write [20, **?**], aimed to leverage Copy on Write within the SSD driver and thus avoid double buffering at the database system level.

2. *Application-level data structures are directly mapped onto physical storage.* Memory-mapped persistent data structures have been proposed such as NV-heaps [**?**] and Mnemosyne [**?**], as well as mechanisms to allocate persistent data structures explicitly [**?**, **?**] or implicitly [**?**] within the application. Such memory-mapped persistent data structures target memory drives [4] or open-channel SSDs [20, 16].

3. *A SSD allocation layer.* The third option is to consider a layer, independent from applications and storage devices, whose primary goal is to manage the interactions between application data structures and physical storage. This requires that physical storage is directly exposed to the host, which is the case with open-channel SSDs. It is the option we explore in this paper. More specifically, we extend and open up the functionalities of solid state drives so that data-intensive applications (such as key-value stores or database systems) can interact efficiently.

We propose an interface, that not only exposes SSDs non-volatile memory directly to the host, but also takes hardware acceleration into account. A SSD already employs a large amount of processing power, and thus even if data placement, garbage collection, etc. are host-managed, several extensions can be made to offload the host.

Given that SSD logic is moved into the host, we consider the following requrements for a sucessful integration:

- **Hardware-independent**: A SSD management layer should accommodate various storage devices, as long as they export physical storage directly to the host. In fact, the management layer can efficiently organize interactions with multiple devices.

  The main problem here is to deal with physical storage on the host. As opposed to the FTL embedded on a SSD controller and expose a read/write interface, the processor cores running the operating system are not directly wired to the underlying flash chips. A second problem is to efficiently integrate multiple SSDs and present them as a uniform storage space available to applications.

- **Extensible**: An generic management library should be specialized and extended to adapt to various application-level data structures (e.g., key-value pairs or index records).

- **Low overhead**: SSD performance has been tuned over years. Introducing significant overhead by moving logic into the host is not acceptable. The overhead introduced by the host-side mapping layer should be low.

- **Durable and consistent**: When data has been written by the application onto the SSD, it remains durable, even in the case of a power loss on the host. The host-side management layer should not introduce any durability anomaly. Likewise, the state of the SSD should remain consistent. The host-side mapping layer should not introduce any consistency anomaly.

We integrate a library that fulfill these requirements in a Linux kernel library similarly to the device mapper (dm) and memory technology devices (mtd) kernel subsystems. The library is placed in the kernel, to both integrate directly with device drivers and to enable the physical address space as either exposed directly to applications, but also more traditional file-systems as a block device. More specifically, this paper makes three contributions:

- We propose a set of abstractions (LightNVM) for open-channel SSDs.
- We explore the design choices for building a management library in the context of the Linux kernel.
- We describe a prototype implementation and evaluate it with both virtualized and physical open-channel SSD hardware.

The abstractions are operating-system independent, and can easily be migrated to other operating systems. The proposed library is publically available on Github and licensed under GPL-2. It provides a resource for researchers and practitioners that intend to develop innovative solutions based on open-channel SSDs.

3 Background and Related Work

SSDs are built from NAND flash, which uses a FTL to expose it the physical flash a block device interface. The behavior of an SSD are given from its NAND flash and in large part its FTL implementation. As no FTL are good for all workloads, it is usually seen that consumer SSDs often are optimized to reads, and burst writes, while enterprise drives cater to long running workloads. To cater for various workloads, each vendor implement different FTLs depending on a given application workload. As the FTLs are hidden behind a block device interface, this information cannot be communicated, leading to unpredictable performance and missed opportunities for application-specific optimizations.

Figure 1 illustrates how the random write performance of two common SSD models differ significantly. With such devices, it is up to applications to deal with the performance characteristics of the idiosyncrasies of SSD management. With open-channel SSDs, the goal is, on the contrary, to adapt SSD management to the idiosyncrasies of applications. In this section, we review the constraints that flash chips introduce for SSD management, first in the context of traditional FTLs, then in the context of open-channel SSDs.

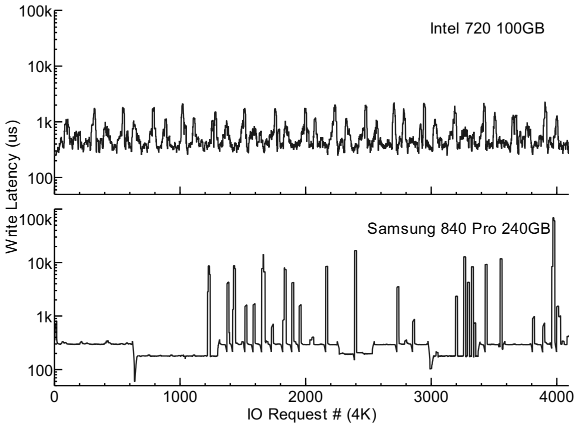

Figure 1: Response time of two SSDs for 16MB Random write using 4KB IOs.

3.1 Flash Constraints

A flash chip is a complex assembly of flash cells, organized by pages (512 to 8192 bytes per page), blocks (64 to 256 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 cell 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 (today 20nm), (ii) in the number of bits per flash cells, (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. For now, this lower performance can be compensated for by increasing parallelism within- and across chips. At some point though, it will be impossible to further reduce the size of a flash cell.

A flash-based SSD contains tens of flash chips wired in parallel to the SSD controller though multiple channels: (C5) Operations on distinct chips can be executed in parallel, so long as there is no contention on the channel. (C6) Operations on a same flash chip are executed serially.

3.2 Embedded FTLs

Traditional SSDs embed a so-called Flash Translation Layer (FTL) on their controller in order to provide a block device interface, hide the aforementioned constraints and leverage parallelism across flash chips and channels. Typically, the FTL implements out-of-place updates to handle C2. Each update leaves, however, an obsolete flash page (that contains the before image). Over time such obsolete flash pages accumulate, and must be reclaimed by a *garbage collector*. A mapping between the logical address space exposed by the FTL and the physical flash space is necessary (a) to support random writes and updates that address C2 and C3 and (b) to support the wear-leveling techniques that distribute the erase counts across flash blocks and mask bad blocks to address C4. How is such a mapping implemented? *Page mapping*, with a mapping entry for each flash page, is impractical with large capacity devices due to the huge size of the map. *Block mapping* drastically reduces the mapping table to one entry per flash block[10]—the challenge then, is to minimize the overhead for finding a flash page within a block. However, block mapping does not support random writes and updates efficiently. More recently, many *Hybrid mapping* techniques [14, **?**, 7] have been proposed that combine block mapping as a baseline and page mapping for update blocks (i.e., the blocks where updates are logged). While each read operation is mapped onto a specific chip, each write operation can be scheduled on an appropriate chip. Besides mapping, existing FTL algorithms also differ on their wear-levelling algorithms [**?**], as well as their management of log blocks, scheduling and garbage collection(block associative [12], fully associative [14], using detected patterns [**?**] or temporal locality [10]).

Both garbage collection and wear-levelling read live pages from a victim block and write those pages, at a location picked by the scheduler, before the block is erased. The garbage collection and wear-levelling operations thus interfere with the IOs submitted by the applications. This is what explains the *noise* we observe in Figure 1.

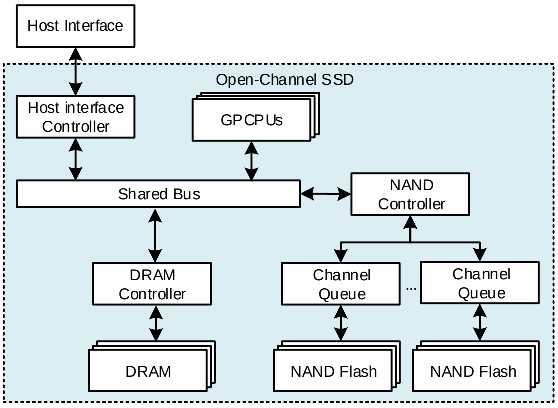

Figure 2: Overview of the logical architecture of an open-channel SSD.

3.3 Open-Channel SSDs

Open-channel SSDs are flash-based SSDs that do not embed a traditional FTL. They directly expose internal flash commands to the host: read and write at the page-level granularity, and erase at block-level granularity.

The logical architecture of an open-channel SSD is depicted in Figure 3. At a minimum it consists of (i) a host interface, such as SATA/SAS [27], NVMe[8], or RapidIO [1], (ii) General purpose CPU(s), (iii) some on-board volatile memory for buffering and processing, and (iv) non-volatile NAND flash for the non-volatile storage.

The directly exposed flash enables the host to direct data placement and control garbage collection at a fine granularity. This also means that the host itself is in charge of managing the flash in an appropriate way. A general FTL have a lot of responsibility. In particular (i) bad blocks management, (ii) accesing flash in regard to flash requirements, (iii) maintain consistency of written data, and (iv) manage metadata. These are all requirements, which the host has to take into account.

In Section 4, we derive the design of LightNVM from these responsibilities. We also explore how open-channel SSDs can embed additional functionalities to enhance performances.

3.4 Products and Prototypes

Companies, such as Fusion-IO and Virident have based their product on a form of open-channel SSD. They employ a host-side driver together with their devices, implementing FTL and garbage collection within the host. This approach enables them to take advantage of the host resources in order to manage their device. The host-managed flash is then both exposed through a generic block interface, but also exposed through optimized interfaces [9, 20, 17]. Fusion-IO exposes these implementation APIs through the OpenNVM initiative, providing an API for applications to take advantage of. The API is a thin layer on top of the actual vendor host driver, thus tightly coupled to the vendor's features.

Organizations that own warehouse scale computers implement their own flash storage [19] to accommodate specific needs [29], co-optimizing their in-house applications and flash-based SSDs. So far, the efforts that have been documented relate to co-design between one application and one SSD.

Open platforms, such as the OpenSSD Jasmine[13], is a hardware platform, that makes it possible to implement custom SSD firmware. This has been used to experiment with the FlashTier and Nameless Writes [23] and implement external sorting using on-the-fly merging [15]. The OpenSSD Jasmine platform is being replaced by OpenSSD Cosmos [26] (Cosmos) that employs better interface and processing power.

To support open-channel development, commercially available, academic and open platforms are all possible to use. Provided they implement the minimum control flow as depicted in Figure 2. I.e., (i) host interface, (ii) general purpose CPU(s), (iii) some onboard volatile memory for fast processing, and (iv) non-volatile NAND flash for the actual storage.

4 LightNVM Design

4.1 Architecture

We propose a set of abstractions (LightNVM) for open-channel SSDs that abstracts a range of open-channel SSDs and makes them available as a common building block for data-intensive applications. Our design is based on the functional (hardware-independent and extensible, durable and consistent) and non-functional (low overhead) requirements listed in Section 1, as well as the constraints introduced by open-channel SSDs, listed in Section 3.3. LightNVM is organized in two layers:

- A core layer that provides common services for open-channel SSDs: address space management, channel management, and block health management. We detail these common abstractions in Section 4.2.
- A layer that provides an open-channel SSD abstraction, that we denote a *target type*, tailored to the needs of a class of applications. We detail the target type abstraction in Section 4.3.

Figure 3 shows LightNVM in the context of the operating system kernel I/O stack. At its core, LightNVM provides block allocation, channel strategies, and

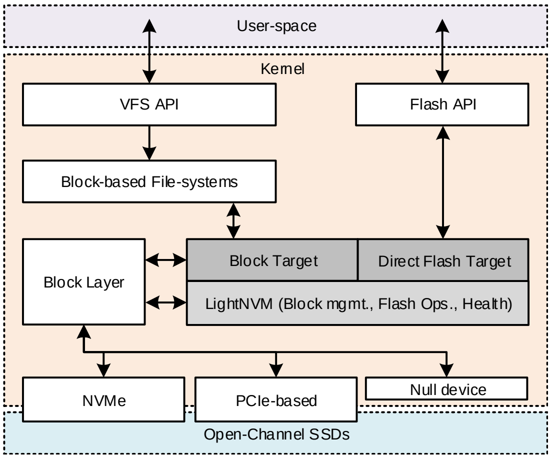

Figure 3: LightNVM Architecture.

flash management (more details in Figure 4). While request flow management is provided by the native Linux block layer. Underneath, the device driver interfaces are hooked into LightNVM. Additional layers contain application-specific SSD management for various targets, including blocks, key-value, or direct-flash. Specialized APIs, or generic interfaces such as the VFS API, expose file-based storage to user-space.

4.2 Common Services

Address space management: LightNVM manages a global pool of physical flash blocks, which is organized as luns. Each lun is a parallel unit withinan SSD. This organization can be leveraged to isolate garbage collection from application workload, or to favor throughput over multiple channels simultaneously, while latency can be decreased if writes are not targeted to chips targeted by reads. Other organizations of the flash block latency (and vice versa). To increase throughput, a buffer writes to multiple pool are possible, e.g., it could be partitioned into groups of blocks sharing similar update frequencies. This is all up to the target to decide how to utilize the underlying flash.

When a target have allocated a flash block, it owns it, and cannot be used by other targets. This allows the library to act as a gatekeeping if QoS is needed. The library can rate-limit the number of blocks that are given to a target, and thus limit the target performance against other targets using same luns. This is very important, as this allows software to provide QoS at a coarser granularity and thus enable million IOPS QoS, without performing inspecting each IO. Thus, QoS decisions are moved outside of the hot data path.

For each active block, the kernel library maintains information about the drive and flash block information. For a drive, it maintains the attached flash latencies, internal device queue depth, pointers to in-use blocks, free blocks and blocks that can be garbage collected. This data is retrieved from the device at boot time, and thus used for bringing up targets with populated data structures. For flash block information, each block has base metadata and target-specific metadata. The metadata maintained by all target types is block state handling, e.g. whether a block is being garbage collected, and pointers reflecting the organization of the global pool of blocks. Each target can maintain its own specialized metadata. A page target might keep information such as the next page to write to, invalidated pages, and when all pages have been written. While a block-based target can omit these and rely on data being written sequentially.

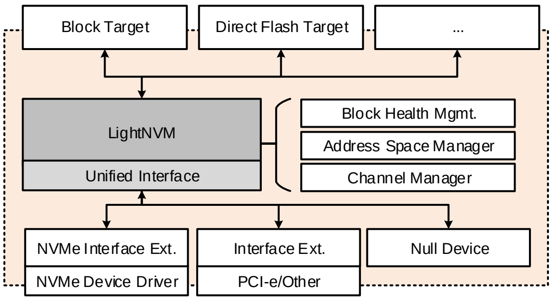

Figure 4: Internal structure of LightNVM.

LUN management: Controlled placement is key to achieving a well-balanced storage system. If all writes and subsequently reads are directed to the same lun, then performance is limited to the performance of the single lun. However, if accesses are distributed across multiple luns, higher throughput is archieved, but higher latency may occur. In the worst case when reads must wait on a full erase.

Depending on the strategy, various performance characteristics are exposed in the SSD. For example, the block target issues write requests in a round-robin fashion across all available luns to utilize the available lun bandwidth. This default placement algorithm can be replaced with more advanced approaches such as CAFTL [**?**], or adaptive scheduling methods that adapt the placement of writes to the targetet luns.

Block health management: As flash blocks erase count increases, the block fails gradually, with single bits being faulty until, at last, no bits can be stored successfully. SSDs typically protect the integrity of each flash block by incorporating an error correction code (ECC) [6] within each block or over multiple blocks [25]. Depending on the scheme, an open-channel SSD vendor might not expose the finest granularity of its flash blocks, but instead expose superblocks, which performs ECC at a higher level in hardware.

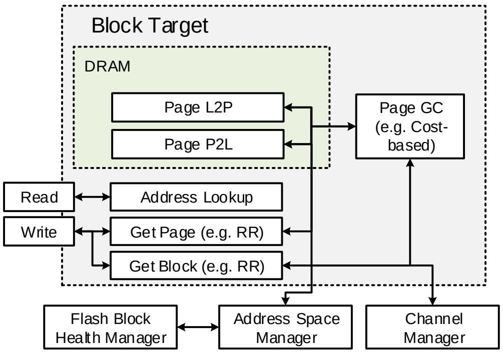

Figure 5: Logical architecture of a block based target.

Generally, ECC should be handled by hardware, and only in very elaborate schemes should the library be involved. However, when a flash block fails, it is up to the library to maintain a parity scheme (at the block level) for retrieving the lost data, or let other layer (e.g., file-systems, such as Ceph or GlusterFS) provide error-recovery.

4.3 Target Types

A target is associated with a group of physical block and exposes a logical address space to the upper layers. The layers could be expose a traditional sector-based interface optimized for subpage writes, key-value accesses, or enabling applications to build their own application targets. Each target thus defines how the physical flash is exposed. For sector-based, over-provisioning ratio and maintaining a logical- to physical (L2P) mapping table are important, while others have other requirements. A target is also responsible for reclaiming blocks through garbage collection. Thus, before a flash block is returned to the library, it should have been erased and ready for new writes.

To evaluate the library, we implemented two target types:

Sector-Based Target Type: The logical address space is a contiguous array of pages, that can be read, written or trimmed. This target provides a fully associative page allocation scheme. This is efficient for random writes, as writes can be buffered sequentially and written in parallel onto flash. The garbage collector picks victim block using a cost-based approach that takes active pages and wear into account [7]. The design can be seen in Figure 5.

Inspiration can be drawn from existing literature while revisiting trade-offs in light of the increased computational resources of operating on the host. For example, Superblock [10] and DFTL [7] represent two possible approaches maintain full associativity at lower memory overhead. A naive approach uses approximately 1GB of metadata per 1TB of flash managed.

Sub flash page allocation poses a significant challenge in terms of durability. As flash page writes are atomic, and flash pages typically are 16-32KB, 4K writes only writes partial of a given flash block. These there have to be buffered either in host or device memory. As the host does not provide power-safety. A method for providing durability is required.

One such approach is used by [7], where only a part of the translation table is kept in memory on the host at any time. We propose a different approach, where the host does not maintain the on-disk translation table durably. Instead we maintain a dual translation map approach, with a global and local translation map.

For each logical address request, a device driver supports a number of physical addresses. In our case, the NVMe implementation supports 64 physical addresses together with a logical address. This enables us to communicate the mapping directly to the device, and thereby the device maintains metadata consistency and durability. On device bring up, the mapping table is loaded from disk and used to retrieve the state of a given target.

To support multiple targets, the device or host will have to implement a form for multiplexing the address space. This either by a device supporting a sparse address space (larger than the physical address space), or by having the host maintain a table of chunks that each are allocated to a given target. We currently only evaluate using a single sector-based target.

Key-value with Flash Blocks Target Type: The logical address space is a key-value store, implementated as a hash table. A block-level allocator, translates logical flash block to physical block by a fully-associative translation map. This mapping is geared toward data requests at the size of a block and/or sequential data patterns. The data must be the size of the physical flash blocks. If smaller data keys are needed, an approach as in NVMKV [**?**] can be used.

The storage requirement for a complete map is only $O(log_2(N) * 2N)$, where N is the number of physical blocks. This can further be improved by grouping blocks together in superblocks if needed. Typically, this allows all blocks to be kept in main-memory and thus supports very fast lookup. Compared to sector-based targets, a garbage collector is not necessary as a key-value delete request is similar to a block erase.

4.4 IO Management

When file-systems submit requests to the underlying storage, they do it in the form of a data structure known as a block io (bio), it contains an address, size and pointer to data. This bio is sent to the block layer [2] that takes

Area	Lines Added	Lines Removed
Library	1222	-
Sector-target	1404	-
Null-block device driver	125	16
SCSI subsystem	128	3
NVMe device driver	1348	746
NVMe QEMU driver	722	27
OpenSSD firmware	860	10
Total	5809	802

Table 1: Lines of code changed to enable LightNVM integration.

specific limitations of the device into account and submits it to the block device. For example, if a bio is too large to fit into a single device request, it is split into several requests and sent to the device driver one at a time.

The library is attached around the block layer. At the top, there is a target, which can span multiple devices, that takes a request and send it to the appropriate device. After a request has been handled by the block layer, the request is passed onto the device driver. In the device driver, the request has its logical address mapped to its physical counter parts.

In the case of NVMe, the read/write command is extended with support to carry physical addresses. With a single address access, the physical address is mapped one to one. While if multiple sectors are accessed, a metadata page is followed with a list of the physical addresses that the logical address maps to. On completion, the completion is similarly extended to communicate if any of the physical addresses had an error. If error, it is up to the library to resubmit requests or return error to the layers above.

4.5 Perspectives

The library is in the process of upstreamed to the Linux kernel. Linux is targeted, as it is the most widely used data center operating system. The version of LightNVM that might be integrated into the Linux kernel will likely differ from the current version on a multitude of details. In particular, there is room for both performance and logic optimizations, as well as the implementation of known concepts in new setting, e.g., deduplication, compression, thin provisioning, integration with virtualization techniques and so forth. However we expect that the core abstractions presented in this paper will remain unchanged.

5 Experimental Framework

We use three storage interfaces to evaluate LightNVM: (i) a SATA interface, through which we access the OpenSSD Jasmine platform [13], (ii) a NVMe target through which we access QEMU with a NVMe null-device, and (iii) a null block device driver to evaluate the host performance.

We enabled the SATA interface to make use of the OpenSSD Jasmine platform. The OpenSSD is a reference SSD implementation, built on top of the Indilinx Barefoot controller. The SSD consists of an ARM7TDMI-S core, SATA 2.0 host controller, and 64MB of SDRAM. It is configured with two active channels with 64GB of NAND flash. Flash timings are 250us read, 1.3ms program, 1.5ms and erases. The page granularity is configured to 2x8KB and block granularity is 2x1MB per physical block.

The OpenSSD firmware was extended with a LightNVM-compatible FTL. It exports the flash erase command to the host and buffers writes to its DRAM store before writing them to flash. See Table 1 for lines of code changed for all the components.

As a point of comparison, the “Greedy” FTL supplied with the OpenSSD is used for comparson. It a page-based FTL, that implements a greedy garbage collector, that only kicks in when no pages are left on the device.

The OpenSSD does not supply any power-safe failure mode, and thus FTL metadata should be written at the same time as the data is written. The OpenSSD platform is from 2011, and today, most SSDs have some form of safe RAM. We therefore ignore the power-safety issue and assume that metadata is safely stored on power failure. A commercial solution would have to implement the necessary fallback modes.

To experiment with faster hardware, we first extended the QEMU NVMe simulator [**?**] with support for LightNVM. The NVMe device driver was changed to use the block layer and use its framework for splitting IOs. With the simulator it was easy to prototype interfaces and extensions.

With the NVMe implementation in place, a NVMe device without an backing store. is used to measure the host overhead together the overhead of driving the device-driver. The NVMe null device has a 35μs latency for 4KB accesses and a maximum bandwidth of 380MB/s.

Finally, we modified the Linux kernel null block device driver to work with LightNVM to test its raw performance. The null block device driver, signals request completion upon its receipt, thus purely stressing the processing overhead of forwarding requests through to a device. In this manner, we achieve throughput not possible present hardware, and bottlenecks otherwise obscured by storage hardware performance are revealed.

All the experiments are performed on a system consisting of an quad-core Intel i7-4770 3.4Ghz CPU, 32GB 1333Mhz DDR3 memory. The software system consists of Ubuntu 14.04, with a LightNVM-patched kernel ver-

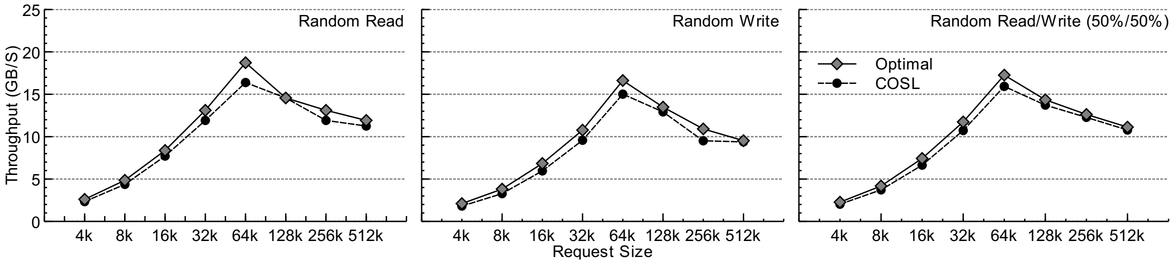

Figure 6: Throughput for 4KB to 512KB request IO accesses at queue depth 32 using the null block device. Left: 100% RR. Middle 100% RW. (3) 50% RR/RW. A slight overhead of LightNVM can be seen on small writes, while it is negligible at high request sizes. The deterioration in throughput for requests larger than 64KB is a consequence of the block layer segmenting them, thus multiple requests are submitted for the larger IO sizes. This can be changed if larger throughput is needed.

sion 3.17-rc3.

6 Experimental Results

We first evaluate our micro-benchmark workloads with the null block device driver and NVMe null-device to show the raw performance of LightNVM and its overhead. We then dig deeper into the overhead of LightNVM from a request perspective and qualify the specific overheads of its implementation. We then compare with the OpenSSD hardware and compares its greedy firmware with our LightNVM firmware. At last we look at the key-value target and evaluate future benefits.

6.1 Microbenchmarks

We use throughput/IOPS, response latency and CPU usage as our main metrics. With respect to evaluating the LightNVM implementation, throughput and latency is used together with CPU overhead, we compare the overhead of LightNVM at the cost of the other two metrics.

6.1.1 Raw Performance

We use the null block driver to represent the optimal throughput attainable on our architecture. Benchmarking with- and without LightNVM yields the overhead incurred by the additional work of translating logical address space requests and maintaining the state of each block on the device. Figure 6 depicts workloads of random read, random write and mixed random reads and writes, respectively. We measure the overhead of address translation compared to not using LightNVM. We call this point “Optimal”, as its the baseline for comparison.

The highest overhead is 16% for random 4KB reads and the lowest 1.2% for 512KB. The high 4KB load is caused by the extra overhead in tracking in-flight IOs, larger mapping overhead and IO request cleanup being executed in interrupt context of the null block driver, thus stalling the IO pipeline. The deterioration in throughput for requests of 64KB and above is a consequence of the block layer segmenting large IOs, thus performing multiple IOs for each large IO. This is by design, though this behavior can be overridden if necessary.

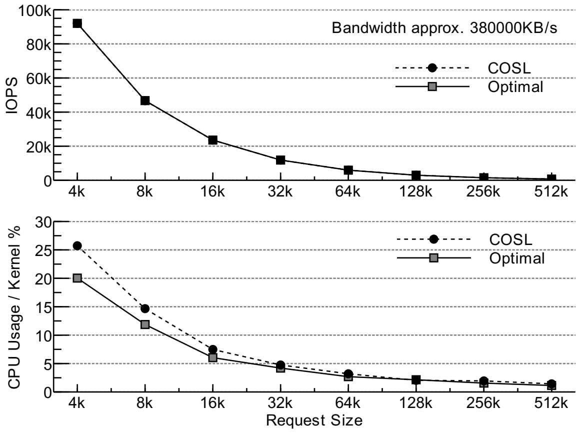

Figure 7: IOPS and CPU kernel usage of 4KB to 512KB random write IO sizes with the null NVMe hardware device. The maximum bandwidth is 380.000KB/s and both 4KB and 512KB utilizes the full bandwidth. The overhead of LightNVM can be seen in CPU usage, with a 6% increase in CPU usage for 4KB to 0.3% increase for 512KB IOs.

This suggests that the overhead is negligible and LightNVM can be used on top of open-channel SSDs. The 4KB accesses are still high, and it could be a viable work to improve the CPU utilization by introducing per-CPU data structures to reduce lock contention, and/or moving IO request cleanup outside the fast interrupt path.

To compare the overhead on real hardware, we benchmark the NVMe null device. We show the measurements in Figure 7, which considers the CPU overhead of LightNVM, while attaining a constant write speed of 380MB/s. Looking at 4KB IO request sizes, the normal overhead is 20% CPU time when managing nearly 100k IOPS. LightNVM increases the CPU load by 6% overhead, declining to 1.3% for 512KB IO requests. For devices, where we also have to pay the price for overhead in regard to DMA buffer management, data transfer, etc.

Because LightNVM increased CPU utilization by 30% in the case of 4KB IO request size, we looked into where the overhead originated. We found that the bottleneck was from IOs being stalled by the heavy in-flight tracking in the IO clean path. Similarly to the null block driver, this could be mitigated by using per-CPU data structures for in-flight tracking. However, in case of the NVMe device, an additional optimization would be to increase the number of hardware queues available, as the device only exposed a single hardware queue and we therefore cannot take advantage of multi-queue opportunities.

6.1.2 Timing Analysis

Table 2 shows an analysis of where time is spent within the block layer, LightNVM, and the three devices for a single request. We focus on the timings of the block layer and LightNVM specifically, as the block layer is known for having the largest overhead, measured to 2-3μs to pass the block layer [4, **?**]. The new block layer [2] has significantly decreased this to only 0.333μs on our system. This makes LightNVM stand out more, as it takes up a modest 0.133μs for reads, and 0.308μs for writes. LightNVM is thus a significant overhead compared to the block layer. The overhead is contributed by locks around in-flight tracking and allocation of pages and blocks by introducing contention. However, the overhead is primarily attributed to post-completion processing of IOs in interrupt context, effectively monopolizing the hardware for the duration of that task.

6.1.3 OpenSSD Comparison

Figure 8 and Table 3 show the performance of the OpenSSD greedy firmware compared with the OpenSSD LightNVM firmware together with the LightNVM host layer. All data is based on 4KB accesses where the disks have been cleanly formatted before use and no garbage collection was run during the benchmarks.

We note that writes for Greedy is significantly slower than LightNVM. This is attributed to the need for Greedy to implement read-modify-write for writes less than 16KB, where the LightNVM firmware instead buffers the incoming writes, efficiently transforming random writes to sequential writes. This is efficient for write throughput, but affects latency, as there are less opportunities for reads.

For LightNVM random writes, we observe three states: (1) a high timing, (2) a more stable writes and (3) finally stable writes with the expected write latency. The first two are a consequence of the aged flash that has been erased intensively during development and therefore exhibits larger program timings. Writing to less used blocks yields a significant drop in latency. This suggests that further tracking of blocks can be beneficial to use for data placement decisions within the host. Targets that wish to take advantage of this can measure write timings and use it to orchestrate its accesses.

The read performance for both approaches are similar. LightNVM is slightly faster at random reads and twice as fast in sequential reads. For random reads, Greedy translates the logical address to the physical address by accessing its DRAM. This allows LightNVM to perform reads faster, as these already have been mapped by the host system, which is significantly faster. Writes must still perform additional work if an on-disk translation map is managed next to the host translation tables. For sequential reads, LightNVM takes advantage of the parallelism in the OpenSSD, while Greedy keeps on reading from the same channel. The trade-offs between the two firmwares show the flexibility in tailoring channel access patterns to its workload, and the resulting difference in behavior.

6.2 Key-Value Store Target

As part of our evaluation of the LightNVM platform, we created an implementation of a key-value-store (KV) utilizing LightNVM to manage the storage. For lookups, it translates a given key into the corresponding physical block. Inserts are handled by allocating a new block from the address space manager, associate the key to it and write the data to the physical block. The translation map is managed by a hash-table, that implements a bounded array for each of the buckets. The KV store is accessed through an IOCTL interface, and thus we created an API user-space applications to issue requests.

In Table 4 we compare sequential read throughput between the Optimal configuration using page-mapping, LightNVM's block device interface utilizing page-based mapping and the key value-store implementation in which each value is being assigned a unique physical block. We find the latency in processing a 1MB sequential write reduced by 14.29%, from 33.02μs to 22.20μs with a corresponding 55.8% improvement in throughput, from 28.1GB/s to 44.7GB/s all while managing a 17.2% reduction in CPU-usage in kernel-space.

The higher performance of LightNVM KV interface is due to the lower overhead of traversing the IO stack

Component	Description	Latency(us)	
		Read	Write
Block Layer	Map application IO to request structure	0.061	
	Insert request into request queue	0.125	
	Prioritization, merging, queuing to device driver	0.086	
	Cleanup and completion of IO to process	0.061	
LightNVM	Address translation / request mapping	0.045	0.211
	Cleanup and completion	0.093	0.097
Time to complete request by HW	Null block device	0.060	
	Null NVMe hardware device	35	
	OpenSSD with LightNVM firmware device	350	

Table 2: Time spent at completing a 4K IO request within the block layer, LightNVM, NVMe and OpenSSD hardware.

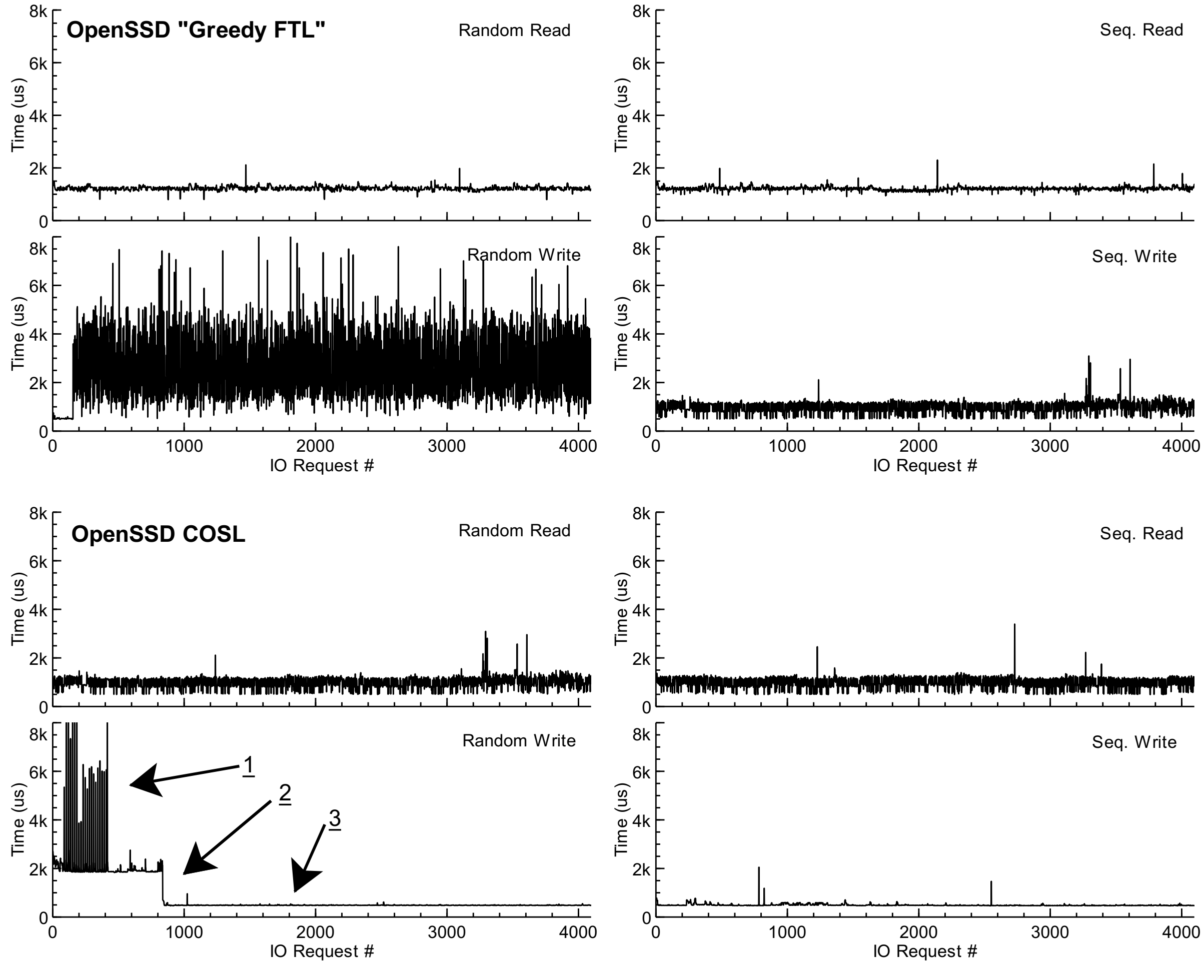

Figure 8: We compare the OpenSSD "Greedy FTL" and the OpenSSD LightNVM firmwares. 4096 4KB requests are issued with respect to random read/write and sequential read/write and its response time is measured in microseconds.

Benchmark	IOPS	Average (μ)	Std.Dev
OpenSSD Greedy			
Random Read	1630	1224	114
Sequential Read	1635	1220	87
Random Write	721	2768	1415
Sequential Write	522	3823	1448
OpenSSD LightNVM			
Random Read	2082	958	222
Sequential Read	4030	494	48
Random Write	4058	490	30
Sequential Write	2032	981	237

Table 3: We compare performance differences observed between the OpenSSD Greedy firmware and the OpenSSD LightNVM firmware and LightNVM host layer for 128MB 4KB writes.

Metric	Optimal	LightNVM-Page	Key-value
Throughput	29GB/s	28.1GB/s	44.7GB/s
Latency	32.04	33.02μs	21.20μs
Kernel Time %	66.03%	67.20%	50.01%

Table 4: Table summarizing the performance differences observed between the page-based block device interface, LightNVM page and key-value target issuing 1MB writes.

for common page IOs. By issuing IO requests through IOCTL commands, we bypass a large part of the legacy IO stack. However, for high-performance applications, where multiple requests are outstanding, a thread must be spawned for each request to account for the synchronicity of IOCTL requests. This uses extra resources and increases latency. This can be mitigated by taking advantage of the existing libaio API, and use it to implement an asynchronous KV interface, allowing the kernel to handle submission and completion of IOCTL within its existing framework.

While the scenario outlined above is tailored to the characteristics of our virtual disk, 1MB values corresponding perfectly to blocks of 128 8KB pages, it clearly exemplifies the potential benefits of exerting direct control over storage device allocation and data management. Mapping overhead is significantly reduced with the overhead of garbage collection reduced correspondingly.

7 Conclusions and Future Work

We proposed a new library that supports a tight integration of open-channel SSDs with applications in Linux. Our design is modular and extensible. Our implementation exhibits low overhead, and good performance even with LightNVM default policies in terms of data placement or garbage collection. Future work includes (i) integrating LightNVM with other types of open-channel SSDs, such as Cosmos, or from vendors that make their products available, (ii) experimenting with LightNVM in the context of various applications, including a key-value store, a distributed object storage and a database system, (iii) experimenting with advanced SSD management policies. Finally, we will push for an adoption of LightNVM in the Linux kernel and thus address the performance optimization opportunities revealed by our evaluation.

References

[1] ASSOCIATION, R. T. RapidIO Interconnect Specification 3.0, 2013.

[2] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block IO: Introducing multi-queue SSD access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference* (New York, NY, USA, 2013), SYSTOR '13, ACM, pp. 22:1–22:10.

[3] BORTHAKUR, D., AND XU, H. The story of rocksdb.

[4] CAULFIELD, A. M., DE, A., COBURN, J., MOLLOW, T. I., GUPTA, R. K., AND SWANSON, S. Moneta: A High-Performance Storage Array Architecture for Next-Generation, Non-volatile Memories. *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture* (Dec. 2010), 385–395.

[5] CAULFIELD, A. M., MOLLOV, T. I., EISNER, L. A., DE, A., COBURN, J., AND SWANSON, S. Providing safe, user space access to fast, solid state disks. In *ASPLOS* (2012).

[6] DONG, G., XIE, N., AND ZHANG, T. On the use of soft-decision error-correction codes in NAND flash memory. *IEEE Transactions on Circuits and Systems 58*, 2 (2011), 429–439.

[7] GUPTA, A., KIM, Y., AND URGAONKAR, B. DFTL: A Flash Translation Layer Employing Demand-based Selective Caching of Page-level Address Mappings. *ACM SIGPLAN Notices 44*, 3 (Feb. 2009), 229.

[8] HUFFMAN, A. NVM Express Specification 1.1. www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1.pdf, 2012.

[9] JOSEPHSON, W. K., BONGO, L. A., LI, K., AND FLYNN, D. DFS: A File System for Virtualized

Flash Storage. *ACM Transactions on Storage 6*, 3 (Sept. 2010), 1–25.

[10] JUNG, D., KANG, J.-U., JO, H., KIM, J.-S., AND LEE, J. Superblock FTL: A Superblock-Based Flash Translation Layer with a Hybrid Address Translation Scheme. *ACM Transactions on Embedded Computing Systems 9*, 4 (Mar. 2010), 1–41.

[11] KANG, J.-U., HYUN, J., MAENG, H., AND CHO, S. The multi-streamed solid-state drive. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)* (Philadelphia, PA, June 2014), USENIX Association.

[12] KIM, J., KIM, J. M., NOH, S. H., MIN, S. L., AND CHO, Y. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics 48*, 2 (May 2002), 366–375.

[13] LEE, S.-W., AND KIM, J.-S. Understanding SSDs with the OpenSSD Platform. *Flash Memory Summit* (2011).

[14] LEE, S.-W., PARK, D.-J., CHUNG, T.-S., LEE, D.-H., PARK, S., AND SONG, H.-J. FAST: A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems 6*, 3 (July 2007).

[15] LEE, Y., QUERO, L., LEE, Y., KIM, J., AND MAENG, S. Accelerating External Sorting via On-the-fly Data Merge in Active SSDs. *HotStorage'14 Proceedings of the 5th USENIX conference on Hot topics in storage and file systems* (2014).

[16] LEE, Y.-S., KIM, S.-H., KIM, J.-S., LEE, J., PARK, C., AND MAENG, S. OSSD: A case for object-based solid state drives. *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)* (May 2013), 1–13.

[17] MÁRMOL, L., SUNDARARAMAN, S., TALAGALA, N., RANGASWAMI, R., DEVENDRAPPA, S., RAMSUNDAR, B., AND GANESAN, S. NVMKV: A Scalable and Lightweight Flash Aware Key-Value Store. *6th USENIX Workshop on Hot Topics in Storage and File Systems* (2014).

[18] O'NEIL, P., CHENG, E., GAWLICK, D., AND O'NEIL, E. The log-structured merge-tree (lsm-tree). *Acta Informatica 33*, 4 (1996), 351–385.

[19] OUYANG, J., LIN, S., JIANG, S., AND HOU, Z. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems* (2014).

[20] OUYANG, X., NELLANS, D., WIPFEL, R., FLYNN, D., AND PANDA, D. Beyond block I/O: Rethinking traditional storage primitives. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* (2011), IEEE, pp. 301–311.

[21] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 1–16.

[22] SALTZER, J. H., AND KAASHOEK, M. F. *Principles of Computer System Design: An Introduction.* Morgan Kaufmann, 2009.

[23] SAXENA, M., ZHANG, Y., SWIFT, M. M., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Getting Real: Lessons in Transitioning Research Simulations into Hardware Systems. *FAST* (2013), 215–228.

[24] SCHLOSSER, S. W., AND GANGER, G. R. Mems-based storage devices and standard disk interfaces: A square peg in a round hole? In *In Proceedings of the USENIX Conferenceon File and Storage Technologies (FAST)* (2004), pp. 87–100.

[25] SHADLEY, S. NAND Flash Media Management Through RAIN. Tech. rep., Micron2011, 2011.

[26] SONG, Y. H., JUNG, S., LEE, S.-W., AND KIM, J.-S. Cosmos OpenSSD : A PCIe-based Open Source SSD Platform OpenSSD Introduction. *Flash Memory Summit* (2014), 1–30.

[27] STEVENS, C. E., AND COLGROVE, D. Technical Comittee TI13 - ATA / ATAPI Command Set - 2. ATA / ATAPI Command Set - 2, 2010.

[28] VOLOS, H., NALLI, S., PANNEERSELVAM, S., VARADARAJAN, V., SAXENA, P., AND SWIFT, M. M. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems* (New York, NY, USA, 2014), EuroSys '14, ACM, pp. 14:1–14:14.

[29] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of LSM-tree based key-value store on open-channel SSD. *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14* (2014), 1–14.

Chapter 7

AppNVM: A software-defined, application-driven SSD

The open-channel SSDs presented in the previous paper constitute a first step towards application-SSD co-design. With open-channel SSDs, the fundamental role of the operating system remains unchanged compared to legacy approaches based on hard drives: It mediates every single IO. As SSD performance keeps on improving, the overhead of traversing layers of operating system code for each IO will become a significant overhead. In this paper, we propose a vision for reorganizing the way applications, operating systems and SSDs communicate.

Large-scale data centers, such as those operated by Google, Facebook, Amazon, and similar require consistent low latency from their network and storage [1, 8]. A user request end-to-end should be served consistently in the low hundred miliseconds. As SSDs approach million IOPS, overhead of host fairness, I/O scheduler, and control of I/Os is significant.

Applications are specially sensitive to I/O outliers [2]. Outliers occur when internal garbage collection or similar is scheduled when user data is scheduled simultaneously. SSDs mitigate outliers by for example limiting incoming requests and allow any background work to be carried out without affecting user I/O performance or data is duplicated multiple locations, where the same request is sent all of them. The first request to serve the content is used. Both examples requires either duplication of data or must deal with subtle scheduling issues.

The problem is furthermore exacerbated in multi-tenancy systems [7, 4]. Fairness is typically achieved by introducing flow barriers. They are controlled by continuously monitor resource utilization and insering flow barries. For tenant I/O, the flow barrier acts as a synchronous point. Similar to blk-mq in Chapter 3, this is incapable to scale to million of IOPS. The question therefore is; How can multi-tenancy systems scale its I/O systems to million of IOPS?

Inspiration from software-defined networking [5] and the recent Arrakis [6] work provide a novel approach. The SSD I/O path is split into a data and control plane. The data plane services user I/Os to the device. Directly outside of the control plane, which in our case the operating system kernel. The control plane controls security and isolation between processes. When the data path is setup between applications and the storage device. It is up to the device to provide the guarantees that the kernel provided. Additionally, to support multi-tenancy systems, the device also provides fairness. The security and isolation are provided by memory mapping queues into the process address space. Fairness is provided by hardware between these queues.

This approach is described in the vision paper "AppNVM: A software-defined application-driven SSD" which was presented at the Non-Volatile Memory Workshop (NVMW) 2015.

AppNVM consists of three parts, that forms a solution where applications bypass kernel for data I/O and fairness is provided by hardware, without kernel being the arbitrator:

- User-space hardware queues; Hardware provides queue and buffer management for the I/O data path. This can either be though virtualized IOMMUs (VT-d) or solutions such as Intel DPDK [3].
- Shared address space; Multiple processes shares the limited storage media. Orchestration is required, either by manually dividing the storage address space into partitions or using a chunk allocator. For example the flash block allocater provided by LightNVM or a traditional file-system block allocator.
- Rule-based interface to define queue characteristics; For each application queue, specialized configuration is possible. The specialization is described through rules in the form quality of service and

behaviors. Rules are expressed similar to the OpenFlow rule language, and sets up rules to characterize flow. When the flow has been configure, the kernel gets out of the data I/O path.

The first two points are engineering challenges. We will not go further into it. The last is important for adoption. A rule-based interface, such as found in OpenFlow [5], provides control based on flows. This is efficient for networking, as data is captured as flows with various rules applied to it. This can similarly be applied to storage.

The rule-based interface is built on top of the interface of the SSD using the LightNVM specification to control internal operations. A rule consists of a condition, state change, action and a priority.

The condition is when the rule is applied. The state change describes any metadata changes within the SSD. Such as mapping table updates or type of garbage collection. The action describes placement of the data onto individual flash dies. This allows data flows to be partitioned internally onto internal flash dies. Applications and kernel can express strict rules for quality of service and the device can provide the required backing storage. Finally, the priority field is to convey any priority between competing rules.

Future work includes building a device prototype that separates data and control plane and implements the logic in the kernel to provide setup/teardown of process I/O queues. Additionally, a full rule-based language must be defined to enable users to express their quality of service requirements.

The long-term goal is to provide an interface for programmable SSDs. Such SSDs allow applications to describe their workload and implements rules which in turn implements a specialized FTL for their workload. Thus, an SSD becomes multi-faceted and optimized for each type of application, be it read-heavy, write-heavy or a mix of both.

Literature

[1] Mohammad Alizadeh, Abdul Kabbani, and Tom Edsall. Less is more: trading a little bandwidth for ultra-low latency in the data center. *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, 2012.

[2] Luc Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding flash IO patterns. In *4th Biennial Conference on Innovative Data Systems Research (CIDR)*, 2009.

[3] Intel Corporation. .Intel Data Plane Development Kit (Intel DPDK) Programmer's Guide. Technical report, 2013.

[4] Rishi Kapoor, George Porter, and Malveeka Tewari. Chronos: predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, 2012.

[5] N McKeown and Tom Anderson. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, 38(2), 2008.

[6] Simon Peter, Jialin Li, Irene Zhang, Dan R K Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, Timothy Roscoe, and E T H Zürich. Arrakis : The Operating System is the Control Plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014.

[7] David Shue, MJ Freedman, and Anees Shaikh. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, 2012.

[8] David Zats, T Das, and P Mohan. DeTail: reducing the flow completion time tail in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 42(12):139—-150, 2012.

AppNVM: A software-defined, application-driven SSD

Matias Bjørling† Michael Wei∗ Jesper Madsen† Javier González† Steven Swanson∗ Philippe Bonnet†
†*IT University of Copenhagen* ∗*University of California, San Diego*

Abstract

We present the design of AppNVM, a software-defined, application-driven solid state drive (SSD) inspired by software-defined networking. AppNVM exposes an application-defined interface without sacraficing performance by seperating the data plane from the control plane. Applications control AppNVM SSDs by installing rules, which define how application requests are handled. A controller then transforms those rules and installs them onto the device, enforcing permissions and global policies such as wear-leveling and garbage collection. The setup enables the application to send requests to the device which are handled in an application specific manner. By seperating the data plane from the control plane, AppNVM easily scales to high-performance million-IOP devices and beyond.

1. Introduction

Current state-of-the-art solid state drives (SSDs) can perform millions of I/O operations per second, and we can only expect future SSDs to scale as the industry continues to make advancements in integration and memory technologies. At the same time, SSDs need to make decisions to map this torrent of I/O requests to the underlying flash. Due to the idiosyncrasies of flash, SSDs must also perform wear-leveling and garbage collection, which only complicates data placement and leads to unpredictable performance. This is at at odds with application developers which want stronger predictability and more control of how the SSD decided to place data.

In the networking community, a similar phenomenon has taken place. As switches evolved from gigabit to 10-gigabit, it quickly became clear that giving system designers the flexibility to route packets is at odds with delivering low latency. Network designers needed a way to control how data flows without being in the path of data itself. As a result, software-defined networking was born. Today, OpenFlow[3] has emerged as the *de facto* standard. Unlike a traditional switch, which performs packet forwarding and routing decisions on the same device, OpenFlow separates the data plane from the control plane by moving routing decisions to a higher level device, called the *controller*. OpenFlow uses an abstraction called a *flow* to bridge the data plane and control plane. Every OpenFlow switch contains a flow table, which controls how packets are routed. OpenFlow controllers install entries into the table, and can inspect the state of flows through counters and messages from the switch.

Given the success of OpenFlow in the networking community, we felt that it would be prudent to analyze how OpenFlow concepts could be used to solve data-placement problems in SSDs. We see the traditional Flash Translation Layer (FTL) as analogous to the traditional switch: a black box which performs both data placement and the placement decisions on the same device. The black box nature of the FTL makes it difficult for the FTL to meet application requirements, which may require flexibility in where data is placed. Furthermore, an FTL cannot reason efficiently about the state beyond the device (for instance, across SSDs which may be potentially in different hosts). Our goal is to move the placement decision logic into a controller, while maintaining the SSDs ability to do fast and efficient data placement.

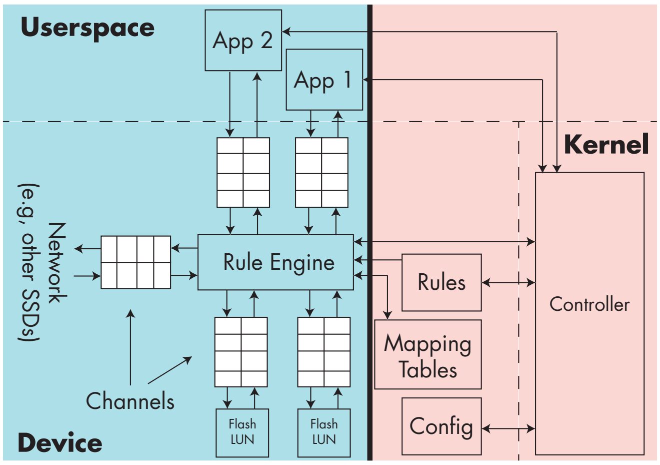

Figure 1: The AppNVM Architecture. Components highlighted in blue on the left belong to the data plane, whereas components highlighted in pink on the right belong in the control plane.

In this paper, we analyze and describe how an OpenFlow-inspired system can be used to control data placement in SSDs. We explore in particular the challenges of dealing with storage: for example, how to deal with garbage collection, wear-leveling and temperature managment.

2. Architecture

Our system design is guided by the principle of seperating the data plane and control plane [4]. Figure 1 illustrates our system architecture. Central to our design are *channels*, which consist of a pair of submission and completion queues. Applications communicate directly to the storage device by sending requests through a channel mapped into user space, bypassing the kernel. Flash channels, or LUNs, are also connected to channels, and there may be several "virtual" channels used by internal processes. In addition, channels may map to a network of remote SSDs, enabling AppNVM to communicate with a storage fabric.

To move data between channels, the SSD applies a set of *rules* to incoming traffic on each queue. Rules are installed

Condition	State Change	Action	Priority
channel.appid == App_0 and header.write	MAP APP_0, header.LBA = NEXT_ADDR	WRITE header.addr = NEXT_ADDR; INSERT TO_LUN	0
header.write	PERFORM GC GREEDY ALL APP	WRITE header.block = VICTIM_BLOCK; INSERT VALID TO ANY_LUN	0
channel.appid == App_0 and header.read		WRITE header.addr = MAP App_0, header.LBA; INSERT ANY_LUN	1

Table 1: A simple set of rules which implement a minimally functional SSD.

to the SSD by a controller, and each rule consists of a set of conditions, state changes, actions and a priority. *Conditions* control whether or not a rule is evaluated for a particular input. For example, a condition may check whether the input is a read, or the value of a certain header. *State changes* affect the internal state of the device. For example, a state change may update a mapping table or internal counter. *Actions* define what the SSD should do with a given input. For example, the device could drop a request, rewrite some headers or place the input on another channel. Finally, the *priority* of the rule determine which rule is evaluated first, in the event that multiple conditions match a given input.

If a rule does not exist for some input, the device sends the input to the controller and requests a new rule. This enables the controller to dynamically learn about traffic and install rules based on incoming requests.

In our current design, the controller is provided as a kernel module. The kernel module makes use of the LightNVM framework [1] to retrieve device configuration and implements a rule-based target. Userspace applications request channels from the kernel using a library. During channel setup, applications can request a set of rules to be installed, or choose from a number of template rules provided by the kernel. These rules may tradeoff parameters such as performance, predictability and media life depending on application requirements. The kernel then inspects those rules and applies any transformations which are necessary to enforce permissions and applies any global policies such as wear leveling and garbage collection. Subsequent I/O requests from the application are then sent directly to the device via the allocated channel without the intervention of the kernel.

2.1. Comparison to Other Architectures

Figure 2 compares AppNVM to other storage architectures. Traditional SSDs present a block-based interface where the data placement is controlled exclusively by the device. Various vendors have produced SSDs which move data placement decisions to the storage fabric or the host.

Willow [5] allows applications to drive an SSD by installing custom software on small processors running within the SSD. However, unlike AppNVM, Willow requires applications to be aware of the internals of the SSD (i.e, programming DMA engines). AppNVM rules, on the other hand, are intended to be device agnostic.

The NVMe standard [2] allows devices to directly expose namespaces to application using user-space submission and completion queues. However, the IO requests are at most controlled using the priority bits and thus decisions are taken on every request, while AppNVM have separated the responsibility.

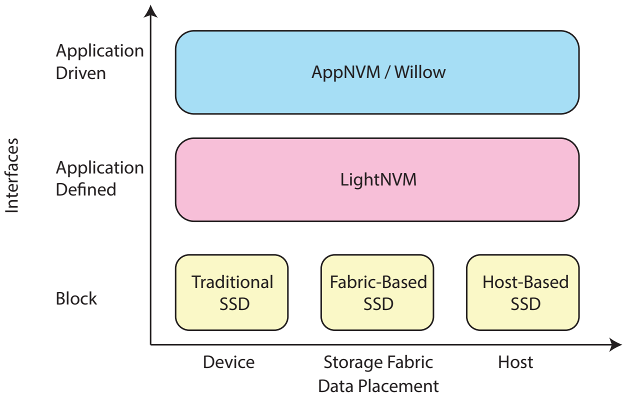

Figure 2: AppNVM compared to other architectures

3. Challenges

While rules provide a flexible interface for applications to control SSD behavior, there are a number of challenges which any data-placement engine must solve. Due to space constraints, we briefly address each challenge below:

Wear-Leveling and Temperature Management. AppNVM rules can be used to track the temperature of data, and rules can also be used to move cold data to provide wear-leveling. These policies could be enforced at the application level or the global level.

Garbage Collection. Garbage collection can be provided by idle rules, and rule priority and the conditions of the garbage collector rules can be used to control when garbage collection takes place.

Allocation. Depending on application requirements, applications can be thin-provisioned, enabling multiplexing, or thick-provisioned, which would provide better performance guarantees.

Quality of Service. While AppNVM does not directly provide quality of service (i.e., an application cannot simply write a rule for 5μs write latency), AppNVM provides a mechanism for a future framework to provide QoS.

Device / Industry on-boarding. AppNVM represents a radical change in SSD design, and AppNVM devices must present channels to the application and be capable of processing rules. Since these devices do not yet exist, we propose a software-based rule engine, similiar to `ovswitch` for OpenFlow, which enables

AppNVM to run on SSDs which support data placement (e.g., Open-Channel SSDs).

References

[1] BJØRLING, M., MADSEN, J., BONNET, P., ZUCK, A., BANDIC, Z., AND WANG, Q. Lightnvm: Lightning fast evaluation platform for non-volatile memories.

[2] HUFFMAN, A. NVM Express Specification 1.2. www.nvmexpress.org/wp-content/uploads/NVM-Express-1_1.pdf, 2014.

[3] MCKEOWN, N., ANDERSON, T., BALAKRISHNAN, H., PARULKAR, G., PETERSON, L., REXFORD, J., SHENKER, S., AND TURNER, J. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review 38*, 2 (2008), 69–74.

[4] PETER, S., LI, J., ZHANG, I., PORTS, D. R. K., WOOS, D., KRISHNAMURTHY, A., ANDERSON, T., AND ROSCOE, T. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, Oct. 2014), USENIX Association, pp. 1–16.

[5] SESHADRI, S., GAHAGAN, M., BHASKARAN, S., BUNKER, T., DE, A., JIN, Y., LIU, Y., AND SWANSON, S. Willow: A user–programmable ssd. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation* (2014), USENIX Association, pp. 67–80.

Chapter 8

Conclusion

The advent of SSDs has already transformed the storage industry. The advent of high-performance SSDs is promising to revolutionize the software industry too. Indeed, secondary storage is no longer a bottleneck; software has to be re-designed to remove bottlenecks and to streamline the data path. The key challenge for data-intensive applications and operating system designers is now to keep up with storage hardware performance and to design software that scales.

In this thesis, my goal has been to tackle this challenge with a focus on operating system support for high-performance SSDs. I have been driven by the opportunity to contribute to the evolution of the Linux kernel IO stack. More specifically, I organized my thesis around two research questions:

1. Is the block device interface still relevant as secondary storage abstraction? I have explored whether and how to transform secondary storage from a memory to a communication abstraction.

2. What are the bottlenecks in the operating systems I/O stack? Beyond the engineering effort required to explore this question, my contribution defines avenues of new research concerning the nature of the communication between applications and secondary storage.

The contributions from this thesis are organized around 6 publications that address various aspects related to these two questions.

First, I analyzed the drawbacks of the block device interface in the context of high-performance SSDs. In the absence of stable performance contract, this narrow interface forces software components on both sides of the interface (i.e., applications and operating system on the host, and FTL on the SSD) to take arbitrary decisions about I/O scheduling and data placement, resulting in sub-optimal decisions, redundancies and missed optimizations. This contributes to unpredictable latency variations, which are increasingly harmful as SSD performance improves.

Exploring novel I/O interfaces requires the ability to evaluate design choices both on the host and on SSDs. My goal was to explore the design space for supporting applications running at hundred of thousands of IOPS. Midway through my thesis, the only form of programmable SSD was the OpenSSD Jasmin SSD. Because of the slow interface speed (300MB/s) and the limited on board processing capabilities, we could only achieve throughput in the tens of thousands IOPS with that device. To evaluate high-performance SSD with novel interfaces, I therefore had to simulate the SSD logic in the host.

This led me to the design of LightNVM, where SSD management was moved to the host in order to simulate high-performance SSDs that could be plugged into a live system and accessed from actual operating system and applications. The initial motivation was to bring SSD simulation from approximate architectural simulators running in simulated time to full-system simulator running in real-time.

The engineering work required to move SSD management to the host opened up a new opportunity: the integration of host-based FTL into the Linux kernel. Even more interesting, LightNVM makes it possible to (a) explore a range of design options for the communication interface between host and SSD, and (b) explore how to specialize SSD management for specific applications. These two aspects constitute a radical transformation of how applications access secondary storage from a model where applications and SSDs are designed independent based on distinct and possibly conflicting goals, to a model where applications and SSDs are co-designed.

My work on open-channel SSDs introduces host-managed SSDs and explores a portion of the co-design space for applications and SSDs. The contribution is a building block for direct storage integrations into applications. By moving parts of the SSD management logic into host-space, software layers that implement duplicated logic can be collapsed into a single layer. For example, today, database systems implements log structured merge tree on top of an extent-based file-system on top of a log-structured FTL. There are obvious redundancies across three layers that manage data placement in a similar way. An Open-Channel SSD enables these layers to be combined into a single layer and exposed as a novel interface, thus removing redundancies and increasing performance.

But if we take this co-design logic even further, the very role of the operating system becomes an issue. What is the role of the operating system when co-designing applications and SSDs? Is it a just overhead? I studied these questions first by focusing on bottlenecks in the Linux kernel and then by laying down new visions.

The work on the multi-queue block layer has shown that the legacy block layer, designed for hard disk drives and single core hosts could be considerably optimized. The inclusion of blk-mq into the Linux kernel has enabled a lot of new work on improving the performance of SSDs. After the paper was published, optimizations have continuously been conducted: e.g., moving a large portion of the device driver stacks to blk-mq, micro-optimizing drivers by removing unneeded locks, preallocating hot-code patch memory allocations, etc. Chasing bottleneck is a continuous process as each fix exposes a new bottleneck somewhere in the IO stack.

Beyond bottleneck chasing, the continuous improvement of SSD performance forces system designers to reconsider basic assumptions about IO processing and software design. In this thesis, I focused on two issues and proposed initial solutions:

1. Should IO block waiting for completion? Until now, it was obvious that any thread issuing an IO should yield the CPU core to make room for useful work. But when SSDs complete an IO in a few microseconds, is blocking still a good option? I conjectured, together with Michael Wei, that speculation was a viable option, which could prove superior to blocking or spinning for next generation SSDs.

2. How to ensure fairness between cores submitting IOs to the same SSD? Until now, fairness was managed by I/O schedulers, such as CFQ, that is now insufficient for high-performance SSDs because of their large synchronization overhead. I presented AppNVM as

a novel way enable SSDs to ensure fairness and even provide quality of service (QoS) guarantees for applications co-designed with a new generation of high-performance SSDs.

The vision is divided into how the host becomes aware of application processes and the other how to communicate the fairness (or QoS) constraints the device. For the first, the storage communication path is split into a data and control path. The data path is provided by exposing the storage address space directly into the application address space. That allows applications to completely bypass the kernel for I/Os. To provide security and isolation for the application, the kernel is used as control path. It sets up channels for each application in the device. This allows the device to be aware of applications and provide fairness across channels.

The second part comes in the form of how to communicate fairness in storage. This is achieved by providing a rule-based interface, which describes the flow of data and processes. The device can then schedule I/O appropriately, without the host uses CPU time to provide fairness.

8.1 Open Issues

The story of this thesis has largely been about creating building blocks for others to build on. We see this with all of the papers. They take a step toward where I saw opportunities, and then allow others to take it further.

The blk-mq paper implemented a new way for IO devices to receive IO requests from applications. There is still significant work needed in the kernel to revamp device drivers to take advantage of this new architecture.

The SSD simulator still requires actual FTL implementations to be useful. Together with the Open-Channel SSD contribution, this enables a coherent implementation of device logic and behavior. The simulator can then be used to evaluate new FTLs on SSDs and allow cross-layer optimizations to be applied, before they are put into hardware.

At last, the work on AppNVM is still a vision. The framework for enabling such a storage system will be a multi-year undertaking, with both industry vendors, academia and user-space developers coming together to build novel ways to co-design applications and SSDs.

8.2 Perspectives for Industry

The storage industry is changing. High-performance SSDs and persistent memories are bound to require a redesign of many of the core pieces of technology that we use today.

The multi-queue block layer has been successful since its adoption into the 3.13 kernel. Device drivers are continuously being ported to the new block layer and upper layers such as device mapper, memory technology devices, etc. are redesigned to take advantage of it. Future work is to continue building convert drivers to this new block layer, provide additional optimizations and removing deprecated code. When all architectures have been redesigned the old block layer can be removed and data structures, etc. can be reduced.

The multi-queue block layer has now been in the kernel for one and a half year. The Open-Channel SSD library is now being accepted upstream. As with the block layer, a lot of work lies ahead in order to enable the software stack to take advantage of these new storage devices. Contrary to the multi-queue block layer, which supports traditional block devices, the open-channel SSDs require new devices on the market. Therefore the adoption of open channel SSDs is tied to the availability of these devices. I have been contacted by many vendors in the last months of my thesis. It appears certain that different models of open channel SSDs will be commercially available in mid 2016. My first goal is to prepare the Linux kernel for such devices. The longer term goal is to move to a single level address space using scale-out storage and share the address space in a distributed system across many diverse applications.

8.3 Perspectives for Research

This thesis has opened up a range of avenues for future research. First, the gap between CPU and IOs has been much reduced with flash memory and promises to be even smaller with next generation Non Volatile Memories (e.g., PCM). Design decisions that assume that storage becomes slower over time is simply not true for NVM-based SSDs. There is an increasing need for solutions that scale with the shrinking gap. An example is how to move computation closer to storage, and make use of its locality revisiting vintage work on active disks and virtual memory in a heterogeneou

Given the increase of performance of SSDs, and the shrinking gap between CPU and IO overhead, quality of service has long seemed out of reach because of the large synchronization overhead it incurs. Solutions based on moving the operating system out of the data path open up for a fresh look on how to provide QoS for secondary storage at low latency.