2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture

2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives

Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang,
Sangyeun Cho, Dong-Gi Lee, Jaeheon Jeong
Memory Business, Samsung Electronics Co., Ltd.

闪存写延迟高，为追求快速写，于数据路径上采用NVRAM（非易失，可保有持久性），在系统确认写请求前，事先“临时且快速”的暂存写入数据（类似写入缓存（与高速缓存cache区别（见附页）），后续或等待空闲时批量写入以达到“快速”的目的），以此手段保证快速写及持久性。

***Abstract*—Performance critical transaction and storage systems require fast persistence 快速写且保有持久性 of write data. (Typically, a non-volatile RAM (NVRAM) is employed on the datapath to the permanent storage, to temporarily and quickly store write data before the system acknowledges the write request.) NVRAM is commonly implemented with battery-backed DRAM. Unfortunately, battery-backed DRAM is small and costly, and occupies a precious DIMM slot. In this paper, we make a case for *dual, byte- and block-addressable solid-state drive* (2B-SSD), a novel NAND flash SSD architecture designed to offer a dual view of byte addressability and traditional block addressability at the same time. Unlike a conventional storage device, 2B-SSD allows accessing the same file with two independent byte- and block- I/O paths. It controls the data transfer between its internal DRAM and NAND flash memory through an intuitive 直观的 software interface, and manages the mapping of the two address spaces. 2B-SSD realizes a wholly different way and speed of accessing files on a storage device; applications can access them directly using memory-mapped I/O, and moreover write with a DRAM-like latency. To quantify the benefits of 2B-SSD, we modified logging subsystems of major database engines to store log records directly on it without buffering them in the host memory. When running popular workloads, we measured throughput gains in the range of 1.2× and 2.8× with no risk of data loss.**

无需如最上注释一般运用写入缓存（写入缓存可在主机端、数据路径或设备端）加速或为缓存写请求至闪存写单元（页面）而设置的写入缓存（此情况下写入缓存无需是PM）

***Keywords*-2B-SSD; non-volatile memory; WAL;**

I. Introduction

For decades, *block* has been the unit of data I/O for a mass storage system [1]. All data managing software, including operating systems, has been ultimately 最终 optimized for block, and the techniques developed thus far are relevant and working very well until these days. The memory hierarchy trinity 三位一体—CPU cache, DRAM, and non-volatile disks—appears to remain strong and even more solidified with the notion of block deeply entrenched 根深蒂固 therein.

Modern data management and storage systems face two very important technology trends, which however, force us to revisit the memory and storage hierarchy design and implementation practices. Obviously, the first trend is irrevocable 不可撤销的 and rapid replacement of hard disk drives (HDDs) by NAND flash solid-state drives (SSDs). The trend is especially pronounced 明显的 in large, hyperscale 超大规模的 datacenters [2–4] and mission-critical enterprise storage platforms [5–7]. As a matter of fact, HDD vendors are starting to drop "performance HDD products" from their offerings [8]. Accordingly, significant efforts are exerted 付出 to revising 修订 the OS [9, 10] and database algorithms [11, 12] as well as the storage I/O protocol itself [13], to overcome inefficiencies introduced by HDD-specific legacy optimizations.

The second trend is renewed interests in byte-addressable, *persistent memory* (PM) technologies [14–16]. While realistic PM products are still unavailable, both software and hardware communities are looking for new strategies to utilize byte-addressable PM to improve system responsiveness 响应 and power consumption [17, 18]. The closest product to PM available today is *non-volatile RAM* (NVRAM), which is typically built with DRAM, NAND flash memory, and battery-backed power supply [19]. Until these days, NVRAM has been a choice by a very limited number of high-end enterprise storage servers to accelerate logging of both user data and metadata. New PM technologies have the potential to revise the current working memory and block storage dichotomy 二分法, and eliminate the (de-)staging and transformation of data between DRAM and disks. 图1（b）结构 【However, it is unlikely that PM will completely replace block storage from systems for many reasons. First, the capacity of PM is very limited today. Most NVRAMs are only 4 ∼ 16 GBs. 图1（c）结构 Moreover, PM is not likely to replace whole working memory because there is a gap between the access latency of DRAM and that of PM [14]. Hence, NVRAM is used as a non-volatile *write buffer* or a small low-latency storage device, rather than replacing whole memory and storage.】图1（b）、图1（c）结构存在的缺陷（图1（c）缺点p427也有陈述）

若PM直接充当二级存储，如图1（b）结构，因可字节寻址，无需软件栈开销

若PM放在主机端，如图1（c）结构，因PM非易失，可于主机端做二级存储的功能来使用，则原来的“主机-设备”数据路径上的数据降级、传输等均可被缓减

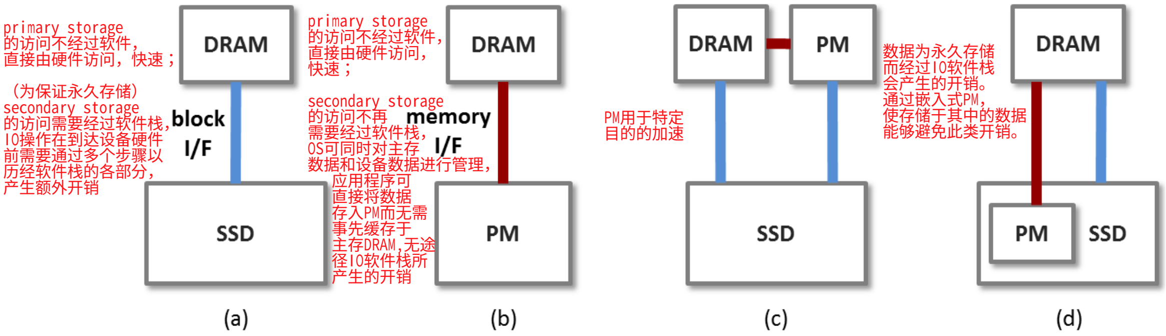
Today's memory-storage hierarchy using NVRAM has another limitation. Data in PM cannot be accessed by block I/O, which has huge benefits in reducing CPU overheads for large data transfer [20]. This implies that data in PM should be de-staged to the permanent storage through the entire I/O stack. This limitation motivated us to think of a *hybrid persistent store* which enjoys the advantage of both block- and byte-addressable interface and eliminates (de-)staging operations and I/O stack overheads all together.

则主机无法通过传统块IO访问PM
这意味着可以将PM中的数据通过整个I/O堆栈降级到永久存储中（因为PM可以由CPU直接存取而无需经过软件栈，PM降级到设备端后，后续可以绕过IO软件栈由另一数据路径存取，这也减少了数据传输开销）而无需担心主机将PM按块设备对待而产生开销→进而提出图1（d）结构

Fortunately, today's PCIe-attached NVMe SSDs incorporate right ingredients that help us realize the idea of hybrid persistent store. These ingredients are the SSD-internal DRAM and the byte addressability of the PCIe interconnect. In this work, we propose *2B-SSD* and its memory management APIs designed for modern NVMe SSDs. To the best of our knowledge, this is the first SSD proposal that allows users to access the same file with two totally separated datapaths: Memory interface and conventional (NVMe) block I/O. By its APIs, any byte from a file can be written or read directly from an application

（通过memory interface）

2575-713X/18/$31.00 ©2018 IEEE
DOI 10.1109/ISCA.2018.00043



primary storage
的访问不经过软件，
直接由硬件访问，快速；

（为保证永久存储）
secondary storage
的访问需要经过软件栈，
IO操作在到达设备硬件
前需要通过多个步骤以
历经软件栈的各部分，
产生额外开销

primary storage
的访问不经过软件，
直接由硬件访问，
快速；

secondary storage
的访问不再
需要经过软件栈，
OS可同时对主存
数据和设备数据进行管理，
应用程序可
直接将数据
存入PM而无需
事先缓存于
主存DRAM，无途
径IO软件栈所
产生的开销

PM用于特定
目的的加速

数据为永久存储
而经过IO软件栈
会产生的开销。
通过嵌入式PM，
使存储于其中的数据
能够避免此类开销。

Figure 1. Logical view of various storage hierarchies: (a) Conventional two-level hierarhcy of DRAM and SSD, (b) PM store, (c) Heterogeneous memory of DRAM and PM, and (d) Hybrid store of PM and SSD. （图中上面的结构表示primary storage，下面的结构表示secondary storage）
（棕色线表示主存主线（字节寻址），蓝色线表示快接口（块寻址））

区别memory和store，前者指主存，后者指存储
→(c)heterogeneous memory是指异质主存，DRAM和PM共同构成主存（primary storage）
→(d)hybrid store是指混合存储，SSD和PM共同构成存储介质（secondary storage）

without being buffered in the host memory. It is especially remarkable that 2B-SSD and its APIs realize data persistence 数据持久性 with a DRAM-like latency. Indeed, we demonstrate, with a product-strength 2B-SSD implementation, that sub-one μs latency is possible for a write of 1 KB or less in size.

In order to implement this novel SSD architecture, we embrace a hardware/software co-design that offers a dual view of byte- and block- addressability at the same time. For allowing memory interface to files stored on SSD, we introduce an internal datapath between a portion of the SSD-internal DRAM and NAND flash memory. We first implement an address translation hardware that redirects references to memory-mapped I/O (MMIO) [21] addresses to those to the former. We then write firmware for the mapping between these two. Also for preventing inadvertent 疏忽的 data updates and ensuring consistency even in the presence of two independent byte- and block- I/O paths, we add a hardware component that gates block I/O requests to NAND pages that are currently mapped to the DRAM for access through memory interface. Lastly for guaranteeing persistence of data in the DRAM across power outages, we turn the DRAM into a persistent memory. We back it up with additional power capacitors, and integrate hardware-software support for power-off detection and recovery. If applications define a mapping between this NVRAM and NAND flash memory using 2B-SSD APIs, the former works as a buffer for the latter. Thus, the data stored on the NAND flash memory can be accessed with memory interface as if they were stored on a fast, byte-addressable memory.

As such, the major contributions of this paper can be summarized as follows:

- To date, 2B-SSD is the first SSD proposal that offers accessing the same file with two separated byte- and block- I/O paths, and has been implemented on a commodity NVMe SSD. 2B-SSD and its APIs allow accessing files from NAND flash memory directly with memory interface. It is especially remarkable that both realize DRAM-like write latency (i.e., 100's of nanoseconds) on high-capacity NAND flash.
- To make good use of 2B-SSD, we redesign *write-ahead logging* (WAL) [22] and successfully ported the revised WAL to widely deployed SQL and NoSQL database engines. To mitigate WAL's performance penalty incurred by small frequent updates, we modified them to employ 2B-SSD APIs so that their log records are directly stored on SSD without being buffered in host memory.
- We measure and report application level results on a real server platform. When our revised database engines run Linkbench [23] or YCSB [24] on 2B-SSD, the throughput gain falls in the range of 1.2× and 2.8× with no risk of data loss.

In the remainder of this paper, we give the background of this work in Section II by discussing storage hierarchy models. In Section III, we describe the overall architecture and detailed design of 2B-SSD and its APIs. Section IV presents a case study of how 2B-SSD concepts help accelerate WAL processing, followed by experimental evaluation results of Section V. We discuss our research findings further in Section VI and related work in Section VII. Finally, Section VIII concludes.

II. Background

A. Storage Hierarchy

1) Traditional architecture: Computer systems traditionally have two storage hierarchy levels, *primary storage* (e.g., main memory built with DRAM) and *secondary storage* (e.g., SSD block storage), as depicted in Fig. 1(a). Once virtual to physical address mapping is established by the OS, a CPU's access to the primary storage is achieved through hardware mechanisms (like TLB) without software intervention. Because memory accesses are the single most frequent operation, the primary storage must provide very high bandwidth and low latency [25].

primary storage的访问
不经过软件，直接由硬
件访问，快速；

secondary storage的访
问需要经过软件栈，IO
操作在到达设备硬件前
需要通过多个步骤以历
经软件栈的各部分，产
生额外开销
（因主机和SSD的数据组
织形式不同，此结构需
软件栈）

On the other hand, a secondary storage access heavily involves software coordination. For example, the Linux I/O stack encompasses a complex structure that is composed of a virtual file system layer, a multitude of file systems, the block I/O layer, I/O scheduler, software and hardware dispatch queues, and device drivers. An I/O operation would walk many steps through these software components before hitting the device hardware. The I/O stack software is constantly evolving to cope with advances in the storage I/O functionality and speed. For example, recent NVMe SSDs [26] boast up to 10× the I/O rate of still popular SATA 3.0 SSDs. A new breed of very low-latency NVMe SSDs are being introduced [27, 28]. To extract the best possible performance out of such devices, applications may bypass the OS kernel and directly access a storage device via a user-level driver [29].

primary storage (e.g.,main memory built with DRAM) and secondary storage (e.g.,byte-addressable PM storage media)

2) Persistent memory (PM) store: In this architecture, persistency is provided by byte-addressable PM storage media on the CPU's main memory bus (see Fig. 1(b)). Some amount of DRAM is provisioned to service memory accesses and to capture hot working and storage data. Fully realizing the benefits of a PM store may require a non-trivial overhaul of the storage system and data management software [30]. The least disruptive method of utilizing the PM store is to build a block storage abstraction on it, mimicking Fig. 1(a). In this case, applications are not required to change, yet they can tap fast and low latency block I/O.

primary storage的访问不经过软件，直接由硬件访问，快速；

secondary storage的访问不再需要经过软件栈，OS可同时对主存数据和设备数据进行管理，应用程序可直接将数据存入PM而无需事先缓存于主存DRAM,快速（计算机中内存地址是以字节为单位的，此结构二级存储PM也是字节寻址，数据组织方式一致，无需软件栈）

In a more integrated system, the OS no longer has to manage in-memory and on-disk data structures separately, and applications can store data directly on the PM store without buffering data first in DRAM. Thus, the I/O stack for locating, transferring and translating data between storage hierarchy levels are unneeded. Algorithms and system support to exploit in-memory persistence are being actively studied [31–35]. Despite several benefits, it is unlikely that the PM store will obviate the traditional block storage, because NAND flash media, the cheapest persistent solid-state memory, is continuously scaling in the vertical direction [36].

3) Heterogeneous memory: As mentioned, NAND flash SSDs are expected to stay for a long life. Other important system considerations are: (1) SSDs (in the standard 2.5-inch form factor) are field serviceable whereas DIMMs[1] are not; (2) NAND flash memory has much lower power consumption requirements on writes than the emerging persistent memory candidate [28], which is extremely important in handling large data sets. SSD是最佳媒介的原因：可现场维护、能耗低

Fig. 1(c) gives a practical architecture where a relatively small PM is provided on the memory bus and a large block storage offers the persistent storage capacity. In this case, the PM space can serve specific purposes, for example, to capture the most latency critical data from a transaction system. Indeed, the criticality of writes may vary considerably in terms of application performance [37]. Small frequent writes like journal and log commits have long been recognized as a perfect target to optimize with a small PM. In today's high-end storage and database platforms, a PM is often implemented with a battery-backed DRAM (or NVRAM).[1] Microsoft SQL Server 2016 takes this architecture to accelerate logging, and obtains a performance gain of 2× [20]. Other use cases of this architecture have been proposed and studied [38–42].

PM用于特定目的的加速

4) Hybrid store: The heterogeneous memory architecture with a limited amount of PM like NVRAM meets practical challenges. For one, an NVRAM consumes a DRAM DIMM slot, which is a scarce resource in large memory servers. To make matters worse, the number of DIMMs per channel decreases with the increase in DRAM clock speeds [43]. For another, an NVRAM is a complex device with a back-up memory (typically NAND flash), an external battery and supporting circuitry. As a result, NVRAM products are two to three times more expensive than equivalently sized DRAM.

(c)的缺点：占用稀缺的SIMM插槽；需要后备存储、外部电源和支持电路，设备复杂；昂贵。

For these reasons, we advocate the hybrid store architecture depicted in Fig. 1(d). Capability-wise, the hybrid store architecture is similar to the heterogeneous memory architecture. The byte-addressable PM capacity in the hybrid store architecture is provided by the SSD. Doing this is feasible with modern I/O interconnects like PCIe, as they can expose an address space that can be accessed with memory instructions. The hybrid store architecture offers an embedded PM store to the software and eliminates the need for data in PM to go through the thick I/O stack for persistence accordingly. Also, it saves a DRAM DIMM slot and is hot-pluggable—the total size of PM in the system can be increased in runtime in contrast with NVRAM. We will further delve into this architecture in Section III.

（MMIO）

数据为永久存储在SSD而经过IO软件栈会产生的开销。通过嵌入式PM，使存储于其中的数据能够避免此类开销。

B. I/O Interconnect with Byte Accessibility

In order to realize the hybrid store architecture, as mentioned, the underlying I/O interconnect must be able to service memory accesses to and from I/O devices. Examples meeting this requirement include PCIe [44] and CCIX [45]. The 2B-SSD architecture builds on PCIe because it is the most widely adopted I/O interconnect technology today. Note however that the architecture can easily be realized with other interconnects like CCIX. In what follows, we will summarize how a PCIe device is recognized in a typical platform.

PCI devices, including NVMe SSDs, have a "configuration space" that is comprised of a set of registers. Upon system reset, a PCIe device is in an inactive and uninitialized state. During the PCI enumeration stage after system reset, the BIOS and the OS discover all PCI devices installed, and initialize them by first reading their configuration space. Among configuration space registers, *base address register* (BAR) is how the device advertises the amount of memory it needs. A PCI device can have up to six 32-bit BARs.

通过PCI设备的配置空间初始化PCI设备

The BIOS or the OS reads each BAR of a given PCI device, and assigns a corresponding address window that

[1] The term NVRAM (non-volatile RAM) is sometimes used interchangeably with persistent memory or non-volatile memory.

[1] DIMM:双列直插式存储器模块[1][2][3]或双线存储器模块[4]（Dual In-line Memory Module，简称DIMM）是指一系列由动态随机存取存储器（DRAM）组成的模块。
DIMM通常是数颗至数十颗DRAM芯片焊接安装于一块已制作好电路的印刷电路板的形式，用于个人电脑、工作站、服务器。相比SIMM两边针脚相连在一起，DIMM两边针脚是独立的。

is allocated from the system memory map. Note that this address range consumes no physical memory on the host, but is for address decoding only. When MMIO [21] is used for this special memory range, a memory read/write to it will be sent to the device. The device is responsible for mapping internal resources, like registers and memory, to the host-visible memory ranges. For example, registers to control and operate an NVMe SSD are defined on the BAR0 address range [13].

只分配内存地址空间

实质就是MMIO若是针对于设备的，则设备执行这个操作，为操作提供所需资源

Applications may map such memory into their user space using the `mmap()` system call [46], and then use memory instructions to access the virtual memory address that will finally be translated to the physical memory space offered by a PCIe device. This method is the mechanism the 2B-SSD uses to open up a PM resource to the CPU.

如何实现MMIO？

一旦映射完成，用户空间的一段内存就与IO设备的内存关联起来

III. 2B-SSD: Design and Implementation

This section describes 2B-SSD at length. Given an I/O interconnect that allows byte accesses (PCIe in our case), a natural instantiation of 2B-SSD is to superimpose byte accessibility on an NVMe SSD. We first portray the basic architecture and explain its key components to show how this is done.

实例化 叠加 描摹

A. Architecture

In a NAND flash SSD, exposing memory interface is not straightforward. Because the underlying storage media does not allow byte-level read and write access to stored data, one cannot simply expose the media onto the memory address space. To overcome this limitation, we employ the internal DRAM as a buffer of data in the NAND flash media. 2B-SSD redirects memory accesses from the host CPU to the SSD-internal DRAM to realize byte-level access to a file. To this end, it implements an internal datapath between the DRAM and NAND flash memory.

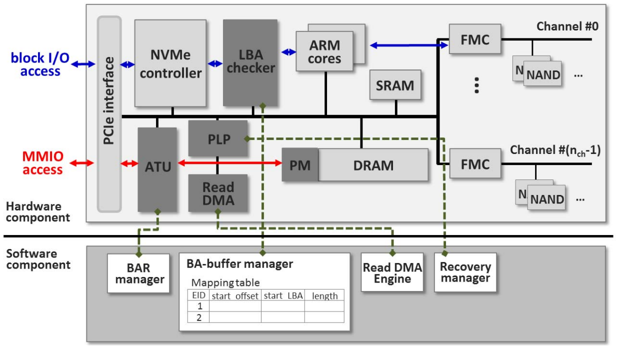
不能直接字节访问闪存阵列

把主机对闪存阵列的访问重定位到对SSD内部DRAM的访问，以实现对文件的字节级访问（2B-SSD中DRAM作为PM提供给主机）

Fig. 2 gives the overall architecture of 2B-SSD. It has four carefully co-designed hardware/software components: *BAR manager*, *BA-buffer*[2] *manager*, *read DMA engine*, and *recovery manager*. The BAR manager enables an additional BAR for byte granule file access and governs the redirection of memory accesses from the CPU into the DRAM. The BA-buffer implements a datapath between the DRAM and NAND flash memory, and provides control APIs to manage this DRAM buffer. The read DMA engine helps accelerate relatively slow memory read on the DRAM. The recovery manager is responsible for back-up and recovery of data in the 2B-SSD's DRAM. In what follows, we will describe in detail each of these components.

颗粒

1) BAR manager: There are two main goals to achieve with the BAR manager. First of all, it opens up a memory space visible to the CPU. In order to do so, it sets up BAR1, a second BAR. Typically, a PCIe device uses a single BAR (i.e., BAR0) for the purpose of device operation. BAR1 is introduced in 2B-SSD such that a separate memory space is prepared specifically for the CPU to communicate with 2B-SSD using memory access instructions.



EID	start offset	start LBA	length
1			
2			

Figure 2. 2B-SSD architecture.

enables an additional BAR (i.e. BAR1) for byte granule file access

Secondly, the BAR manager employs an *address translation unit* (ATU), a specially designed hardware structure to link the address space exposed to the CPU and a memory space in the 2B-SSD's internal DRAM. In essence, the ATU implements inbound and outbound address translation windows from/to the PCIe interface. The BAR manager leverages the ATU to redirect a memory read or write to BAR1's memory range into the DRAM. From the CPU's viewpoint, it has access to a DRAM capacity that is physically provided by 2B-SSD. We will explain in Section III-A4 how this DRAM capacity is presented to the CPU as a persistent memory across power outages.

即暴露给主机的闪存阵列地址空间

governs the redirection of memory accesses from the CPU into the DRAM

It is worth mentioning that the host system must treat the persistent memory of 2B-SSD carefully to obtain good write performance. We exploit the *write combining* (WC) mode of the underlying CPU [47]. By combining individual writes into a larger write burst in CPU's WC buffer (64 bytes in size in current x86 CPUs), it leads to a significant reduction of memory accesses. By reserving BAR1's address space for WC usage, our BAR manager enables a high-speed memory write to 2B-SSD.

向PM批量写数据以提高写性能

2) BA-buffer manager: The BA-buffer manager maintains a memory hierarchy of the DRAM and the NAND flash memory in 2B-SSD. The BA-buffer is the capacity we reserve in the DRAM to construct a persistent memory. BA-buffer resident pages are cached pages from the NAND flash memory. We implement the BA-buffer logic in software that runs on an ARM core within 2B-SSD. Primarily, this logic maps the BA-buffer onto a set of NAND pages and handles data transfer between the buffer and the NAND pages.

映射逻辑实在ARM核上实现的，即2B-SSD是在SSD嵌入式控制器上实现

The mapping information between DRAM addresses and NAND flash data is stored in a table. As shown in Fig. 2, each table entry includes (1) `entry_id`, (2) `start_offset` in the BA-buffer, (3) `start_LBA` of a given file, and (4) `length`. The operation of the BA-buffer logic is explicitly

表中的第（2）（3）项即建立了DRAM address (start_offset)
和NAND flash data (start_LBA)之间的映射

BAR manager BAR1
ATU
BA-buffer manager
的区别:(1)BAR manager 主要完成重定位（主机内存空间上的联系），为字节寻址做准备；
(2) BA-buffer manager主要完成设备端映射及数据拷贝。

[2] The term BA-buffer refers to byte-addressable buffer. Unless otherwise noted, we will use BA to mean byte-addressable hereafter.

(BA)

orchestrated by a set of host-side APIs, because the BA-buffer belongs to the host address space. For example, calling `BA_PIN()` would establish a mapping between a DRAM address (`start_offset`) and NAND flash data (`start_LBA`), copy the NAND flash data (of `length`) to the DRAM area, and store the mapping information in the mapping table. Because the host CPU's memory references to the BA-buffer (handled by hardware) are much more frequent than BA-buffer management events (handled by software), the software overheads are insignificant. Let us further describe the BA-buffer management APIs in Section III-C.

精心策划

API管理BA-buffer映射逻辑（映射的建立&数据由闪存阵列到DRAM的拷贝等）、应用BA-buffer上存储空间的分配、应用对BA-buffer上文件的读写等。

The BA-buffer management APIs are designed to enable applications to allocate memory on the BA-buffer, and read and write files using them. Upon an application's memory read or write request, the BA-buffer manager loads data from NAND pages to the BA-buffer or flushes buffered data to the corresponding NAND pages. As such, any NAND page on 2B-SSD can be accessed by two totally separated datapaths: Memory interface (BA-buffer) and conventional (NVMe) block I/O. Note that accessing the same file with two independent paths, unless properly controlled, may result in data inconsistency in persisted data. To prevent inadvertent data updates and ensure consistency in persistent data, the BA-buffer manager gates block I/O requests to NAND pages that are currently mapped to the BA-buffer for access through memory interface. We design and leverage a hardware logic named "LBA checker" to snoop on every I/O request's LBA to see if the corresponding NAND pages need be protected.

闪存阵列到SSD上DRAM之间的映射关系一旦建立，且使用API进行数据拷贝后，闪存阵列和DRAM上均有应用所需数据，则有两种访问方法/数据路径：
Memory interface
(BA-buffer)
和
conventional (NVMe) block I/O

但多种访问方会导致数据不一致的问题，为避免此问题，2B-SSD将阻止对于已经映射到BA-buffer的闪存阵列页面的块IO请求。图2中LBA checker检查每个IO请求的LBA以筛查各闪存阵列页面以找出上述情况进行块IO阻止操作。
（之后只需使用后文中BA_FLUSH（EID）这一API同步数据即可

3) Read DMA engine: As explained in Section III-A1, the BA-buffer achieves high write bandwidth by leveraging write combining of the underlying CPU [47]. In contrast, reading data from the BA-buffer with memory interface is slower than reading with block I/O commands. For example, reading out 4 KB in the BA-buffer takes an order of magnitude longer than an equivalent block read (150 *us* vs. 13.2 *us* as shown in Figure 7 (a)). This is because the BA-buffer is an "uncacheable" region, so the reading from the BA-buffer is split into 8-byte read PCIe transactions [48].

字节寻址无读缓存机制且读单元比块IO小，则读延迟比块IO大

In order to accelerate memory reads, 2B-SSD takes advantage of a DMA facility called *read DMA engine* (Fig. 2). This engine can offload expensive memory operations such as copying of large data from 2B-SSD to a host-designated destination. Performance gains come in two aspects, by reducing (in-SSD) CPU cycles needed to transfer data and by leveraging the concurrency offered by the dedicated, independent DMA facility. We provide an intuitive API to utilize and integrate the read DMA engine in user applications. Experimental results in Section V-B indicate that a read operation on 2 KB or larger data will benefit significantly from using the read DMA engine.

即对大量数据的读操作

to Read DMA engine

因拷贝数据是通过DMA传输的

对于某些数据量大的读操作，Read DMA engine可将其卸载，直接通过DMA处理（对应后文中API：BA_READ_DMA(EID...))

4) Recovery manager: The user data content in the BA-buffer must appear persistent across power outages. However, in the physical sense, the BA-buffer capacity is drawn from volatile DRAM. We design a *recovery manager* to turn the BA-buffer into a persistent memory.

Modern NVMe SSDs have already their power loss protection subsystem including power loss detection circuitry and back-up power capacitors [27, 49]. Our recovery manager leverages this subsystem in cost-effective manner. First of all, we add a little more capacitance into the power supply subsystem of 2B-SSD (see Fig. 6). The provided back-up power capacity is sufficient to securely save BA-buffer contents and all related information (e.g., the BA-buffer mapping table) in a reserved area of the NAND flash memory before 2B-SSD turns completely off. Secondly, we write a software recovery logic that runs data protection procedures launched by power loss detection circuitry. When power comes back, it restores both BA-buffer contents and management information from the saved area.

电容

在SSD原有数据恢复系统上（1）添加一个额外的小电容确保掉电时BA-buffer内容和相关信息（映射表）能被刷新到闪存阵列的保留区域内（DRAM是易失的，但闪存阵列是非易失的）（2）添加用于恢复BA-buffer内容和管理信息的软件恢复逻辑。

B. Durability Guarantee

Memory writes to 2B-SSD become durable once they are stored in the BA-buffer. However, in WC mode, they are not immediately flushed to the BA-buffer. Instead, they are first cached in the WC buffer of the underlying CPU, sent to the PCIe root complex, and finally stored in the BA-buffer. The problem is, during this write path, they are arbitrarily reordered [41] and cached in the WC buffer for a period of time. The latter is particularly critical because data in this CPU's buffer will be lost at the time of power outage. To guarantee durability as well as to enforce the ordering of these memory writes, we introduce two types of flush and memory barrier support as depicted in Fig. 3.

根复合体

为在WC mode下保证WC buffer中数据的持久性和有序性而采取的措施

- **WC buffer → Root complex:** clflush (cache line flush) followed by mfence (memory fence) [50] is used to flush data in the WC buffer to the root complex in order. clflush is an instruction to force a cache line of a relevant address to be flushed, and mfence places an order with respect to all load and store instructions. clflush is only ordered by mfence [51], so the former should be followed by the latter. Applications that access 2B-SSD through memory interface can use this pair of instructions to ensure that prior stores are included in the writeback to the root complex.
- **Root complex → BA-buffer:** We define and exploit an operation that we call "write-verify read" to enforce write ordering and flushing at the PCIe root complex. In a PCIe system, the root complex connects CPUs and the memory subsystem to the PCIe switch fabric where one or more PCIe devices are attached [44]. The problem is, PCIe memory writes (e.g., memory writes to 2B-SSD) become "posted" transactions that do not wait for a completion response. In our design, we realize the write-verify read operation with a memory read of zero byte from the BA-buffer. This works because the read and write transactions are sequentialized at the root

交换设备

发布型

顺序的

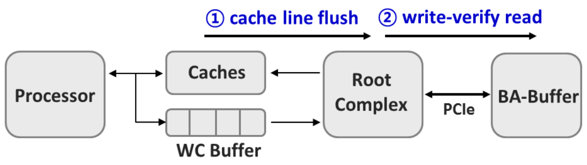


Figure 3. Two steps for ensuring ordering and durability of writes.

complex, and read access is a “non-posted” transaction, which ensures that all prior writes are committed.

C. 2B-SSD API

We provide the following APIs to make the BA-buffer programming straightforward for application developers. Currently, the functionalities of the APIs are defined in the context of our 2B-SSD, but not as of yet defined in the NVMe standard specification. In our current design, we exploit the Linux ioctl infrastructure to implement the APIs such that the 2B-SSD APIs can pass through the file system.

- **BA_PIN(EID, offset, LBA, length):** This API brings pages in NAND flash memory to the BA-buffer, pins them, and then adds a new entry to the BA-buffer mapping table. The requested pages range from `LBA` to `LBA + length`, and can be one or multiple 4 KB pages. On receiving this request, the BA-buffer manager first checks whether the specified range overlaps with the BA-buffer cached data by looking up the mapping table. If not, it reads data from NAND flash memory into the given `offset` in the BA-buffer. Finally, it adds the `EID`th entry with mapping of `LBA` onto `offset` to the mapping table. Only applications with permission to access the requested LBA range are allowed to use this API. Otherwise, the OS will block the attempt of applications.
- **BA_FLUSH(EID):** Given NAND flash memory whose LBA range is referred to by the `EID`th mapping entry, this API synchronizes their state in the BA-buffer with NAND flash memory. That is, it transfers all their contents in the BA-buffer to their corresponding NAND pages. We perform this mechanism because 2B-SSD APIs enable a CPU’s direct access to the BA-buffer, and 2B-SSD cannot determine which data are dirty or not. On receiving this request, 2B-SSD stores all the BA-buffer’s contents referred to by the given mapping entry to NAND flash memory. If the operation is successful, the BA-buffer manager deletes the `EID` entry from the mapping table.

（保持一致性）

对BA-buffer中数据的操作被
加速实现后，此API将EID下
数据段刷新传输回闪存阵列，
此时BA-buffer中不再保有此
数据段，可删去该映射←以此
保证数据一致性

- **BA_SYNC(EID):** This API aims at ensuring persistence of contents in the BA-buffer. Unlike the above two APIs, it requires no read/write from/to NAND flash memory and causes no change in the mapping table accordingly. This API performs three sub-functions in sequence: (1) Determining which pages in the BA-buffer are referred to by the given `EID`th mapping entry, (2) issuing clflush followed by mfence for the specified BA-buffer pages, and (3) sending a write-verify read command to 2B-SSD.

再次重复图3中的
写操作以确保该EID
对应BA-buffer处仍
有写入数据（即确保
数据持久性）

- **BA_GET_ENTRY_INFO(EID):** Given an `EID`, this API shows the details of the `EID`th entry from the mapping table. It is used by BA_SYNC(`EID`) when determining the BA-buffer pages pinned by the `EID`th mapping. The host application can also use this API to obtain corresponding mapping information.
- **BA_READ_DMA(EID, dst, length):** Given contents in the BA-buffer referred to by the `EID`th mapping entry, this API exploits the 2B-SSD’s read DMA engine and programs it to copy the contents up to `length` bytes into the range beginning at `dst`. The host finally receives an interruption from 2B-SSD when the operation is done.

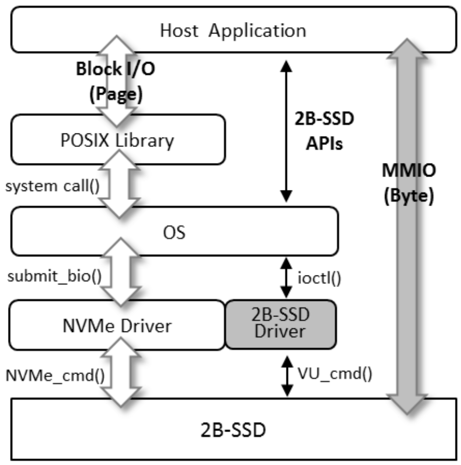


Figure 4. Two access paths of software stacks for 2B-SSD.

Finally, Fig. 4 shows the entire software stack including a device driver developed for 2B-SSD. This device driver gives features needed to initialize 2B-SSD. We use ioctl() and SSD-specific vendor unique command facilities to pass our APIs, so there is no change in the traditional block I/O path (the left side of Fig. 4). Host applications can access the BA-buffer through MMIO (the right side of Fig. 4) once all initialization has been taken care of with the 2B-SSD APIs.

IV. Case Study: Database Logging

As our case study, we focus on SQL and NoSQL database systems because their log commits are notorious for small frequent writes. This section describes how we revise database logging mechanism to exploit 2B-SSD’s memory interface. 2B-SSD is also a good fit for file system journaling and line-of-business applications where critical small writes harm application performance.

臭名昭著

A. 2B-SSD's Impact on Logging

WAL is a standard technique used by database systems to enforce atomicity and durability of transactions [22]. When using WAL, all database modifications should be written to log files before they are applied. The problem is that writing WAL logs causes small frequent writes that imply significant deterioration in performance. First, even though WAL logs are usually much smaller than a single page (typically 4 KB in size), the I/O size should be aligned with the page size. Moreover, log writes are usually followed by fsync() function to guarantee immediate persistence. This function incurs writing all modified data in device cache to NAND pages. Some data in the device cache may be irrelevant to database systems, but they should wait until SSD has written all cached data.

2B-SSD tackles this significant burden of log commits using its memory interface, whose benefits are as follows:

Byte-level read/write. 2B-SSD enables database systems to write data in a byte granularity to its NVRAM, which noticeably reduces the write burden in the conventional SSDs. Absorbing small frequent writes using an incorporated memory is particularly effective for NAND solutions where NAND write latencies may be much longer and even extremely vary compared to read latencies because of their read/write asymmetries [52].

Low overhead in making transactions persistent. Conventional SSDs should bear the heavy burden of programming or erasing NAND pages for writes, whereas 2B-SSD need not carry it for data persistence. Once written to the BA-buffer, WAL logs become persistent under guarantee of capacitors. Only CPU cache flushing followed by *write-verify read* is required for their persistence. Compared to the high NAND manipulation overhead, the overhead by this pair of instructions is negligible.

Reduced write amplification factor (WAF). WAL logs are sequential records that that are completely ordered by time and appended to log files. As mentioned in this section, a WAL log is commonly smaller than a log page, and database systems do not create a log page in the presence of a partially filled one. Thus, the same log page may be written to storage multiple times. In contrast, the target, where a log page is written, is changed from NAND pages to the BA-buffer in our case. Also, multiple pages filled with WAL logs are flushed from the BA-buffer to NAND pages at once. A single NAND write per log page reduces WAF [53] and optimizes both tail latencies and SSD lifespan.

B. Implementation of WAL for 2B-SSD

To demonstrate the effectiveness of 2B-SSD described in Section IV-A, we make a case for WAL for 2B-SSD (*BA-WAL*). It is a new WAL scheme designed to take advantage of the 2B-SSD's byte-accessibility. Unlike the conventional WAL scheme, BA-WAL stores WAL logs directly on 2B-SSD without buffering them in the host memory. 2B-SSD provides fast persistence of write data and easily makes log commits off the performance-critical path, which significantly improves the response time and throughput in database transaction processing.

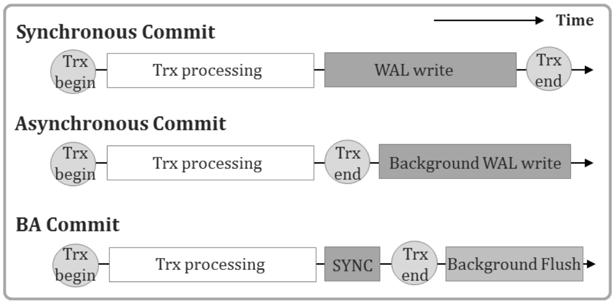


Figure 5. Possible transaction commit modes in 2B-SSD.

Fig. 5 compares commit modes to complete transactions. The conventional WAL scheme supports *synchronous* and *asynchronous* commit modes [54], and our BA-WAL scheme offers *BA commit* mode. Synchronous commit ensures data persistence at the cost of performance. In this mode, database systems should wait for WAL logs to be flushed to the permanent storage. In contrast, asynchronous commit emphasizes performance over the risk of data loss and enables a time window between a transaction completion and the time that WAL logs are persistent.

Different from the two above, BA commit comprises three phases; logging, commit, and flushing. In the logging phase, database systems append WAL logs to the BA-buffer. The point is, logs are written as much as exactly necessary. During the commit phase, they mark the commit point of a transaction on the BA-buffer and invoke BA_SYNC(). By ensuring that WAL logs in the BA-buffer become durable, this phase completes the given transaction. Finally, in the flushing phase, a group of WAL logs from multiple transactions is transferred from the BA-buffer to the pinned NAND pages by a single BA_FLUSH() call. BA commit has the best of the conventional commit modes; it not only achieves throughput as high as performance-oriented asynchronous commit, but also ensures durability as strongly as synchronous commit. The fast persistence of 2B-SSD mainly contributes to this improvement. WAL logs become durable as soon as they are written to the BA-buffer so that database systems can complete a transaction after the transaction is truly committed by persistent WAL logs.

A detailed example of implementing BA commit in database engines is as follows. To hide flushing latencies, we adopt "double buffering technique" so that logging and flushing phases can proceed in parallel. WAL logs are appended to one half of the BA-buffer, while logs in the other half are flushed to the file. After flushing, logging subsystem re-pins the half so that the BA-buffer can be used for accepting new logs.

BA-WAL for PostgreSQL. PostgreSQL [55] is an open-source, standard compliant relational database system. It includes a logging subsystem called *XLOG*, where log files are stored as a set of segment files, each of which is normally 16 MB in size. We selected a notably recent version of PostgreSQL (9.6.0), and revised XLOG to directly use the 2B-SSD APIs. Specifically, we (1) set the size of a log segment file to half of the BA-buffer size for double buffering, (2) mmap the BA-buffer into the user address space of PostgreSQL and allocate its log buffer to the BA-buffer, (3) disable block I/O writes to a log segment file, and (4) execute pinning and flushing (i.e., BA_PIN() and BA_FLUSH()) in parallel during transactions.

BA-WAL for RocksDB. RocksDB [56] is an embedded persistent key-value store developed by Facebook. Its three basic constructs are a memory-resident *memtable*, an *SST file* (Sorted String Table), and a log file. The newly inserted data is stored to both the memtable (to be flushed to SST files when it becomes full) and the corresponding log file. Upon a crash, it can recover as much as the memtable state that is reflected to its log file. RocksDB builds up a maximum of two memtables and log files—one is active for storing new data while another is already full and flushed out to the storage device. Similar to PostgreSQL, the 2B-SSD APIs are used to implement BA-WAL for RocksDB and double buffering is applied. We selected version 5.1.4 of RocksDB and overrode its WAL-related `WritableFile` class. The constructor of overridden class now creates a new log file and pins their LBAs to the BA-buffer. Each log file is set to a quarter of the size of the BA-buffer because a half of the BA-buffer should be reserved for double buffering. Its *append* function is also rewritten to append WAL logs to the BA-buffer using `memcpy()` instead of block I/O.

BA-WAL for Redis. Redis [57] is an open-source, in-memory key-value store, which is used as persistent database or cache. It uses a single-threaded design and sequentially serves requests. As the main persistence option, it uses an *append-only file* (AOF) that logs every single write operation that the system receives. We selected version 3.2.4 of Redis and revised the AOF subsystem using the 2B-SSD APIs. Specifically, we (1) set the size of AOF to be identical to the BA-buffer size, (2) rewrite *append* function to append WAL logs to the BA-buffer using `memcpy()`, and (3) execute pinning and flushing during transactions. We did not apply double buffering in this case to avoid violating the single threaded design concept of Redis.

V. Evaluation

This section empirically illustrates the potential benefits of 2B-SSD using performance-critical workloads. The results are presented in two parts, basic performance and application level performance. The evaluations reveal that memory interface of 2B-SSD makes small frequent writes off the critical path and significantly improves the performance in SQL/NoSQL systems.

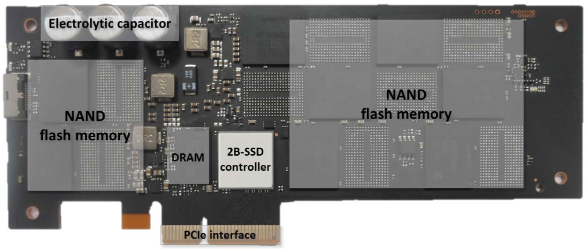


Figure 6. 2B-SSD hardware. Only one side is shown.

Table I
2B-SSD SPECIFICATION.

Item	Description
Host interface	PCIe Gen.3 ×4
Protocol	NVMe 1.2
Capacity	800 GB
SSD architecture	Multiple channels/ways/cores
Storage medium	Single-bit NAND flash
Capacitance of electrolytic capacitors	270 μF ×3
BA-buffer size	8 MB
Max. entries of BA-buffer	8

A. Experimental Setup

We conducted experiments on a system comprised of a Dell PowerEdge R730 server. The server is equipped with two Intel Xeon(R) CPU E5-2699 @2.30GHz (18 threads per socket) and 256 GB DRAM. 64-bit Ubuntu 14.04 with kernel 4.6.3 is chosen as the OS system. We developed a prototype of 2B-SSD based on the state-of-the-art NVMe SSD [58] depicted in Fig. 6. The details of our SSD are listed in Table I. Note that the performance of MMIO on 2B-SSD is constant regardless of the performance of SSD on which it piggybacks, because 2B-SSD's memory interface relies solely on byte-addressability support of PCI interface. For performance comparison, PM963 [49] and Z-SSD [27] were used, which are a latest datacenter-class SSD (DC-SSD) and an ultra-low latency SSD (ULL-SSD), respectively.

请注意，MMIO在2B-SSD
上的性能是恒定的，而
不管其搭载的SSD的性能
如何，因为2B-SSD的内
存接口仅依赖于PCI接口
的字节寻址能力支持。

B. Basic Performance Results

As a basic performance measure, we want to compare throughput as well as latency of three SSDs. We examine performance of MMIO on 2B-SSD and those of block I/O on the other two. First, we measure read/write latencies as a function of request size for both types of I/Os. For MMIO, we use `memcpy()` standard library, where memory read/write latencies indicate time taken to complete a single request. For block I/O, we used Linux `FIO`. In the case of 2B-SSD's latencies, we additionally observe memory reads accelerated by BA_READ_DMA() and memory writes

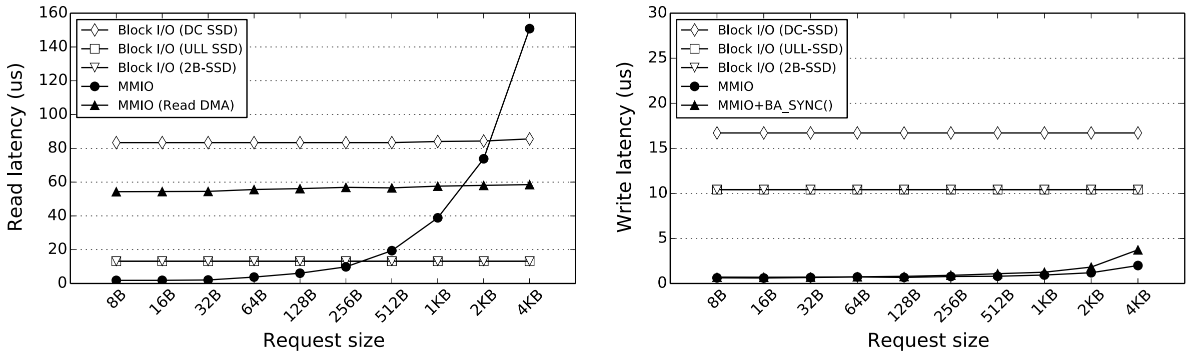


Figure 7. Read/write latencies of block I/O and MMIO: (a) Read latency. (b) Write latency.

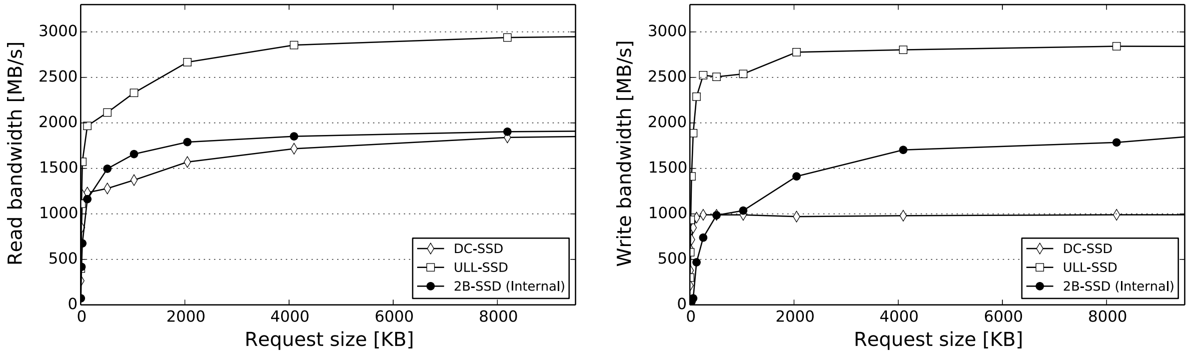


Figure 8. Bandwidths of block I/O and internal I/O (in case of 2B-SSD): (a) Read bandwidth. (b) Write bandwidth.

including BA_SYNC() overhead. Secondly, we measure bandwidths of all target SSDs as a function of request size. In the case of 2B-SSD, read/write bandwidths from its internal datapath (i.e., the path between the BA-buffer and NAND pages) are observed. To get these internal bandwidths, we measure the time elapsed during BA_PIN() for read and BA_FLUSH() for write. Similar to latencies, `FIO` is used for measuring throughputs of DC-SSD and ULL-SSD.

Read/write latencies. In this experiment, we measure read-/write latencies by varying request size from 8 bytes to 4 KB. For DC-SSD and ULL-SSD, we examine block read/write latencies at queue depth of one. For 2B-SSD, we also observe memory read/write latencies to store data on the BA-buffer. Note that these latencies indicate the time taken for memory operations and do not include required time to transfer data through the internal datapath.

Fig. 7(a) plots the read latencies. In the case of block reads, both ULL-SSD and 2B-SSD show 6.3× shorter latencies than DC-SSD. ULL-SSD offers low latencies as its name implies; unsurprising, 2B-SSD has the exactly identical block read latencies to ULL-SSD on which it piggybacks. In the case of memory reads, 2B-SSD considerably shortens the read latency when the read request size is below 256 bytes—there are differences of $3 \sim 11$ μs between ULL-SSD and MMIO. The read latency of MMIO increases in proportion to the read request size and it consumes much more time than ULL-SSD and DC-SSD at a read request size of approximately 350 bytes and 2 KB, respectively. This is because MMIO read is a non-posted transaction that requires a read completion to indicate success or failure of the transaction [44]. Even worse, it is split into small-size (up to 8 bytes on current x86 CPUs) read transactions to guarantee atomicity. Thus, 2B-SSD provides the read DMA engine for MMIO. The read DMA engine accelerates the latency of MMIO read by 2.6× at 4 KB and shows the latency of approximately 58 μs. This value is 40% shorter than that of DC-SSD. However, the read DMA engine still consumes 4.7× more time than ULL-SSD because ULL-SSD performs fully hardware-automated read operation for 4 KB [58].

Fig. 7(b) plots the measured write latencies. As shown, write operations are faster than read operations—ULL-SSD and 2B-SSD take 10 μs whereas DC-SSD takes 17 μs. To significantly reduce the write latency, latest SSDs including ULL-SSD and DC-SSD use persistent write buffer for their datapath. The write operations are completed when the data is placed in the write buffer instead of reaching stable NAND pages, which requires time-consuming NAND program operation that costs hundreds of μs. MMIO has

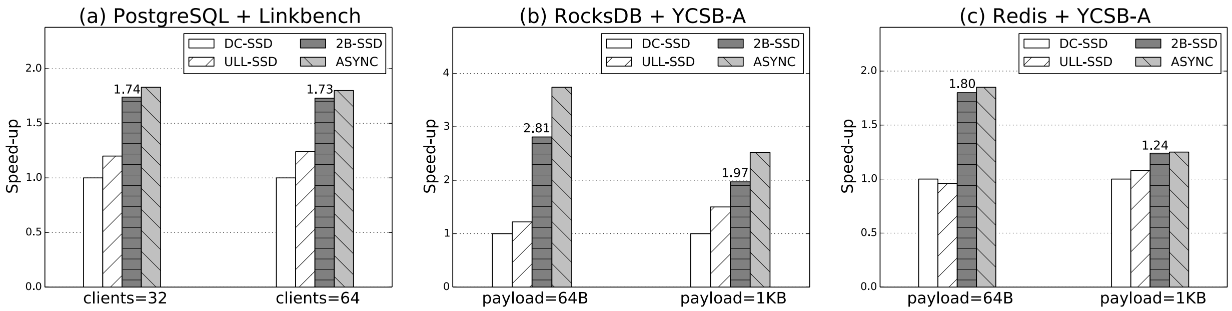


Figure 9. Application-level results.

16.6× shorter latency than modern SSDs—8-byte MMIO write only consumes 630 *ns*. When the write request size increases, the latency of MMIO also increases from 630 *ns* to 2 μs. Even so, remarkable latency differences remain between MMIO and block I/O of latest SSDs because MMIO write is a posted transaction that does not wait for completion response of PCIe bus transaction [44]. Furthermore, our BA-buffer manager chooses WC usage for MMIO as mentioned in III-A1, and individual MMIO can be combined into larger write transaction (up to 64 bytes) in the CPU's WC buffer, which enables one to significantly reduce the number of transactions. Compared to MMIO, the latency of persistent MMIO includes an additional latency for BA_SYNC(). When the write request size is small, this burden is negligible and results in approximately 15% longer latency. When the write request size increases, the latency also increases up to 47% at 4 KB. However, persistent MMIO still takes 6 μs shorter latency than ULL-SSD. As later shown, the remarkable MMIO write of 2B-SSD mitigates the performance drop incurred by small frequent updates of both SQL/NoSQL systems.

Read/write bandwidth. In this experiment, we measure bandwidths of read/write operations at queue depth of one by varying the request size from 4 KB to 16 MB. Again, it is internal read/write in the case of 2B-SSD—there is no need to transmit data from 2B-SSD to the host over a host interface. The results do not include the legacy block I/O of 2B-SSD since it is equal to that of ULL-SSD.

Fig. 8 plots the measured bandwidths. ULL-SSD shows superior bandwidths for both read and write and achieves maximum bandwidth limited by the host interface (at around 3.2 GB/s with PCIe Gen.3 ×4) despite the queue depth of one. The internal bandwidth of 2B-SSD lies between ULL-SSD and DC-SSD; it is lower than ULL-SSD by about 1 GB/s at a request size of at least 4 MB even if it does not include data transfer between SSD and the host. This is because the hardware-automated datapath of ULL-SSD is designed and optimized to move data from/to outside of the device, and thus 2B-SSD does not exploit it fully. Consequently, the software firmware that runs on ARM cores is mainly involved in the internal datapath of 2B-SSD, similar to the case of DC-SSD. The bandwidth of 2B-SSD outperforms DC-SSD by about 700 MB/s at a request size of at least 4 MB for the write. In terms of read bandwidth, when the read request size increases, their performance gap is considerably decreased. Since latest SSDs use a heuristic read-ahead mechanism to accelerate sequential read operation [59], it unfortunately cannot be adopted to 2B-SSD because the BA-buffer is for MMIO and thus its content should be explicitly controlled by the host. However, this bandwidth still contributes remarkable performance improvement in application-level results, as shown in the next section.

C. Application level results

This section presents experimental results to illustrate benefits of 2B-SSD in actual database engines. We chose PostgreSQL-9.6.0 (relational DBMS) [55], Redis-3.2.4 (in-memory key-value engine) [57], and RocksDB-5.1.4 (persistent key-value store) [56]. As mentioned in Section IV-B, in order to elevate byte-accessibility of 2B-SSD, we rewrote portions of transactional logging system in PostgreSQL, Redis, and RocksDB to directly store WAL logs on the BA-buffer via MMIO. This allows database engines to reduce the overhead of transaction commit tremendously (up to 26×) compared to the conventional logging system using legacy block I/O; thus the overall query performance is improved. Note that amount of code changes for BA-WAL is negligible: Fewer than 100 lines for Redis, and fewer than 200 lines for PostgreSQL and RocksDB.

We ran Linkbench [23] for PostgreSQL and Yahoo! Cloud Serving Benchmark (YCSB) [24] for RocksDB and Redis. Linkbench is a SQL database benchmark for a large-scale social graph. Specifically, it reflects Facebook's social graph workloads where most read requests are served by a caching layer. It is read intensive with about 30% writes. YCSB is a key-value benchmark for representing cloud workloads inside Yahoo!. Workload-A is chosen, since it is write-heavy workload mix of 50% reads and 50% updates.

Comparison to traditional architecture. In this experiments, we first measure the performance of traditional architecture and the proposed hybrid store architecture in

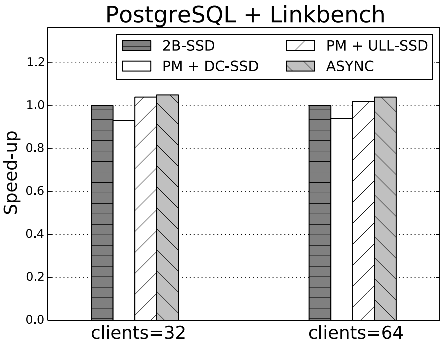


Figure 10. Performance in the heterogenous memory and hybrid store architectures.

Figs. 1(a) and (d). To fully focus on the effect of logging system on database engines, we assumed that all user data fits in DRAM, and only WAL logs are written to a log device. As a log device, we used DC-SSD, ULL-SSD, and 2B-SSD, respectively. Again, we adopted BA-WAL for 2B-SSD as mentioned in Section IV-B. The results of 2B-SSD with block I/O are excluded since they are equal to those of ULL-SSD. We also included the results of *asynchronous commit* as a theoretical maximum achievable performance, which is shown next to ASYNC results in Fig. 9. Although asynchronous commit enables transactions to complete more quickly, there is a time window between the transaction completion and the time that WAL logs are written to the persistent log device. If the database crashes during this risk window, changes made during that transaction will be lost.

Fig. 9 gives the results. ULL-SSD (and 2B-SSD with block I/O) performs better than DC-SSD because of its lower write latency. As shown in Section V-B, ULL-SSD has 70% lower write latency than DC-SSD, so the maximum improvement of ULL-SSD reaches 1.5× in RocksDB on 1 KB of payload. Redis, which is an in-memory and single-threaded key-value engine, does not enjoy this write latency and shows similar performance of ULL-SSD and DC-SSD. In contrast, all SQL/NoSQL systems take full advantage of our BA-WAL and 2B-SSD achieves $1.2 \sim 2.8\times$ and $1.15 \sim 2.3\times$ speed-up compared to DC-SSD and ULL-SSD, respectively. For RocksDB and Redis, the performance gain highly depends on the payload size, which presents the write request size to the log device for every key-value pair insertion. In the conventional logging system, an entire log page write (typically 4 KB in size) occurs regardless of the payload size, whereas in BA-WAL, the system can write as much as necessary. Because the payload size is decreased, the difference of written WAL logs between two systems increases; thus the performance gap increases. It is important to point out that 2B-SSD achieves $75 \sim 95\%$ from ASYNC without the risk of data loss. This mainly comes from both byte-accessibility and low overhead of 2B-SSD in making transactions persistent.

Comparison to heterogeneous memory architecture. In this experiment, we compare performance in heterogeneous memory architecture (Fig. 1(c)) and hybrid store architecture (Fig. 1(d)). As mentioned in Section II, the former architecture comprises a small PM and a large block device. To instantiate it, we use emulated DRAM and two NVMe SSDs: DC-SSD and ULL-SSD, which were used in previous experiments. To take advantage of this architecture, WAL should be rewritten so that WAL log writes to the PM buffers are written to the log device through the I/O stack in a lazy manner. As far as we know, only PostgreSQL has reference code for this revised WAL scheme [60]. In order to eliminate any bias in implementation and evaluation, we presented PostgreSQL result only. We examined its Linkbench throughput in both architectures.

Fig. 10 compares throughputs of four configurations. Here, baseline (*2B-SSD*) is the performance in the hybrid store architecture using 2B-SSD. *PM + ULL-SSD* and *PM + DC-SSD* denote performance evaluated on the heterogeneous architecture using ULL-SSD and DC-SSD as a log device, respectively. Lastly, *ASYNC* is theoretical performance of the performance-oriented asynchronous commit mode. The normalized throughput of baseline is compared with the other configurations. As shown in this figure, baseline, PM + DC-SSD, and PM + ULL-SSD report almost identical performance, which appears notably similar to that of ASYNC. Compared to baseline, PM + DC-SSD and PM + ULL-SSD give approximately 0.6% lower and 0.4% higher throughputs, respectively. The performance gap between the two above originates from the overhead of flushing WAL logs in the PM to the log device—ULL-SSD offers 70% lower write latency than DC-SSD as shown in Fig. 7(b). Deploying emulated DRAM and state-of-the-art NVMe SSD as the PM and block storage is ideal. Indeed, not only emerging PM can hardly offer DRAM latencies, but also latest NVMe SSDs are too expensive to use as a log device. Thus, real heterogeneous memory architecture can hardly achieve comparable performance to baseline and ASYNC.

VI. Discussions

Based on our results and experiences, we make several observations, findings, and remarks.

Costs for enabling memory interface. 2B-SSD requires an incorporated DRAM and capacitors for the device-level NVRAM. Currently, 2B-SSD exploits a portion of the SSD-internal DRAM that is not used by vanilla firmware of the piggybacked SSD, and 8 MB of NVRAM can be provided at a small additional cost for back-up power capacitors. Even with this small NVRAM, by fully exploiting internal datapath between the NVRAM and NAND flash memory, i.e., double buffering mechanism, we can achieve significant performance improvement. As shown in Fig. 8, the maximum internal bandwidth between the NVRAM and NAND flash memory is achieved when the NVRAM size is about 8 MB. Larger NVRAM capacity can provide more usability and reduce BA_FLUSH() overhead, but we do not expect better performance. We also managed to keep 2B-SSD's footprint as small as possible; the additional binary size for the internal datapath is only 4.09 KB. With the minimum use of in-device resources, we guarantee that block I/O has no performance degradation even when memory interface is enabled.

Is 2B-SSD for all? Not all applications would benefit from 2B-SSD's memory interface but they should be satisfied with some conditions. First, their data must require sufficiently small byte-granular write as well as bulk read. This type of data is everywhere (e.g., log, journal, and tiny data from IoT devices) and continuously cumulated in their systems. In many cases, these data are stored in real-time and read by batch processing only if required. We confirmed that smaller write size shows better performance gain than large one did. Secondly, they must be latency critical rather than bandwidth critical. If a bandwidth-intensive application exploits 2B-SSD's memory interface, the bandwidth can be monopolized by the internal datapath so that other applications accessing with block I/O would not be able to get it enough.

垄断

Furthermore, we saw a chance to apply 2B-SSD to the workload of bulk write as well as small size of read, the opposite case of the above. The powerful bandwidth of block I/O is the most perfect way to write a bulk data and, with preloading (pinning) from NAND flash memory to the NVRAM of 2B-SSD, the read latency can be superb as shown in Fig. 7(a). Applications need not read the whole page to get only several bytes so that they can save DRAM or other resources.

Unified data structure between byte and block. In 2B-SSD, the data on the BA-buffer is transferred to NAND flash memory in format originally written. Typically, WAL logs and journals can be written without re-formatting, only need building pages. However, user data requires a sort of transformation between the form of byte and page when erasure coding or loading occurred. This is a clear obstacle to apply 2B-SSD generally. For this, in-storage computing [61] should be supported. However, in the era of PM store as depicted in Fig. 1(b), we expect that this heterogeneity of block and byte would be diminished consequently.

初步理解：
1.日志通过字节寻址传输至设备端，则具有字节的数据组织形式，则PM与闪存阵列间数据传输无需格式转换；
2.用户数据通过传输块O传输至设备端（因用户数据大多不具备频繁小写的特征），不具有字节的数据组织形式，则当某些用户数据需要通过PM加速时，PM与闪存阵列间数据传输需格式转换。

VII. Related Work

It is well known that techniques to ensure data persistency, such as write-ahead logging, journaling [62] and shadow paging [63] incur performance-critical writes. Academia and industry have long recognized their overheads and attempted to satisfy their urgency with battery-backed NVRAM.

Some recent prior works exploit emerging PM like phase change memory, and propose database engines that buffer WAL logs in the PM directly instead of in DRAM [41, 60, 64, 65]. A heterogeneous memory architecture (like Fig. 1(c)) is commonly assumed. Note however, as described in Section II, log writes to the PM buffers are temporary, and should be periodically written through the thick I/O software stack to a permanent storage. In contrast, 2B-SSD promotes a byte-addressable storage of Fig. 1(d) and stores WAL logs directly and permanently without any intervention of I/O stack.

There are two pieces of recent work comparable to ours in that they incorporate NVRAM in an SSD. First, Kang et al. [66] build *DuraSSD*, an SSD equipped with a back-up capacitor like ours. They revise MySQL to buffer modified data pages in the embedded DRAM rather than in NAND flash memory. This releases the database system from the burden of frequent `fsync()` calls for data durability guarantee. While DuraSSD exposes NVRAM area to applications, it is fundamentally different from 2B-SSD, because the DuraSSD's NVRAM can only be accessed through the block I/O stack. In contrast, 2B-SSD allows applications to directly write to, with memory instructions, its NVRAM.

Secondly, there is a standardization effort in the NVMe SSD community to define a "Persistent Memory Region (PMR)" feature [67]. Like 2B-SSD, PMR exposes a portion of SSD-internal NVRAM to be accessed in byte granularity. However, there is a crucial difference between these two. A PMR-enabled NVMe SSD does not offer interchangeable byte and block access interface. It features no internal data mapping and transfer path between its NVRAM and NAND flash memory. For this reason, data transfer between them should go through the host I/O stack. In contrast, 2B-SSD maps its NVRAM to NAND flash memory and efficiently transfers data between them via an internal datapath.

VIII. Conclusions

This paper described the motivation, design, and implementation of a byte- and block-addressable solid-state drive (2B-SSD). We also detailed its novel hardware-software interface. To date, this work is the first proposal and full implementation that exposes interchangeable byte- and block-level access interfaces out of an SSD. Through its APIs, applications can write and read any number of bytes on 2B-SSD without forcing the data being buffered in the host memory. Our design achieves DRAM-like write latency on an SSD. We demonstrate as the result that major database engines can see a throughput gain of up to 2.8× without the risk of data loss. 2B-SSD allows these applications to enjoy the benefits of having an expensive NVRAM freely, without consuming a valuable DIMM slot. As SSDs become a default storage device in large datacenters serving interactive applications, the proposed 2B-SSD architecture has the potential to make those applications more responsive without incurring costs to the infrastructure.

Acknowledgment

We thank the anonymous reviewers for their constructive comments, which helped improve the quality of this paper. Many researchers and individuals have contributed at various stages to this work, including: Donguk Kim, Hyungwoo Ryu, Kwanghyun La, Yunseok Kang, Hong-Moon Wang, Wonmoon Cheon, Jongyoul Lee, and Pankaj Mehra.

References

[1] C. J. Bashe, *IBM's Early Computers*. The MIT Press, 1986.

[2] "New ssd-backed elastic block storage." https://aws.amazon.com/ko/blogs/aws/new-ssd-backed-elastic-block-storage/, 2014.

[3] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash reliability in production: The expected and the unexpected," in *USENIX FAST*, pp. 67–80, 2016.

[4] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A large-scale study of flash memory failures in the field," in *ACM SIGMETRICS*, pp. 177–190, 2015.

[5] "Oracle launches new all flash fs1 storage system." https://www.oracle.com/corporate/pressrelease/oracle-all-flash-array-FS1-storage-system-082715.html, 2015.

[6] "Emc declares 2016 the "year of all-flash" for primary storage." https://www.emc.com/about/news/press/2016/20160229-04.htm, 2016.

[7] "The industry's first all-flash array with 15 tb ssds." http://community.netapp.com/t5/Tech-OnTap-Articles/The-Industry-s-First-All-Flash-Array-with-15-TB-SSDs/ta-p/120003, 2016.

[8] "Ssds kill the 15k hdd, rolls out last generation." http://www.tomshardware.com/news/-hdd-15k-ssd-enterprise,32920.html, 2016.

[9] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *USENIX FAST*, pp. 3–3, 2012.

[10] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom, "OS I/O path optimizations for flash solid-state drives," in *USENIX ATC*, pp. 483–488, 2014.

[11] S.-W. Lee, B. Moon, and C. Park, "Advances in flash memory ssd technology for enterprise database applications," in *ACM SIGMOD*, pp. 863–870, 2009.

[12] D. Tsirogiannis, S. Harizopoulos, M. A. Shah, J. L. Wiener, and G. Graefe, "Query processing techniques for solid state drives," in *ACM SIGMOD*, pp. 59–72, 2009.

[13] "Nvm express." https://nvmexpress.org/.

[14] R. F. Freitas and W. W. Wilcke, "Storage-class memory: The next storage system technology," *IBM J. Res. Dev.*, vol. 52, no. 4, pp. 439–447, 2008.

[15] C. H. Lam, "Storage class memory," in *IEEE ICSICT*, pp. 1080–1083, 2010.

[16] G. W. Burr, "Storage class memory," in *Non-volatile Memories Workshop*, pp. 1–25, 2010.

[17] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *EuroSys*, pp. 1–16, 2016.

[18] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent page management for two-tiered main memory," in *ACM ASPLOS*, pp. 631–644, 2017.

[19] D. Schwalb, T. Berning, M. Faust, M. Dreseler, and H. Plattner, "nvm malloc Memory allocation for nvram," in *ADMS*, pp. 61–72, 2015.

[20] T. Talpey, "Persistent memory in windows," in *SNIA Persistent Memory Summit*, 2016.

[21] N. Y. Song, Y. J. Yu, W. Shin, H. Eom, and H. Y. Yeom, "Low-latency memory-mapped i/o for data-intensive applications on fast storage devices," in *SCC*, pp. 766–770, 2012.

[22] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM TODS*, vol. 17, no. 1, pp. 94–162, 1992.

[23] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "Linkbench: a database benchmark based on the facebook social graph," in *ACM SIGMOD*, pp. 1185–1196, 2013.

[24] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *ACM SoCC*, pp. 143–154, 2010.

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 5th ed., 2011.

[26] "Samsung ssd pm1725a." http://www.samsung.com/semiconductor/global/file/insight/2016/08/Samsung_PM1725a-1.pdf.

[27] "Samsung shows off a z-ssd: With new z-nand." http://anandtech.com/show/11206/samsung-shows-off-a-z-ssd.

[28] "The intel optane ssd dc p4800x (375gb) review: Testing 3d xpoint performance." http://anandtech.com/show/11209/intel-optane-ssd-dc-p4800x-review-a-deep-dive-into-3d-xpiont-enterprise-performance.

[29] H.-J. Kim, Y.-S. Lee, and J.-S. Kim, "Nvmedirect: A user-space i/o framework for application-specific optimization on nvme ssds," in *USENIX HotStorage*, 2016.

[30] D. Narayanan and O. Hodson, "Whole-system persistence," in *ACM ASPLOS*, pp. 401–410, 2012.

[31] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better i/o through byte-addressable, persistent memory," in *ACM SOSP*, pp. 133–146, 2009.

[32] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," in *ACM ASPLOS*, pp. 105–118, 2011.

[33] T. Hwang, J. Jung, and Y. Won, "HEAPO: heap-based persistent object store," *ACM TOS*, vol. 11, no. 1, pp. 3:1–3:21, 2015.

[34] S. Chen and Q. Jin, "Persistent b+-trees in non-volatile main memory," *VLDB Endowment*, vol. 8, no. 7, pp. 786–797, 2015.

[35] I. Oukid, D. Booss, A. Lespinasse, W. Lehner, T. Willhalm, and G. Gomes, "Memory management techniques for large-scale persistent-main-memory systems," pp. 1166–1177, 2017.

[36] C. Kim, J. Cho, W. Jeong, I. Park, H. W. Park, D. Kim, D. Kang, S. Lee, J. Lee, W. Kim, J. Park, Y. Ahn, J. Lee, J. Lee, S. Kim, H. Yoon, J. Yu, N. Choi, Y. Kwon, N. Kim, H. Jang, J. Park, S. Song, Y. Park, J. Bang, S. Hong, B. Jeong, H. Kim, C. Lee, Y. Min, I. Lee, I. Kim, S. Kim, D. Yoon, K. Kim, Y. Choi, M. Kim, H. Kim, P. Kwak, J. Ihm, D. Byeon, J. Lee, K. Park, and K. Kyung, "11.4 A 512gb 3b/cell 64-stacked WL 3d V-NAND flash memory," in *IEEE ISSCC*, pp. 202–203, 2017.

[37] S. Kim, H. Kim, S.-H. Kim, J. Lee, and J. Jeong, "Request-oriented durable write caching for application performance," in *USENIX ATC*, pp. 193–206, 2015.

[38] J. Jung, Y. Won, E. Kim, H. Shin, and B. Jeon, "FRASH: exploiting storage class memory in hybrid file system for hierarchical storage," *ACM TOS*, vol. 6, no. 1, pp. 3:1–3:25, 2010.

[39] I. H. Doh, Y. J. Kim, J. S. Park, E. Kim, J. Choi, D. Lee, and

S. H. Noh, “Towards greener data centers with storage class memory: Minimizing idle power waste through coarse-grain management in fine-grain scale,” in *ACM CF*, pp. 309–318, 2010.

[40] E. Lee, H. Bahn, and S. H. Noh, “Unioning of the buffer cache and journaling layers with non-volatile memory,” in *USENIX FAST*, pp. 73–80, 2013.

[41] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won, “Nvwal: Exploiting nvram in write-ahead logging,” in *ACM ASPLOS*, pp. 385–398, 2016.

[42] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. E. Anderson, “Strata: A cross media file system,” in *ACM SOSP*, pp. 460–477, 2017.

[43] “Fujitsu server primergy memory performance of xeon e5-2600 v4 (broadwell-ep) based systems.” https://sp.ts.fujitsu.com/dmsp/Publications/public/wp-broadwell-ep-memory-performance-ww-en.pdf, 2016.

[44] R. Budruk, D. Anderson, and T. Shanley, *PCI Express System Architecture*. MINDSHARE, 2004.

[45] “Cache coherent interconnect for accelerators (ccix).” http://www.ccixconsortium.com, 2016.

[46] “mmap.” https://en.wikipedia.org/wiki/Mmap.

[47] Intel, “Write combining memory implementation guidelines,” in *Technical Report 244422-001*, Intel Corporation, 1998.

[48] “Intel 64 and ia-32 architectures software developer’s manual.” https://software.intel.com/en-us/articles/intel-sdm.

[49] “Samsung ssd pm963.” http://www.samsung.com/semiconductor/global/file/insight/2016/08/Samsung_PM963-1.pdf.

[50] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm, “Implications of cpu caching on byte-addressable non-volatile memory programming,” in *HP Labs Technical Reports*, 2012.

[51] “Clflush: Flush cache line (x86 instruction set reference).” http://x86.renejeschke.de/html/file_module_x86_id_30.html.

[52] G. E. Blelloch1, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun, “Efficient algorithms with asymmetric read and write costs,” in *ESA*, pp. 14:1–14:18, 2016.

[53] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka, “Write amplification analysis in flash-based solid state drives,” in *ACM SYSTOR*, 2009.

[54] R. Johnson, I. Pandis, R. Stoica, M. Athanassoulis, and A. Ailamaki, “Scalability of write-ahead logging on multicore and multisocket hardware,” *VLDB Endowment*, vol. 21, no. 2, pp. 239–263, 2011.

[55] M. Stonebraker and G. Kemnitz, “The postgres next generation database management system,” *Communications of the ACM*, vol. 34, no. 10, pp. 78–92, 1991.

[56] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Stumm, “Optimizing space amplification in rocksdb,” in *CIDR*, 2017.

[57] M. Paksula, “Persisting objects in redis key-value database,” University of Helsinki, Department of Computer Science, 2010.

[58] W. Cheong, C. Yoon, S. Woo, K. Han, D. Kim, C. Lee, Y. Choi, S. Kim, D. Kang, G. Yu, J. Kim, J. Park, K.-W. Song, K.-T. Park, S. Cho, H. Oh, D. D. Lee, J.-H. Choi, and J. Jeong, “A flash memory controller for 15us ultra-low-latency ssd using high-speed 3d nand flash with 3us read time,” in *IEEE ISSCC*, pp. 338–339, 2018.

[59] J. Do, D. Zhang, J. M. Patel, D. J. DeWitt, J. F. Naughton, and A. Halverson, “Turbocharging dbms buffer pool using ssds,” in *ACM SIGMOD*, pp. 1113–1124, 2011.

[60] T. Horikawa, “Non-volatile memory (nvm) logging,” in *PGCon*, 2016.

[61] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho, *et al.*, “Biscuit: A framework for near-data processing of big data workloads,” in *ACM/IEEE ISCA*, pp. 153–165, 2016.

[62] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Analysis and evolution of journaling file systems,” in *USENIX ATC*, pp. 105–120, 2005.

[63] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson, “System r: Relational approach to database management,” *ACM TODS*, vol. 1, no. 2, pp. 97–137, 1976.

[64] J. Huang, K. Schwan, and M. K. Qureshi, “Nvram-aware logging in transaction systems,” *VLDB Endowment*, vol. 8, no. 4, pp. 389–400, 2014.

[65] T. Wang and R. Johnson, “Scalable logging through emerging non-volatile memory,” *VLDB Endowment*, vol. 7, no. 10, pp. 865–876, 2014.

[66] W.-H. kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh, “Durable write cache in flash memory ssd for relational and nosql databases,” in *ACM SIGMOD*, pp. 529–540, 2014.

[67] “Nvme ssd with persistent memory region.” https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170810_FM31_Chanda.pdf.