Not your Grandpa's SSD: The Era of Co-Designed Storage Devices

Alberto Lerner
alberto.lerner@unifr.ch
University of Fribourg
Fribourg, Switzerland

Philippe Bonnet
phbo@itu.dk
IT University of Copenhagen
Copenhagen, Denmark

ABSTRACT

Gone is the time when a Solid-State Drive (SSD) was just a fast drop-in replacement for a Hard-Disk Drive (HDD). Thanks to the NVMe ecosystem, nowadays, SSDs are accessed through specific interfaces and modern I/O frameworks. SSDs have also grown versatile with time and can now support various use cases ranging from cold, high-density storage to hot, low-latency ones. The body of knowledge about building such different devices is mostly available, but it is less than accessible to non-experts. Finding which device variation can better support a given workload also requires deep domain knowledge. This tutorial's first goal is to make these tasks—understanding the design of SSDs and pairing them with the data-intensive workloads they support well—more inviting.

The tutorial goes further, however, in that it suggests that a new kind of SSD plays an essential role in *post-Moore* computer systems. These devices can be *co-designed* to align their capabilities to an application's requirements. A salient feature of these devices is that they can run application logic besides just storing data. They can thus gracefully scale processing capabilities with the volume of data stored. The tutorial's second goal is thus to establish the design space for co-designed SSDs and show the tools available to hardware, systems, and databases researchers that wish to explore this space.

CCS CONCEPTS

• **Information systems** → **Database management system engines**; **Flash memory**; **Storage architectures**; • **Hardware** → **Hardware-software codesign**.

KEYWORDS

SSD co-design; database performance

ACM Reference Format:
Alberto Lerner and Philippe Bonnet. 2021. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3448016.3457540

SIGMOD '21, June 20–25, 2021, Virtual Event, China

ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457540

1 INTRODUCTION

To situate the era of co-design, let us take a historical look back.

Initial SSDs. F. Matsuoka invented Flash-memory in 1980. It was not until 1987 that Toshiba started commercializing NAND Flash and a few years went by before the first SSD appeared on the market in 1991. At the time, HDDs were the secondary storage of choice. To gain quick acceptance, SSDs adopted the same interface.

HDDs and SSDs are, however, very different devices. Suffices to say that HDDs allows updates in place, and SSDs do not. HDDs can only execute requests sequentially, while SSD cannot reach peak performance without performing operations in parallel. HDDs have a low error rate, while SSDs depend on elaborate error correction codes. Put simply, SSDs are more sophisticated than HDDs.

Adopting the simpler interface forced SSDs to hide their internal mechanisms and emulate a block device. Logical block addresses in SSDs do not reflect their physical position. Consequently, applications lost the ability to control data placement, i.e., to decide which data blocks to store contiguously. Moreover, an SSD does not respond to a block request as an HDD does; SSDs have no arms to seek. The operating system (OS) has no visibility into how to reorder IO requests to minimize latency, as it did with HDDs.

For a brief period, applications may have benefitted from SSDs' notably higher IOPS capacity (IOs per second). However, developers quickly realized they were not given a replacement for the rotational disk plates model's predictability. SSDs are fast, but their performance is hard to model and can be erratic at times.

Diversification of SSDs. One needs to look no further than database systems to understand why a performance model is essential. The importance of SSDs in those systems has long been recognized [29]. However, database systems have historically taken control of data placement and IO scheduling to meet the requirements of the disparate workloads on which they depend.

Table 1 presents some of these workloads. For instance, the characteristics involved in log writing (small sequential, synchronous writes) are very different from buffer manager flushes (random read/writes with many requests in flight), in turn distinct from tree structures traversal (read of non-contiguous blocks in strides with potential for prefetching) or checkpoint writing (large sequential blocks written in parallel). What performance profile would an SSD present under these circumstances?

This is not a question a database administrator can answer. It would depend on a given SSD's particular characteristics since SSDs are not a uniform class of devices. They behave differently depending on specific aspects of their architecture: whether they use single-cell or multi-cell NAND-Flash, DRAM or DRAM-less caches,

	access type	operation	queue depth	block size	observations
log writing	sequential	write	1	small	circular, latency sensitive
checkpointing	sequential	write	high	large	may never be read, low priority
buffer flushes	random	write	highly variable	average	bursty
index traversal	mixed	read	mixed	average	prefetching opportunities

Table 1: Characterization of some of the workloads generated by a database system.

light or heavy overprovisioning, low-latency or high-throughput orientation, single or multi-plane, etc. The better question for a database administrator is the following: How to choose the SSD(s) best suited to a given workload? Matching any given SSD to a workload is a daunting task, if not done systematically, just because of the sheer diversity of SSDs.

Modern Interfaces and Abstractions. Instead of choosing SSDs that match a given workload, some have suggested equipping SSDs with new *external* interfaces that give back control to applications. One notable example of such an abstraction is the Open Channel SSDs interface. Simply put, an OCSSD device exports its *geometry*, the equivalent of an HDD numbering its blocks, sectors, and tracks, making the device a white box vis-a-vis the application. Moreover, an OCSSD device also relinquishes control of the scheduling of IOs to an application. The features allow systems and applications to control data placement once again and to reorder IO requests, respectively. However, they need to adapt to the NAND-Flash idiosyncrasies on a given device.

Other interfaces appeared that allow application to describe policies, instead of implementing the underlying mechanims. A notable example of such an interface is called *streams*. The abstraction supports declaring the group to which a file belongs. Operations on a group of files should not impact the operations on another group of files. An SSD that support streams remains a black box for the application, but it guarantees the separation of resources among file groups. Streams reduce interferences among workloads, one of the most common source of performance problem.

Figure 1 contrasts these two kinds of external interfaces, the black-box and white-box variations. It also suggests the interfaces can come in internal flavors, which we discuss next.

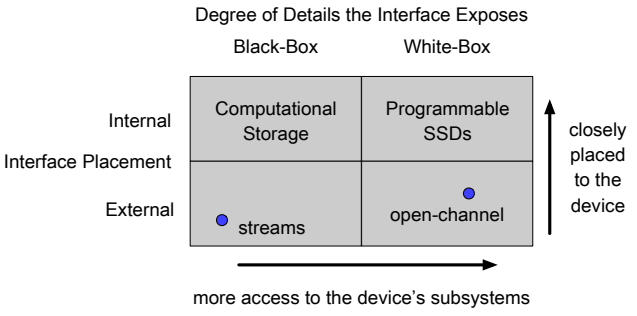

Figure 1: Interfaces fall into quadrants depending on how closely they are placed relative to the device and the offered visibility into its mechanisms.

Specialization and Co-Design. Now, in the co-design era, devices no longer serve just to store data; application logic can be offloaded to SSDs. This can be done in at least two ways. In *computational storage*, the devices have a reserved area for application logic, usually in a co-located FPGA. The logic in the FPGA does not need to cross a bus, e.g., the PCIe subsystem, to reach the device. Computational storage devices rely on an internal, black-box SSD interface to alter requests and data coming in and out of the device.

A classic use case for computational storage devices is early predicate evaluation [9]. A caller would want to scan a data set's blocks but would not be interested in entries in those blocks that do not pass a filtering condition. The caller then pushes the predicate evaluation logic into the device, and the latter applies it to the blocks, generating a filtered stream as a result. In other words, the filtered data never leaves the device, saving expensive data transfers.

The second type of co-designed devices are called *programmable* devices, SSDs whose internal mechanisms can be programmed. The assumption here is that a programmable device is modular, and the internal interfaces delineate replaceable components. A simple example is a device in which a generic page caching mechanism is replaced with a specific one that fits exactly the needs of a workload. Our definition is broader than simple component replacement. Application logic can be embedded in new components. In the cache replacement policy example above, the new component could detect when prefetching opportunities arise and execute them proactively. Programmable devices leverage internal, white-box SSD interfaces.

Not your grandpa's SSD, indeed.

Contributions. In this tutorial we show that SSD diversity and application control need not be mutually exclusive. Both should motivate new SSD designs that are well-suited for data-intensive systems and applications. In summary, we argue that:

- revisiting external interfaces is a pre-requisite for co-design;
- understanding internal interfaces is just as essential;
- the design space for SSD co-design is well defined;
- the tooling to explore this space is mostly available.

We structure the rest of the paper to reflect these four topics.

Our main contributions with this tutorial are (i) a simple but powerful taxonomy of interfaces, and (ii) a continnum of techniques that uses the taxonomy to produce new or evaluate existing SSD variants.

2 EXTERNAL INTERFACES

The traditional storage abstractions are the block device abstraction and POSIX I/O. They have been stable for decades, successfully reducing the complexity of software that relies on stored data at a negligible performance cost. Today, however, these abstractions are failing.

The block device abstraction consists of a flat logical address space, quantized in blocks. Each block has its own address (Logical Block Address or LBA). A block device is a memory abstraction, with a simple read/write interface. Historically, the implicit performance contract associated with this simple abstraction was that (i) sequential accesses are orders of magnitude faster than random accesses and (ii) contiguity in the logical address space favors sequential accesses. SSDs do not respect this performance contract. Attempts at defining new performance contracts under the block device abstraction failed to expose SSD parallelism, and still hid data placement and command scheduling.

New abstractions have emerged outside the block device abstraction to address its limitations:

- Open-Channel [8, 14, 37] redefines the SSD interface, as we mentioned above, using NVMe extensions. It exposes a device's geometry and supports vectored I/Os based on read, write and erase commands. The granularity of read, write, and erase is defined by the SSD. The host is in charge of data placement and I/O scheduling. In particular, the host is in charge of managing write pointers that keep track of where data should be written to guarantee sequential writes within each block.
  OCSSDs are supported by Linux via the LightNVM framework. `Pblk` is a full-fledged host-based Flash Translation Layer (FTL) inside the Linux kernel providing a block device abstraction to legacy systems relying on POSIX I/O. Despite adoption in the SPDK framework and by companies such as Alibaba, attempts to standardize Open-Channel (within and outside NVMe) failed. Open-Channel SSDs thus remain either proprietary or experimental devices.
- ZNS [7, 13] has been part of the NVMe standard since July 2020. It defines zoned namespaces, i.e., a partitioning of the logical address space in zones, composed of logical blocks, with the constraint that logical blocks must be written sequentially within a zone. This corresponds to the interface of shingled magnetic recording (SMR) HDDs. In essence, it is a variant of Open-Channel. ZNS provides two operational modes: (1) a host managed mode, where the host is responsible for issuing write commands, maintaining a write pointer per zone and managing explicit zone transitions, and (2) a host aware mode, where the SSD is responsible of enforcing sequential writes within a zone. In host aware mode, the host issues asynchronous append commands; zone transitions are implicit. ZNS drives are not (yet) commercially available. There is ZNS support in QEMU, and in the Linux stack.
- Extensions of the block device interface, so-called *directives*, make it possible for hosts to cooperate with the embedded FTL. The addition of the `TRIM` command (initially in the context of SATA and SCSI) falls in this category. The `TRIM` command allows the host to inform an SSD that certain blocks are no longer in use and should be erased.
  Another extension is the *stream* directive that we briefly discussed above. It allows a host to indicate that files belong to separate streams (formulated as lifetime hints in the `fcntl` system call). SSDs equipped with support for multi-streams will isolate data from different streams on different blocks, thus ensuring that they can be garbage collected together [45].

The evolution of the SSD interface is accompanied by an evolution of the I/O framework on the host. POSIX I/O initially defined a simple and elegant abstraction relying on files as arrays of bytes as main storage abstraction. The interface makes it possible to create/delete, open/close and read/write from a file. A file looks like an array of disk blocks. The default behavior relies on the existence of a page cache in memory that contains disk blocks. There is usually a 1-1 mapping between virtual memory pages and disk blocks. The interface includes a function that flushes dirty pages from memory to disk (fsync). This is the textbook example of a deep module [35]. However, the problems with this abstraction are piling up.

The first problem is that the original read/write interface was synchronous, which was acceptable for HDDs with hundreds of millisecond latency. But such interface became untenable when disk latency was reduced by a couple of orders of magnitude. An asynchronous POSIX I/O interface was introduced in the early 2000s. However, its implementation in Linux was never satisfactory.

In addition, the latency of modern SSDs requires that device driver overhead be reduced. As a result, new mechanisms have been introduced to handle I/Os: SPDK in user-space [47] and `io_uring` [2] in the Linux kernel. SPDK relies on a user-space NVMe driver running in *polling mode*, thus trading increased CPU usage for lower I/O latency. SPDK provides not only block devices, but also ZNS and Open-Channel abstractions.

`Io_uring` was introduced in the Linux kernel in 2019 as a flexible and efficient mechanisms to manage I/O submissions and completions. `Io_uring` relies on shared circular queues between user-space and kernel-space and can be used to access SSDs either via the file system abstraction or directly via a NVMe driver. In the default mode, `io_uring` is interrupt-based (as legacy asynchronous I/Os) but it can be configured to run in polling mode with a dedicated kernel thread (as can SPDK in user-space).

Reducing software overhead also requires avoiding redundancies and missed optimization opportunities across layers. A strict layering of POSIX I/O on top of block devices results in the log-on-log problem [46], unpredictable tail latencies [18] and resource under-utilization [10]. New storage abstractions make it possible to tackle these problems. The alternative is to explore system design based on cross-layer optimizations.

In summary, the block device and POSIX I/O are no longer universal storage abstractions. They are still useful but have lost their role as the narrow waist of the storage stack in the NVMe ecosystem. In fact, NVMe's extensible design has proven valuable for what is yet another promising step in the evolution of modern storage stacks: computational storage. We talk about the latter and other *internal* interfaces next.

3 INTERNAL INTERFACES AND MECHANISMS

Several internal mechanisms in an SSD influence its behavior and performance. It follows that each of these mechanisms could be made modular by establishing an interface that allows to replace them. We call those interfaces *internal*. To narrow down the most promising interface sites, we look for the sub-systems that can control or attenuate interference among workloads.

We found empirically that those sub-systems were: caching, scheduling, data placement, and error-correction codes (ECC) calibration. In the rest of this section, we talk about one sub-system/interface at a time and note that they are critical interfaces in a Programmable Device (cf. Figure 1). We also talk about what we called Computational Storage in that figure. Such devices bring an embedded platform capable of executing application logic.

3.1 Caching

A DRAM buffer in a Flash controller can serve as a cache. Many SSDs use it to absorb writes and serve reads off of it, with the expected performance improvements.

A cache managed with a typical LRU policy may not reflect the priority of different operations. For instance, a workload with larger operations (more pages per operation) will occupy larger portions of the cache than a smaller operations workload, even if the workloads were similar in terms of read/writes frequencies. Therefore, some works attempt to manage the cache replacement policy differently [25, 26].

3.2 Scheduling

A similar effect occurs when scheduling operations to LUNs. A LUN is the basic unit in which to organize NAND-Flash cells. An SSD is made of many LUNs, and each of them can serve one request at at time. Typically, read and write commands are broken down into individual Flash (page-sized) operations. Write operations, in particular, have the option of being assigned to any available LUN.

The larger operations will tend to engage a larger number of LUNs simultaneously, creating a "delay" effect in other operations that may be ongoing. In other words, there is very little *isolation* between in-flight operations. Some works address the issue by establishing fair scheduling among workloads [42] while others resort to suspending/re-scheduling expensive operations that are root causes of interference [44].

3.3 Data Placement

In most SSDs, pages from different workloads may be placed on a same block. Mixing pages at the block level have at least two negative performance implications. First, if the workloads present different page lifecycles, their pages would be invalidated (deleted) at different times. This minimizes the chance of used but empty blocks occurring naturally. Empty blocks are critical to fast garbage collection because they allow erase commands to execute immediately, skipping the need to *copy-back* still valid pages from victim blocks onto new blocks.

The second performance implication is that hot pages of different workloads placed in the same block cannot be read in parallel. Recall, a LUN's page buffer can only serve one read operation at a time. Some works address those issues by separating the blocks that can be used by each workload [24]. Some go even further; they "carve" regions of the device and assign those to different workloads in an attempt to achieve deterministic response times [28].

3.4 ECC Calibration

Delivering pages without errors requires robust ECC techniques. However, these techniques are expensive in terms of operations' latency and circuit area. Once again, different storage managers, and possibly different workloads, may be tolerant to different error management disciplines.

Consider a workload that deals with images. If a page stores an image's pixels, perhaps some applications would tolerate a few bits on the page to be flipped in exchange of performance. This is a shift of some ECC responsibility onto applications, allowing them to decide whether to consume pages with a controlled degree of error [32]. Moreover, if a calibrated degree of error is accepted, there are techniques such as *shallow* erases [20] or *wear unleveling* (selecting aging of pages according to the individual signal strength) [21] that can significantly increase a device's longevity.

3.5 Computational Devices

A task force at SNIA, a trade group representing storage companies, has defined terminology and architectures for computational storage [14]. They denote the processing units integrated with storage computational storage processors. Together with storage, these processors provide Computational Storage Services (CSS) to hosts. Storage nodes equipped with appropriate drivers access CSS via a network (Ethernet or PCIe).

SNIA distinguishes between fixed computational storage services, based on predefined functions such as compression or encryption, and programmable computational storage services, via code upload. Four mechanisms are listed for code upload: operating system image, containerized application, FPGA bitstream and eBPF bytecode (eBPF is a vendor-neutral Instruction Set Architecture already used for offloading code to Network Interface Cards). This suggests that computational storage programs can be packaged as containers. They should accommodate hardware acceleration (on FPGA) and code shipping (of eBPF code).

An extension of the NVMe standard for computational storage is expected in 2022. While it will define mechanisms for uploading code to computational storage, it will not address how to develop computational storage programs that are efficient and secure for a given data-intensive system. This remains an open issue.

4 THE CO-DESIGN CONTINUUM

We classify techniques for co-designing SSDs and data-intensive systems along a *continuum*, ranging from treating SSDs as complete black-boxes to allowing increasing degrees of behavior configuration, up to practically designing SSDs "à la carte." Figure 2 depicts these possibilities. The levels in this continuum are, in order of adaptivity, the following.

Adapt by simply shaping the workload. In the absence of other techniques, an application can unilaterally try to modify the workloads it sends to SSDs to foster performance. The techniques at this level involve adjusting the queue depth (number of outstanding-ing requests), changing read/write operation sizes via batching of operations, and transforming random workloads into sequential ones whenever possible [19]. To aid in these changes, applications may consider using specialized data structures more adapted to SSDs than to HDDs [12]. The techniques at this level treat the SSDs as black boxes.

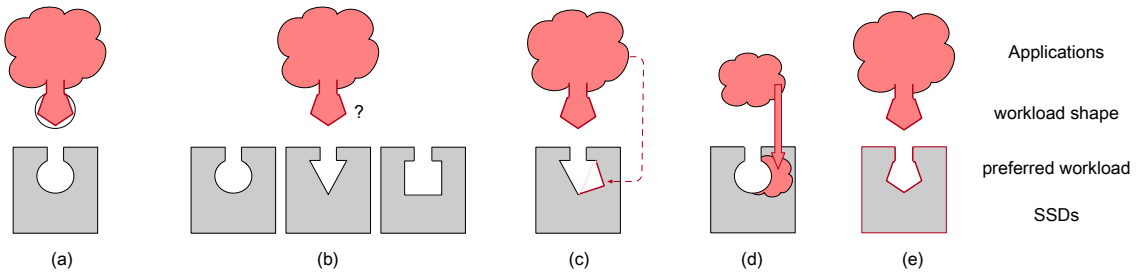

Figure 2: Continuum of adjustments an application, on top, and a device, on bottom, can make in order to interact efficiently. The interaction occurs through an application-generated workload, to which a device may be more or less efficiently tuned. The steps on the continuum are: (a) the application unilaterally adapts to the SSD characteristics to perform well; (b) the application weighs in among alternative SSDs before selecting one; (c) the application configures some aspects of a device to its needs through an external interface before using the device; (d) a computational device serves as an application platform, running part of its logic through an internal, black-box interface; (e) a programmable device replaces or adjusts some of its components through an internal, white-box interface to fit the exact needs of an application.

Adapt by matching the shaped workload with a carefully chosen commercial device. The next step in the continuum of techniques involves not only shaping applications workloads but also pairing them with compatible devices. At this level, one hopes to find among the many available SSDs a device with features that, although still general-purpose, can perform better under the application's scenarios. For instance, enterprise SSDs come in read-heavy, write-heavy, and mixed workload variations. Their caching and allocation strategies are optimized for each specific workload. (Aside from performance, endurance is also considered, with write-heavy devices with much more longevity than the others.)

Workload-mix is but one of the axes around which commercial devices vary. Many other "variation axes" for SSDs exist. Some devices behave better according to a target queue-depth. For instance, some devices excel well when only one I/O request is outstanding (called queue depth 1), such as seen in database transaction logs. Some other devices compromise cost and writing speed by using a different mix of single- and multi-cell NAND-Flash memories. The write performance decreases as an inverse proportion as the number of bits per cell. Several other criteria can further tailor devices to particular workloads, such as DRAM and DRAM-less devices, overprovisioning amounts, etc. In practice, there is no shortage of SSDs variations in the market today.

Adapt by further tuning the device to the workload. The pairing strategy described above has many limitations. The main one being risking performance regressions as systems and devices evolve. Some devices can implement external interfaces that allow them to adopt specific behavior *in the field*.

Several relevant interfaces have been recently discussed. For example, devices with stream support [24, 45] aim at providing workload isolation by controlling caching, scheduling, and data placement at once. As we discussed, streams is an external, black-box interface that requires very little change on the application side. Other relevant interfaces exist that are already standardized. Recent versions of the NVMe standard propose several performance directives such as Predictable Latency Mode, NVM sets, Host Memory Buffers, and IO Performance and Endurance hits [1]. These are also external, black-box type of interfaces.

In short, all the external interface methods discussed in Section 2, be them black-box or white-box, apply here.

Bypass fixed interfaces via Computational Storage. So far, the techniques we mentioned fine-tune both the application and storage stack, but without altering such stacks. This next category shows that device specialization can take a step further by using the internal interfaces described in Section 3. We discuss the techniques of this category via some examples.

The KV-SSD (key-value) [5] can entirely bypass the OS and the NVMe layers and expose a low-level KV interface into the device that manipulates objects, e.g., `put()`, `get()`, and `delete()` calls. The Graph-SSD [34] can use NVMe extended commands to export a graph manipulation API, which stores and traverses edge and vertex data.

We note that the Graph-SSD is built upon the OpenSSD prototyping platform that we discuss shortly in Section 5. Besides the OpenSSD, there exist commercial options for computational storage on the market. Samsung has recently introduced its SmartSSD [40] that brings an FPGA-based platform that applications can customize. Some frameworks emerged that make it easy to define application-specific storage interfaces. The OX framework has been used, for instance, to provide a batched I/O interface to a host running LLAMA/Bw-tree [36].

Lastly, A few projects proposed frameworks for dynamically offloading application components to computational storage, most notably Willow [41], Biscuit [17] and Insider [39].

Design specialized devices that meet the needs of a given workload. Device specialization can be accomplished without locking the device to an individual workload category. The devices in this category also export the internal interfaces discussed in Section 3. We present a few examples of such devices next.

The 2B-SSD [3] supports two different interfaces: the traditional one (NVMe based) and a *byte-addressable*, low-latency one. The application may use the latter to talk to the device's low-latency side without a block interface, e.g., via memory mapping. A software interface allows an application to assign a given page to the "fast" or the "slow" sides and to change its side at any time. This arrangement proved helpful for database transaction logs workloads.

The Cognitive-SSD [33], learns access patterns through Machine Learning techniques. It then creates a low-latency path in the device for data retrieval, for instance, by using predictive techniques for caching. The Cognitive-SSD is also implemented atop the OpenSSD.

The difference between this and the previous category is subtle. In computational devices, we emphasize moving application logic to the device, whereas in programmable devices we showed aplication-agnostic devices with high degree of adaptivity. In practice, we may see caracteristics of both classes at once in some devices [11].

5 TOOLING

There is a large body of tools that allows us to build and evaluate interface-implementation pairings. We separate these tools into three classes: simulation-based, hardware prototyping, and a hybrid approach. Simulations have the advantage of a quick development cycle, but simulation time can be an issue for large workloads. Hardware prototyping is what companies use to test new designs. Thanks to a powerful academic platform, it is also available to the research community. The lack of availability of NAND-Flash alternatives, however, may constraint the prototyping platform. The hybrid platform presents a good balance of flexibility, accuracy, and development speed. We discuss each of these platforms in turn.

5.1 Simulation

A recent class of SSD simulation platforms take into consideration many low-level aspects of an SSD and, as such, are capable of approximating its behavior. Examples of such platforms are MQSim [43] and Amber [15]. These simulators consider many internal mechanisms we discuss here and, therefore, can approximate actual devices' behavior to some degree. Moreover, the simulators come with an existing array of functionalities that can be configured with different Flash package parameters. They can be extended through software programming to test changes in devices' behavior.

An aspect that makes these tools particularly useful is their integration with a whole-system simulator called gem5 [4]. Gem5 can run an existing binary on an emulated CPUs. The SSD emulators can be the storage stack attached to this CPU. As with any simulation environment, the experiments can be particularly time-consuming. However, these tools' maturity makes them excellent candidates to study new devices or the latter's interaction with database systems or prototypes.

Another category of simulators that target Flash memory specifically are also available, such as NANDFlashSim [22]. This simulator is limited in that it does not consider many system-level aspects such as channel designs and the utilization of caches. However, it is invaluable as a studying tool because it incorporates several aspects that contribute to variability in performance at the Flash level [16].

5.2 Prototyping

Rapid prototyping platforms exist that can allow the actual development of new SSDs. A particularly flexible one is the OpenSSD platform, including the Cosmos+ [27] and some commercial derivatives. The Cosmos+ is an FPGA-based, full-fledged NVMe device with a large portion of the functionality implemented as open source, both in firmware (in C) and hardware (in Verilog).

The device manages up to 2TB of hybrid SLC/MLC Flash divided in 8 channels, each with 8 planes, and can be configured or altered in many ways. Thanks to this platform, there exist an Open Channel implementation [30] as well as a micro-architecture performance measurement framework [31]. We mentioned in Section 4 that several research works have used the OpenSSD as a prototyping platform for both internal and external interface experiments.

It is also worth noting a nascent effort to leverage FPGA platforms for peripheral development in general, where controller components are made available independently of the prototyping platform. One example is the OpenExpress [23], which implements a fully functional NVMe controller.

5.3 Hybrid

Hardware prototyping is limited by the availability of NAND-Flash documentation, often regarded as trade secrets. Moreover, the development cycle for every new NAND-Flash type may be prohibitive in an experimental environment. A hybrid approach to overcome these difficulties is to combine a hardware prototype such as the OpenSSD with an emulated NAND-Flash platform. A suitable platform to the latter is the recently standardized M.2 Accelerator Module Hardware Specification [38].

The M.2 module combines a processing element and DRAM. For example, there exist implementations in the market where the processing element is an FPGA [6]. The idea is to use the M.2 module to emulate different NAND-Flash types at a time behind a PCIe interface. In other words, the M.2 module can pretend it is built out of a specific kind of flash memory and emulate the response time that accessing that memory would entail. The PCIe interface would be the conduit to issue requests and deliver responses. The hybrid approach would have none of the simulation platforms' speed limitations and neither the lack of NAND-Flash variety of the prototyping ones.

6 CONCLUSION

This tutorial discussed various ways to harmonize applications' workloads and SSDs designs. It classified the latter's interfaces in a taxonomy based on two axes: how close to the device an interface is placed and how much access that interface opens into the device. We showed that a continuum of techniques derived from this taxonomy allows for increasingly deeper levels of co-design.

We hope to have persuaded hardware, systems, and database researchers that co-designing applications and SSDs is desirable and possible with the current tooling. Exploring this integration continuum has just begun. We strongly believe that a body of techniques, abstractions, and devices may emerge from this effort that is fundamental for designing new high-performance systems.

ACKNOWLEDGMENTS

The first author received funding from the European Research Council (ERC) under the European Union Horizon 2020 Research and Innovation Programme (grant agreement 683253/Graphint). The second author received funding from the European Union's Horizon 2020 Research and Innovation Programme (grant agreement No 957407/Daphne).

REFERENCES

[1] Nick Adams and David Woolf. NVMe 1.4 Features and Compliance: Everything You Need to Know, 2019.

[2] Jens Axboe. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf, 2019.

[3] D. Bae, I. Jo, Y. A. Choi, J. Hwang, S. Cho, D. Lee, and J. Jeong. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. In *International Symposium on Computer Architecture*, 2018.

[4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The Gem5 Simulator. *SIGARCH Comput. Archit. News*, 39(2), 2011.

[5] T. Bisson, K. Chen, C. Choi, V. Balakrishnan, and Y. Kee. Crail-KV: A High-Performance Distributed Key-Value Store Leveraging Native KV-SSDs over NVMe-oF. In *International Performance Computing and Communications Conference*, 2018.

[6] Bittware. Xilinx® kintex® ultrascale+™ fpga on m.2 accelerator module.

[7] Matias Bjørling. From Open-Channel SSDs to Zoned Namespaces. USENIX Vault, 2019.

[8] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. LightNVM: The Linux Open-Channel SSD Subsystem. In *USENIX Conference on File and Storage Technologies*, 2017.

[9] Wei Cao, Yang Liu, Zhushi Cheng, Ning Zheng, Wei Li, Wenjie Wu, Linqiang Ouyang, Peng Wang, Yijing Wang, Ray Kuan, Zhenjun Liu, Feng Zhu, and Tong Zhang. Polardb meets computational storage: Efficiently support analytical workloads in cloud-native relational database. In *USENIX Conference on File and Storage Technologies*, 2020.

[10] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Symposium on High Performance Computer Architecture*, 2011.

[11] Jaeyoung Do, Sudipta Sengupta, and Steven Swanson. Programmable solid-state storage in future cloud datacenters. *Communications of the ACM*, 62(6), 2019.

[12] Athanasios Fevgas, Leonidas Akritidis, Panayiotis Bozanis, and Yannis Manolopoulos. Indexing in flash storage devices: a survey on challenges, current approaches, and future trends. *The VLDB Journal*, 29(1), 2020.

[13] Javier Gonzalez. Zoned Namespaces: Standardization and Linux Ecosystem | SNIA. SDC EMEA, 2020.

[14] Javier González. The Denali Open-Channel Standard: Impact and Future. In *Flash Memory Summit*, 2019.

[15] D. Gouk, M. Kwon, J. Zhang, S. Koh, W. Choi, N. S. Kim, M. Kandemir, and M. Jung. Amber*: Enabling Precise Full-System Simulation with Detailed Modeling of All SSD Resources. In *IEEE/ACM International Symposium on Microarchitecture*, 2018.

[16] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf. Characterizing flash memory: Anomalies, observations, and applications. In *IEEE/ACM International Symposium on Microarchitecture*, 2009.

[17] Boncheol Gu, Andre S. Yoon, Duck-Ho Bae, Insoon Jo, Jinyoung Lee, Jonghyun Yoon, Jeong-Uk Kang, Moonsang Kwon, Chanho Yoon, Sangyeun Cho, Jaeheon Jeong, and Duckhyun Chang. Biscuit: A framework for near-data processing of big data workloads. *SIGARCH Comput. Archit. News*, 44(3), 2016.

[18] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. 2016.

[19] Jun He, Sudarsun Kannan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. The Unwritten Contract of Solid State Drives. In *European Conference on Computer Systems (Eurosys)*, 2017.

[20] Jaeyong Jeong, Sangwook Shane Hahn, Sungjin Lee, and Jihong Kim. Lifetime Improvement of NAND Flash-based Storage Systems Using Dynamic Program and Erase Scaling. In *12th USENIX Conference on File and Storage Technologies*, 2014.

[21] Xavier Jimenez, David Novo, and Paolo Ienne. Wear Unleveling: Improving NAND Flash Lifetime by Balancing Page Endurance. In *USENIX Conference on File and Storage Technologies*, 2014.

[22] M. Jung, E. H. Wilson, D. Donofrio, J. Shalf, and M. T. Kandemir. Nandflashsim: Intrinsic latency variation-aware NAND flash memory system modeling and simulation at microarchitecture level. In *Symposium on Mass Storage Systems and Technologies*, 2012.

[23] Myoungsoo Jung. OpenExpress: Fully Hardware Automated Open Research Framework for Future Fast NVMe Devices. In *USENIX Annual Technical Conference*, 2020.

[24] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The Multi-streamed Solid-State Drive. In *USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage 14, 2014.

[25] Minji Kang, Soyee Choi, Gihwan Oh, and Sang-Won Lee. 2R: Efficiently Isolating Cold Pages in Flash Storages. *PVLDB*, 13(12), 2020.

[26] J. Kwak, J. Lee, D. Lee, J. Jeong, G. Lee, J. Choi, and Y. H. Song. GALRU: A Group-Aware Buffer Management Scheme for Flash Storage Systems. *IEEE Access*, 8, 2020.

[27] Jaewook Kwak, Sangjin Lee, Kibin Park, Jinwoo Jeong, and Yong Ho Song. Cosmos+ OpenSSD: Rapid Prototype for Flash Storage Systems. *ACM Trans. Storage*, 16(3), 2020.

[28] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. DC-Store: Eliminating Noisy Neighbor Containers using Deterministic I/O Performance and Resource Isolation. In *USENIX Conference on File and Storage Technologies*, 2020.

[29] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A Case for Flash Memory SSD in Enterprise Database Applications. In *ACM SIGMOD International Conference on Management of Data*, 2008.

[30] Sangjin Lee and Yong-Ho Song. Experimental Results of Implementing NVMe-based Open Channel SSDs. In *Flash Memory Summit*, 2017.

[31] Alberto Lerner, Jaewook Kwak, Sangjin Lee, Kibin Park, Yong Ho Song, and Philippe Cudré-Mauroux. It Takes Two: Instrumenting the Interaction between In-Memory Databases and Solid-State Drives. In *Conference on Innovative Data Systems Research (CIDR)*, 2020.

[32] Fei Li, Youyou Lu, Zhongjie Wu, and Jiwu Shu. ASCache: An Approximate SSD Cache for Error-Tolerant Applications. In *Design Automation Conference*, 2019.

[33] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A Deep Learning Engine for In-Storage Data Retrieval. In *USENIX Annual Technical Conference*, 2019.

[34] Kiran Kumar Matam, Gunjae Koo, Haipeng Zha, Hung-Wei Tseng, and Murali Annavaram. GraphSSD: Graph Semantics Aware SSD. In *International Symposium on Computer Architecture*, 2019.

[35] John Ousterhout. *A Philosophy of Software Design.* Yaknyam Press, 2018.

[36] Ivan Luiz Picoli. *OX: Deconstructing the FTL for Computational Storage.* PhD thesis, 2019.

[37] Ivan Luiz Picoli, Niclas Hedam, Philippe Bonnet, and Pinar Tözün. Open-Channel SSD (What is it Good For). In *Conference on Innovative Data Systems Research, (CIDR)*, 2020.

[38] Open Compute Project. M.2 accelerator module hardware specification v1.0. http://files.opencompute.org/oc/public.php?service=files&t=b5a78cf8a8941ef6dd4b5590aa79b824&download.

[39] Zhenyuan Ruan, Tong He He, and Jason Cong. Insider: Designing in-storage computing system for emerging high-performance drive. In *USENIX Annual Technical Conference*, 2019.

[40] Balavinayagam Samynathan. Computational Storage For Big Data Analytics. In *Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures*, 2019.

[41] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[42] A. Tavakkol, M. Sadrosadati, S. Ghose, J. Kim, Y. Luo, Y. Wang, N. Mansouri Ghiasi, L. Orosa, J. Gómez-Luna, and O. Mutlu. FLIN: Enabling Fairness and Enhancing Performance in Modern NVMe Solid State Drives. In *International Symposium on Computer Architecture*, 2018.

[43] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQSim: A Framework for Enabling Realistic Studies of Modern Multi-Queue SSD Devices. In *USENIX Conference on File and Storage Technologies*, 2018.

[44] Guanying Wu and Xubin He. Reducing SSD Read Latency via NAND Flash Program and Erase Suspension. In *USENIX Conference on File and Storage Technologies*, 2012.

[45] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. AutoStream: Automatic Stream Management for Multi-Streamed SSDs. In *ACM International Systems and Storage Conference (Systor)*, 2017.

[46] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't stack your Log on my Log. 2014.

[47] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. SPDK: A Development Kit to Build High Performance Storage Applications. In *International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017.