A Case for Hardware-Based Demand Paging

Gyusun Lee*[†] Wenjing Jin*[‡] Wonsuk Song[†] Jeonghun Gong[‡] Jonghyun Bae[‡]
Tae Jun Ham[‡] Jae W. Lee[‡] Jinkyu Jeong[†]

[†]Sungkyunkwan University [‡]Seoul National University

{gyusun.lee, wonsuk.song}@csi.skku.edu, jinkyu@skku.edu
{wenjing.jin, jh.gong, jonghbae, taejunham, jaewlee}@snu.ac.kr

***Abstract*—The virtual memory system is pervasive in today's computer systems, and demand paging is the key enabling mechanism for it. At a page miss, the CPU raises an exception, and the page fault handler is responsible for fetching the requested page from the disk. The OS typically performs a context switch to run other threads as traditional disk access is slow. However, with the widespread adoption of high-performance storage devices, such as low-latency solid-state drives (SSDs), the traditional OS-based demand paging is no longer effective because a considerable portion of the demand paging latency is now spent inside the OS kernel. Thus, this paper makes a case for *hardware-based demand paging* that mostly eliminates OS involvement in page miss handling to provide a near-disk-access-time latency for demand paging. To this end, two architectural extensions are proposed: *LBA-augmented page table* that moves I/O stack operations to the control plane and *Storage Management Unit* that enables CPU to directly issue I/O commands without OS intervention in most cases. OS support is also proposed to detach tasks for memory resource management from the critical path. The evaluation results using both a cycle-level simulator and a real x86 machine with an ultra-low latency SSD show that the proposed scheme reduces the demand paging latency by 37.0%, and hence improves the performance of FIO read random benchmark by up to 57.1% and a NoSQL server by up to 27.3% with real-world workloads. As a side effect of eliminating OS intervention, the IPC of the user-level code is also increased by up to 7.0%.**

***Index Terms*—demand paging, virtual memory, page fault, operating systems, CPU architecture, hardware extension**

I. Introduction

The storage system stack builds on the long-standing assumption of a large performance gap between a fast CPU and a slow disk [14]. However, the widespread adoption of high-performance storage devices, such as solid-state drives (SSDs), is rapidly narrowing down this performance gap. Today's ultra-low latency SSDs, such as Intel's Optane SSD [31] and Samsung's Z-SSD [64], feature only a few microseconds of access time [45], which takes at most tens of thousands of CPU cycles, instead of tens of millions. With this ever-shrinking gap between CPU and storage performance it is necessary to reexamine the design decisions made for the conventional storage system stack to fully harness the performance potentials of these emerging storage devices [45], [74].

Modern computer systems employ page-based virtual memory (paging), and demand paging is the key enabling mechanism, wherein the main memory is used as a cache for disks [55]. Upon a miss on memory (i.e., page fault), CPU raises an exception, and the page fault handler in the operating system (OS) fetches the missing page from disk. The OS typically performs a context switch to run other threads as traditional disk access is slow. However, this page fault handling does more harm than good for ultra-low latency SSDs. Our analysis reveals that the OS's page fault handler not only incurs a significant direct cost (up to 76.29% of the device access time) in terms of miss handling latency but also charges hidden costs such as microarchitectural resource pollution [66] (e.g., caches and branch predictors), thereby slowing down a user application. This motivates us to take a vertically integrated approach from the hardware architecture to software to improve the performance of demand paging.

This paper makes a case for *hardware-based demand paging*, whose time, we believe, has finally come. The main goal of the proposed scheme is to eliminate wasted CPU cycles inside the OS kernel to service a page fault with a near-disk-access-time latency without compromising page protection capabilities provided by OS. To this end, we propose two architectural extensions: i) a *logical block address (LBA)-augmented page table* with an extended page table walker in memory management unit (MMU) and ii) a novel hardware structure called *storage management unit* (SMU). The former allows the CPU to understand the storage layout [6], so upon a page miss, the CPU can locate the missing page in the storage device. The latter handles storage device-specific I/O operations (e.g., the NVMe protocol [51] in our prototype). With the proposed architectural extension, when a CPU accesses a non-resident page, it does not raise an exception but stalls its pipeline and transfers I/O data directly in hardware. Consequently, the latency of miss handling, as well as architectural resource pollution by frequent OS intervention can be greatly reduced.

These new architectural extensions demand proper support from OS [48], and we take a plane separation approach [59]. That is, the data plane (i.e., page miss handling) is performed in hardware, thereby significantly reducing the miss penalty. In the control plane, necessary information is provided by the OS kernel: preserving page-level protections, augmenting a page table entry (PTE) with an LBA during page replacement,

*These authors contributed equally to this work.

978-1-7281-4661-4/20/$31.00 ©2020 IEEE
DOI 10.1109/ISCA45697.2020.00093

allocating a device command queue, supplying free pages to the SMU, and synchronizing OS metadata (e.g., LRU list) with updated hardware usage by hardware. This plane separation not only detaches these management costs from the critical path when handling a page miss but also improves the execution efficiency of these operations by executing them in batch [66].

We evaluate the effectiveness of the proposed architectural extensions using both a detailed cycle-level simulator [26] and a real x86 machine. Our evaluation using Samsung's Z-SSD [64] demonstrates that our hardware-based demand paging reduces the latency of demand paging by 37.0% compared to the conventional OS-based demand paging. To evaluate the impact of our proposal on real-world workloads, we run FIO random reads as a microbenchmark, and DBBench readrandom [21] and YCSB (Yahoo Cloud Serving Benchmark) [18] with RocksDB [60] NoSQL store. The results show that our scheme improves overall throughput by up to 27.3% and user-level IPC by up to 7.0% compared to the OS-based demand paging. All these benefits come at a tiny area cost, 0.004% of the CPU die area.

To the best of our knowledge, this paper is the first to successfully demonstrate the feasibility of hardware-based demand paging. Our contributions can be summarized as follows:

- We analyze the latency of the conventional OS-based demand paging with today's low-latency SSDs and its performance impact on user applications.
- We redefine the roles of hardware and OS to accomplish demand paging with a near-disk-access-time latency. Our micro-architectural design detaches time-consuming OS operations from the critical path of demand paging, so the critical path mostly runs in hardware. The OS-level support acts as a control plane to support the critical path.
- We provide a detailed evaluation of the proposed architecture and OS support using a cycle-accurate full-system simulator and a real x86 machine. The evaluation results demonstrate that the proposed scheme accelerates the performance of demand paging and also improves the execution efficiency of user-level instructions.

II. Background and Motivation

A. Demand Paging and Page Fault

Many applications handling a large amount of data rely on virtual memory and demand paging for its convenience of use. While non-volatile memory (NVM) technologies have recently hit the market as an option to expand main memory [29], they are far from being the mainstream partly due to their lower-than-expected performance and cost efficiency and higher performance variations [34], [54]. Instead, a hierarchy of DRAM backed by high-performance SSDs is still the most popular choice for applications processing a large volume of data. Those applications often map their data stored in a disk to virtual memory and access them without programmers' intervention to load requested data into memory. When a non-resident page is accessed in virtual memory, demand paging is triggered that transfer data from disk to memory. Application programmers are also freed from the concerns of buffer replacement. Data are not usually fit in memory, hence requiring data replacement. By using virtual memory, the page replacement algorithm in the OS kernel preserves recently used data in memory and evicts less-likely used data to disk.

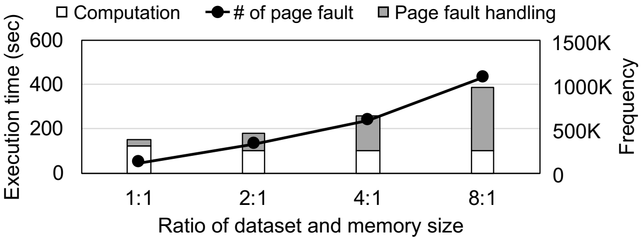


Fig. 1. Execution time breakdown of YCSB-C [18] with various ratio of dataset and memory size

This memory mapped I/O using *mmap()* has the following advantages over the conventional read/write I/O: i) it saves memory space with no redundant buffering in both user and kernel spaces as the user program does not require explicit buffering [37]; ii) it is easier to make persistent a complex in-memory object without invoking frequent fine-grained I/O system calls; iii) the cached portion of the memory-mapped file can be accessed faster due to fewer context switches [22], [67], [73]. The memory-mapped I/O is particularly well suited for applications featuring large working sets and random access patterns. Accordingly, many NoSQL applications [49], [60], in-memory MapReduce [3], [67], graph analytics [57], [58], web servers [2] adopt this technique. Also, programming guides for mobile devices [4], [5] often provide performance optimization tips using *mmap()*.

For such applications, the performance of demand paging is important since it is a form of cache miss that affects the critical path. Figure 1 shows the execution time of the YCSB-C workload and the fraction in time spent for demand paging with varying the ratio of physical memory and dataset size; X:1 means the dataset size is X times larger than the physical memory size. As shown in the figure, as the dataset size increases, a large fraction of time is spent on demand paging (i.e., page fault) while the time spent in the user application (i.e., compute time) remains similar.

A page fault is a key mechanism to implement OS-based demand paging. When a virtual address references a non-resident page in memory (*page miss*), CPU raises an exception, and the page fault handler in the OS kernel handles the exception (*page miss handling*). The page fault handler first allocates a new page frame and issues a read request to the storage I/O stack in the kernel. After the device driver sends a device-specific I/O request to the device-side command queue, the kernel typically switches the context of the current CPU to save CPU cycles because of slow disk access times. Once an I/O request is completed, the blocked thread is woken up, and I/O software stack returns to the fault handler. The fault handling path updates the OS-managed metadata structure (e.g.,

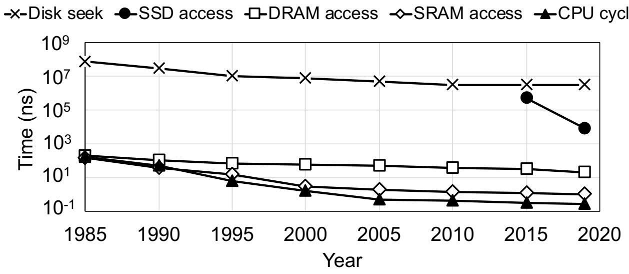


Fig. 2. Performance trends of components in computer systems [14]

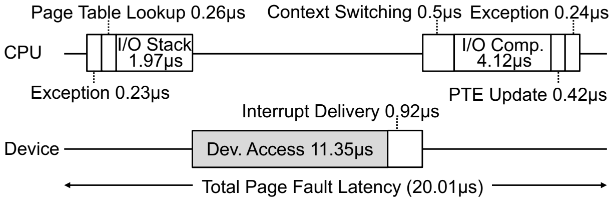


Fig. 3. Time breakdown of a single page fault handling (Drawn not to scale)

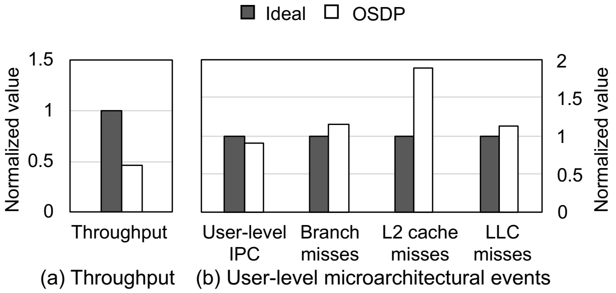
LRU list and reverse mapping), updates the PTEs to make it point to the new page, and finally returns from the exception.

B. Limitations of Page Fault with Ultra-low Latency SSDs

The problem is that recent significant performance advance of SSDs makes the overheads of page fault handling (i.e., OS-based page miss handling) no longer lightweight. Faster storage media have introduced [16], [33], [63]. These devices are now attached to a high-speed I/O bus (e.g., PCI-express 3.0), whose I/O controller is integrated inside a processor die. A more efficient storage access protocol (e.g., NVM express (NVMe) [51]) has been introduced to increase protocol-level efficiency and to reduce CPU overheads [52]; an NVMe protocol requires a single 64 bytes cacheline write to memory and a single PCIe register write to dispatch a single I/O command to a device. The NVMe protocol also supports up to 64K command queues and 64K entries in each queue to scale with multi-core CPUs and high performance of storage media.

Because of these device-side advances, today's ultra-low latency SSDs can deliver up to 3 GB/s I/O bandwidth and only a few microseconds of I/O latency [31], [64]. This trend rapidly narrows the long-standing huge performance gap between CPU and disk, as shown in Figure 2. As of 2019, a disk access time is tens of million CPU cycles, whereas the access time of ultra-low latency SSDs is only tens of thousand CPU cycles.

Figure 3 shows the latency breakdown of a single page fault handling. As shown in the figure, each operation in the fault handling path takes a small amount of time but the total account for the large fraction of the device time: 2.45% on exception and page table walk, 9.85% on I/O submission in the I/O stack, 2.5% on interrupt delivery, 9.85% on context switch, and 20.6% on I/O completion. However, because of the shrunken device time, the aggregated overhead becomes 76.3% of the device time, which is quite substantial.



(a) Throughput (b) User-level microarchitectural events

Fig. 4. Performance impact of page faults on YCSB-C: (a) normalized throughput and (b) normalized user-level IPC and microarchitectural events

The page fault handling incurs not only such direct costs in terms of latency but also indirect costs such as microarchitectural resource pollution [66]. Figure 4 shows throughput and user-level microarchitectural events (i.e., IPC, miss events) while running the YCSB-C workload with and without page faults. The workload is configured to have a dataset fit in memory and we compare two cases: *ideal* experiencing no page faults with the whole dataset pre-loaded into memory and `MAP_POPULATE` enforced, and *OSDP* experiencing frequent page faults with no pre-loading. The results show that *OSDP* only has less than a half throughput of *ideal*. The cost of page fault handling can be divided into direct and indirect costs. The direct cost of page faults is additional CPU cycles to execute the page fault handler in OS and perform disk I/O. Furthermore, as Figure 4(b) shows, page faults also degrade the execution efficiency of user-level instructions by polluting microarchitectural resources such as caches and branch predictors [66], hence lowering the user-level IPC.

C. New Direction: Hardware-based Demand Paging

If demand paging can be supported directly by hardware, no exception needs to be raised to eliminate OS intervention. The latency of demand paging can also be greatly reduced because it is directly performed by CPU. To accomplish this hardware-based demand paging, however, three following important issues need to be resolved.

- **How to make a CPU understand storage layout [6]?** This is important for a CPU to correctly locate missing data in a storage device when a page miss occurs in memory. Since storage layout is managed by the OS kernel (e.g., file system or swapping system), this semantic gap between CPU and the kernel should be bridged.
- **How to make a CPU control a storage I/O device?** A storage device is typically controlled by a device driver in the OS kernel, and various, often complex device control protocols exist depending on the type of device. Hence, this issue includes two sub-issues: i) how to make an isolated I/O path for a CPU to control a storage device and ii) how to make a CPU handle a device control protocol.
- **How to handle remaining page fault operations in the OS kernel?** During a page fault, OS prepares a free page and updates the OS-managed metadata structures

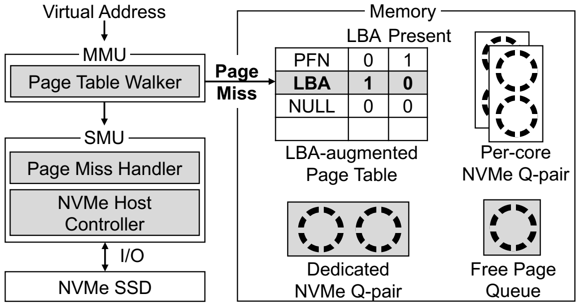


Fig. 5. Hardware-based demand paging overview. Extended or newly added hardware or data structures are shaded in grey.

associated with the page table. When the hardware directly handles page faults without the OS intervention, these tasks should be handled by the OS in some way.

Our work addresses these three issues through new architectural extensions and OS support. Specifically, Section III-B explains how our work resolves the first issue through an architectural extension. Section III-C describes how we address the second issue with the introduction of a novel hardware component. Finally, Section IV introduces the operating system support required to address the third issue.

III. Hardware-based Demand Paging

A. Overview

The proposed hardware-based demand paging handles a page miss in hardware without exception. Figure 5 presents the overall structure of the proposed scheme. The proposed architectural extension is comprised of the following two major components:

- **Logical block address (LBA)-augmented page table** is proposed to make the CPU understand the storage layout. Each PTE for non-resident pages is now augmented LBA, and upon a page miss, the CPU can identify the location of missing data in storage without the intervention by OS.
- **Storage management unit (SMU)** handles storage device-specific I/O operations in cooperation with a memory management unit (MMU) handling address translation. SMU exposes an interface for OS to install proper information to make CPU issue an I/O command directly. This includes an isolated I/O command queue for CPU and free page frames for page miss handling.

With the proposed architectural extension, when a page miss occurs, CPU does not raise an exception but stalls its pipeline and issues I/O operations directly in hardware. Consequently, the latency of demand paging can be greatly reduced because of the elimination of page fault handling overhead in the OS kernel. A user application can also execute its instructions efficiently due to the reduced architectural resource pollution caused by frequent OS intervention. We discuss both components in greater detail in the rest of this section.

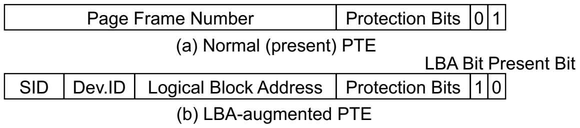


Fig. 6. PTE layouts for (a) a present page and (b) a non-present but LBA-augmented page.

B. LBA-Augmented Page Table and MMU Extensions

Page Table Structure. The PTE of an invalid page holds the logical block address (LBA) to enable hardware-based demand paging. We call this page table an *LBA-augmented page table*. Figure 6 shows the PTE layout when a page is present in memory (a) and when a page is non-present and LBA-augmented (b).

The LBA bit set to one indicates that a page miss of this virtual page should be handled by hardware and that the PTE holds a valid LBA in the disk for the page. This bit is set whenever a non-resident page is made (e.g., `mmap()` or page replacement), and the OS decides to use hardware acceleration for its page miss handling (Section IV-B). When a PTE is augmented LBA, it also holds proper protection bits to preserve page-level permission after its page miss handled in hardware. Whenever the LBA of a file page in the disk is changed, the LBA update is applied to its corresponding LBA-augmented PTE if it is not present in memory.

Page Miss Handling with LBA-augmented PTE. During a page table walk, MMU checks both present and LBA bits in the PTE of the requested virtual address. If both bits are clear (i.e., not resident and not LBA-augmented), MMU raises an exception, and OS handles the page fault normally. In contrast, if the page is not present and the LBA bit is set, MMU requests SMU to fetch the missing virtual page from the disk. Once the requested page is loaded to memory, MMU then receives the newly allocated physical frame address from SMU and resumes the stalled address translation.

The LBA bit is recognized by MMU only when its present bit is clear. In the conventional system, when the present bit is clear, the other bits are ignored, thus OS usually keeps architecture-independent metadata in PTE (e.g., swap offset when a virtual page is swapped out). With the LBA bit set, the page frame number (PFN) bits are used by an LBA, as in Figure 6, which are utilized by MMU. This extension can be readily accommodated by OS without any conflict as it can now put an LBA (i.e., physical block address) instead of a swap offset (virtual block address).

When the LBA bit is set, the page frame number (PFN) field in a PTE should point to a unique storage block in a system. The PFN bits are decomposed into three fields: socket ID (SID), device ID and LBA. The SID field is used to identify the home SMU to handle a page miss to the corresponding page. An <SID, device ID> pair identifies a block device (i.e., an NVMe namespace) in the system. LBA finally locates a unique block within the block device. In our prototype, we assume to use 3 bits for SID (i.e., up to 8 sockets), 3 bits for

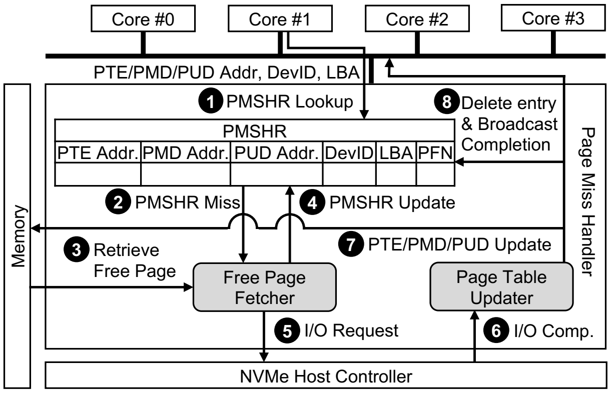


Fig. 7. Page miss handler

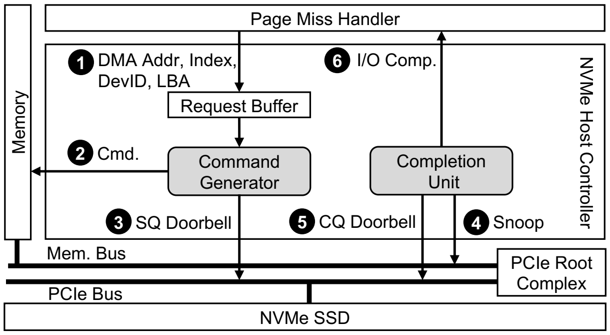


Fig. 8. NVMe host controller

device ID (i.e., up to 8 devices per socket) and 41 bits for LBA (i.e., up to 1 PB storage capacity). The remaining 17 bits are for storing protection bits (12 bits) and architecture-specific features (e.g., 1 bit for no-execute (NX) and 4 bits for page protection key in x86 architecture [32]).

The LBA bit is also added to entries in upper-level page tables (e.g., page middle directory (PMD) and page upper directory (PUD) in Linux on the x86_64 architecture [19], [32]), but is used for a different purpose. Specifically, a set LBA bit in the upper-level entries indicates that its corresponding last-level page table has one or more PTEs whose page miss has been handled by hardware, but OS metadata is not yet updated. Thus, this bit is set when the SMU finishes page miss handling for the relevant PTE and then later cleared by OS when the PTE updates are completely synchronized with the rest of OS data structures. We have empirically found that marking this information in the next two levels up is sufficient to keep the overhead of finding unsynchronized PTEs low. The details are discussed in Section IV-C.

C. Storage Management Unit (SMU)

The storage management unit (SMU) is the key extended architectural component to handle page miss directly in CPU. It consists of two major parts: *page miss handler* and *NVMe host controller*. Figure 7 and 8 depict the structure of the two components. The former handles page misses issued by cores and coalesces duplicated page misses. The latter handles storage-device specific I/O control.

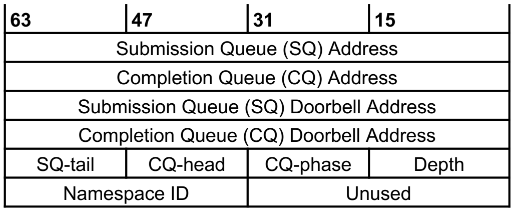
Page Miss Handler. When a page miss occurs, MMU sends a page miss handling request to SMU specified by Socket ID (SID) of the PTE. The request has five parameters, the addresses of the three entries (PUD entry, PMD entry, and PTE), device ID, and LBA. The address of a PTE is an identifier of a page miss since it is the unique key of a virtual page. The device ID and LBA are used to issue a block I/O request, and the three addresses of the entries are used to update their values after handling the page miss.

The page miss handler maintains a set of *page miss status holding registers* (PMSHR). Its structure is similar to that of miss status holding register (MSHR) [41], [69] in that it coalesces duplicated page miss handling requests. Each entry of PMSHR holds the status of an outstanding page miss. The number of entries in PMSHR determines the maximum concurrent outstanding I/O requests supported by SMU. Our prototype empirically chooses 32 entries for PMSHR, which works well in our setup.

When requested to handle a page miss, SMU first looks up any outstanding page miss to the same page in PMSHR (❶ in Figure 7) by using the PTE address as the key. If found, the request returns immediately and the page table walk in MMU enters a pending state. When the page miss handler completes its page miss handling, it broadcasts a completion message with the PTE address, the value of the PTE, and the result of the page miss handling. Then, the pending page table walk can continue. If not found, ❷ the page miss handler allocates an entry in PMSHR and initializes it with the parameters of the new page miss handling request.

For the next step, ❸ the *free page fetcher* allocates a new page frame to hold new data from disk. In our scheme, SMU maintains a buffer of free pages called a *free page queue*, a circular queue residing in memory containing a set of <PFN, DMA address> pairs. To consume each entry in the queue, our hardware maintains three registers, queue base address, head and tail pointers. This queue has a single consumer (the free page fetcher hardware) and a single producer (a page refill routine in the OS kernel), and consequently, no synchronization is necessary. When the queue is empty, SMU invalidates the entry in PMSHR and notifies MMU of the failure of page miss handling. Then, MMU raises a page fault exception and the OS page fault handler handles this page miss. Since the free page queue is empty, the kernel also refills the free page queue by using its page allocator (Section IV-D). A naive implementation of free page fetching can potentially expose a whole memory round-trip latency because it reads an entry from memory. To avoid this issue, our hardware eagerly prefetches and buffers a few free pages into SMU.

For the next step, ❹ the page miss handler completes the initialization of the entry in PMSHR by writing the allocated PFN, which is later used to update PTE. Then, ❺ it sends an I/O request to the NVMe host controller (described in the next paragraph). After ❻ the I/O completes, ❼ the *page table updater* updates the PTE as well as PMD and PUD entries



63	47	31	15
Submission Queue (SQ) Address			
Completion Queue (CQ) Address			
Submission Queue (SQ) Doorbell Address			
Completion Queue (CQ) Doorbell Address			
SQ-tail	CQ-head	CQ-phase	Depth
Namespace ID		Unused	

Fig. 9. NVMe queue descriptor registers allocated for each block device.

by using their addresses. Hence, the LBA field in the PTE is replaced by PFN, and the LBA bits of the upper-level entries are set. Note that SMU does not clear the LBA bit of the PTE to ensure OS later updates the metadata associated with the PTE (see Section IV-C). Once all these updates are finished, ❽ SMU broadcasts a message to notify cores of the completion of page miss handling and finally invalidates the PMSHR entry.

NVMe Host Controller. When the page miss handler decides to issue an I/O command, it transfers the DMA address, device ID, LBA, and the index of PMSHR entry to the NVMe host controller (❺ in Figure 7 connected to ❶ in Figure 8). The NVMe host controller handles commands for a 4KB read without a physical region page (PRP) list and I/O completion in the NVMe protocol [51]. It maintains several sets of descriptor registers (Figure 9) each of which stores necessary information to fetch a block from a block device (or an NVMe namespace[1] [51]). Our prototype assumes to support 8 block devices for each SMU with 3-bit device ID. When the OS kernel enables the proposed hardware-based demand paging for a certain file on a block device, it allocates a new NVMe I/O queue pair, which is isolated from other OS-managed I/O queue pairs, and initializes one set of the NVMe queue descriptor registers for the new I/O queue pair. Then, by augmenting a PTE with a proper SID, device ID and LBA, the SMU can correctly locate and fetch the requested file block from the block device.

To issue an I/O command, the NVMe host controller generates a 64-byte NVMe command and ❷ writes it to memory at the address specified by SQ base address + SQ tail. Then, ❸ it rings the SQ doorbell (i.e., write to the PCIe register) to notify the NVMe device of a new request arrival.

For I/O completion, our scheme disables interrupt for the I/O command queues allocated to SMU [51] to avoid OS intervention. Instead, the completion unit ❹ monitors all memory write transactions from the PCIe root complex via snooping the memory address (CQ base address + CQ head). When it happens, ❺ the completion unit handles the completion protocol (progressing NVMe CQ pointer, ringing CQ doorbell, updating the CQ phase register if necessary), and ❻ percolates the completion upward to the page miss handler. Note that each NVMe command is tagged with the index of PMSHR entry, which is used to find the corresponding PMSHR entry during I/O completion handling.

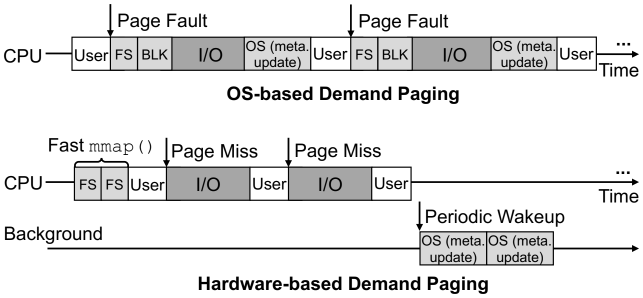


Fig. 10. Comparison between OS-handled demand paging and the proposed hardware-based demand paging

IV. Operating System Support

A. Overview

Figure 10 compares the traditional OS-based demand paging and the proposed hardware-based demand paging. First, the system call API is extended so that page miss on certain pages can be handled in hardware (Section IV-B). Second, OS metadata updates are detached from the critical path and are batched in the background (Section IV-C). Third, free page allocation does not happen every time for a page miss. Instead, a number of free pages are allocated in batch during a device I/O time or through the use of a background kernel thread (Section IV-D). Finally, I/O block layer operations are removed from the OS since the hardware SMU, specifically, NVMe host controller (Section III-C) handles I/O communication. Below, we discuss each aspect in detail.

B. Fast File mmap()

For flexible deployment of hardware-based demand paging on a per-virtual memory area basis, we extend the `mmap()` system call so an application selects virtual memory areas requiring the fast demand paging. For example, database files of a NoSQL application or graph data of a graph analytics application are the target of the fast file `mmap()` because the latency of page miss handling on those files critically affects the performance of those applications.

To this end, a new flag is introduced to the POSIX `mmap()` system call. When the flag is specified, the PTEs within the memory area are all LBA-augmented. During the `mmap()` system call, the kernel checks whether a corresponding page is in the OS page cache. If so, the page is linked in the PTE. Otherwise, the kernel consults the file system to retrieve LBA and records the LBA to the PTE with LBA bit set. When a page belonging to the fast `mmap()` area is evicted from memory, i) the LBA is updated to the PTE, ii) the PTE's present bit is cleared, and iii) LBA bit is set.

Such the LBA augmentation leads to a potential increase in memory usage for the `mmap()` call compared to the original kernel since the original kernel allocates a page for a page table only after at least one of the PTEs in the page table is accessed. However, in our scheme, memory spaces are allocated for the whole page table since all PTEs are populated to store either PFN or LBA. However, the space overhead of PTE for each

[1] A namespace is a storage volume organized into logical blocks, typically managed by a single file system.

TABLE I
Descriptions of PTE, PMD and PUD status according to the values of present and LBA bits

Type	LBA bit	Present bit	PFN field	Description
PTE	`0`	`0`	`0s`	Non-resident, not-LBA-augmented, page miss will-be-handled by OS
	`1`	`0`	LBA	Non-resident, LBA-augmented, page miss will-be-handled by hardware
	`1`	`1`	PFN	Resident, page miss is already handled by hardware, OS metadata is not updated yet
	`0`	`1`	PFN	Resident, identical to conventional PTEs
PUD or PMD entry	`0`	`X`	PFN of next-level table	No PTE in the last-level page table(s) requires OS metadata update
	1	`X`		Last-level page table(s) has one or more PTEs whose associated OS metadata is not updated

4KB page is limited to 0.2% (i.e., 128MB for a 64GB file). In addition, in a case where most of the memory-mapped pages are eventually accessed, this is not really an overhead since most PTEs are eventually populated. The LBA augmentation may also increase the latency of an `mmap()` call. However, this latency increase is not significant since `mmap()` is usually in a control path, which does not affect application performance. If such overhead is a problem, the asynchronous population of PTEs [17] can be applied to alleviate this overhead.

When the block mapping of a file is changed (e.g., file block updates on copy-on-write or log-structured file system [11], [61]), the changed block location should be reflected to its LBA-augmented PTEs. To this end, when a file is mapped using LBA augmentation, the file is marked. Then, whenever a file system changes its block mapping, the routine also updates the LBA field of the PTEs. LBA-augmented PTEs are reverted to normal PTEs when a process forks as our scheme currently does not support sharing of virtual pages across multiple address spaces. We discuss this in greater detail in Section V.

C. Updating OS Metadata for Hardware-handled Page Misses

In conventional OS-based demand paging, the OS kernel handles a page miss, and thus OS-managed metadata associated with the handled page miss can be easily updated. However, in hardware-based demand paging, a page miss is handled by hardware without OS's intervention. Accordingly, updating the relevant OS-managed metadata becomes a nontrivial problem.

To address this problem, our approach introduces a kernel thread (*kpted*) that runs in background and updates the relevant OS-managed metadata for the hardware-handled page misses. This thread periodically (e.g., every 1 second) scans page tables that contain memory area mapping a file using the fast `mmap()`.

If this thread finds a PTE having both LBA and present bits set, it updates the relevant OS metadata for this PTE. Specifically, it i) inserts the page to an LRU list for the page replacement, ii) updates the metadata of the page, iii) adjusts other related metadata (e.g., reverse mapping), iv) inserts the page to the OS page cache if the page is shared, and so forth. The kernel thread finally clears the LBA bit of the PTE to indicate that this PTE's relevant OS metadata is updated.

The PTE scanning cost can be substantial if the thread needs to scan all PTEs in all fast `mmap()`'ed memory areas in every period. However, not all PTEs require the update of their relevant OS metadata. Some of them can be non-resident in memory or serviced OS metadata updates before. For the efficient retrieval of PTEs requiring metadata updates, the kernel thread utilizes LBA bits in upper-level page tables (i.e., PUD and PMD) (explained in Section III-B). An LBA bit in the upper-level entry indicates that its leaf page table(s) has one or more PTEs of which page misses are handled in hardware, hence requiring OS metadata updates. For example, if an LBA bit of a PMD entry is set, the page table pointed by the entry has one or more PTEs having hardware-handled page misses. If an LBA bit of a PUD entry is set, the PMD pointed by the PUD entry has entries whose LBA bit is set. This recursively indicates the leaf page tables belonging to each PMD entry has PTEs awaiting OS metadata updates. As a result, the kernel thread can skip a bunch of PTEs if an upper-level entry has LBA bit clear. In order to guarantee the scanning condition, *kpted* clears the LBA bit of an upper-level entry (e.g., a PUD entry or PMD entry) before inspecting the lower-level table (PMD or last-level page table). Table I summarizes the possible combinations of LBA and present bits in page table entries and their semantics.

Since the OS metadata is updated asynchronously, they should be carefully managed to prevent accesses to inconsistent OS metadata. In our prototype, three system calls, `msync()`, `fsync()`, and `munmap()`, are modified to update OS metadata before their operations. For the `munmap()` system call, it is required to prevent potential races between SMU's page miss handling and PTE unmapping. Before unmapping PTEs, the kernel waits for the completion of all outstanding page misses associated the PTEs to be unmapped. The actual implementation may require an additional instruction (e.g., SMU barrier). Process termination naturally avoids the potential races because it internally calls the modified `munmap()` for all virtual memory areas of a terminating process.

D. Free Page Refill

As stated in Section III-C, when SMU detects a lack of free pages in the free page queue, SMU makes MMU raise a page fault exception. The OS's page fault handler first handles the fault, and then refills the free page queue if the queue is empty. We overlap the refill operation with current device I/O time to hide the latency of page refill operation as in AIOS [45].

This synchronous page refill, however, may increase the number of OS-handled page misses, each of which experiences longer latency than hardware-handled page misses. To reduce the OS-handled cases, we run another kernel thread called *kpoold* that periodically refills the free page queue. In our experiment, the use of *kpoold* reduces the number of page faults for the synchronous refill by 44.3–78.4%.

V. Discussion

Page Aliasing. PMSHR detects and coalesces duplicated page misses to prevent page aliases, and it uses the PTE address as the key to detect identical page misses. Hence, no page alias is made in a multi-thread process running on multiple cores. However, if multiple processes share the same file, the current scheme cannot prevent page aliases. Thus, our scheme reverts LBA-augmented PTEs to normal PTEs when a process forks. The support for file mapping shared across multiple processes is left as future work.

Long Latency I/O. Long-latency I/O operations can unnecessarily occupy a logical core for a long time. Storage-side queuing delays can be alleviated by using storage-side I/O scheduling features (e.g., urgent priority in the NVMe protocol [51]). However, If such a long read delay is caused by device-internal limitations, a millisecond-scale delay is unavoidable, thereby wasting a huge number of CPU cycles. Many attempts have made to alleviate such long delays for reads [35], [40], [72] because reads usually reside in a critical path. However, if such long delays are still unavoidable, one remedy is a timeout-based exception followed by context switching by OS. This may save the wasted CPU cycles.

Prefetching Support. Our scheme currently does not consider memory accesses having spatial locality. There exist workloads taking benefits from prefetching (or readahead) [23], [46], [56], [71]. Prefetching support in SMU is left for future work.

Huge Page Support. Supporting huge pages is possible but not a first-class feature of our design. It is because the use of huge pages for disk-backed files is not well supported in mainstream OS/file systems due to their I/O traffic bloat for writeback [43] and difficulties of the implementation [13]. The virtual memory system in Linux does not support swapping at a huge page granularity, either. However, the proposed scheme can be easily extended to support huge pages. If a PMD holds a huge page mapping, a huge page flag (e.g., PS bit in x86 [32]) is set. If this flag is set, the LBA bit indicates whether an LBA-augmented PTE is used or not; if not, the same bit indicates whether the corresponding last-level page table contains hardware-handled PTEs or not as described in Section IV-C.

Demand Paging Support for Anonymous Page. Our current design only accelerates a major page fault which involves a disk I/O. However, we believe the same architecture can be extended to accelerate demand paging for anonymous pages (e.g., stack and heap). Such anonymous pages trigger minor faults at the first access to them. To accelerate minor faults, we can reserve a pre-defined constant for the LBA field to mark the first access and make SMU bypass I/O processing when it meets the constant. Accelerating swap-in of anonymous pages is straightforward. For swap-out of non-shared pages, the OS kernel can update the LBA field with the LBA on swap space and set the LBA bit of the corresponding PTE. Also, the proposed design does not accelerate copy-on-write (CoW) as shared pages among multiple processes are not supported.

Enforcing OS-level Resource Management Policy. The free page queue in SMU is a global architectural context to make it difficult to apply OS-level memory management policy (e.g., page coloring [68], NUMA policy [44], or memory cgroup [62]). One possible solution is to use per-core free page queues to enforce a memory management policy independently for each thread context. We leave this as future work.

TABLE II
Experimental configuration

Server	Dell R730
OS	Ubuntu 16.04.6
Kernel	Linux 4.9.30
CPU	Intel Xeon E5-2640v3 2.8GHz 8 physical cores (HT)
Storage devices	Samsung SZ985 800GB Z-SSD
Memory	DDR4 32GB

Architecture and OS Dependency. The proposed design primarily targets a 64-bit architecture with a hardware page table walker. 32-bit architectures are not considered as a 32-bit PTE is too small to support sufficient storage capacity. A single NVMe command can read a block of up to 8KB without using an additional PRP list. Thus, pages of 8KB or smaller can be readily supported with the proposed hardware. The support of larger pages (e.g., 64KB, 1MB, 2MB or 1GB page) may require additional modifications to the proposed hardware, which is currently not necessary due to lack of support for huge page file mapping [13]. In addition, the proposed architectural extensions are carefully designed to have no dependency to any specific OS. Although our prototype is based on Linux, we believe other OSs (e.g., FreeBSD) can also easily accommodate the proposed hardware since they maintain similar metadata for virtual memory and perform similar operations for demand paging.

VI. Evaluation

A. Experimental Setup

Methodology. Our architectural proposal requires thorough evaluation not only in micro-architecture-level but also in end-to-end system-level. We take a vertically integrated approach to evaluating the proposed system by using a cycle-level simulator and a real x86 machine.

We first measure the average page miss handling latency with our hardware and OS extensions by running a micro-benchmark FIO [24] with the *mmap* engine on a Gem5-based full system simulator integrated with the SSD model [26]. We then take off the device I/O time to obtain the average number of CPU cycles spent in SMU and modified MMU per page fault, say, T1. Then, we run the same benchmark with our modified OS for SMU emulation (explained in the next paragraph) on a real x86 machine (Table II) to measure the time cost of a page fault in CPU cycles, excluding the device I/O time, say, T2. This is the time from SQ doorbell access to the write of a CQ entry by the device. The difference between the two time costs (T2-T1) is the delta in CPU cycles between the proposed hardware-based scheme and the software-emulated SMU running on a real machine. To estimate the performance gains from our proposed hardware we adjust the measurements from the software-emulated SMU by applying the delta. This

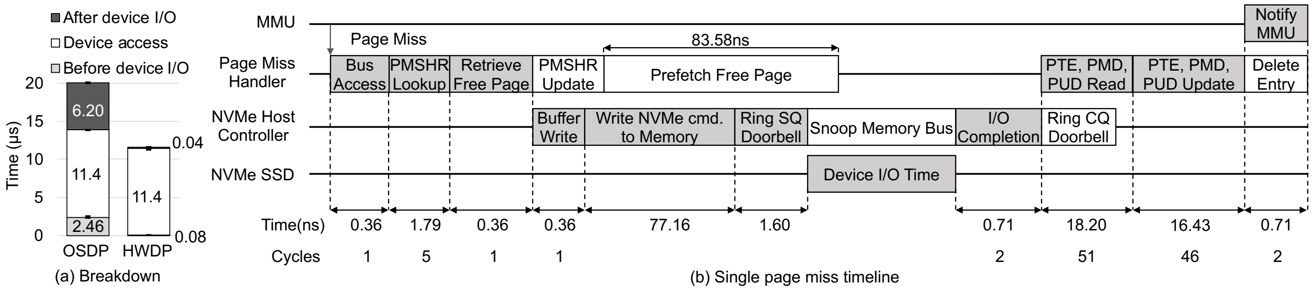


Fig. 11. Single page fault (a) breakdown compared to OS-based demand paging and (b) hardware-based demand paging timeline

approach conservatively estimates the performance of our scheme if it were run with the proposed hardware and OS extensions since the real workloads include contentions in various system components and those are not excluded by the time cost adjustment.

We compare this result with the vanilla, unmodified Linux kernel. The readahead within file `mmap` area is disabled because the readahead (i.e., page prefetch) results in performance degradation for the workloads we tested. Throughout the evaluation, the vanilla kernel is denoted as *OSDP* and our scheme is denoted as *HWDP*.

OS Extension. To test the real-world workloads on the real machine, we modify the original page fault handler in the Linux kernel (version 4.9.30) for x86 as follows. At the early stage of the fault handler, we insert a routine to check an LBA bit (bit 10 used). If set, the routine jumps to a function that emulates the behaviors of storage management unit (SMU); checking and inserting page miss status holding registers (PMSHR), issuing an NVMe command. To emulate the behavior of the memory bus monitoring, we utilize `monitor/mwait` instructions [50]. Once the NVMe I/O finishes and the interrupt is raised, the modified interrupt handling routine touches the memory address that the emulated SMU routine's `mwait` instruction was waiting for. At this point, the routine continues emulating the behaviors of the storage management unit and eventually completes the page miss handling.

B. Page Miss Latency Analysis

Figure 11 shows the breakdown and timeline of single page miss handling latency using the cycle-accurate simulator. We refer to the part before disk I/O time as *before device I/O* and the latter part as *after device I/O*. Because of the disparity between the simulator's device I/O time and a device I/O time in a real machine, we use the value of the host device time in Figure 3. As shown in Figure 11(a), before device I/O and after device I/O are decreased by 2.38μs and 6.16μs compared to OSDP, respectively. Replacing kernel operations with the custom hardware logic greatly reduces the latency overheads to nano-second scale.

Figure 11(b) shows the timeline of actions for a single page miss handling in HWDP. First, before device I/O happens, two register writes and one lookup of content addressable memory (CAM) takes, 1, 1, 5 cycles, respectively. Memory write for writing an NVMe command is the most time-consuming operation, 77.16ns. A single PCIe register write takes 1.60ns. In addition, a memory access needs to happen to fetch a free page; however, we prefetch the free page entries for the future accesses during the device I/O time to hide this memory latency.

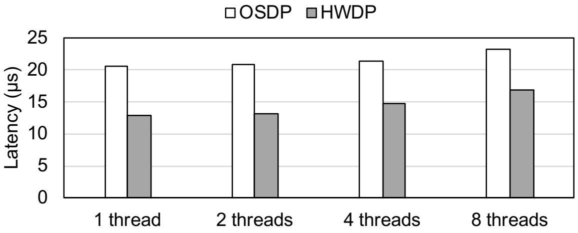


Fig. 12. Demand paging performance (4KB read latency) with varying the number of threads

After the device I/O, the most time-consuming operation is reading and updating the three entries (PTE, PMD and PUD entries). We observe that these accesses rarely miss the CPU's last level cache (LLC), and thus assume these operations take 97 cycles (i.e., three LLC reads and writes). I/O completion (2 cycles) is a register operation. Finally, SMU notifies MMU including bus access and completion checking (2 cycles).

C. Application Performance

This section reports the end-to-end performance of the proposed scheme on the real x86 machine.

Parameter Configuration. In this evaluation, the depth of the free page queue is configured to 4096 (16 MB of memory, 256 pages/core) which is 0.05% of total memory. The period of *kpoold* is 4 milliseconds. Hence, *kpoold* refills pages at a rate of 250 MB/s, which is similar to the bandwidth of 4KB random read using a single thread.

The period of *kpted* may affect the page replacement policy and is set to one second in our test. We believe that the one second period does not affect OS's page replacement policy because the Linux kernel adopts a variant of clock algorithm [25], and our physical memory requires at least 10 seconds to rotate the whole pages in the LRU list; 32 GB physical memory with 3GB/s of read bandwidth of the SSD.

Demand Paging Latency. First, we measure the demand paging performance and depict the results in Figure 12. The FIO benchmark with `mmap` engine [24] is used to measure

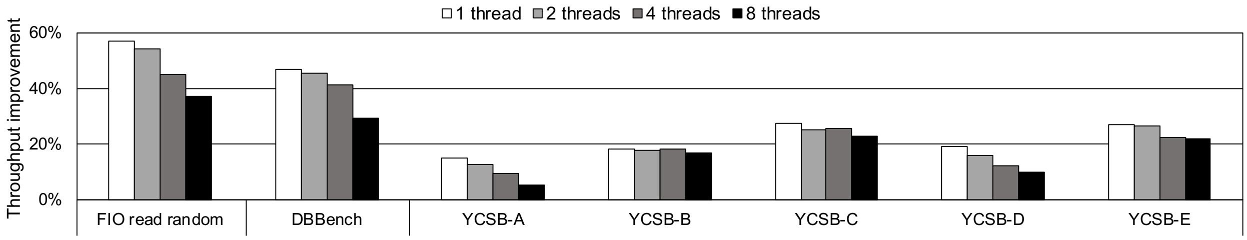


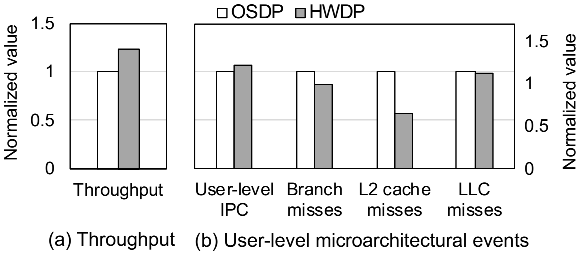
Fig. 13. Throughput improvement by HWDP over OSDP with varying the number of threads

the performance of demand paging. The workload repeatedly accesses 4GB memory-mapped file randomly so as to incur cold page misses. As shown in the figure, our hardware-based demand paging outperforms the traditional OS-based demand paging; reducing the latency by up to 37.0%.

The single thread environment shows the lowest latency, and as parallelism increases, the latency gap compared to OSDP becomes smaller. With eight threads, hence using all the physical cores, the latency gain is reduced to 27.0%.

Realistic Workloads. Then, we measure the performance impact of our scheme through several types of realistic workloads. We tested three workloads: FIO with `mmap` engine to expose application-perceived demand paging performance, DBBench readrandom [21] on RocksDB NoSQL store [60] to test general key-value store performance, and YCSB workloads [18] on RocksDB to test the performance key-value stores in the cloud. For all the workloads, the dataset size is configured to 64GB, which is unable to fit in 32GB physical memory. For the DBBench workload, we conduct four million operations of record size 4KB. The YCSB workloads are configured to run 64GB of dataset and execute 32 million operations of record size 4KB (128 GB of data access footprint in total).

Figure 13 shows the throughput gain by HWDP over OSDP in the three workloads. The results can be summarized as follows. First, access pattern affects the performance. FIO and DBBench workloads achieved the biggest performance gain, 29.4%–57.1%. This is because their memory access pattern is uniform. In contrast, the YCSB workloads follow the realistic access patterns so the performance gain is reduced to 5.3%–27.3%. The maximum performance gain among the YCSB workloads is achieved with YCSB-C (up to 27.3%) because this is a read-only workload. Other workloads have read+write or read-modify-write access patterns. Hence, the workloads show higher read I/O latency than read-only workloads due to contention caused by writes in the SSD. As the read I/O latency increases, the portion of the reduced latency by our scheme becomes smaller, hence yielding lower performance gains. Finally, as the number of thread increases, the performance improvement is slightly reduced for two different reasons. First, a read latency gets higher with more threads because of the increased number of write I/Os by the increased number of threads. This has especially happened in the YCSB-A and YCSB-D workloads. Second, FIO and DBBench show the decrease in performance improvement with 4 or 8 threads because of the increased contention in page tables and PMSHR; we believe this is due to the limitation of our software-based model as PMSHR is organized not into registers in SMU but into a memory table, which incurs cache line contention with a large number of threads.



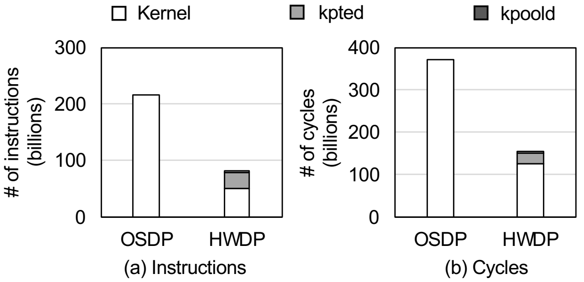
(a) Throughput (b) User-level microarchitectural events

Fig. 14. (a) Normalized throughput and (b) normalized user-level IPC and user-level microarchitectural events in the YCSB-C workload with four threads

Architectural Resource Pollution. To verify whether the proposed scheme is effective in reducing architectural resource pollution by frequent OS intervention, we measured several user-level architectural events and depict the results in Figure 14. We used the hardware performance monitoring unit [32] and measured architectural events in the YCSB-C workload using four threads.

As shown in the figure, our scheme shows improved user-level IPC by 7.0% as compared to OSDP. This is because frequent OS intervention is mostly eliminated. In this workload, 99.9% of page faults are replaced to the hardware-based page miss handling. The figure also depicts the number of miss events on caches and branch prediction, and most of the miss events are decreased. We believe that this result indicates architectural resources are not polluted by OS context [66].

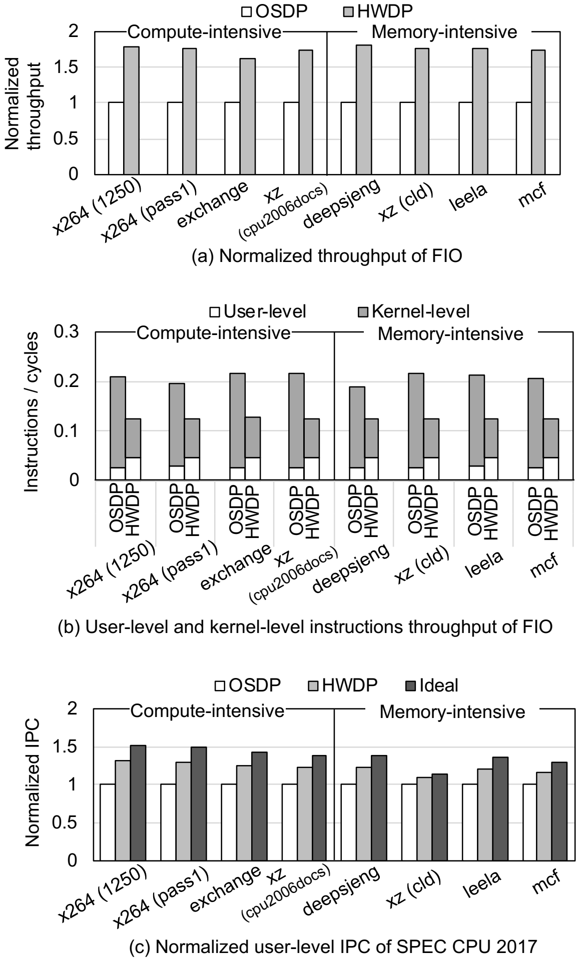
Kernel Costs. Figure 15 shows the number of instructions and CPU cycles spent by the kernel context while running the same YCSB-C workload using four threads. The results of HWDP includes the instructions and cycles spent by the two background kernel threads, *kpted* and *kpoold*. Hence, the *kernel* in the figure denotes kernel contexts in the application thread. As shown in the figure, a total 62.6% of retired instructions are reduced in HWDP. The reasons are twofold: first, the block layer is removed in HWDP and second, the batched OS metadata update efficiently utilizes instructions. The reduction in the number of CPU cycles stays similar to the reduction in the number of instructions. However, the number of CPU



(a) Instructions

(b) Cycles

Fig. 15. The number of kernel-level retired instructions and CPU cycles in the YCSB-C workload with four threads



(a) Normalized throughput of FIO

(b) User-level and kernel-level instructions throughput of FIO

(c) Normalized user-level IPC of SPEC CPU 2017

Fig. 16. Normalized throughput of FIO (a), throughput of user- and kernel-level instructions of FIO (b), and normalized user-level IPC of workloads in SPEC CPU 2017 [20] (c)

cycles for *kpted* is reduced due to its batching of OS metadata update operations.

Polling vs. Context Switching. In HWDP, a pipeline stalls when it misses a page. The core may continue to run a few more instructions having no dependency on the stalled instruction. Nevertheless, the thread context may reach to waiting for the stalled pipeline. Hence, the resources (e.g., functional units) of the core stay idle for a device I/O time. In contrast, OSDP performs context switching during I/O waiting. So, the CPU resources can be utilized during a device I/O time if there is a thread to be scheduled.

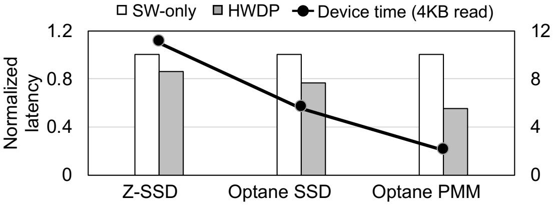


Fig. 17. Benefits of hardware support

However, when our scheme collaborates with simultaneous multi-threading (SMT), the wastage of CPU resources can be greatly reduced and efficiency becomes even better than OSDP. To verify this argument, we configured an experiment as follows. We run two threads, one I/O-bound thread (FIO) and one CPU-bound thread (from the SPEC CPU 2017 benchmark [20]). Each thread is pinned to one of the logical cores sharing one physical core; our testbed supports two hardware threads in each physical core. We ran the two workloads for 30 seconds and collect the following results: throughput of FIO, the number of instructions executed by FIO, and the number of instructions executed by the SPEC thread. Figure 16 shows the three results.

As shown in the figure, in all the cases, HWDP outperforms OSDP. First, HWDP shows more than 1.72× performance improvement over OSDP (Figure 16(a)). Second, each SPEC workload shows higher IPC with HWDP than that with OSDP (Figure 16(c)). In summary, HWDP shows improved throughput of the two co-running workloads in all the cases. The reasons can be explained using Figure 16(b). First, demand paging performance of HWDP is higher than OSDP, so the FIO thread shows improved throughput with HWDP. In Figure 16(b), we can see that FIO executes more user-level instructions with HWDP than with OSDP. Because of the fast demand paging of HWDP, FIO can quickly issue the subsequent demand paging request. Second, the FIO thread with HWDP leaves more CPU resources available to the co-located hardware thread than that with OSDP. In Figure 16(b), the total number of instructions executed by the FIO thread with HWDP is always smaller than that with OSDP by up to 42.4%. Hence, the remaining issue slots and functional units can be utilized by the co-running SPEC thread. As a result, the co-running SPEC thread with HWDP can retire more instructions than that with OSDP.

Software-only vs. Hardware Support. Figure 17 quantifies the performance gains from hardware support by comparing the performance of our fast software-only implementation described in Section VI-A (SW-only) and the proposed design (HWDP). The figure reports the single-fault latency of both schemes normalized to the software-only implementation using three different fast block devices: Z-SSD [64], Optane SSD [31], and Optane DC PMM (used as a storage device in App-direct mode) [29]. It also reports the measured device time for a 4KB read on the three devices, ranging from 2.1μs (Optane

DC PMM) to 10.9μs (Z-SSD). Compared to our fast software-only implementation leveraging LBA-augmented PTEs, HWDP achieves substantial reduction of the page fault latency, and the benefits of using the hardware are more pronounced as the device time gets shorter. For Z-SSD which is the slowest of the three HWDP has 14% lower latency than the software-only implementation. However, for Optane DC PMM whose device time is only around 2μs, HWDP has only about a half of the latency (44% reduction). Thus, the hardware support becomes more important as the device time continues to scale down with the introduction of faster non-volatile memory devices and higher-performing storage interfaces.

D. Area Overhead

We utilize McPAT [27] to coarsely estimate the area overhead of SMU. We use the SRAM and register models in the McPAT CPU model for area estimation of SMU. The die size of the target processor, Intel Xeon E5-2640 v3 at a 22nm technology, is 354mm^2 [12], and the total area of the SMU is 0.014mm^2 (0.004% of the processor), which is a negligible fraction of the processor die size. Our PMSHR has a total of 32 entries, and the size of each entry is 300 bits (three 64-bit addresses, a 64-bit PFN, a 41-bit LBA, and a 3-bit device ID). Since the PMSHR is a fully associative CAM structure, it accounts for 87.6% of the SMU area. In addition, the SMU has eight 352-bit registers for NVMe device control, and they account for 6.7% of the SMU area. The prefetch buffer (16 entries of <PFN, DMA address> pair) accounts for 3.7% of the SMU area. The other miscellaneous registers occupy the remaining 2.0% of the area.

VII. Related Work

Memory-mapped File I/O Optimization on SSDs. To meet the ever-increasing demand for memory capacity from data-intensive applications, optimizing memory-mapped file I/O is an important research issue. SSDAlloc and Chameleon [7], [8] are among the first attempts to expand the DRAM capacity using an SSD with providing low latency page access. They use an address translation module that translates a virtual memory address to the location on the SSD. FlatFlash [1] proposes a lightweight page promotion mechanism using a promotion lookaside buffer that manages hot-cold pages using a byte-addressable SSD. FlashMap [28] mitigates the overheads of the three-stage address translation between memory, storage, and device-level indirection by proposing a unified address. These studies still do not eliminate the OS-handling overhead, which is no longer lightweight with ultra-low latency SSDs. However, our scheme efficiently reduces the overhead with new architecture and OS extension.

Reducing I/O Latency for Ultra-low Latency SSDs. User-level I/O drivers [30], [39], [53], [59], file systems [38], [42] and hybrid approaches [15] are proposed to provide a user-level direct access to an SSD to achieve low I/O latency. These approaches can deliver an I/O to an application in a low latency because of the elimination of high-overhead kernel I/O from the I/O path. However, they require to transfer the full control of an SSD to a user program [30], [39], [53] or extend a storage interface [15] to provide safe user-level access [59]. There also exist studies to optimize the high-overhead kernel I/O stack [45], [65]. Their approaches, however, are contained only to software. DC-express [70] alleviates storage protocol-level overheads. DevFS [36] implements a file system inside an SSD and provides APIs to directly access a DevFS-enabled SSD with minimized OS-level overheads. FlashShare [74] proposes a new I/O submission/completion hardware with OS support. However, its hardware support is limited to I/O submission and completion. In contrast to these related studies, our scheme focuses on the modification of CPU architecture to mostly eliminate page fault exceptions and to provide low latency storage access during demand paging.

Reducing Page Table Walker Latency. Prior research suggests a number of new mechanisms to boost page fault handling [9], [10], [23], [47]. SPAN [23] proposes page prefetching that records and replays the access pattern of pages. ASAP [47] reduces the latency of page table walk by prefetching page table entries from the last two-level page tables and by concurrently checking the entries with top-level page tables. Tempo [10] suggests enabling the memory controller to complete the translation in-place, so as to immediately prefetch the data for which the address translation is being carried out. These optimizations can be seamlessly combined with our work, which would further reduce the page fault latency.

VIII. Conclusion

This paper proposes architectural extensions and OS supports to demonstrate a case for hardware-based demand paging. Two novel hardware extensions are proposed: i) LBA-augmented page table and ii) storage management unit (SMU), and the OS acts as a control plane of the extended hardware components. As a result, the data path of demand paging is handled mostly in hardware with a near-disk access time. The modified OS kernel supports the proposed hardware components; frequent intervention by OS page fault handler is mostly eliminated, and its tasks are detached from the critical path and are batched in background. The proposed scheme is tested using a vertically integrated evaluation approach; using a cycle-accurate simulator to verify the architectural extensions and a real x86 machine to demonstrate the end-to-end performance of the proposed system. The performance evaluation with real-world workloads shows that the proposed scheme improves the performance as well as the user-level IPC of the workloads. We hope that our work vitalizes the memory hierarchy research with the ever-shrinking performance gap between disk and CPU.

Acknowledgment

We thank the anonymous reviewers for their valuable comments. This work was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1702-05. Jinkyu Jeong is the corresponding author.

References

[1] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu, "FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2019, pp. 971–985.

[2] "Apache HTTP server," https://httpd.apache.org.

[3] "Apache Spark," https://spark.apache.org/.

[4] Apple developer, "File system advanced programming topics: Mapping files into memory," https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemAdvancedPT/MappingFilesIntoMemory/MappingFilesIntoMemory.html#//apple_ref/doc/uid/TP40010765-CH2-SW1.

[5] Apple developer, "File system programming guide: Performance tips," https://developer.apple.com/library/archive/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/PerformanceTips/PerformanceTips.html#//apple_ref/doc/uid/TP40010672-CH7-SW1.

[6] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[7] A. Badam and V. S. Pai, "SSDAlloc: hybrid SSD/RAM memory management made easy," in *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2011, pp. 211–224.

[8] A. Badam, V. S. Pai, and D. W. Nellans, "Better flash access via shape-shifting virtual memory pages," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*. ACM, 2013, pp. 3:1–3:14.

[9] T. W. Barr, A. L. Cox, and S. Rixner, "SpecTLB: A mechanism for speculative address translation," in *Proceedings of the 38th Annual International Symposium on Computer Architecture*. IEEE Press, 2011, pp. 307–317.

[10] A. Bhattacharjee, "Translation-triggered prefetching," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 63–76.

[11] J. Bonwick and B. Moore, "Zfs: The last word in file systems," 2007.

[12] B. Bowhill, B. Stackhouse, N. Nassif, Z. Yang, A. Raghavan, O. Mendoza, C. Morganti, C. Houghton, D. Krueger, O. Franza *et al.*, "The xeon® processor e5-2600 v3: A 22 nm 18-core product family," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 1, pp. 92–104, 2015.

[13] N. Brown, "Transparent huge pages in the page cache," https://lwn.net/Articles/686690/, 2016.

[14] R. E. Bryant and O. David Richard, *Computer systems: a programmer's perspective 3rd edition*. Pearson Education, 2015.

[15] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012, pp. 387–400.

[16] 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. Song, K. Park, S. Cho, H. Oh, D. D. G. Lee, J. Choi, and J. Jeong, "A flash memory controller for 15µs ultra-low-latency ssd using high-speed 3d nand flash with 3µs read time," in *2018 IEEE International Solid - State Circuits Conference - (ISSCC)*, Feb 2018, pp. 338–340.

[17] J. Choi, J. Kim, and H. Han, "Efficient memory mapped file I/O for in-memory file systems," in *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems*. USENIX Association, 2017, p. 5.

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

[19] J. Corbet, "Five-level page tables," https://lwn.net/Articles/717293/, 2017.

[20] S. P. E. Corporation, "SPEC CPU 2017," https://www.spec.org/cpu2017/, 2017.

[21] "RocksDB benchmarking tool," https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[22] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014.

[23] V. Fedorov, J. Kim, M. Qin, P. V. Gratz, and A. L. N. Reddy, "Speculative paging for future NVM storage," in *Proceedings of the International Symposium on Memory Systems*. ACM, 2017, pp. 399–410.

[24] "Flexible I/O tester," https://github.com/axboe/fio, 2016.

[25] M. Gorman, *Understanding the Linux virtual memory manager*. Prentice Hall Upper Saddle River, 2004.

[26] 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 *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 2018, pp. 469–481.

[27] Hewlett Packard, "McPAT," https://github.com/HewlettPackard/mcpat.

[28] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified address translation for memory-mapped SSDs with FlashMap," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015, pp. 580–591.

[29] Intel, "Intel Optane DC persistent memory," https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html.

[30] Intel, "Storage Performance Development Kit," http://www.spdk.io/.

[31] Intel, "Intel Optane SSD DC P4800X/P4801X," https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-ssd-dc-p4800x-p4801x-brief.pdf, 2018.

[32] Intel, "Intel 64 and ia-32 architectures software developer's manual," https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf, 2019.

[33] Intel and Micron, "A revolutionary breakthrough in memory technology," http://investors.micron.com/static-files/7b934cfe-139c-4a6f-93e5-b86240642351.

[34] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane DC persistent memory module," 2019.

[35] W. Kang and S. Yoo, "Dynamic management of key states for reinforcement learning-assisted garbage collection to reduce long tail latency in ssd," in *Proceedings of the 55th Annual Design Automation Conference*. ACM, 2018, pp. 8:1–8:6.

[36] S. Kannan, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, Y. Wang, J. Xu, and G. Palani, "Designing a true direct-access file system with DevFS," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*. USENIX Association, 2018, pp. 241–256.

[37] M. Kerrisk, *The Linux programming interface: a Linux and UNIX system programming handbook*. No Starch Press, 2010.

[38] H. Kim and J. Kim, "A user-space storage I/O framework for NVMe SSDs in mobile smart devices," *Proceedings of the IEEE Transactions on Consumer Electronics*, vol. 63, no. 1, pp. 28–35, 2017.

[39] 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 *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX Association, 2016.

[40] S. Kim, J. Bae, H. Jang, W. Jin, J. Gong, S. Lee, T. J. Ham, and J. W. Lee, "Practical erase suspension for modern low-latency ssds," in *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX Association, 2019, pp. 813–820.

[41] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 195–201.

[42] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 460–477.

[43] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel, "Coordinated and efficient huge page management with ingens," in *Proceesings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2016, pp. 705–721.

[44] C. Lameter, "Numa (non-uniform memory access): An overview," *Queue*, vol. 11, no. 7, p. 40–51, Jul. 2013. [Online]. Available: https://doi.org/10.1145/2508834.2513149

[45] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong, "Asynchronous I/O stack: A low-latency kernel I/O stack for ultra-low latency SSDs," in *Proceedings of the 2019 USENIX Annual Technical Conference*. USENIX Association, 2019, pp. 603–616.

[46] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn't, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 2:1–2:29, 2012.

[47] A. Margaritov, D. Ustiugov, E. Bugnion, and B. Grot, "Prefetched address translation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 1023–1036.

[48] J. C. Mogul, A. Baumann, T. Roscoe, and L. Soares, "Mind the gap: Reconnecting architecture and os research," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*. USENIX Association, 2011, pp. 1–1.

[49] "MongoDB," https://www.mongodb.org.

[50] "Linux mwait," https://github.com/torvalds/linux/blob/master/arch/x86/include/asm/mwait.h, 2019.

[51] "NVM express," https://nvmexpress.org.

[52] "NVM express overview," https://www.nvmexpress.org/wp-content/uploads/NVMe_Overview.pdf.

[53] "OpenMPDK," https://github.com/OpenMPDK/uNVMe.

[54] O. Patil, L. Ionkov, J. Lee, F. Mueller, and M. Lang, "Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using intel optane DC persistent memory modules," in *Proceedings of the International Symposium on Memory Systems*. ACM, 2019, p. 288–303.

[55] D. A. Patterson and J. L. Hennessy, *Computer organization and design*. Newnes, 2013.

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

[57] R. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, Nov 2010, pp. 1–11.

[58] R. Pearce, M. Gokhale, and N. M. Amato, "Faster parallel traversal of scale free graphs at extreme scale with vertex delegates," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, p. 549–559.

[59] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," *ACM Trans. Comput. Syst.*, vol. 33, no. 4, pp. 11:1–11:30, Nov. 2015.

[60] "RocksDB," https://github.com/facebook/rocksdb.

[61] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Trans. Storage*, vol. 9, no. 3, Aug. 2013.

[62] R. Rosen, "Resource management: Linux kernel namespaces and cgroups," *Haifux, May*, vol. 186, 2013.

[70] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. L. Moal, T. Bunker, J. Xu, S. Swanson, and Z. Bandić, "DC express: Shortest latency protocol for reading phase change memory over PCI express," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*. USENIX Association, 2014, pp. 309–315.

[63] Samsung, "Ultra-Low Latency with Samsung Z-NAND SSD," https://www.samsung.com/semiconductor/global.semi.static/Ultra-Low_Latency_with_Samsung_Z-NAND_SSD-0.pdf, 2017.

[64] Samsung, "Samsung Z-SSD SZ985," https://www.samsung.com/semiconductor/global.semi.static/Brochure_Samsung_S-ZZD_SZ985_1804.pdf, 2018.

[65] W. Shin, Q. Chen, M. Oh, H. Eom, and H. Y. Yeom, "OS I/O path optimizations for flash solid-state drives," in *Proceedings of the 2014 USENIX Annual Technical Conference*. USENIX Association, 2014, pp. 483–488.

[66] L. Soares and M. Stumm, "FlexSC: Flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2010, pp. 33–46.

[67] N. Y. Song, Y. Son, H. Han, and H. Y. Yeom, "Efficient memory-mapped I/O on fast storage device," *ACM Trans. Storage*, vol. 12, no. 4, pp. 19:1–19:27, May 2016.

[68] G. Taylor, P. Davies, and M. Farmwald, "The tlb slice—a low-cost high-speed address translation mechanism," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 355–363. [Online]. Available: https://doi.org/10.1145/325164.325161

[69] J. Tuck, L. Ceze, and J. Torrellas, "Scalable cache miss handling for high memory-level parallelism," in *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 2006, pp. 409–422.

[71] F. Wu, H. Xi, J. Li, and N. Zou, "Linux readahead: less tricks for more," in *Proceedings of the Linux Symposium*, vol. 2. Citeseer, 2007, pp. 273–284.

[72] S. Yan, H. Li, M. Hao, M. H. Tong, S. Sundararaman, A. A. Chien, and H. S. Gunawi, "Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND ssds," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies*. USENIX Association, 2017, pp. 15–28.

[73] T. Yoshimura, T. Chiba, and H. Horii, "EvFS: User-level, event-driven file system for non-volatile memory," in *11th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX Association, 2019.

[74] J. Zhang, M. Kwon, D. Gouk, S. Koh, C. Lee, M. Alian, M. Chun, M. T. Kandemir, N. S. Kim, J. Kim, and M. Jung, "Flashshare: Punching through server storage stack from kernel to firmware for ultra-low latency ssds," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, 2018, pp. 477–492.