Taming the Killer Microsecond

Shenghsun Cho, Amoghavarsha Suresh, Tapti Palit, Michael Ferdman and Nima Honarmand
Department of Computer Science, Stony Brook University
Stony Brook, NY, USA
{shencho, amsuresh, tpalit, mferdman, nhonarmand}@cs.stonybrook.edu

***Abstract*—Modern applications require access to vast datasets at low latencies. Emerging memory technologies can enable faster access to significantly larger volumes of data than what is possible today. However, these memory technologies have a significant caveat: their random access latency falls in a range that cannot be effectively hidden using current hardware and software latency-hiding techniques—namely, the microsecond range. Finding the root cause of this "Killer Microsecond" problem, is the subject of this work. Our goal is to answer the critical question of why existing hardware and software cannot hide microsecond-level latencies, and whether drastic changes to existing platforms are necessary to utilize microsecond-latency devices effectively.**

We use an FPGA-based microsecond-latency device emulator, a carefully-crafted microbenchmark, and three open-source data-intensive applications to show that existing systems are indeed incapable of effectively hiding such latencies. However, after uncovering the root causes of the problem, we show that simple changes to existing systems are sufficient to support microsecond-latency devices. In particular, we show that by replacing on-demand memory accesses with prefetch requests followed by fast user-mode context switches (to increase access-level parallelism) and enlarging hardware queues that track in-flight accesses (to accommodate many parallel accesses), conventional architectures can effectively hide microsecond-level latencies, and approach the performance of DRAM-based implementations of the same applications. In other words, we show that successful usage of microsecond-level devices *is not* predicated on drastically new hardware and software architectures.

***Index Terms*—Killer microseconds, Emerging storage, Data-intensive applications, FPGA**

I. Introduction

Accessing vast datasets is at the heart of today's computing applications. Data-intensive workloads such as web search, advertising, machine translation, data-driven science, and financial analytics have created unprecedented demand for fast access to vast amounts of data, drastically changing the storage landscape in modern computing. The need to quickly access large datasets in a cost-effective fashion is driving innovation across the computing stack, from the physics and architecture of storage hardware, all the way to the system software, libraries, and applications.

The traditional challenge in accessing data is the increased latency as the data size grows. This trend is observed at virtually all levels of storage, whether the data reside in on-chip storage, main memory, disk, or remote servers. Therefore, architects have designed a variety of mechanisms to *hide* data access latency by amortizing it across bulk transfers, caching hot data in fast storage, and overlapping data transfer with independent computation or data access.

Current latency-hiding mechanisms and storage interfaces were designed to deal with either nanosecond-level fine-grained accesses (e.g., on-chip caches and DRAM) or millisecond-level bulk-access devices (e.g., spinning disks). Hardware techniques, such as multi-level on-chip caches, prefetching, superscalar and out-of-order execution, hardware multi-threading, and memory scheduling are effective in hiding nanosecond-level latencies encountered in the memory hierarchy. On the other hand, millisecond-level latencies of disks and networks are hidden through OS-based context switching and device management.

While these techniques are effective for conventional memory and storage devices, the computing landscape, especially in data centers and warehouse-scale computers, is introducing new devices that operate in the gap between memory and storage—i.e., at latencies in the microsecond range. A variety of such technologies are being deployed today, such as Flash memories (latencies in the tens of microseconds) [1] and 40-100 Gb/s Infiniband and Ethernet networks (single-digit microseconds) [2]–[4]. New devices, such as 3D XPoint memory by Intel and Micron (hundreds of nanoseconds) [5], are being introduced over the next several years.

Unfortunately, existing micro-architectural techniques that handle fine-grained accesses with nanosecond-level latencies cannot hide microsecond delays, especially in the presence of pointer-based serial dependence chains commonly found in modern server workloads [6]. At the same time, OS-based mechanisms themselves incur overheads of several microseconds [7], and while such overheads are acceptable for large bulk transfers in traditional storage, they would fundamentally defeat the benefits of emerging low-latency devices.

This fact, dubbed the "Killer Microsecond" problem [8], [9], although clearly understood at an intuitive level, completely lacks quantification and examination in technical literature. In particular, in the context of microsecond-latency storage, which is the target of this work, it is not clear what aspects of current hardware and software platforms are responsible for their inability to hide microsecond-level latencies. As a result, it is not known whether effective usage of such devices is preconditioned on fundamental hardware/software changes in existing systems, or simply requires straightforward modifications. In this paper, we undertake a quantitative study of modern systems to find an answer to this question.

As we will discuss in Section II, we believe that adoption and disruptive use of microsecond-latency storage devices in data-intensive applications is incumbent on their integration as memory, capable of serving fine-grained (i.e., cache-line size) data accesses whose latency must be hidden from application developers by the hardware/software platform. To investigate the limits of such integration in modern systems, we developed a flexible hardware/software platform to expose the bottlenecks of state-of-the-art systems that prevent such usage of microsecond-latency devices.

On the hardware side, we developed an FPGA-based storage device emulator with controllable microsecond-level latency. This hardware allows us to experiment with different device-interfacing mechanisms (Section III) commonly found in modern systems. On the software side, we used a synthetic microbenchmark that can be configured to exhibit varying degrees of memory-level parallelism (MLP) and compute-to-memory instruction ratios—both of which are well-known factors in the performance of data-intensive applications.

We used the microbenchmark and our FPGA platform to find the bottlenecks of a cutting-edge Xeon-based server platform in hiding microsecond-level latencies. Then, to verify that the identified bottlenecks are relevant to real workloads, we ran three open-source data-intensive applications on our platform, and compared their performance with that of our microbenchmark, showing that they follow the same general trends and suffer from the same bottlenecks.

Our results indicate that, although current systems cannot effectively integrate microsecond-latency devices without hardware and software modifications, the required changes are not drastic either. In particular, we show that simple software modifications such as changing on-demand device accesses into prefetch instructions combined with low-overhead user-mode context switching, and simple hardware changes such as properly sizing hardware structures that track pending memory accesses (e.g., the prefetch queues) can go a long way in effectively hiding microsecond-level latencies.

We also show that software-managed queue-based interfaces, such as those used in NVMe and RDMA devices, are *not* a scalable solution for microsecond-latency storage. Instead, these devices should be memory mapped—ideally, connected to low-latency interfaces such as QPI links or DDR buses—and accessed through load, store, and prefetch instructions. Although today's off-the-shelf processors do not readily support such storage devices, the required modifications are simple.

Overall, the main contribution of this work is not in proposing new hardware/software mechanisms to hide microsecond-level latencies; rather, it is in showing—perhaps surprisingly and in contradiction to our intuitive understanding of the Killer Microsecond problem—that novel mechanisms are not strictly necessary. We do so not through simulations, whose representativeness of real systems are often hard to establish, but through careful experimentation with real, commercially-relevant hardware and software.

II. Background

A. Trends in Low-Latency Data Access

The storage landscape is rapidly changing, especially in data centers and warehouse-scale computers. On the one hand, data-intensive workloads require fast access to large datasets. On the other hand, density, cost-per-bit, and power scaling of volatile (DRAM) and non-volatile (spinning disks and Flash) storage technologies have already stagnated or are expected to do so in the near future [10], [11].

TABLE I: Common hardware and software latency-hiding mechanisms in modern systems.

Paradigm	HW Mechanisms	SW Mechanisms
Caching	On-chip caches Prefetch buffers	OS page cache
Bulk transfer	64-128B cache lines	Multi KB transfers from disk and network
Overlapping	Super-scalar execution Out-of-order execution Branch speculation Prefetching Hardware multithreading	Kernel-mode context switch User-mode context switch

Given these trends, the current solution to the "big-data-at-low-latency" problem is to shard the data across many in-memory servers and interconnect them using high-speed networks. In-memory data stores [12], in-memory processing frameworks [13], [14], in-memory key-value stores [15], [16], and RAM Clouds [17] are all examples of this trend. In these systems, although the storage medium (i.e., DRAM) provides access latencies of tens of nanoseconds, a remote access still takes a few hundreds to thousands of microseconds. This extra latency is commonly attributed to the software stack (networking and OS scheduling), network device interface, and queuing effects in both user and kernel spaces [18], [19].

Fortunately, new non-volatile memory (NVM) and networking technologies can provide a way out of this situation. Low-latency high-density NVM devices, such as 3D XPoint [5], will allow large amounts of data to be stored on a single server, without requiring a lot of expensive and power-hungry DRAM. Furthermore, where multiple servers are needed—for capacity, availability, quality-of-service, or other reasons—fast RDMA-capable networks, such as Infiniband and Converged Ethernet [3], [4], can bring the remote data within a few microseconds. However, successfully leveraging these devices requires effective mechanisms to hide microsecond-level latencies, as well as hardware/software interfaces that avoid the high overheads of OS-based device management.

B. Latency-Hiding Mechanisms

Among the existing latency-hiding techniques, three paradigms can be identified. *Caching* is useful in applications with substantial temporal locality. *Bulk transfers* is useful if there is significant spatial locality. *Execution overlapping* is helpful if there is enough independent work to overlap with the accesses, and the overhead of discovering and dispatching independent work is less than the access latency. These paradigms are implemented using different mechanisms at different layers of the computing stack, as listed in Table I.

Any storage technology can be optimized for either fast fine-grained accesses (cache-line size) or slow bulk accesses (page size or larger). The main determinants in this decision are 1) the random access latency of the device, and 2) the effectiveness of hardware and software mechanisms in hiding that latency.[1]

Slow bulk accesses are only effective where there is substantial spatial locality to amortize the access time across many useful data elements. Creating spatial locality requires careful attention to the data layout and access patterns, and is a notoriously difficult task. It can therefore be argued that fast fine-grained accesses are preferable to bulk transfers, particularly in the context of modern server applications that exhibit little spatial locality [6].

On the other hand, in the presence of fine-grained accesses and limited locality, latency hiding falls primarily onto execution overlapping techniques. Existing hardware mechanisms have low overheads, but they can only find small amounts of useful work to overlap with data accesses. Super-scalar and out-of-order execution are limited to windows of about 100 instructions, and hardware multithreading can expand the amount of available independent work only by a small factor. Overall, these techniques have short look-ahead windows and are only effective for hiding fast accesses. Notably, the main benefit of these techniques is leveraging memory-level parallelism (MLP) by finding and overlapping multiple independent memory operations, as logic and arithmetic operations alone have limited potential for hiding long DRAM latencies.

Software techniques, however, can look for independent work across many threads. They can potentially hide larger latencies, but come with significantly higher overheads. For example, one study [7] found that the overhead of a kernel-mode context switch can range from several to more than a thousand microseconds. Obviously, such high-overhead mechanisms are not effective for hiding microsecond-level latencies.

We argue that, for microsecond-latency devices to have a disruptive impact on data-intensive workloads—to give them bigger data at lower latencies than possible today—they must accommodate fast, fine-grained (cache-line size) accesses. Therefore, the key question is whether existing latency-hiding techniques (Table I), in conjunction with existing device-interfacing mechanisms (Section III), allow effective use of microsecond-latency devices as fast, fine-grained memory.

III. Device Access Mechanisms

In this section, we enumerate the existing mechanisms that could be used for interfacing with microsecond-latency devices, and describe their interaction with the latency-hiding mechanisms of the previous section. For some mechanisms, simple reasoning clearly demonstrates their poor applicability to fast fine-grained accesses. We describe these mechanisms here, but omit them from the evaluation results. For the rest, we quantitatively analyze their performance to uncover their limitations in Section V.

In the process of studying the wide range of mechanisms described here, we found that, by their nature, the implementation of all device access mechanisms corresponds to a pair of queues, one for requests and one for responses. To perform a data access, a processor core writes a message into the request queue; to indicate completion of the data access, the device writes a message into the response queue. The existence of these queues is sometimes non-obvious, hidden by hardware abstractions. Nevertheless, it is the behavior of the queues, and their interactions with the various software and hardware components in the system, that dictate the performance characteristics of the device interfaces. We will elaborate more on this point as we describe each mechanism.

A. Software-Managed Queues

Kernel-Managed Software Queues. The age-old approach to device access is through kernel-managed software queues. The kernel allocates a structure in host memory, which is both the request and the response queue. To perform an access, an application performs system calls, and the kernel manages the queues on behalf of the application.

Upon receiving the system call, the kernel stores the request into a queue and performs a *doorbell* operation—usually a memory-mapped I/O (MMIO) write—to inform the device about the presence of a new request Then, the application thread is de-scheduled and placed on an OS wait queue.

Triggered by the doorbell, the device reads the head element of the request queue from memory, performs the access, stores the result in the host memory, and updates the response queue to indicate request completion. Finally, it raises an interrupt to signal the host to check the response queue.

Kernel-managed queues perform well for millisecond-level devices with bulk accesses. Although each request has a high overhead—the system call, doorbell, context switch, device queue read, device queue write, interrupt handler, and the final context switch, adding up to tens or hundreds of microseconds—the costs are still acceptable for millisecond-level bulk-access devices. For microsecond-latency devices, however, these overheads dwarf the access latency, making kernel-managed queues ineffective. Therefore, we omit this access mechanism from further consideration.

Application-Managed Software Queues. For faster devices (e.g., Infiniband NICs), the excessive overheads of kernel-managed queues have inspired alternative designs where the kernel permits applications to directly manipulate the queue contents and the doorbell mechanism.

Although this technique removes the kernel overheads, it also loses important kernel functionality. First, the kernel cannot schedule other threads while an access is outstanding. Second, because this technique avoids interrupts, it requires polling the completion queue.

To overcome these limitations, application-managed queue implementations must rely on a cooperative *user-level scheduler* to keep the processor core busy after sending out a request. The scheduler, invoked after each request, de-schedules the current user-level thread, checks the response queue to identify completed requests, and "wakes up" the user-level threads that originally requested the newly-completed accesses.

[1]There are other factors—such as read and write durability, power and storage density, and persistence properties—that should be considered, but latency and ability to hide it have historically been the most important.

Although this approach avoids the system calls, kernel-level context switches, and interrupt handling, it still operates a costly MMIO doorbell after enqueuing each request. Doorbell overheads can be reduced with a flag in the host memory that indicates if the doorbell is needed for the next request. The device continues checking the request queue until it reaches a pre-defined limit or when the request queue is empty. It then sets the *doorbell-request* flag. The host checks this flag to determine whether it should operate the doorbell on the next request, and clears the flag after doing so.

Additionally, reading one request at a time from host memory unnecessarily serializes request processing. To avoid serialization, the device can perform a *burst read* of several requests from the head of the queue. This potentially over-reads the available requests in the queue, but minimizes the latency and overheads of reading the queue in situations where multiple requests are typically present.

In modern systems, an application-managed software queue, with a doorbell-request flag and burst request reads, is in fact the best software-managed queue design for microsecond-latency devices. We experimented with mechanisms lacking one or both of these optimizations and found them to be strictly inferior in terms of maximum achievable performance. Therefore, we choose this highly-optimized mechanism to evaluate the best performance achievable through software-managed queues. Nevertheless, as our evaluation results indicate, the overhead of software queue management manifests itself as a major bottleneck for microsecond-latency devices.

B. Hardware-Managed Queues

High performance storage and networking interfaces have traditionally relied on software-managed queues. However, with the emergence of Storage Class Memory (SCM), system architects may want to use *memory interfaces* for storage devices. Memory interfaces are built around hardware-managed queues, incurring lower overheads than software queues. Moreover, they allow device storage to be accessed using ordinary load and store instructions, similar to DRAM, mitigating software development challenges.

On-Demand Accesses. While traditional uses of memory-mapped IO (MMIO) are for device control and status checks, an MMIO region behaves much in the same way as DRAM and allows software to access the device like byte-addressable memory. Software can insert a request into the hardware request queue simply by performing a load or store operation on the corresponding address. Response handling is performed automatically by the hardware, waking up any dependent instructions and allowing the load instruction to complete and retire, all without any software overhead for explicit polling or processing of the response queue.

Using hardware queues and their automatic management entails both advantages and disadvantages for request handling. On the upside, all of the processor's latency-hiding mechanisms are automatically leveraged. For example, MMIO regions marked "cacheable" can take advantage of locality,

Listing 1: Prefetch-based device access function

```
int dev_access(uint64* addr) {
  asm volatile ("prefetcht0␣%0" :: "m"(*addr));
  userctx_yield();
  return *addr;
}
```

while an out-of-order core can issue multiple parallel accesses to the device and overlap them with independent work.

On the downside, the size of the reorder buffer inherently limits the amount of work that can overlap with a request. Compared to DRAM accesses, a load from a microsecond-latency device will rapidly reach the head of the reorder buffer, causing it to fill up and stall further instruction dispatch until the long-latency access completes and the load instruction retires. Even worse, whereas the software-managed approach is able to quickly add requests belonging to multiple software threads into the request queue to perform in parallel, the processor core is limited to discovering such accesses only within a single software thread, severely restricting the extent to which access-level parallelism can be exploited.

Modern CPUs implement simultaneous multi-threading (SMT), partitioning the core resources into multiple hardware contexts that execute independent software threads. SMT offers an additional benefit for on-demand accesses by allowing a core to make progress in one context while another context is blocked on a long-latency access. In this way, SMT is able to extract higher performance from a processor core when device accesses are encountered. However, the number of hardware contexts in an SMT system is limited (with only two contexts per core available in the majority of today's commodity server hardware), limiting the utility of this mechanism.

Software Prefetching. The software- and hardware-managed queues described above have been used with a wide range of data access mechanisms. They accent the dichotomy between storage and memory: storage devices use software-managed queues, paying large overheads for queue management, but benefiting from the ability to switch threads and continue using the processor while requests are outstanding; memory devices use hardware-managed queues, avoiding the queue management overheads, but they are limited to the latency-hiding techniques of the processor and the instruction-level parallelism found within a small window of a single thread.

A best-of-both-worlds approach combines the performance of hardware-managed queues with the flexibility of user-level threading. Hardware queue management is needed to lower the overhead of request queuing and completion notification, while ultra-lightweight user-level threading is needed to overlap a large amount of independent work with device accesses, and enable many threads to perform device accesses in parallel. Fortunately, today's processors already offer the necessary building blocks of this approach in the form of *software prefetching* and the associated hardware queues.

Listing 1 presents the device access function for the prefetch-based access mechanism. The non-binding `prefetcht0` instruction enqueues a memory address in the hardware request queue. The prefetch request does not block the processor pipeline, allowing the software to perform a user-level context switch and continue running another thread, without being limited by the size of the reorder buffer. If the new thread also performs a device access, it issues another non-binding prefetch, followed by a context switch to yet another thread. The idea of hiding access latency like this, using a prefetch followed by a context switch to other independent threads, is not common today, but has been previously considered in some systems. For example, Culler et al. use a similar idea in the Threaded Abstract Machine [20], and the MIT Alewife uses context switching after a cache miss [21].

As user-level threads continue to execute, the hardware is responsible for automatically handling access requests in the queue. Device accesses are issued to MMIO regions marked as "cacheable," accessing the microsecond-latency device and reading a cache block of data surrounding the requested address. When the access completes, the hardware is responsible for completing the request and installing the cache block in the L1 cache of the requesting core.

Eventually, all user-level threads run out of work, either because they all issued a prefetch and performed a context switch, or because they encountered a synchronization operation that prevents further progress. The user-level scheduler then switches back to one of the threads that performed a prefetch request and performs a regular load, which ideally hits in the L1, and execution proceeds. If there are not enough threads to hide the entire latency of a device access, the regular load operation will automatically identify the presence of the requested address in the hardware queue (MSHR, the miss status holding register) and await its return. In the meantime, the out-of-order core will continue to issue subsequent independent instructions until the reorder buffer fills up. When the prefetch request completes, the hardware automatically, and with no software overhead, wakes up all instructions dependent on the long-latency access and allows the waiting load instruction to retire.

This mechanism can be effective at hiding microsecond-level latencies provided that 1) there are enough threads on each processor core, 2) switching among threads is inexpensive, and 3) the hardware queues are large enough to accommodate the outstanding requests from all threads and all cores. Our results indicate that, while the first two requirements are relatively easy to meet, it is the small size of the hardware queues—at multiple levels of the memory hierarchy—that prevent sufficiently-many in-flight device accesses to hide microsecond-level latencies. Therefore, increasing the queue sizes can be a simple remedy for the Killer Microsecond problem, without requiring drastic architectural changes in modern processors, as we discuss in Section V.

IV. Evaluation Methodology

To evaluate the device access mechanisms of Section III, we use a PCIe-enabled FPGA board to emulate a configurable microsecond-latency device. We use a Xeon-based server to host the FPGA board and run the corresponding software library. Together, our hardware and software platform covers the three access mechanisms that we study in this work: on-demand memory-mapped access, prefetch-based memory-mapped access, and application-managed software queue.

Instead of using commercially available NVM storage devices that allow both bulk and byte-level accesses (such as Microsemi's Flashtec drives [22]), we chose to build an FPGA-based storage emulator for several crucial reasons. First, we needed to experiment with different interface options and have full control over their features (e.g., the queue-management details of software queues). Second, we needed to experiment with different ranges of device latency. Finally, and most importantly, we needed to ensure that the internal performance of the device does not become an artificial bottleneck that could limit the number of in-flight accesses and thus mask the system-level bottlenecks we were looking for. As we explain in the next section, our FPGA design employs several non-trivial design tricks to meet this challenge. Also, we initially experimented with placing the emulator on the QPI interconnect using the Intel HARP platform [23], but found the HARP system overly restrictive due to some inherent assumptions in its architecture, while the PCIe interface gave us the needed flexibility in experimenting with different interfaces.

For the on-demand and prefetch mechanisms, the FPGA is exposed to the host as a cache-line addressable memory, accessible using standard memory instructions. Our prefetch-based implementation performs a light-weight user-level context switch immediately after triggering a device accesses using software prefetch instructions. To study the software-managed queues, a different FPGA design interacts with in-memory software-managed descriptor queues to receive requests and provide responses to the software. For this, we use the same lightweight user-mode context switching mechanism, but rather than performing a prefetch and relying on the hardware to manage the request, software-managed in-memory descriptor structures are used to communicate requests to the device and software is used to check for request completion.

A. *Microsecond-Latency Device Emulator*

For the host system, we use an Intel Xeon E5-2670v3 dual-socket server, with one CPU removed to avoid cross-socket communication, and with hyperthreading disabled. Hardware prefetching is also disabled to avoid interference with the software prefetch mechanism. An Altera DE5-Net FPGA board is connected via a PCIe Gen2 x8 link to the host. The FPGA receives requests from the host and responds with the requested cache line after a configurable delay. The configured response delays account for the PCIe round-trip latency ($\sim$800ns on our system).

Because we study big-data applications, replete with pointers and data-dependent accesses, the emulated device must faithfully reply to host requests with the correct memory contents. This requires us to use the FPGA's on-board DRAM to store large amounts of data. Unfortunately, the FPGA DRAM interface (DDR3 at 800Mhz) has high latency and low bandwidth, which makes it impossible to emulate a high-

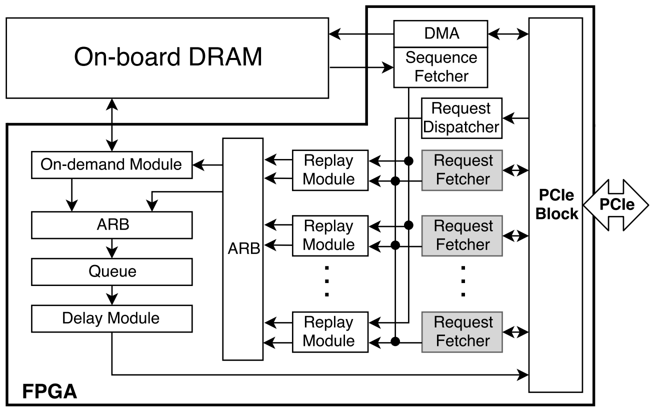


Fig. 1: Hardware design of the microsecond-latency device emulator. Request Fetchers (gray boxes) are used by the software-managed queue. Memory-mapped accesses use the Request Dispatcher.

performance microsecond-latency device by performing on-demand accesses to slow on-board DRAM.

To overcome this problem, we developed an *access replay* mechanism using the FPGA device. In this method, we run each experiment twice. In the first run, we record the application's data access sequence and save the read addresses and their corresponding data. Before starting the second run, we load this sequence into the FPGA's on-board DRAM using a DMA engine. During this second run, the pre-recorded sequence is continuously streamed using bulk on-board DRAM accesses well in advance of the request from the host, allowing the FPGA to precisely control the response latency of the emulated accesses. This second run is the one whose performance we measure and report in the paper.

We ensure that the memory access sequence remains deterministic across these runs and different cores, which enables us to reuse the same recorded access sequence (after applying an address offset) to handle requests from multiple cores in our multi-core experiments, significantly reducing the bandwidth and capacity requirements of the on-board DRAM.

The complexity of this design (detailed below) is necessary to ensure that the internal device logic does not become the limiting factor when we increase the number of parallel device requests. This is crucial for finding performance bottlenecks of the host system; otherwise, the FPGA-induced limits would prevent the manifestation of the host bottlenecks.

Memory-Mapped Hardware Design. Figure 1 presents our emulator's internal architecture. For the on-demand and prefetch-based mechanisms, the FPGA exposes a byte-addressable memory region over a PCIe BAR (Base Address Register) to the host. We modified the host processor's MTRRs (Memory Type Range Registers) to map the BAR region as cacheable, enabling us to use regular loads and software prefetch instructions to perform device accesses through the processor's cache hierarchy The FPGA also exposes a control interface to perform DMA transfers of the recorded access sequences into the on-board DRAM.

Replay modules maintain synchronization between the pre-recorded access sequences stored in on-board DRAM and the host requests. Because PCIe transactions do not include the originating processor core's ID, we subdivide the exposed memory region and assign each core a separate address range, to enable steering core requests to their corresponding replay modules. Once a host request is matched by a replay module, a response is enqueued in a *delay module*, which sends the response to the host via PCIe after a configurable delay. To ensure precise response timing, incoming requests are timestamped before dispatch to a replay module, enabling the delay module to determine when a response should be sent.

The above scheme works for the vast majority of host accesses. However, a CPU occasionally experiences cache hits or performs wrong-path speculative accesses, resulting in the replay modules observing missing, reordered, or spurious accesses compared to the pre-recorded sequence. A naïve replay module implementation quickly locks up due to a mismatch between the replay and request sequences. To overcome these deviations, our replay module tracks a sliding window of the replay sequence and performs an age-based associative lookup for each request.

CPU cache hits result in replay entries that never match a host request and are thus skipped. We do not immediately age out older entries once a match is found, but instead temporarily keep skipped accesses in the window to ensure they are found in case of access reordering.

Spurious requests present a greater challenge. Although they are initiated by wrong-path instructions, it is nevertheless critical to respond to them correctly, because the responses are cached in the CPU's on-chip hierarchy and responding with incorrect data may break later execution. When the replay module cannot match a host request within the lookup window, the request is sent to the *on-demand module*, which reads the data from a copy of the dataset stored in a separate on-board DRAM. The ratio of spurious requests compared to regular accesses is minute. Therefore, the on-board DRAM channel on which these data are accessed is lightly loaded, and we can still meet the response delay deadlines for nearly all accesses.

Software-Managed Queue Design. In this interface, the software puts memory access descriptors into an in-memory *Request Queue* and waits for the device to update the corresponding descriptor in an in-memory *Completion Queue*. Each descriptor contains the address to read, and the target address where the response data is to be stored. The device ensures that writes to the Completion Queue are performed after writes to the response address.

Given this protocol, the emulator's architecture differs from that of the memory-mapped design in a number of ways. The hardware interface exposes per-core doorbell registers that are associated with the *request fetcher* modules. After adding a request to request queue, the host software triggers the request fetcher by performing an MMIO write to the corresponding doorbell. Once triggered, the request fetcher continuously performs DMA reads of the request queue from host memory, and forwards these requests to the corresponding core's replay module. Unlike the memory-mapped design, where the delay module sends a single read completion packet to complete the request, here the delay module performs two write transactions: one to transfer the response data and another to write to the completion queue. Also, because the device accesses are explicitly generated by the software, there are no missing or spurious accesses, leaving the on-demand read module unused for this interface.

To amortize the PCIe overheads of descriptor reads, the request fetcher retrieves descriptors in bursts of eight, starting from the most-recently observed non-empty location in the request queue, and continues reading so long as at least one new descriptor is retrieved during the last burst. Continually fetching new descriptors eliminates the need for the host software to perform a costly doorbell operation after enqueuing each request. Instead, when no new descriptors are retrieved on a burst, the request fetchers update an in-memory flag to indicate to the host software that a doorbell is needed to restart the fetcher for the next access.

B. Support Software

We developed a parallel, high-performance software framework to interact with microsecond-latency devices. The framework includes kernel drivers to expose the device (via MMIO or software-managed queues as described in Section IV-A), a user-level threading library, and an API to perform fine-grained device accesses. We built and evaluated the platform on Ubuntu 14.04 with kernel 3.18.20. We used the GNU Pth library [24] for user-level threading, extending it with our device access mechanisms and optimizing heavily for modern hardware and our use cases. To reduce interference from unrelated system threads in our experiments, we use Linux's *isolcpu* kernel option to prevent other processes from being scheduled on the cores where we perform our measurements.

The library and its API are designed to minimize application source code changes compared to traditional software that expects its data to be in memory. To avoid changing the application code, the exposed API only requires the application to use the standard POSIX threads, and to replace pointer dereferences with calls to `dev_access(uint64*)`. This device access function is synchronous to the calling thread and returns only after the device access is complete. The library transparently handles all low-level details of performing the accesses and switching user-level threads to overlap access latency with execution of other threads. This design, keeps changes to the application source code minimal.

API functions for the prefetch model are implemented with a sequence of "prefetch, scheduler call, load" (expecting the load to hit on the prefetched line in the L1), as shown in Listing 1. In this case, the scheduler simply switches between threads in a round-robin fashion.

The software-managed queue model, on the other hand, requires a more sophisticated support software that requires close integration between the scheduler and queue-management code. The scheduler polls the completion queue only when no threads remain in the "ready" state. The threads are managed in FIFO order, ensuring a deterministic access sequence for replay.

Finally, taking advantage of devices with microsecond-level latencies requires an extremely optimized context switch and scheduler. After significant optimizations, we were able to reduce the context switch overheads from 2 microseconds in the original Pth library to 20–50 nanoseconds, including the completion queue checks.[2]

C. Benchmarks

We begin our study using a carefully-crafted microbenchmark that gives us precise control over the memory accesses it performs. We then validate our microbenchmark behavior and relate it to real-world scenarios by porting three open source applications to our framework. In both the microbenchmark and applications, only the main data structures are emulated to be stored in the microsecond-level storage, while the application code, as well as hot data structures (including stack, global variables, etc.) are all placed in the main memory (i.e., DRAM).

Microbenchmark. The microbenchmark enables controlled analysis of the performance characteristics of the various access mechanisms. Its main loop includes a device access followed by a set of "work" instructions that depend on the result of device access, mimicking real applications that would use the accessed value. To avoid perturbing the memory hierarchy, the work comprises only arithmetic instructions, but is constructed with sufficiently-many internal dependencies so as to limit its IPC to $\sim$1.4 on a 4-wide out-of-order machine. The microbenchmark supports changing the number of work instructions performed per device access. We refer to this quantity as the *work-count* throughout the paper.

A single-threaded microbenchmark is used to measure the DRAM baseline and the on-demand access mechanism. We use multi-threading to overlap microsecond-latency accesses for the prefetch and software-managed queue mechanisms. In either case, each thread repeatedly performs a device access followed by work instructions.

In all cases, we make each microbenchmark access go to a different cache line, ensuring that results are straight forward to interpret because there is no temporal or spatial locality across accesses. All reported results are averaged over 1 million iterations of the microbenchmark loop. For the baseline, we replace the device access function with a pointer dereference to a data structure stored in DRAM.

We report microbenchmark performance as "normalized work IPC". We define "work IPC" as the average number of "work" instructions retired per cycle. To compute the normalized work IPC, we divide the work IPC of the microbenchmark run (with device accesses) by the work IPC of the single-thread DRAM baseline.

Applications. In addition to the microbenchmark, we ran the following open source software on our platform, using

[2]This required sacrificing some functionality, such as the ability to deliver pending signals to individual threads, that was not important for our use case.

the microsecond-level device for their key data structures. Importantly, we kept the code changes to a minimum by maintaining the standard POSIX threading model and the demand-access programming model used in the original code.

- The BFS (breadth first search) benchmark is a graph traversal benchmark from Graph500 [25]. BFS begins with a source vertex and iteratively explores its neighbors, until reaching the destination vertex. Graph traversal is a central component of many data analytics problems, such as recommendation systems, social media modeling, and route optimization.
- The Bloom filter benchmark is a high-performance implementation of lookups in a pre-populated dataset. Bloom filters are space-efficient probabilistic data structures for determining if a searched object is likely to be present in a set. Bloom filters are routinely employed in many large-scale computations such as genome analysis.
- The Memcached benchmark performs the lookup operations of the Memcached [26] in-memory key-value store that is widely used to speed up dynamic content generation applications by caching the results of expensive sub-operations, such as database queries.

We are interested in observing the behavior of these applications with regard to performing their core data structure accesses, but without being biased by the memory-system behavior of the work that they do after the data accesses (the work differs between implementations and relies on additional auxiliary data structures not stored in the microsecond-latency device). Therefore, we modify the applications, removing code not associated with accessing the core data structures, and replacing it with the benign work loop from our microbenchmark. For each application, we report its "normalized performance" obtained by dividing the execution time of the device-access version by the execution time of a single-threaded baseline version where data is stored in DRAM.

V. Results and Analysis

In this section, we identify the performance bottlenecks faced by microsecond-latency devices when used as high-capacity storage with fine-grained accesses. Our goal is to understand the integration options and limitations imposed by widely-available modern systems that are most likely to host such devices.

A. On-Demand Access

We first consider the case of unmodified software that simply uses the microsecond-level device as normal memory using on-demand memory-mapped accesses. For these experiments, we use our microbenchmark which performs a memory load operation to access a word of data followed by work instructions with an IPC of ∼1.4. The requests are serviced by a 1µs-, 2µs-, and 4µs-latency device which, providing a 64-byte cacheable block in response to a load.

Figure 2 shows the device performance compared to our baseline that stores the data in DRAM. As expected, within the

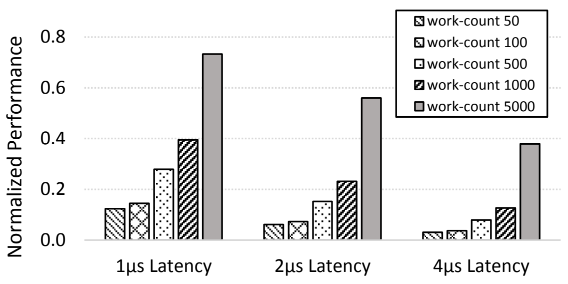


Fig. 2: On-demand access of microsecond-latency device. The values are normalized to the single-threaded on-demand DRAM baseline.

range of a reasonable number of work instructions per memory load, the performance drop is abysmal and likely precludes the use of microsecond-level devices in any performance-conscious system. Only when there is a large amount of work per device access (e.g., 5,000 instructions), the performance impact of the device access is partially abated. However, with such infrequent device accesses, the usefulness of the microsecond-level devices would be questionable to begin with.

***Implications*.** It is impractical to use microsecond-latency devices as drop-in DRAM replacements with unmodified software and hardware. Given the instruction window size of modern processors (∼100–200 instructions), out-of-order execution cannot find enough independent work—in particular, independent device accesses—to hide device latency. Therefore, software changes are required to assist hardware in discovering independent work.

B. Prefetch-Based Access

Next, we consider the technique where a software prefetch instruction is used to access the device before performing a light-weight context switch to another thread (Listing 1).

Hardware Queues with Prefetch-Based Access. Figure 3 shows the performance of this system with 1µs, 2µs, and 4µs device access latencies. The values are normalized to the single-threaded on-demand DRAM baseline. As the number of threads rises, the system effectively hides a greater portion of the microsecond-level latency, improving performance relative to the DRAM baseline. Longer device latencies result in a shallower slope because a smaller fraction of the device latency is overlapped.

These results show that the simple round-robin user-mode context switch has minimal performance impact. At 10 threads and 1µs device latency, the performance is similar to running the application with data in DRAM. Notably, the microsecond-latency device marginally outperforms DRAM. This is because the DRAM baseline suffers from DRAM latency, which is not fully overlapped with useful work. In comparison, prefetch-based system overlaps all device accesses with work and device accesses from other threads, thereby reducing the perceived device latency. Figure 4 shows the effect when more work is performed per device access. As expected, with more

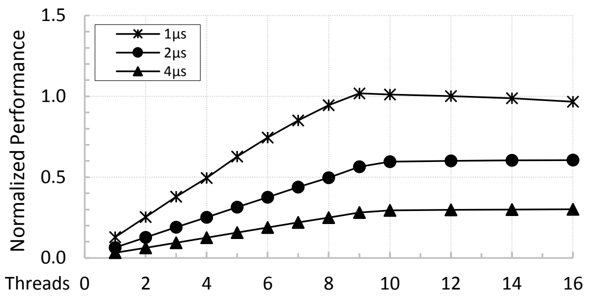


Fig. 3: Prefetch-based access with various latencies. The values are normalized to the single-threaded on-demand DRAM baseline.

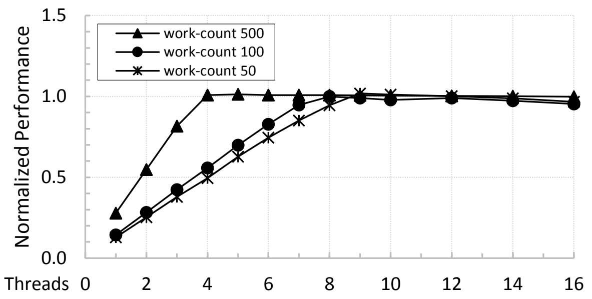


Fig. 4: 1µs prefetch-based access with various work counts. The values are normalized to the single-threaded on-demand DRAM baseline.

work, fewer threads are needed to hide the device latency and match the performance of the DRAM baseline.

Figure 3 also reveals the first limitation of the prefetch-based technique on modern hardware. Once requests are issued with software prefetch instructions, the outstanding device accesses are managed using a hardware queue called Line Fill Buffers (LFBs) in Intel processors. In essence, LFBs are miss status holding registers (MSHRs) and are used to track pending cache misses in case of memory accesses. To the best of our knowledge, all state-of-the-art Xeon server processors have at most 10 LFBs per core, severely limiting number of in-flight prefetches. As a result, after reaching 10 threads, additional threads do not improve performance. This is particularly noticeable for slower devices, where more threads are needed to hide device latency.

Multicore Effectiveness. To understand the multicore scalability of prefetch-based accesses, Figure 5 shows the behavior of the same microbenchmark running concurrently across multiple cores on the same CPU. We continue to plot the performance as a function of the number of threads per core, and normalize all results to the performance of a single-core DRAM baseline to maintain consistency across figures.

With a few threads per core, the multi-core performance scales linearly compared to the baseline. Each core uses its LFBs to track outstanding device accesses, providing aggregate performance. This is best seen for 4µs latency, which shows that the multi-core systems are not limited to 10 simultaneous accesses and can exceed the performance of a single

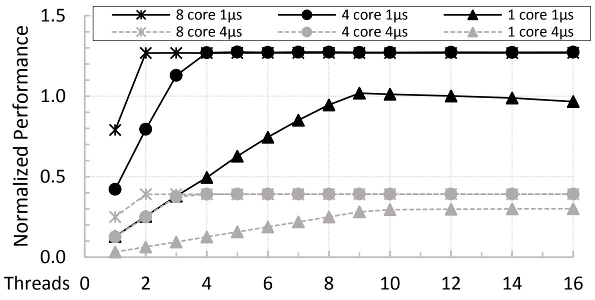


Fig. 5: Multicore prefetch-based access with various latencies. The values are normalized to the single-threaded on-demand DRAM baseline.

core (capped at 10 threads by the LFB limit). Unfortunately, another hardware cap emerges, limiting this technique from being effective on multi-core systems. Although the LFBs across cores are independent, the on-chip interconnect between the cores and the PCIe controller has another hardware queue which is shared among the cores. This shared queue becomes yet another bottleneck that limits performance improvements as the thread count increases.

We do not have sufficient visibility into the chip to determine the location of the shared queue. However, we have experimentally verified that the maximum occupancy of this queue is 14. We have also verified that the queue is not a fundamental property of the on-chip ring interconnect by confirming that a larger number of simultaneous DRAM accesses can be outstanding from multiple cores (e.g., at least 48 simultaneous accesses can be outstanding to DRAM). Regardless of the exact nature of this bottleneck, its existence limits the use of any tricks for leveraging the LFBs of multiple cores to increase the number of simultaneous device accesses and obtain higher performance.

Impact of MLP. The microbenchmark results presented thus far are straightforward to interpret, but are not necessarily representative of real applications. In particular, real applications are likely to perform multiple independent data accesses for a given amount of work, allowing for the latency of the data accesses to be overlapped. For example, in Memcached, after key matching is done, value retrieval can span multiple cache lines, resulting in independent memory accesses that can overlap with each other. We therefore consider variants of our benchmark having memory-level parallelism (MLP) of 2.0 and 4.0. We modify the code to perform a single context switch after issuing multiple prefetches. In the DRAM baseline, the out-of-order scheduler finds multiple independent accesses in the instruction window and issues them into the memory system in parallel. In each case, the microsecond-latency device results are normalized to the DRAM baseline with a matching degree of MLP.

The results comparing the MLP variants are presented in Figure 6. We label these variants as 1-read, 2-read, and 4-read, and the 1-read case is the same microbenchmark used in the previous graphs. Similarly to the 1-read case, the 2- and 4-read variants gain just as much performance from the first several

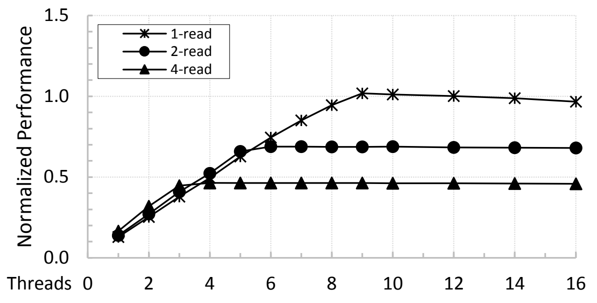


Fig. 6: 1μs prefetch-based access at various levels of MLP. Each line is normalized to the corresponding DRAM baseline.

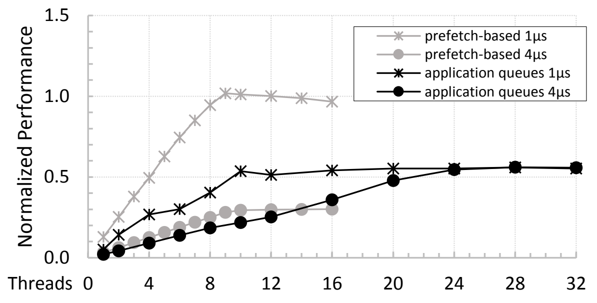


Fig. 7: Comparison of the application-managed queues and prefetch-based accesses with 1μs and 4μs device latencies.

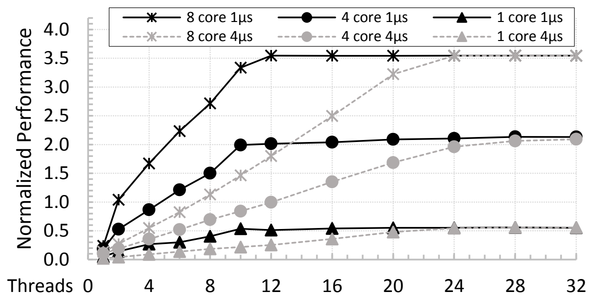


Fig. 8: Multicore comparison of software-managed queues with 1μs and 4μs device latencies.

threads and are just as effective as the 1-read case. However, performing multiple accesses per thread prior to context switch consumes LFBs more rapidly, and significantly reduces the number of threads that can be leveraged. While the 1-read case can scale to 10 threads before filling up the LFBs, the 2-read system tops out at 5 threads, and the 4-read system peaks at 3 threads. In summary, we see that the LFB limit is more problematic for applications with inherent MLP, severely limiting their performance compared to the DRAM baseline. For brevity, we omit the multi-core and higher-latency results with MLP because they follow identical trends as the 1-read microbenchmark, topping out at 14 aggregate accesses.

***Implications*.** The combination of fast user-mode threading and prefetch-based device access has the potential to hide microsecond-level latencies. The main obstacle in realizing this potential in modern systems is the limited hardware queue sizes that prefetch requests encounter on their way to the device. If the per-core LFB limit of 10 could be lifted, given enough threads, even 4μs-latency devices could match the performance of DRAM. Moreover, if the chip-level queue-size limit of 14 were increased, the prefetch-based mechanism could effectively scale to multicore systems.

Given these results, some simple back-of-the-envelope calculations can determine the required queue sizes at different levels of the memory hierarchy. Each microsecond of latency can be effectively hidden by 10-20 in-flight device accesses per core. Therefore, the per-core queues (of which LFB is one example), should be provisioned for approximately "20 $\times$ expected-device-latency-in-microseconds" parallel accesses. Chip-level shared queues, of which the 14-entry queue is an example, should support "20 $\times$ expected-device-latency-in-microseconds $\times$ cores-per-chip" to have sufficient access parallelism for the whole chip. Processor vendors can make similar calculations for all queues encountered by the prefetch instructions on their path to the device and size them accordingly, to ensure a balanced system design.

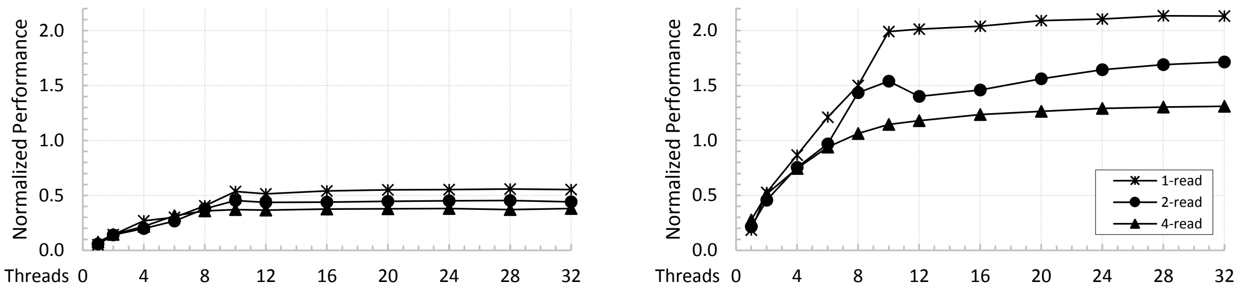
It appears that shared hardware queues on the DRAM access path are larger than on the PCIe path. Therefore, integrating microsecond-latency devices on the memory interconnect in conjunction with larger per-core LFB queues may be a step in the right direction.

We must emphasize that such hardware changes are not sufficient alone. Software changes (i.e., prefetching + fast context switch) are needed to create a high degree of accesses-level parallelism. With this approach, the programming model remains unchanged and the application changes are minimized, with the device access complexity encapsulated in the threading library. Using such software, and with the hardware queues properly sized, we do not foresee any reason that would prevent conventional processors from effectively hiding microsecond-level latencies and approaching the performance of DRAM for data-intensive applications.

C. *Application-Managed Software Queues*

The hardware queues in modern processors limit the prefetch-based access performance of microsecond-latency devices. We therefore consider systems that do not rely on hardware queues and instead keep the queues in memory, using a light-weight descriptor and doorbell interface to interact between queues stored in host memory and the device.

Effectiveness of application-managed queues. We begin with a direct comparison of the prefetch-based access and application-managed queues in Figure 7. Two effects are evident. First, for higher latency, when the prefetch-based access encounters the LFB limit, the application-managed queues continue to gain performance with increasing thread count. Second, although the thread count is not bounded by a hardware queue, the application-managed queues incur significant queue management overhead, which fundamentally limits the peak performance relative to the prefetch-based access. At 10 threads and 1μs, or 24 threads and 4μs, all of the device latency is effectively overlapped and peak performance is achieved. However, the queue management overhead incurred



(a) 1 Core

(b) 4 Cores

Fig. 9: Impact of MLP on software-managed queues with one and four cores. Each line is normalized to the corresponding DRAM baseline.

for each access limits the peak performance of the application-managed queues to just 50% of the DRAM baseline.

Multicore effectiveness. Figure 8 shows the scalability of the application-managed queues to multiple cores. Unlike the prefetch-based mechanism that was limited to 14 requests across cores, the application-managed queues have no such limitations and achieve linear performance improvement as core count increases.

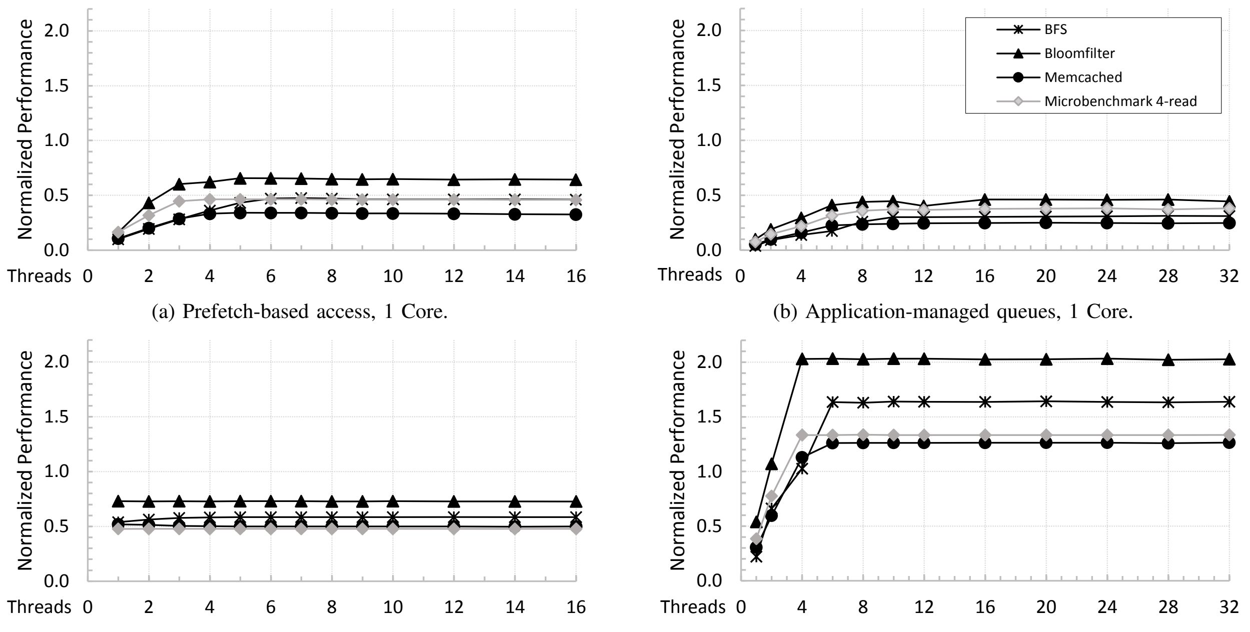
Unfortunately, at eight cores, the system encounters a request-rate bottleneck of the PCIe interface, limiting further performance improvements. The PCIe protocol includes significant per-request overhead beyond its payload. In our case, the response data size is only one cache line (64 bytes), but there is a 24-byte PCIe packet header added to each transaction, a 38% overhead. In addition, the device must send PCIe reads to access the descriptors, followed by writes of the actual response data into host memory, and then writes of request completion indications. This significantly increases the number of PCIe transactions required to serve each device access request, compared to the prefetch-based mechanism. Although we took significant steps to minimize the overheads (e.g., burst request reads to amortize the PCIe transaction costs), the sheer number of small requests wastes the available PCIe bandwidth. As a result, of the 4GB/s theoretical peak of our PCIe interface, the eight-core system is only able to use 2GB/s (50%) to transfer useful data.

Impact of MLP. Figure 9 shows the performance of the application-managed queues in the presence of MLP. Similar to prefetch-based systems, the relative performance compared to the DRAM baseline with MLP is noticeably worse than without MLP. The queue management overhead in software increases with the number of device accesses, even when the accesses are batched before a context switch. As a result, the peak performance of the application-managed queues on a workload with MLP of 2.0 is 45%, relative to the DRAM baseline. Going to an MLP of 4.0, the impact is even greater, lowering the normalized peak performance to only 35% of the corresponding DRAM baseline. Figure 9b compares the behavior of workloads with different MLPs for four cores. The increased amount of data transfer per unit of work puts greater strain on the PCIe bandwidth, reaching peak performance at four cores (instead of eight cores in the case of MLP of 1.0) and much more quickly (at below 16 threads) for MLP of 4.0.

These results show that the impact of MLP on application-managed queues is even more severe than on the prefetch-based design. With an MLP of 4.0, the queue management overheads and PCIe bandwidth constraints limit the four-core system to just 1.3× performance relative to the DRAM baseline. We do not include the results for 2µs and 4µs latencies for brevity; they are analogous, achieving identical peaks to 1µs, but at proportionally higher thread counts.

Implications. On systems where prefetch-based access is limited by hardware queues, application-managed queues can present a scalable solution to the killer-microsecond problem. However, they have significantly higher software overheads that primarily arises from queue management. Performance is determined by the number of parallel requests, which in turn is limited by the queue management overhead and interconnect bandwidth. Given the current trends of increasing bandwidth with every new generation of PCIe and other interconnects—hundreds of gigabytes per second are not out of reach—the bandwidth is not likely to be major bottleneck for high-end devices. Queue-management overheads, however, are not easily remedied, as software must work to submit requests and to check for completion notifications from the device.

Additionally, from a programmability point-of-view, prefetching has a clear advantage over application-managed queues: hardware support for cache coherence. If microsecond-latency storage is to be treated as memory, they should support both read and write requests from multiple processor cores. In the case of prefetching, the device data is stored in hardware caches and kept coherent across cores in the event of a write. With the software-managed queues, every read or write request to the device will use a different DRAM location for the response data, making hardware-based coherence moot. Supporting writes with software-managed queues will either preclude sharing across cores, or require complex software cache coherence, neither of which is palatable to software developers seeking easy-to-use, high-performance memory interfaces.



(a) Prefetch-based access, 1 Core.

(b) Application-managed queues, 1 Core.

(c) Prefetch-based access, 8 Cores.

(d) Application-managed queues, 8 Cores.

Fig. 10: One- and eight-core performance at 1µs latency for the application benchmarks, shown alongside a 4-read microbenchmark for comparison.

D. Application Case Studies

To confirm that our microbenchmark results are broadly applicable to more practical applications, we extracted the main data-access code of three open source data-intensive applications (Section IV-C) and compared their performance trends to the microbenchmark. For the baseline DRAM implementations of these applications, data accesses are performed on-demand on DRAM without software modification. For the device-based implementations, we modify the applications to batch device requests. The nature of the applications permits batches of four reads for Memcached and Bloomfilter, but limits us to two reads for BFS due to inherent data dependencies.

Figure 10 presents the relative performance of the applications using the prefetching and application-managed queues, with 1µs device latency, normalized to their respective DRAM baselines. It is immediately evident that the DRAM baselines allows the cores to find MLP and issue memory accesses in parallel, just as we are able to do through our manual batching effort. As a result, the application behavior is very similar to the microbenchmark behavior in the presence of MLP.

Comparing Figures 10a and 10b shows the effectiveness of the two access mechanisms (prefetch vs. queue). Although the single core performance of the prefetch-based access is always limited by the LFBs, it can still achieve adequate performance (between 35% to 65% of the DRAM baseline) before reaching the LFB limit. On the other hand, due to software overheads, the application-managed queues only reach 20% to 50% of the baseline performance with a single core.

Comparing Figures 10c and 10d reveals the scalability of the access mechanisms while running on eight cores. The hardware queue limitations fundamentally prevent the prefetch-based access from achieving adequate application performance. On the other hand, while the scalability of the application-managed queues is not inherently limited, the aggressive use of interconnect bandwidth for queue management takes a significant toll on the peak performance that can be achieved, with the final performance of the eight-core runs peaking at between 1.2x to 2.0x of the DRAM baseline performance of a single core.

Implications. Data-intensive applications exhibit some memory-level parallelism, exacerbating the need to support a large number of in-flight requests to microsecond-level devices. If the bottlenecks identified in the previous sections are lifted, we expect applications to show scalability while effectively hiding microsecond-level device latency with high thread counts. However, on existing hardware, microsecond-latency devices can achieve only modest performance (∼50% of baseline DRAM) on a single core. On multiple cores, modern systems fair extremely poorly compared to multi-core DRAM baselines.

VI. Related Work

Barroso [8] coined the phrase "Killer Microsecond" to refer to microsecond-level latencies for which we do not have appropriate latency-hiding mechanisms. To the best of our knowledge, our work is the first to measure the extent of this problem in existing server platforms. Inspired by the need for fast context switching mechanisms to hide latencies, Barroso et. al [27] propose a low-latency context caching mechanism in hardware. Such mechanisms are complimentary to our user-threading based methods, and make them more efficient, but are not a requirement. Sukhwani et al. [28] described an FPGA-based proprietary DDR emulator. Our emulation system is similar in spirit, but we selected the PCIe interface because it provides sufficient flexibility to experiment with both hardware- and software-managed queuing mechanisms in one platform.

There are many proposals for architecting NVM devices as something other than a simple drop-in replacement for block-based disks. The MMIO-based access mechanisms evaluated in this paper can be applied to any such proposal as long as they expose the storage/memory device to applications using load/store instructions for fine-grained accesses [29]–[41].

Although, this work only focuses on read accesses, a significant number of proposals on NVM devices are motivated by their write-related properties, such as write endurance, write latency, and persistence and consistency issues. Some proposals target efficient designs to support transactional semantics for persistent updates [31], [34]–[38], [40], [42], [43] to improve the performance in the presence of various atomicity and consistency requirements. Some proposals are motivated by wear-leveling requirements and limited write endurance of Flash and other NVM devices [30], [44], [45], or their high write latencies [46], [47]. These techniques often absorb the writes into faster storage elements, typically DRAM, that act as a cache with respect to the NVM device. Finally, some modern processors now include instructions to support proper handling of cached persistent data [48].

VII. Conclusion

Although emerging technologies can accommodate the need for vast amounts of low-latency storage, existing mechanisms for interacting with storage devices are inadequate to hide their microsecond-level latencies. In this work, we use a microsecond-latency device emulator, a carefully-crafted microbenchmark, and three data-intensive applications to uncover the root causes of this problem in state-of-the-art systems. We then show that simple changes to existing systems are sufficient to support microsecond-latency devices. In particular, we show that by replacing memory accesses with prefetches and user-mode context switches and enlarging hardware queues for in-flight accesses, conventional architectures can effectively hide microsecond-level latencies and approach the performance of DRAM-based implementations of the same applications. In other words, we show that successful usage of microsecond-level devices is not predicated on employing drastically new hardware and software architectures.

It is worth mentioning that, in this paper, we only investigated the performance impact of microsecond-latency device reads, and did not consider writes. Fortunately, because writes do not have return values, are often off the critical path, and do not prevent context switching by blocking at the head of the reorder buffer, their latency can be more easily hidden by later instructions of the same thread without requiring prefetch instructions. On the other hand, as discussed in Section V-C, write operations have significant programmability implications, and should be further investigated in future work.

Acknowledgements

This work was supported by a Google Faculty Research Award and NSF Grant #1452904. The experiments were conducted with equipment donated by Intel Corporation and purchased through NSF Grant #1405641.

References

[1] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. Manasse, and R. Panigrahy, "Design Tradeoffs for SSD Performance," in *Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404019

[2] M. Technologies, "InfiniBand Performance," http://www.mellanox.com/page/performance_infiniband.

[3] M. Technologies, "RoCE vs. iWARP Competitive Analysis," http://www.mellanox.com/related-docs/whitepapers/WP_RoCE_vs_iWARP.pdf.

[4] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N. S. Islam, H. Subramoni, and D. K. Panda, "Performance Analysis and Evaluation of InfiniBand FDR and 40GigE RoCE on HPC and Cloud Computing Systems," in *Proceedings of 20th Symposium on High-Performance Interconnects*, ser. HOTI '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: http://dx.doi.org/10.1109/HOTI.2012.19

[5] Intel, "Intel and Micron Produce Breakthrough Memory Technology," https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology, July 2015.

[6] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," in *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2150976.2150982

[7] C. Li, C. Ding, and K. Shen, "Quantifying the Cost of Context Switch," in *Workshop on Experimental Computer Science*, 2007. [Online]. Available: http://doi.acm.org/10.1145/1281700.1281702

[8] L. A. Barroso, "Landheld Computing," ISSCC 2014 Panel on "Data Centers to Support Tomorrowś Cloud". http://www.theregister.co.uk/Print/2014/02/11/google_research_three_things_that_must_be_done_to_save_the_data_center_of_the_future.

[9] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the Killer Microseconds," *Commun. ACM*, vol. 60, Mar. 2017. [Online]. Available: http://doi.acm.org/10.1145/3015146

[10] I. T. R. for Semiconductors, "Process Integration, Devices, and Structures," 2011.

[11] L. M. Grupp, J. D. Davis, and S. Swanson, "The Bleak Future of NAND Flash Memory," in *Proceedings of 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208463

[12] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *Annual Technical Conference*, ser. USENIX ATC'13. Berkeley, CA, USA: USENIX Association, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2535461.2535468

[13] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of 2010 International Conference on Management of Data*, ser. SIGMOD '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[14] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of 2nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1863103.1863113

[15] B. Fitzpatrick, "Distributed caching with memcached," *Linux J.*, vol. 2004, Aug. 2004. [Online]. Available: http://dl.acm.org/citation.cfm?id=1012889.1012894

[16] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling memcache at facebook," in *Proceedings of 10th USENIX Conference on Networked Systems Design and Implementation*, ser. nsdi'13. Berkeley, CA, USA: USENIX Association, 2013. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482626.2482663

[17] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, G. Parulkar, M. Rosenblum, S. M. Rumble, E. Stratmann, and R. Stutsman, "The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM," *SIGOPS Oper. Syst. Rev.*, vol. 43, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1713254.1713276

[18] G. Blake and A. G. Saidi, "Where does the time go? characterizing tail latency in memcached," in *Intl. Symposium on Performance Analysis of Systems and Software*, March 2015.

[19] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "Thin servers with smart pipes: Designing soc accelerators for memcached," in *Proceedings of 40th International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2485922.2485926

[20] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken, "Tam-a compiler controlled threaded abstract machine," *J. Parallel Distrib. Comput.*, vol. 18, Jul. 1993. [Online]. Available: http://dx.doi.org/10.1006/jpdc.1993.1070

[21] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung, "The mit alewife machine: Architecture and performance," in *Proceedings of 22nd International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: ACM, 1995. [Online]. Available: http://doi.acm.org/10.1145/223982.223985

[22] "Flashtec NVRAM Drives," https://www.microsemi.com/product-directory/storage-boards/3690-flashtec-nvram-drives.

[23] "Intel-Altera Heterogeneous Architecture Research Platform Program," http://bit.ly/1Pwo9IM.

[24] R. S. Engelschall, "GNU Pth - The GNU Portable Threads," https://www.gnu.org/software/pth/.

[25] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, 2010.

[26] A. Vorobey and B. Fitzpatrick, "memcached - memory caching daemon."

[27] L. A. Barroso, J. Laudon, and M. R. Marty, "Low latency thread context caching," Jul. 5 2016, uS Patent 9,384,036. [Online]. Available: https://www.google.com/patents/US9384036

[28] B. Sukhwani, T. Roewer, C. L. Haymes, K.-H. Kim, A. J. McPadden, D. M. Dreps, D. Sanner, J. Van Lunteren, and S. Asaad, "Contutto: A novel fpga-based prototyping platform enabling innovation in the memory subsystem of a server class processor," in *Proceedings of 50th International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124535

[29] A. Badam and V. S. Pai, "SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy," in *Proceedings of 8th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'11. Berkeley, CA, USA: USENIX Association, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1972457.1972479

[30] A. Badam, V. S. Pai, and D. W. Nellans, "Better Flash Access via Shape-shifting Virtual Memory Pages," in *Proceedings of 1st Conference on Timely Results in Operating Systems*, ser. TRIOS '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2524211.2524221

[31] 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 *Proceedings of 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950380

[32] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System Software for Persistent Memory," in *Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: ACM, 2014. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592814

[33] J. Huang, A. Badam, M. K. Qureshi, and K. Schwan, "Unified Address Translation for Memory-mapped SSDs with FlashMap," in *Proceedings of 42nd International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2749469.2750420

[34] A. Kolli, S. Pelley, A. Saidi, P. M. Chen, and T. F. Wenisch, "High-Performance Transactions for Persistent Memories," in *Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '16. New York, NY, USA: ACM, 2016. [Online]. Available: http://doi.acm.org/10.1145/2872362.2872381

[35] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency," in *Proceedings of 41st International Symposium on Computer Architecture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2665671.2665712

[36] P. M. Programming, "Linux NVM Library," http://pmem.io/.

[37] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory," in *Proceedings of 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11. Berkeley, CA, USA: USENIX Association, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=1960475.1960480

[38] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *Proceedings of 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950379

[39] C. Wang, S. S. Vazhkudai, X. Ma, F. Meng, Y. Kim, and C. Engelmann, "NVMalloc: Exposing an Aggregate SSD Store As a Memory Partition in Extreme-Scale Machines," in *Proceedings of 26th International Parallel and Distributed Processing Symposium*, ser. IPDPS '12. Washington, DC, USA: IEEE Computer Society, 2012. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2012.90

[40] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the Performance Gap Between Systems with and Without Persistence Support," in *Proceedings of 46th International Symposium on Microarchitecture*, ser. MICRO-46. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540744

[41] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Proceedings of 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12. Berkeley, CA, USA: USENIX Association, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208464

[42] 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 *Proceedings of 22nd Symposium on Operating Systems Principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629589

[43] Z. Deng, L. Zhang, N. Mishra, H. Hoffmann, and F. T. Chong, "Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in nvms," in *Proceedings of 50th International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3123939.3124548

[44] E. Ipek, J. Condit, E. B. Nightingale, D. Burger, and T. Moscibroda, "Dynamically Replicated Memory: Building Reliable Systems from Nanoscale Resistive Memories," in *Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XV. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1736020.1736023

[45] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory As a Scalable DRAM Alternative," in *Proceedings of 36th International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555758

[46] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *Proceedings of 36th International Symposium on Computer Architecture*, ser. ISCA '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1555754.1555760

[47] V. Technology, "Non-Volatile Memory and Its Use in Enterprise Applications," http://www.vikingtechnology.com/uploads/nv_whitepaper.pdf.

[48] Intel, "Intel Architecture Instruction Set Extensions Programming Reference," https://software.intel.com/sites/default/files/managed/b4/3a/319433-024.pdf, February 2016.