Memory-Mapped I/O on Steroids

Anastasios Papagiannis*
Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS)
Heraklion, Greece
apapag@ics.forth.gr

Manolis Marazakis
Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS)
Heraklion, Greece
maraz@ics.forth.gr

Angelos Bilas*
Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS)
Heraklion, Greece
bilas@ics.forth.gr

Abstract

With current technology trends for fast storage devices, the host-level I/O path is emerging as a main bottleneck for modern, data-intensive servers and applications. The need to improve I/O performance requires customizing various aspects of the I/O path, including the page cache and the method to access the storage devices.

In this paper, we present *Aquila*, a library OS that allows applications to reduce I/O overhead by customizing the memory-mapped I/O (*mmio*) path for files or storage devices. Compared to Linux *mmap*, *Aquila* (a) offers full *mmio* compatibility and protection to minimize application modifications, (b) allows applications to customize the DRAM I/O cache, its policies, and access to storage devices, and (c) significantly reduces I/O overhead. *Aquila* achieves its *mmio* compatibility, flexibility, and performance by placing the application in a privileged domain, non-root ring 0.

We show the benefits of *Aquila* in two cases: (a) Using *mmio* in key-value stores to reduce I/O overhead and (b) utilizing *mmio* in graph processing applications to extend the memory heap over fast storage devices. *Aquila* requires 2.58× fewer CPU cycles for cache management in RocksDB, compared to user-space caching and *read*/*write* system calls and results in 40% improvement in request throughput. Finally, we use Ligra, a graph processing framework, to show the efficiency of *Aquila* in extending the memory heap over fast storage devices. In this case, *Aquila* results in up to 4.14× lower execution time compared to Linux *mmap*.

CCS Concepts: • **Information systems → Flash memory**; **Storage management**; • **Software and its engineering → File systems management**; **Memory management**.

*Also with the Department of Computer Science, University of Crete, Greece.

EuroSys '21, April 26–29, 2021, Online, United Kingdom

ACM ISBN 978-1-4503-8334-9/21/04…$15.00
https://doi.org/10.1145/3447786.3456242

Keywords: memory-mapped I/O, Linux mmap, fast storage devices, I/O caching, key-value stores

ACM Reference Format:
Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2021. Memory-Mapped I/O on Steroids. In *Sixteenth European Conference on Computer Systems (EuroSys '21), April 26–29, 2021, Online, United Kingdom.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3447786.3456242

1 Introduction

Fast storage devices provide high sequential throughput, high random IOPS, and low access latency. For instance, block-addressable NVMe devices attached to PCIe are typically capable of more than 500K random IOPS, with access latency lower than 10 μs [28]. Byte-addressable NVM devices attached to memory DIMMs are already capable of 300 *ns* access latency while providing 10*s* of GBs of throughput [31]. Despite these technology trends, modern data-intensive applications do not benefit proportionally: The I/O path is becoming a significant bottleneck in terms of overhead (CPU cycles) and scalability with the number of cores. Ideally, future servers should consume as many CPU cycles as possible for performing application processing rather than I/O. Today, we are far from this ideal situation.

To improve I/O performance, data-intensive applications resort to customizing the I/O path: Typically, I/O-intensive applications use a user-space cache (Figure 1(b)) instead of the system-wide kernel-space buffer cache (Figure 1(a)). The former approach avoids frequent system calls to the kernel, thus reducing I/O overhead. In addition, a user-space cache allows for custom policies and can be combined with user-space access to dedicated storage devices via libraries such as SPDK [61] to eliminate the use of system calls and the involvement of the kernel.

However, even in these cases, each data access requires a lookup in the I/O cache for *all operations, including hits.* Therefore, even in the case of hits, I/O cache lookups result in high CPU overhead for cache management (in user or kernel space). Harizopoulos et al. [24] claim that management of a user-space cache consumes about one-third of the total CPU cycles of a database system running OLTP workloads. Papagiannis et al. [48] claim that about half of the total CPU cycles are spent on cache management in RocksDB [20], a widely used persistent key-value store.

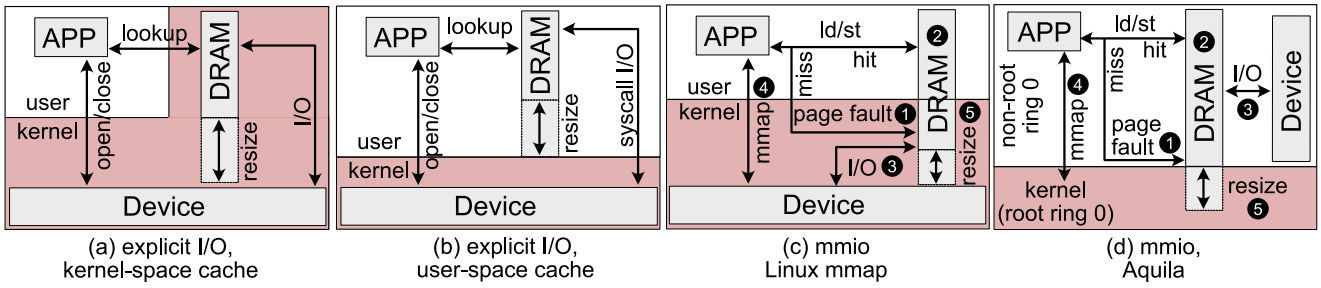


Figure 1. Storage cache configurations. Transitions to the kernel (dark/red color) require a protection domain switch.

Memory-mapped I/O through memory-mapped files or devices, which we refer to as *mmio* for the rest of the paper, has the potential to reduce I/O stack overheads over fast storage devices by eliminating the cost of cache hits [1, 7, 11, 18, 27, 45, 47, 48, 51, 58]. I/O cache hits in *mmio* are handled via virtual memory mappings and do not incur any software overhead: The application maps a storage device (or a file) in the process virtual address space and the user can access it via processor *load*/*store* instructions. In the case of a hit, a valid mapping in the page table already exists. The virtual to physical translation is handled entirely in hardware.

However, *mmio*, as provided by Linux *mmap*, has several shortcomings: (a) It uses a single, shared, kernel-level I/O page cache that does not allow easy customization. (b) It does not allow customizing I/O requests during misses and evictions, as these are handled by the kernel. (c) It requires expensive page faults in the case of misses (Figure 1(c)). Page fault handlers modify virtual memory mappings in the page table and fetch new data in memory. These shortcomings render Linux *mmap* impractical for many applications that could benefit from a custom *mmio* path.

Recent efforts to improve the *mmio* path have focused on providing a custom path in the Linux kernel, such as DI-MMAP [18] and FastMap [50], replacing *mmap* for specific uses. However, customizing the *mmio* path in the Linux kernel is not straight-forward, creates new limitations, is not easy to deploy, and cannot satisfy diverse application requirements. Moreover, even with a custom in-kernel *mmio* path, performance can still suffer due to the high overhead for I/O cache misses that require expensive page faults.

In this paper, we design and implement *Aquila*, a library OS that allows applications to reduce software I/O overhead by using a custom *mmio* path without the shortcomings of *mmap*. *Aquila* offers applications an *mmio* interface compatible with *mmap* which requires minimal changes. Applications are able to provide their own *mmio* handling, e.g., by using a custom I/O page cache or device access method. In addition, *Aquila* improves performance compared to *mmap* by optimizing *mmio* operations in the common path. *Aquila* is collocated with the application in a privileged execution domain (non-root ring 0) along with common-path *mmio* functionality, while still offering full *mmio* functionality by interacting with the hypervisor (Figure 1(d)). Essentially, *Aquila* runs the application as a guest OS and provides full protection semantics by leveraging hardware support for virtualization [4, 62].

In our work, we first observe that although I/O cache hits are free with *mmio*, misses are expensive and trigger five main operations (Figure 1(c)): ❶ Handle page-faults; ❷ perform DRAM I/O cache replacements, including evictions and write-backs; and ❸ access the I/O device. Additionally, *mmio* needs to ❹ create or destroy file mappings (i.e. *mmap* and *munmap*), and ❺ dynamically resize the DRAM cache. Then, we observe that these operations occur with different frequencies: ❶–❸ are common path operations while ❹–❺ are less frequent, but are still required for full *mmio* functionality.

Based on these observations, *Aquila* places common-path operations, ❶–❸ (page faults, I/O cache replacements, device access) and the application in the same context, non-root ring 0 to allow both for customization of the common *mmio* path and to reduce overhead without compromising protection. At the same time, *Aquila* provides operations ❹–❺ (coarse-grain memory allocation and file mapping management) by interacting with the hypervisor, allowing for the full *mmio* functionality and minimizing the required modifications to application code, without incurring a significant performance penalty.

Overall, the contributions of this paper are:

1. We design and implement a new approach, based on *mmio*, which, similar to user-space I/O caches, enables customization of the I/O path (page cache structure, mechanisms, policies, device access) while, similar to *mmio*, it eliminates the cost for cache hits.
2. We provide this flexibility at low overhead, by placing common path operations in non-root ring 0 and only requiring interaction with the hypervisor for uncommon path operations.
3. As opposed to Linux *mmap* that is used mainly for shared library and executable management, we provide a scalable I/O page cache tailored for I/O-intensive applications.

4. We provide direct access to storage devices from non-root ring 0 using SPDK and remove the need for expensive *syscall*/*vmcall* operations in our customizable *mmio* path.
5. We offer compatibility with Linux *mmap*, allowing its use in different cases. We show the effectiveness of our approach in two scenarios: (i) storage I/O in key-value stores that are broadly used today and (ii) extending the application heap over fast storage devices for handling large datasets.

We implement *Aquila* in Linux and evaluate its efficiency with micro-benchmarks and real applications. For RocksDB [20], *Aquila* requires 2.58× fewer CPU cycles for cache management compared to user-space caching and *read*/*write* system calls, and results in up to 40% higher throughput. *Aquila* reduces the average page fault latency by 45.3% compared to Linux *mmio*. We show that *Aquila* achieves both low average and tail latency, while maintaining high device throughput. We also examine the use of *mmio* for extending the heap of Ligra [57], a graph-processing, data-intensive application over fast storage devices. *Aquila* reduces execution time up to 4.14× compared to *mmap*, with minimal modifications to the application, and only during initialization.

2 Background

In this section we provide a brief summary of Linux *mmio* [42] and Intel VT-x [62] virtualization extensions.

2.1 Linux *mmap*

The Linux kernel *mmap* system call creates new virtual memory mappings to physical memory for a specific process. Mappings can be either anonymous or backed by files. Anonymous mappings are private to each process and are mainly used for user, heap-based memory allocation, i.e. *malloc*. In this paper we examine I/O over persistent storage, an inherently shared resource. Therefore, we consider only shared memory mappings backed by a file or block device, as also required by Linux *mmio*.

File-backed mappings can be either private to each process or shared among processes. Private file mappings are used to load executables and shared libraries. These are typically mapped with read-execute permissions in the text segment and include portions mapped with read-write permissions in the data segment. Any modification to the data segment must not reach the underlying file. For this reason, these are Copy-On-Write mappings. On the other hand, shared file mappings persist after a process exits or a failure occurs, and therefore, are appropriate for storage purposes. In this paper we target only shared file-backed mappings.

2.2 Intel VT-x CPU Virtualization

Intel VT-x is a set of processor hardware extensions to accelerate the operation of hypervisors. The *x86_64* CPU has two major operating modes, VMX root and VMX non-root. VMX root is similar to privileged non-virtualized CPU operation; the hypervisor and the host OS typically run in this mode. VMX non-root mode is used to run the guest operating system. In this mode, CPUs have protection limitations for privileged instructions. Both VMX root and VMX non-root support a separate set of protection rings, where ring 0 is the most privileged and ring 3 is the least privileged. Commonly, the operating system (host and guest) runs in ring 0 and user applications run in ring 3. Ring 1 and ring 2 are not used in modern operating systems. Figure 2 shows the different CPU modes and rings.

Executing *vmlaunch* or *vmresume* CPU instructions from VMX root changes the mode to VMX non-root, and starts or continues to execute guest OS code. This transition is named *vmentry*. The hypervisor handles privileged instructions after a transition from VMX non-root to VMX root mode. This transition is named *vmexit*. *vmexits* occur upon events predefined by the hypervisor. These events and several other configuration options are stored in the per-CPU VM Control Structure (VMCS) memory buffer. Besides the predefined events (privileged instructions), the guest can generate a *vmexit* explicitly by issuing a *vmcall* instruction. For both *vmentry* and *vmexit* events, processor hardware handles the steps for saving and restoring architectural state to/from VMCS.

An important aspect of Intel VT-x [62] is the use of *Extended Page Tables (EPTs)* which accelerate address translation, as follows. When the guest is executing, there are two levels of address translations: (i) From Guest Virtual Address (GVA) to Guest Physical Address (GPA), using regular page tables without requiring a *vmexit*; (ii) From Guest Physical Address (GPA) to Host Physical Address (HPA) requiring a *vmexit*. EPTs accelerate this second step under the control of the hypervisor. During the access of a GPA, if the translation does not exist, an EPT-fault occurs and the hypervisor handles the fault in a way similar to regular page faults.

3 *Aquila* Design

Aquila provides a fast, customizable I/O path by collocating I/O functionality and the application in non-root ring 0 (Figure 2). To provide fast access to data, *Aquila* uses *mmio* which eliminates the cost of hits to the I/O cache and handles page faults that are necessary for misses in non-root ring 0 (❶). To allow for customization (mechanisms and policies), *Aquila* places the I/O page cache in non-root ring 0 (❷), and allows for different methods to access storage devices (❸) from non-root ring 0. Finally, to provide the full *mmio* functionality, *Aquila* offers management of file mappings (❹) and the ability to resize the I/O cache (❺). Although these operations require interaction with the hypervisor, they do not affect performance as they occur less frequently. Figure 2(right) shows where *Aquila* places the application, as

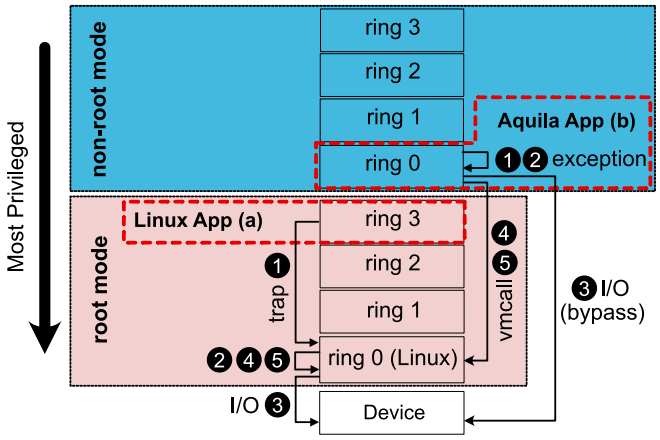


Figure 2. Protection rings and operation path in Linux (left) and *Aquila* (right).

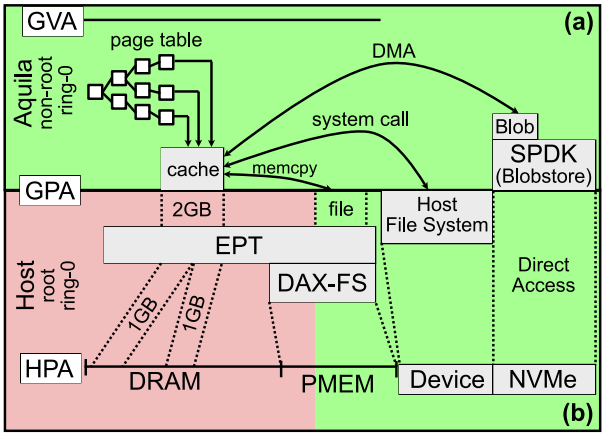


Figure 3. Page tables, device I/O, DRAM cache, and DRAM allocation in *Aquila*.

opposed to common Linux applications (Figure 2(left)). Next, we discuss each of these operations in more detail.

3.1 Handling Page-Faults for I/O Cache Misses

I/O cache misses in *mmio* require expensive page faults in the kernel, due to protection domain switching to update protected resources, such as the page tables and TLB. Page faults in *mmio* are required to modify virtual to physical memory mappings. Essentially, these mappings allow *mmio* to eliminate the cost for I/O cache hits. *Aquila* collocates the application and the DRAM cache in non-root ring 0, in order to allow customization of the *mmio* path, as well as to reduce the cost of page faults. However, handling page faults in non-root ring 0 introduces significant challenges. We continue with the presentation of our design, and discuss implementation challenges in more detail in Section 4.

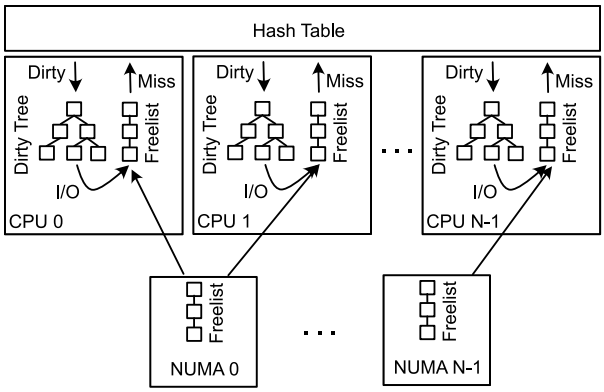


Figure 4. DRAM cache organization in *Aquila*.

Page faults are handled via a page table, as specified by the underlying architecture and OS. *Aquila* uses a shared page table for all threads of each process to store these memory mappings, similar to modern OSes. Figure 3 shows that *Aquila*'s page table resides in non-root ring 0 and translates application virtual addresses to DRAM cache pages. With Intel VT-x [62] there are two levels of virtual to physical address translation. First, in VMX non-root mode, a regular page table translates from Guest Virtual Address (GVA) to Guest Physical Address (GPA). In VMX root mode, an *Extended Page Table (EPT)* translates from Guest Physical Address (GPA) to Host Physical Address (HPA). *Aquila* maps GVA addresses to GPA addresses.

Today, page faults switch from ring 3 to ring 0 as follows. For user applications, page fault exceptions occur in ring 3. The page fault initiates a protection domain switch to ring 0, changes the stack, and stores state information about the process exception in the kernel stack. Then, execution branches to the appropriate fault handler. Upon page fault completion, control returns to the user-space code in ring 3 via the *iret* instruction.

In *Aquila*, the application already runs in ring 0 using virtualization support. Page fault exceptions occur in ring 0 and a protection domain switch is not required. Execution branches directly to the appropriate fault handler. Upon page fault completion, control returns to the application without the need for a transition from ring 0 to ring 3. Handling page faults in non-root ring 0 requires dealing with TLB invalidations and handling exception handler execution. We discuss these two aspects in more detail in Section 4.

3.2 DRAM Cache Design

A DRAM cache is necessary for hosting data fetched from I/O devices. FastMap [50] has shown that the Linux kernel buffer cache used by *mmap* does not scale well with increasing the number of application threads. Therefore, *Aquila* aims to reduce contention and improve DRAM cache scalability. The main observation of FastMap [50] is that, unlike Linux, dirty

pages need to be maintained in a separate structure from clean pages. Figure 4 shows the organization of the DRAM cache in *Aquila*, with the following structures and operations that target specifically the caching functionality required by *mmio*.

Cache lookups: Cache lookups are an important component of the *mmio* path. *Mmio* eliminates the software cost for cache lookups in the case where a page resides in the cache. In the case of a page fault, the handler first checks if the requested page is in the DRAM cache. Although this is a page fault, it may occur that upon checking the DRAM cache as part of the page fault handling routine, the page has been brought in the cache. For this reason, the handler uses a lock-free hash table to perform a fast lookup, similar David et al. [16]. If the requested page resides in the DRAM cache, *Aquila* creates a mapping to the page table between the faulting virtual address and the physical page. If the page is not present in the cache, it (1) allocates new cache pages from the freelist, (2) evicts pages from the cache, (3) cleans dirty pages, and (4) issues write/read I/O to the underlying device, as follows.

Freelist and evictions: *Aquila* uses a hierarchical 2-level freelist to manage DRAM cache pages, as follows. The first level consists of a queue per NUMA node, while the second level of a queue per core. When a page is required, the core checks, in order, its local (core) queue, the local NUMA node queue, and the remote NUMA node queues. All queues contain free pages. If all queues are empty, *Aquila* tries to evict a batch of pages (512) synchronously. We choose which pages to evict via an approximation of *LRU*. *Aquila* updates the *LRU* list based on page faults. When a page is evicted from the cache, it is placed in the local core queue. If the number of pages in the local core queue exceeds a threshold, they are moved to the appropriate NUMA queue. All page movement between first and second level queues is performed in batches (4096 pages in our evaluation). By implementing lock-free freelist queues and using batching in our two-level allocator, we do not observe high contention.

Dirty page write-back: *Aquila* maintains dirty pages in a data structure separate from the hash table to accelerate writeback and *msync* operations. We track dirty pages via page faults similar to FastMap [50]. In the case of a read fault, we map a page as read-only. A write on this page results in an additional page fault where we only mark the page as dirty. In the case of a write fault we also mark the page as dirty during the initial page fault. Dirty pages need to be sorted by device offset and this is not facilitated by our hash table. For this reason, and to reduce contention on a single lock, we use per-core red-black trees. When a page is selected for replacement, we write pages to the device in the order defined by their page offsets. Having multiple sorted red-black trees simplifies merging of pages in larger I/Os for writebacks similar to the Linux kernel [42]. For reads, we require synchronous I/Os and cannot apply any batching. Based on the possibly supplied *madvise* arguments we also perform read-ahead to improve sequential reads.

3.3 Device I/O

Device I/O typically requires kernel involvement and incurs high overheads. A user-space storage cache requires expensive system calls. In Linux *mmio*, as a page fault requires a trap to the kernel, the I/O does not require an additional protection domain switch. However, since *Aquila* moves the application into non-root ring 0, serving I/Os in the kernel (root ring 0) could require a *vmcall*, which is even more expensive compared to system calls. *Aquila* removes the need for an additional protection domain switch with direct access to storage devices from non-root ring 0. *Aquila* allows user applications to customize device I/O based on their performance and flexibility. This spans from a common file system with strict POSIX semantics to user-space optimized frameworks. This section provides details on using direct access to fast block-addressable and byte-addressable devices, and specific optimizations for each case.

Figure 3 shows the I/O path in *Aquila* and how it provides direct access to storage devices. Next, we discuss how *Aquila* achieves this for both block-addressable storage devices, such as PCIe-based (NVMe), and for byte-addressable, DIMM-based, non-volatile memories (NVM).

Direct access to NVMe: We use SPDK [61], a broadly-used user-space framework to access NVMe devices. *Aquila* provides a system call interception and file abstraction over SPDK and applications can run with minimal changes. Using SPDK, *Aquila* bypasses the host OS and issues I/O operations directly to devices (Figure 3*(a – right)*). Direct access requires that the devices are not shared with other processes and also that they are visible directly from *Aquila* at non-root ring 0. This is the case, e.g., with modern NVMe devices attached to PCIe, because device configuration registers can be mapped directly to user space.

To provide applications with a file abstraction, we leverage *Blobstore* [60]. *Blobstore* provides a flat namespace of *blobs*, where each *blob*, identified by a unique number, can be created/resized/deleted at runtime, and also supports extended attributes. *Aquila* supports the translation from files to blobs transparently. For this purpose, we intercept *open* and *mmap* calls in non-root ring 0. *Aquila* uses the direct I/O path of *Blobstore*, which does not buffer data, as opposed to *BlobFS* [59], which buffers data in its local DRAM cache.

Direct device access requires dedicated devices for protection purposes. This can also take the form of dedicated device partitions. Today, such an approach is common for systems such as key-value stores and data processing frameworks that fully manage their storage and data on their own.

Direct access to NVM:. Byte-addressable, NVM devices can be accessed through processor *load*/*store* instructions. When using NVM as a storage device, it can be either mapped directly to the program address space or used as a backing device for a DRAM I/O cache. The two approaches have different tradeoffs for access latency and throughput [31]. *Aquila* targets setups that provide DRAM caching over byte-addressable NVM devices. *Aquila* maps NVM in the application address space in non-root ring 0 and uses the *Direct Access (DAX)* framework for device access purposes (Figure 3*(a – left)*). For I/O we use memory copy (*memcpy*) between DAX-*mmap*ed files and our DRAM cache. We provide protected sharing of NVM between different processes and forward all metadata operations to the host OS.

Aquila uses an effective optimization for memory copies that transfer data between DRAM and NVM, as follows. Memory copies for reads have a size of $4KB$, equal to the page size. The Linux kernel cannot use SIMD instructions for memcpy because this requires a full FPU state save and restore, which is extremely costly. For *SSE* it has to save 512 bytes and for *AVX* 832 bytes. We measure the cost to save and restore *AVX* state using the *XSAVEOPT* and *FXRSTOR* instructions to be around 300 cycles. Furthermore, we measure the cost of a $4KB$ memcpy, without using SIMD instructions to be about about 2400 cycles. Instead, an optimized memcpy of $4KB$ using *AVX2* streaming (i.e. processor cache bypass) instructions requires about 900 cycles. With the cost of save and restore FPU state this increases to 1200 cycles, i.e. 2× faster than non-SIMD memcpy. For these reasons *Aquila* uses an *AVX2* optimized memcpy and pays the cost of saving and restoring FPU state only in the case of page faults that require a memory copy.

Finally, *Aquila* can use other approaches for I/O as well. These include the common synchronous *read*/*write* system calls and asynchronous approaches, such as *libaio* or *io_uring* [32] (Figure 3). All these strike a different balance between sharing and performance. We leave the evaluation of different configurations for future work.

3.4 File-mapping Management

Virtual address range update operations, such as *mmap*, *munmap*, and *mremap* are used to create, destroy, expand, or shrink mappings to files or devices. These operations happen less frequently. Virtual address lookups happen frequently in the common path (operation ❶), because every page fault checks if the faulting address refers to a valid, properly mapped, virtual address. Therefore, the challenge in *Aquila* is to perform virtual address range updates using root ring 0, while supporting efficient lookups in non-root ring 0. We should handle virtual address range updates in *Aquila* without interacting with the host OS, except from virtual memory operations not related to file mappings.

Linux, uses a red-black tree to keep all active Virtual Memory Areas (VMA), protected by a read-write lock. Operations that modify address ranges (*mmap*, *munmap*, and *mremap*), need to acquire this as a write lock, while page faults acquire it as read lock. Other work [8, 12, 13] has shown that this lock can limit scalability in servers with a large number of cores, even in cases where it is acquired as a read lock.

For this reason, *Aquila* uses a radix tree, similar to RadixVM [13], instead of a balanced tree to avoid contention and provide scalable manipulation and access of virtual address ranges. In the case of page faults, the radix tree is used for two purposes: (1) check if the page fault occurred in a valid address and (2) lock the specific entry to avoid concurrent modifications for the same page.

RadixVM uses per-core page tables in order to keep metadata about which TLB contains specific mappings and enable targeted TLB invalidations. RadixVM targets anonymous mappings where the design tradeoffs are different. We choose to have a single page table shared by all cores, similar to what common OSes do. This approach reduces the number of total page faults and as we use a batched TLB shootdown approach it does not negatively affect the performance.

Finally, we do not use the RadixVM *refcache* mechanism for page reference counting. We provide explicit page management as described in the previous section. In the cases where reference counting is required (i.e. radix tree metadata), we use a single shared reference count which does not incur significant overhead, as it is not in the common path.

3.5 Dynamic Cache Resizing

Aquila provides the ability to resize its DRAM cache dynamically, similar to the Linux page cache. This occurs less frequently, compared to other operations during *mmio*. Therefore, its cost is of secondary importance.

Figure 3 shows this path. The host operating system is responsible to allocate additional DRAM to *Aquila* and reclaim it, when needed. *Aquila* uses a set of ranges in Guest Physical Addresses (GPAs) for its DRAM cache. An *Extended Page Table (EPT)* translates GPA to to Host Physical Address (HPA). Accesses to a GPA, where an *EPT* mapping does not exist, result in an *EPT* fault in the host OS. This is similar to common page faults but has higher cost due to the required *vmexit*. Similar to Dune [4], during an *EPT* fault *Aquila* checks the normal page table to determine if the access is valid and then adds the translation to the *EPT*. *Aquila* reduces the number of *EPT* faults with huge pages only for GPA to HPA translations. We support both $2MB$ and $1GB$ pages; in our evaluation we only use $1GB$ pages for cache resizing purposes. Allocating cache in multiples of $1GB$ does not result in any DRAM utilization issues. *Aquila* can also support both $2MB$ and $1GB$ pages in other cases. For GVA to GPA translations we use only regular $4KB$ pages to provide fine grain accesses to the application.

In *Aquila*, similar to common OSes, all threads of a process share the same page table; therefore we use a single *EPT* per

process. *Aquila* modifies *EPT* management in Dune to replace its single *EPT* per thread with a single *EPT* per process.

4 *Aquila* Implementation

Aquila uses Dune [4] to access and configure Intel VT-x extensions. Dune uses hardware virtualization extensions and provides direct and protected access to hardware features, such as rings, page tables, and tagged TLBs.

Aquila consists of about 20*K* lines of both *C* and *C++* source code, excluding third-party libraries. This code handles virtual address range management, DRAM caching, including the dedicated page allocator, dirty page management, *LRU* eviction, and I/O to and from devices. It also handles page faults and intercepts system calls. We are able to run user applications, such as RocksDB, with minimal changes. These changes are a single function call where *Aquila* initializes its context during the application startup (i.e. in main function) and a single function call for each new thread to switch it over to the *Aquila* mode.

4.1 Batched TLB invalidations

Section 3.1 provides details on how *Aquila* adds new translations in the page table during page faults. An additional important operation is the modification and removal of mappings. This is required in the case of page evictions or when updating protection flags in existing mappings (i.e. *mprotect*). These operations require a TLB invalidation. In *x86_64*, each CPU can only invalidate its local TLB. *x86_64* provides Inter-Processor Interrupts (IPIs) so the OS can notify other cores to invalidate their TLB (aka TLB shootdown). Other work [2, 3] has shown that for anonymous mappings this can limit scalability with high core counts. An optimized *mmio* path for shared mappings has also shown similar scalability issues [50] for TLB shootdowns. Handling TLB shootdowns in VMX non-root ring 0 introduces challenges for both correctness and performance. To reduce this cost, *Aquila* uses a batched TLB shootdown approach. We remove the mappings for multiple pages (512 in our evaluation) and send a single TLB invalidation for all pages. We use posted IPIs as provided by hardware virtualization extensions, together with a mechanism similar to Shinjuku [33]. Shinjuku sends and receives IPIs without the need of a *vmexit*, by mapping the Advanced Programmable Interrupt Controller (APIC) directly to the user.

An additional problem to address in *Aquila*, as it targets unmodified user applications, is the case where a malicious process performs a denial-of-service attack by issuing a large number of interrupts to a specific core. To address this, we choose to use a *vmexit* in the send path by writing to a Model-Specific Register (MSR) register. A *vmexit* transfers control to a protected domain (i.e. hypervisor), where we can limit the rate of interrupts and avoid a denial-of-service attack. Requiring a *vmexit* on the send path results in increasing the cost from 298 to 2081 cycles [33]. However, due to batching this cost is amortized over the entire batch and we show in our evaluation that this is negligible compared to other costs (Figure 8(b)). Finally, similar to Shinjuku, *Aquila* uses the *vmexit*-less receive path.

4.2 Exception stack management

Aquila uses two exception handlers: one for page faults and one for Inter-Processor Interrupts (IPIs). Using the same stack for both user data and exception handling in *Aquila* (non-root ring 0) can cause corruption due to the *red zone* compiler optimization.

The *red zone* is a fixed-size area in each function stack frame, beyond the current stack pointer. A function may use the *red zone* to store local variables without the additional overhead of modifying the stack pointer. The x86-64 ABI uses a 128-byte *red zone*, starting directly under the stack pointer. Corruption may occur when an interrupt/exception is triggered and the user code is using the *red zone*: The handler will overwrite the *red zone* and will corrupt user data. For this reason, OS code is compiled with the *red zone* disabled to avoid this type of data corruption. In addition, Linux currently uses a separate stack for exceptions that can occur while applications are executing, these being: Double Fault Exceptions, non-maskable interrupts, hardware debug interrupts, and Machine Check Exceptions. In *x86_64* there can be up to seven alternative stacks.

Aquila uses two additional stacks for its two handlers. *Aquila* handlers run with interrupts enabled, because e.g., in memory-mapped I/O a page fault can lead to an I/O operation, which can take several thousands of cycles to complete. Delaying IPIs during a prolonged interval would negatively affect the performance of other threads. For this reason, similar to Linux, in both *Aquila* handlers, we start by disabling interrupts, then allocate the exception frame on an alternative stack, and avoid the *red zone*. Afterwards, we copy the exception frame back to the stack of the currently running application function and re-enable interrupts.

A simple solution that requires recompiling the application and all libraries, including the standard C library (*libc*), is to disable at compile time the *red zone* optimization. However, this is not practical, as we want to minimize application modifications. To overcome this limitation we use an *x86_64* feature which provides the ability to change the stack in hardware when an interrupt/exception occurs.

4.3 Dependence on architecture and OS

Memory-mapped files are supported on most commonly used OSes. OSes based on Linux and BSD support *mmap* system calls, while Windows provide similar functionality with the *MapViewOfFile* system call. *Aquila* is not tightly coupled to a specific CPU architecture, OS, or hypervisor: We use *x86_64* to make our case; however, other architectures, such as ARM [63], Intel Itanium [30], and IBM Power [26] provide

hardware-assisted virtualization. Finally, *Aquila* can leverage nested virtualization [6] that modern hypervisors [41, 44, 64, 66] provide to run within a virtual machine.

4.4 System Call Interception

Similar to Dune, *Aquila* implements a subset of the guest operating system's system calls and redirects the rest to the host operating system, using *vmcall*. This requires a custom handler for system calls in *MSR_LSTAR*, which contains the address of the system call handler. We modify Dune to also enable system call interception in ring 0. Then, we intercept all virtual memory related system calls, specifically *mmap*, *munmap*, *mremap*, *madvise*, *mprotect* and *msync*. These calls are handled in *Aquila* and do not result in a *vmcall*. Therefore, they incur the overhead of a regular function call, as they do not trigger a protection domain switch.

We acknowledge that systems which run applications in a privileged domain, similar to where a guest OS runs in virtual machines incur increased costs for system calls that have to go into the host OS [4]. In our case, we assume that applications using *Aquila* leverage memory-mapped files/devices to interact with storage (rather than system calls) and that they use RDMA for networking. Therefore, system calls are not in the common path. Common networking-related system calls if used, will incur increased latency in *Aquila*: a *vmexit* adds about 750 cycles (250 ns) [4], which is relatively small compared to other network path costs. Furthermore, *Aquila* can be combined with techniques from IX [5], ZygOS [55], and Shinjuku [33] to enable fast *vmexit*-less networking.

4.5 Security implications

Aquila provides a similar security model to a guest virtual machine running on a host operating system.

5 Experimental Methodology

Our testbed consists of a dual-socket server equipped with two Intel(R) Xeon(R) E5-2630 v3 CPUs running at 2.4 GHz, each with 8 physical cores and 16 hyperthreads, for a total of 32 hyperthreads. The storage device used in our experiments is a PCIe-attached Intel Optane SSD DC P4800X [28] with a capacity of 375 GBs. The server is equipped with 256 GB of DDR4 DRAM at 2400 MHz and runs CentOS v7.3, with Linux kernel 4.14.72.

During our evaluation we limit the available capacity of DRAM (using *cgroups* [35]) as required by different experiments. To reduce variability in our experiments, we disable swap and Transparent Huge Pages (THP), and set the CPU scaling governor to *performance*. In experiments where we want to stress the software path of the Linux kernel we also use a *pmem* [54] block device. This emulates a fast byte-addressable (NVM) block device backed by DRAM.

In our evaluation we first use a custom multithreaded microbenchmark. It uses a configurable number of threads that issue *load*/*store* instructions at randomly generated offsets within the memory mapped region. We ensure that each *load*/*store* results in a page fault.

Table 1. Standard YCSB Workloads.

	Workload
A	50% reads, 50% updates
B	95% reads, 5% updates
C	100% reads
D	95% reads, 5% inserts
E	95% scans, 5% inserts
F	50% reads, 50% read-modify-write

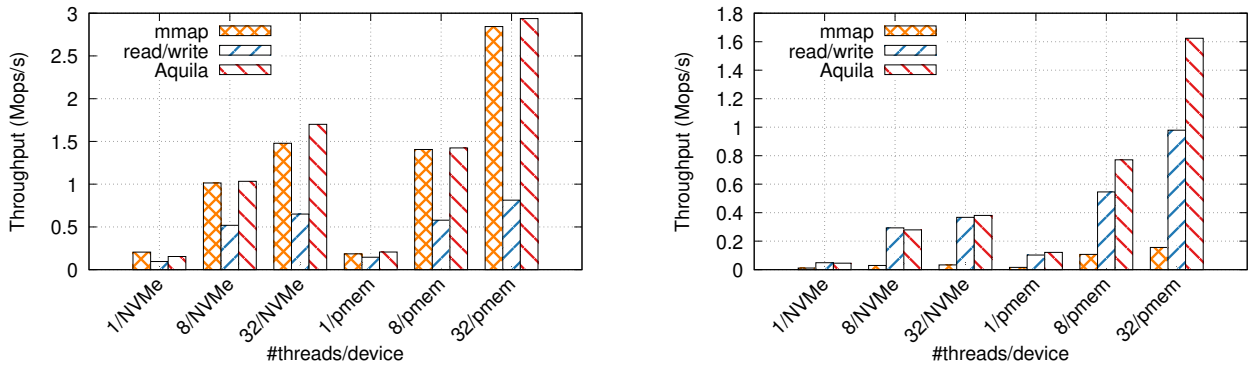
Next, we use RocksDB [20] (v6.8.0), a persistent key-value store developed by Facebook and widely used in production systems. It is based on LSM-trees [46], with each level organized in fixed-size files (64*MB* by default), named Static-Sorted-Tables (*SST*s). *SST*s are placed in the mount point specified by the user and are organized in a flat namespace. RocksDB provides different ways to read and write data from files: direct I/O with a user-space cache, buffered *read*/*write* in the Linux kernel, and *mmio*. The recommended mode of operation is to use explicit *read*/*write* calls, in direct I/O mode, combined with a user-space cache [9].

Then, we use Kreon [48, 49], a persistent key-value store designed from the ground-up to use *mmio* in the common path. Kreon is based on LSM-trees [46] but instead of *SST*s uses a log to store all keys and values and a B-Tree index per level for indexing. This approach increases random accesses to devices but reduces I/O amplification and CPU cycles in the common path. Kreon provides a custom *mmio* path in the Linux kernel, named *kmmap*, and places its data in a single file/device, using a custom allocator for space management.

We have ported RocksDB and Kreon to *Aquila* with minor changes to their initialization code. We experiment with the original YCSB workloads [15] using a C++ implementation of YCSB [56] to eliminate high JNI overheads. Table 1 summarizes the YCSB workloads we use.

Furthermore, we use Ligra [57], a lightweight graph processing framework for shared memory with OpenMP-based parallelization. We run the Breadth First Search (BFS) algorithm to evaluate *Aquila*'s effectiveness in extending the virtual address space beyond physical memory over fast storage devices. For this purpose, we convert all *malloc*/*free* calls of Ligra to allocate space over a memory-mapped file on a fast storage device. Ligra uses OpenMP for parallelization, which shows that *Aquila* can be integrated with real-life, complex runtime systems.

We run all experiments three times and report averages across runs. The variation we observe across runs in our experiments is negligible.



(a) in memory (b) out of memory

Figure 5. *Aquila* compared to *mmap* and *read*/*write* for RocksDB, with (a) a dataset that fits in memory, and (b) a dataset that does not fit in memory.

6 Experimental Analysis

In this section, we evaluate experimentally:

1. *Aquila* custom *mmio* vs. explicit *read*/*write* I/O calls.
2. Extending the application heap with *Aquila*.
3. Reducing the I/O cache overhead with *Aquila*.
4. Reduced overhead of *Aquila* vs. Linux *mmap*.
5. Improved scalability of *Aquila* vs. Linux *mmap*.

Next, we discuss our results.

6.1 *Aquila* custom *mmio* vs. explicit *read*/*write* I/O

In this section we examine how *Aquila* compares to Linux explicit I/O in terms of throughput. For this purpose, we use RocksDB, which has been designed to perform explicit I/O to storage devices. RocksDB also provides an option to use *mmio* when reading data from SSTs. The developers of RocksDB suggest [22] that using *mmap* for an in-memory database with a read-intensive workload increases performance. They also suggest [21] that *mmap* sometimes causes problems when data does not fit in memory and is managed by a file system over a block device.

We compare RocksDB with explicit I/O (direct I/O and a user-space cache of 8*GB*), RocksDB with Linux *mmap* (8*GB* page cache limited with *cgroups*), and RocksDB with *Aquila* (8*GB* DRAM cache). We use YCSB with workload C (100% random reads), with 1*KB* sized values, 30*B* sized keys, and the uniform YCSB distribution. We use a dataset of 8*M* records (8*GB*) that fits in the cache and a dataset of 32*M* records (32*GB*), which is 4× larger than the cache size.

Figure 5(a) shows the results for this experiment using both NVMe and *pmem* devices, with a dataset that fits in the cache. We see that similar to what the developers of RocksDB suggest, *mmap* is faster than *read*/*write* calls. In this case *Aquila* is up to 1.15× faster compared to Linux *mmap*.

Figure 5(b) shows our results for a dataset that does not fit in the cache, both for NVMe and *pmem*. We see that Linux *mmap* performs poorly compared to *read*/*write* I/O. The main reason is that *mmap* prefetches 128*KB* for 1*KB* reads.

The *pmem* device shows the potential of *Aquila* as storage devices become faster. In this case, *Aquila* results in higher RocksDB throughput by 1.18× for 1 thread and by 1.65× for 32 threads. With the NVMe device, *Aquila* and direct I/O have similar performance (between 0.96× up to 1.06×) because throughput is limited by the device itself. Therefore, *Aquila* is able to improve upon explicit I/O performance, even for large reads, a case where explicit I/O performs best.

In all previous cases *Aquila* provides better average and tail latency. Using the dataset that fits in cache, *Aquila* achieves from 1.28× to 1.39× lower average latency compared to direct I/O and from 1.09× to 1.27× compared to *mmap* with the NVMe device. With *pmem*, improvements are between 1.06× and 1.21× for average latency and between 1.01× and 1.15× for tail latency.

Using the dataset that does not fit in the cache, we compare *Aquila mmio* with Linux direct I/O, excluding Linux *mmap* as it performs poorly. For the NVMe device, average latency improves similarly to the in-memory dataset, between 0.98× and 1.12×. However, for *pmem*, *Aquila* achieves even lower average latency compared to the out-of-memory dataset, between 1.24× and 1.28×.

We notice larger improvements in all cases for tail (p99.9) latency. For in-memory datasets, *Aquila* achieves 3.88× lower tail latency on average compared to Linux explicit I/O. For out-of-memory datasets, *Aquila* achieves 1.26× lower tail latency on average.

Finally, we do not provide an evaluation of write operations in RocksDB, generated by compactions. Compactions (and writes) in RocksDB take place in background threads and they are optimized to issue large (1-2MB) I/O requests.

In this case the only bottleneck is the device itself, rather than the software stack.

6.2 Extending the application heap with *Aquila*

We evaluate *Aquila* for extending the virtual address space of an application beyond DRAM and over fast storage devices. Using *mmap* (and *Aquila*) a user can easily map a file over fast storage and provide an extended address space, limited only by device capacity. Although this is not practical today because of *mmio* overhead, it is a straightforward manner to allow for large datasets in applications, without modifying them to handle I/O.

Ligra is a demanding workload in terms of memory accesses and commonly operates on large datasets. Ligra assumes that the dataset (and metadata) fit in main memory. For our evaluation we generate a R-Mat [10] graph of 100M vertices, with the number of directed edges is set to 10× the number of vertices. We run BFS on the resulting 18GB graph, thus generating a read-mostly random I/O pattern. Ligra requires about 64GB of DRAM throughout execution. To evaluate *Aquila* against Linux *mmap*, we run experiments where we limit the main memory in our server to 8GB and 16GB, and use both a *pmem* and a *NVMe* device.

Figure 6(a) shows our results with 8GB of DRAM cache. First, we notice that *mmap* with a *pmem* device results in a substantial slowdown, up to 11.8× compared to in-memory execution. This gap in performance leads applications to handle large datasets that do not fit in memory by extensive application re-design to perform I/O. *Aquila* allows Ligra to use large datasets with few modifications, while significantly improving the performance over *mmap*. We observe that for BFS with 1 thread, *Aquila* is 1.56× faster compared to *mmap* (33.6s vs 52.5s). At 8 threads, *Aquila* is 2.54× faster compared to *mmap* (8.4s vs 21.4s). At 16 threads, the difference is significantly higher, 4.14× (6.74s vs. 27.9s), due to the better scalability of the custom *Aquila* I/O cache. Figure 6(b) shows our results with 16GB of DRAM cache. Compared to 8GB of DRAM cache, absolute performance improves due to reduced cache misses. However, even in this case *Aquila* results in similar benefits compared to *mmap* (up to 2.3× with 16 threads). As a reference point, we also include measurements from in-memory experiments with Ligra. *Aquila* closes the gap of *mmio* compared to in-memory execution from 5× to 3.23× slower at 1 thread, from 6.4× to 2.5× slower at 8 threads, and from 11.8× to 2.8× slower at 16 threads. We also see similar benefits with a real *NVMe* device both in terms of scalability and actual execution time.

Figure 6(c) shows the breakdown of execution time with 8GB and 16 threads. With a *pmem* device, these benefits come from the reduced system time from 61.79% with *mmap* to 43.82% with *Aquila*. This leaves more CPU time for user space processing (55.92% in *Aquila* vs. 10.61% in *mmap*). *Aquila* also reduces the idle time. We see similar improvements with the *NVMe* device.

This breakdown shows that *mmap* is not efficient in extending the application heap over fast storage devices. *mmap* incurs several inefficiencies that result in increased system and idle time for cache management, device I/O, and handling page faults. *mmap* results in a 11.8× slowdown compared to using only DRAM, which is a significant penalty for extending the virtual memory of a user application. *Aquila* reduces system and idle time by 8.31×, resulting in significantly lower slowdown (2.8×) compared to using only DRAM. Therefore, *Aquila* makes it practical to support large heaps (and datasets) without application redesign, and only requires limited modifications during initialization.

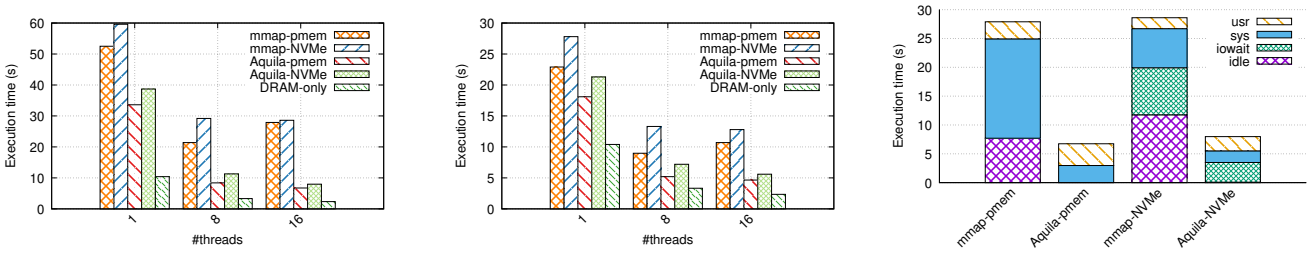
6.3 Reducing I/O cache overhead with *Aquila*

In this section we show that *Aquila* outperforms user-space caching with direct I/O even in the case where the dataset is larger that the available DRAM cache. We use RocksDB with a similar setup as in Section 6.1.

Figure 7 shows that RocksDB with a user-space cache requires 65.4K cycles on average for random reads. We break this down to three sections: (a) Device I/O, which excludes system call overhead, (b) cache management, which includes system call overhead, and (c) RocksDB *get*, excluding cache accesses. Device I/O is the lowest cost at about 4.8K cycles. Cache management accounts for approximately 45.2K cycles. We further break this cost down to (i) system calls for misses and (ii) user-space cache lookups and evictions. System calls cost around 13K cycles and user-space lookups and evictions around 32K cycles. Finally, RocksDB *get* incurs a cost of about 15.3K cycles.

RocksDB with *Aquila* requires 3.9K cycles for I/O. This improvement, compared to 4.8K cycles with the user-space cache, comes from our optimized *memcpy* (Section 3.3). Cache management costs about 17.5K cycles. Cache management in *Aquila* consists of two main parts: Page fault cost during misses of about 5.6K cycles and user-space data processing in RocksDB of about 11.8K cycles. RocksDB *get* now requires 18.5K cycles. This is higher compared to the 15.3K cycles when using explicit I/O because of increased TLB misses, as *Aquila* modifies memory mappings and flushes the TLBs more frequently.

It is apparent that managing the user-space I/O cache requires about 69% of the total CPU cycles for RocksDB reads in the case of explicit I/O. Faster storage devices will stress this path further, increasing even more the related overheads. *Aquila* allows applications to take advantage of fast storage devices more effectively. With *Aquila*, RocksDB requires 2.58× fewer CPU cycles for cache management. *Aquila* spends 43.7% of the total CPU cycles consumed by RocksDB for I/O, while also achieving 40% higher end-to-end, user-perceived throughput.



(a) 8GB DRAM Cache (b) 16GB DRAM Cache (c) Breakdown

Figure 6. (a)-(b) Execution time for Ligra running BFS with *mmap*, *Aquila* (with *pmem* and *NVMe* device), and *DRAM-only* (with *malloc/free*). (c) Execution time breakdown for Ligra running BFS with 16 threads and 8GB of DRAM as cache.

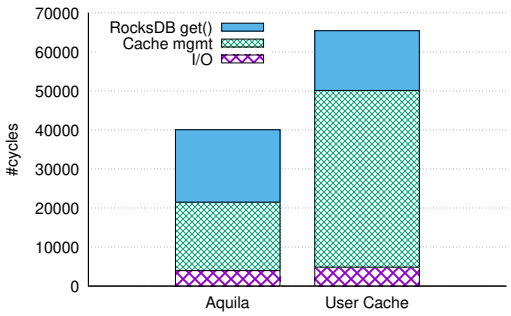


Figure 7. RocksDB execution time breakdown (in cycles) for reads with *Aquila* and user-space caching.

6.4 Reduced overhead of *Aquila* vs. Linux *mmap*

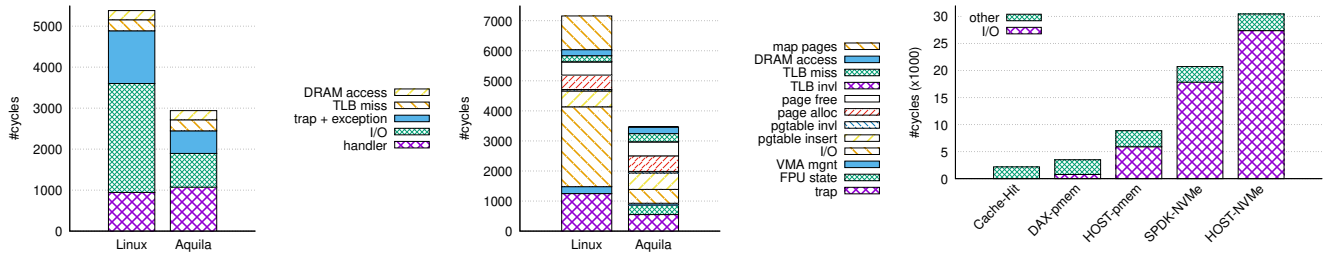
First, we look at the cost of protection domain switching. Figure 8(a) shows the average overhead for a page fault over a memory-mapped file to be about 5380 cycles in total. Two major components of this overhead are: (1) 49% is due to device I/O overhead and (2) 24% is due to the cost of the protection domain switch (trap). If we exclude device I/O, e.g., when the page resides in the page cache, the cost of a page fault (2724 cycles or 1.13μs) is comparable to accessing fast storage devices. We measure the protection domain switch cost (excluding the handler itself) to be 1287 cycles (536ns). In this measurement, if we exclude I/O cost, then switching domains is 1287 out of 2724 cycles, thus 52.7% of the page fault cost. Reducing the protection domain switch overhead will affect all page faults that happen in the common path. Figure 8(a) shows that the trap cost in non-root ring 0 (*Aquila*) is 552 cycles (230ns), which is 2.33× lower compared to exceptions from ring 3.

Figure 8(b) shows the average overhead breakdown for the case where the dataset does not fit in main memory and evictions happen in the common path. In this case we use 8GB for the DRAM cache and a 100GB dataset. *Aquila* achieves 2.06× lower overhead compared to Linux *mmap*. We observe that in this case the major sources of overhead are protection domain switching and I/O to the underlying device. Finally, we observe that in *Aquila*, no single source of overhead dominates in the common path, even in the case where evictions occur in the common path: Each component of the *Aquila mmio* path accounts for less than 10% of overhead. We omit the presentation of results for writes as we observe similar behavior and performance to reads.

Figure 8(c) demonstrates how the I/O path affects the performance in *Aquila*. *Cache-Hit* is the case where no I/O is required and the total cost in this case is 2179 cycles. *DAX-pmem* uses our optimized path for byte-addressable devices and *HOST-pmem* uses direct I/O system calls to the host OS. In this case, *Aquila* achieves 7.77× lower overhead. This stems from the fact that we remove system calls and use a SIMD-optimized *memcpy*. *SPDK-NVMe* uses SPDK to bypass the host OS for PCIe attached devices. *HOST-NVMe* uses direct I/O, similar to *HOST-pmem*. In the case of *Aquila*, bypassing the host OS reduces overhead by 1.53×. In all cases, the remaining cost, excluding the I/O, remains the same. This shows that the method used to access storage devices in *Aquila* affects overall performance. Removing the interaction with the host OS reduces overhead by up to 7.77×.

Finally, we use *Aquila* with Kreon, a persistent key-value store designed to use *mmio* in the common path. Kreon provides a custom *mmio* path, named *kmmap*, which improves several aspects of Linux *mmap*. We compare Kreon over *kmmap* with Kreon over *Aquila*. We run all YCSB workloads using a single thread to show how overhead reduces in the single-thread path, with a dataset of 16M records (16GB) and a 8GB cache.

Figure 9 shows our results. Using NVMe, *Aquila* achieves on average 1.02× higher throughput for all YCSB workloads. In this case the bottleneck is the NVMe device itself given the request size (4KB) and a single outstanding I/O. Latency improves significantly: *Aquila* achieves 1.29× lower average latency and 3.78× tail (p99.9) latency compared to *kmmap*. With *pmem*, where device throughput is not the dominating bottleneck, *Aquila* achieves on average 1.22× higher throughput. We also see significant improvements in terms



(a) page fault cost (b) page fault cost (w. evictions) (c) I/O cost

Figure 8. Breakdown in cycles for (a) page fault overhead (with microbenchmark) for Linux and *Aquila*, using a *pmem* device (backed by DRAM) and a dataset that fits in memory (*Aquila* incurs only exception cost, not trap) (b) page fault overhead (with microbenchmark) for Linux and *Aquila*, using a *pmem* device (backed by DRAM) and a dataset that does not fit in memory, and (c) different approaches to access the storage device in *Aquila*.

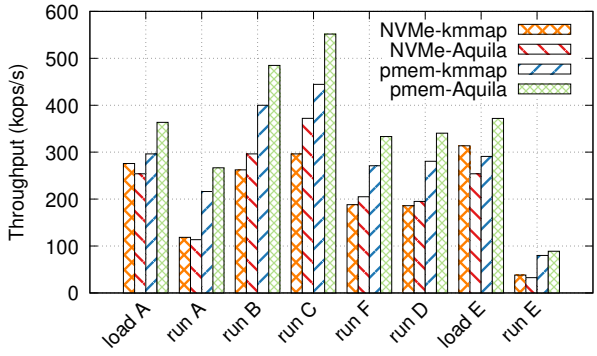


Figure 9. *Aquila* vs. Kreon *kmmap* for a dataset that does not fit in memory for NVMe and *pmem* using a single thread.

of latency: *Aquila* achieves 1.43× lower average latency and 13.72× lower tail (p99.9) latency.

6.5 Improved scalability of *Aquila* vs. Linux *mmap*

Next, we examine how *Aquila* scales with an increasing number of cores. We distinguish two cases for accessing files with *mmio*: when all threads access a single shared file and when each thread accesses a different file.

Figure 10 shows our results for a dataset (100*GB*) in two cases, one where it fits in memory (100*GB* DRAM) and one where it does not fit in memory (8*GB* DRAM). In both cases, as we increase the number of threads *Aquila* scales as follows: For a single shared file that fits in memory, *Aquila* achieves 1.81× higher throughput with 1 thread and 8.37× with 32 threads. In the case where the dataset does not fit in memory, the improvement is even more pronounced compared to Linux *mmap*: *Aquila* performs better by 2.17× at 1 thread and by 12.92× at 32 threads.

We use profiling to identify the reason for the large improvement over Linux *mmap* for a single shared file. We find that in Linux, a single lock protects the radix tree of cached pages, and, as a result, is highly contended. *Aquila* replaces this single lock with a lock-free hash table which stores all cached pages. We notice similar behaviour for writes in Linux, as this lock is also required to mark a page as dirty. *Aquila* uses per-cpu red-black trees to store dirty pages, thus overcoming this limitation.

Using a separate file per thread, we also see significant improvements: *Aquila* achieves higher throughput between 1.82× (1 thread) and 1.99× (32 threads) using the in-memory dataset and between 2.21× (1 thread) and 2.84× (32 threads) for the out-of-memory dataset.

Aquila also provides consistently better latency for all these experiments, both average and tail. Using a single thread and the dataset which does not fit in main memory, *Aquila* achieves 2.07× lower average latency. Furthermore, *Aquila* has 13.6× (p99) and 2.1× (p99.9) lower tail latency.

Using 32 threads the improvements of *Aquila* in terms of latency are even greater. Using a shared file, *Aquila* achieves 8.52× (average), 177× (p99), and 213× (p99.9) lower latency. For a separate file per thread *Aquila* achieves lower latency by 1.64× (average), 42.64× (p99), and 53.2× (p99.9). Eliminating the shared contended lock in the shared file case provides huge improvements in tail latency for *Aquila*.

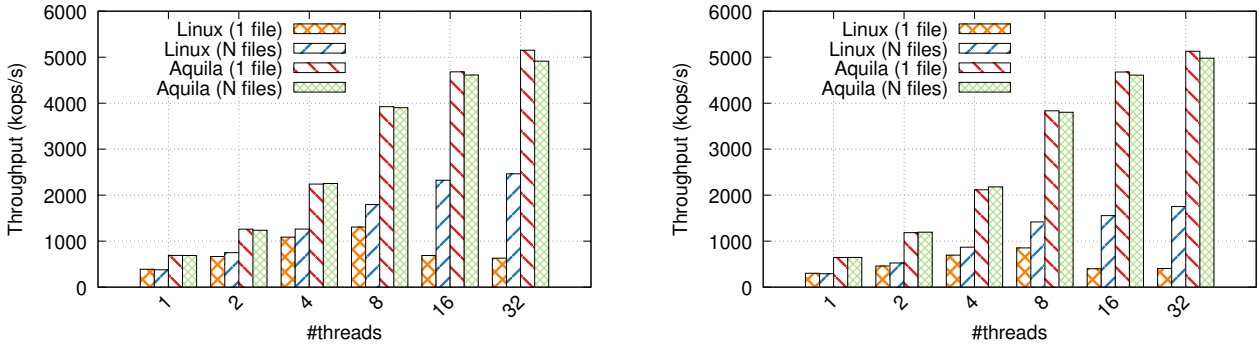
We see similar behaviour in writes compared to reads. For this reason, we omit the presentation of write results.

7 Related Work & Discussion

We categorize related work in three areas: (a) I/O over fast devices, (b) improvements to *mmio*, and (c) dataplane OSes.

7.1 I/O over fast storage devices

Synchronous *read*/*write* system calls result in significant overheads and thus several different approaches have been



(a) in memory

(b) out of memory

Figure 10. Scalability of *Aquila* vs. Linux *mmap* using random reads for both a shared and a private file per thread with (a) a dataset that fits in memory and (b) a dataset that does not fit in memory.

proposed to mitigate their overhead. A user-space cache is often used to eliminate expensive system calls. However, a user-space cache still incurs high lookup overhead for frequent hits [24]. In addition, it becomes even less effective when misses that require system calls start to increase, especially for larger datasets.

Using asynchronous system calls and batching, such as *libaio* and *io_uring*[32] can help mitigate system call overhead. The latter is the state-of-the-art approach for asynchronous I/O. It allows batching in the issue path, with a single system call initiating multiple I/O operations. In the completion path, it does not require any system calls as it uses shared memory between user and kernel space, and merely checks for completion in user-space memory. Asynchronous I/O reduces the required CPU cycles in the I/O path and increases throughput in most cases. However, it also increases tail latency due to batching and can negatively affect latency-sensitive applications. Moreover, asynchronous I/O is significantly more complex for applications to use.

An alternative to traditional (synchronous or asynchronous) system calls for explicit I/O is to provide user-space access to certain types of devices, such as byte-addressable NVM and block-addressable NVMe devices [37, 43, 61]. This path does not require a system call and typically lacks strong protection and sharing of resources. Moreover, as the user-space frameworks bypass the kernel, they need to use polling when waiting for completions, resulting in higher CPU utilization, a problem that is particularly acute with current technology trends. Most of the user-space frameworks also employ user-space caching to reduce the number of accesses to storage devices, e.g., BlobFS [59] which is part of SPDK. However, these systems incur the high hit overhead of user space caches.

Aquila takes a different approach by using *mmio* in two respects: Firstly, it allows synchronous I/O which is important for latency purposes and is easiest for the programmer. Secondly, it removes the user-space software cache lookups from the common path by using hardware address translation for lookups. Therefore, *Aquila* has the potential to reduce fundamental overheads for fast storage devices. The challenge we address is to improve *mmio* in a way that separates common from uncommon path operations which up to this point have been traditionally bound together.

Byte-addressable NVM devices today are used as an extension of DRAM (*pmem*) and essentially always used with DAX. DAX allows direct mapping of NVM to the user address space, next to available DRAM. In this case, user-space libraries [14, 23, 25, 39, 52, 65, 67] can be used to provide higher level abstractions. However, NVM latency and throughput are about 3x worse than DRAM [31] making management of this hybrid memory space (DRAM and NVM) a significant challenge. Instead, *Aquila* uses DRAM as a cache to *pmem* with low overhead. Although *Aquila* has the additional potential to transparently extend the process address space over *pmem*, hiding device heterogeneity, our primary goal in this paper is to improve persistent storage I/O.

7.2 Improvements to *mmio*

The evaluation of Tucana [47], a key-value store which uses *mmio*, shows that for write-intensive workloads, Linux *mmio* results in excessive and unpredictable traffic to devices, leading to freezes during execution and consequently high tail latency.

Kmmap [48] is a custom *mmio* path in the Linux kernel tailored for Kreon, a memory-mapped key-value store. Kmmap provides several improvements in the Linux *mmio* path to reduce performance variability due to the aggressive writeback policy of Linux. Additionally, kmmap provides a

custom *msync* operation, based on Copy-On-Write semantics of Kreon. Copy-On-Write ensures that Kreon updates only newly allocated pages and there will be always only one timestamp for the latest (single) update. *msync* in kmmap writes pages based only on this timestamp. Kmmap does not address scalability issues with the number of user threads for a single memory mapping. Similar to kmmap, *Aquila* uses a lazy writeback strategy compared to Linux. However, *Aquila* provides the ability to use custom cache and device access mechanisms and policies combined with the reduced cost of the removal or protection domain switched.

FastMap [50] demonstrates that the Linux *mmio* path fails to scale efficiently with increasing application threads. To overcome this issue, the authors (a) separate clean and dirty-trees to avoid all centralized contention points, (b) use full reverse mappings instead of Linux object-based reverse mappings to reduce CPU processing, and (c) introduce a scalable DRAM cache with per-core data structures to reduce latency variability. FastMap requires a custom Linux kernel and, being a loadable kernel module, it does not allow per-application customization. Similar to FastMap, *Aquila* uses separate data structures for clean and dirty pages to reduce contention. Unlike FastMap, *Aquila* enables applications to use custom cache and device access mechanisms and policies, on an unmodified Linux kernel. In addition, *Aquila* reduces the overhead (CPU cycles) in the page fault path by eliminating the need for protection domain switches.

DI-MMAP [18, 19] removes the swapper from the critical path and implements a custom (FIFO based) eviction policy using a fixed-size memory buffer for all *mmap* calls. UMap [51] is a user-space memory-mapped I/O framework which adapts different policies to application characteristics and storage features. Handling page faults in user-space (using *userfaultfd* [36]) introduces additional overheads that are not acceptable with fast storage devices. Song et al. [58] propose an optimized page reclamation procedure with a new page recycling method to reduce context switches. This makes it possible to use extended vector I/O – a parallel page I/O method. In *Aquila* we take a more fundamental approach, by placing the application in non-root ring 0 and eliminating protection domain switch overheads.

7.3 Dataplane operating systems

Exokernel [17] proposes to separate the OS dataplane and the OS control plane to provide higher throughput and lower latency with specialized library OSes. Systems, such as IX [5], ZygOS [55], Shinjuku [33], MICA [40], and Chronos [34] optimize the network path, while *Aquila* targets the storage path. Although storage and networking share similarities, they need to address different challenges related to virtual memory management.

Arrakis [53] targets storage I/O in addition to networking. Arrakis uses SR-IOV [29] to provide multiple virtual PCIe devices and handle protection and multiplexing in the I/O controller. *Aquila* takes a more holistic approach that includes the user-space storage cache and improves virtual memory management and device access in *mmio*.

Similar to IX [5], ZygOS [55], and Shinjuku [33] *Aquila* uses Dune [4] to have access to privileged hardware features. These systems target networking while *Aquila* reduces access overhead for fast storage devices. ReFlex [38] provides an optimized path to access remote flash storage. It uses Dune to closely integrate networking and storage processing, and achieves low latency and high throughput in the storage server. In *Aquila* we assume that the server has access to fast local storage devices. Approaches similar to ReFlex can be used orthogonally to our work for remote storage.

8 Conclusions

In this paper we argue that *mmio* can reduce I/O overhead over fast storage devices by eliminating the cost of I/O cache hits compared to other alternatives, such as moving the I/O page cache in user space. However, using *mmio* via Linux *mmap* introduces several limitations, including the necessity to use the kernel-space shared I/O page cache, specific mechanisms for device access, and expensive page faults.

We design *Aquila*, a library OS that addresses these limitations by collocating the application and *mmio* functionality in non-root ring 0. *Aquila* allows applications to use *mmio* efficiently by reducing the cost of page faults, *and* customize the I/O path (I/O cache and device access mechanisms and policies), *while* requiring limited application modifications by maintaining compatibility with Linux *mmap*.

We implement *Aquila* using Dune, and evaluate it in detail with micro-benchmarks, two persistent key-value stores, and a graph processing framework. We observe significant performance gains both in terms of throughput and latency. *Aquila* requires 2.58× fewer CPU cycles for cache management in RocksDB, compared to user-space caching and *read*/*write* system calls, and improves request throughput by up to 40%. For a graph-processing application with its heap extended over fast storage devices, *Aquila* reduces execution time up to 4.14× compared to Linux *mmap*, with minimal modifications to the application.

Acknowledgements

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the project EVOLVE (Grant Agreement ID: 825061). Anastasios Papagiannis is also supported by the Facebook Graduate Fellowship. We would like to thank Ioannis Malliotakis for his insightful comments and constructive feedback. Finally, we thank the anonymous reviewers for their insightful comments and our shepherd Mark Silberstein for his help with preparing the final version of the paper.

References

[1] Alexandra Fedorova. [n.d.]. Getting storage engines ready for fast storage devices. https://engineering.mongodb.com/post/getting-storage-engines-ready-for-fast-storage-devices. Accessed: October 13, 2022.

[2] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 27–39. https://www.usenix.org/conference/atc17/technical-sessions/presentation/amit

[3] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't Shoot down TLB Shootdowns!. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 35, 14 pages. https://doi.org/10.1145/3342195.3387518

[4] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[6] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. 2010. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 423–436.

[7] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Commun. ACM* 51, 12 (Dec. 2008), 77–85. https://doi.org/10.1145/1409360.1409380

[8] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2010. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 1–16.

[9] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 209–223. https://www.usenix.org/conference/fast20/presentation/cao-zhichao

[10] D. Chakrabarti, Y. Zhan, and C. Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *SIAM International Conference on Data Mining*. http://www.cs.cmu.edu/~christos/PUBLICATIONS/siam04.pdf

[11] Jungsik Choi, Jiwon Kim, and Hwansoo Han. 2017. Efficient Memory Mapped File I/O for In-Memory File Systems. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotstorage17/program/presentation/choi

[12] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2012. Scalable Address Spaces Using RCU Balanced Trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 199–210. https://doi.org/10.1145/2150976.2150998

[13] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: Scalable Address Spaces for Multithreaded Applications. In *Proceedings of the 8th ACM European Conference on Computer Systems* (Prague, Czech Republic) *(EuroSys '13)*. Association for Computing Machinery, New York, NY, USA, 211–224. https://doi.org/10.1145/2465351.2465373

[14] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 143–154. https://doi.org/10.1145/1807128.1807152

[16] Tudor David, Rachid Guerraoui, and Vasileios Trigonakis. 2015. Asynchronized Concurrency: The Secret to Scaling Concurrent Search Data Structures. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. Association for Computing Machinery, New York, NY, USA, 631–644. https://doi.org/10.1145/2694344.2694359

[17] D. R. Engler, M. F. Kaashoek, and J. O'Toole. 1995. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain, Colorado, USA) *(SOSP '95)*. Association for Computing Machinery, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[18] Brian Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. 2015. DI-MMAP–a Scalable Memory-map Runtime for Out-of-core Data-intensive Applications. *Cluster Computing* 18, 1 (March 2015), 15–28. https://doi.org/10.1007/s10586-013-0309-0

[19] B. V. Essen, H. Hsieh, S. Ames, and M. Gokhale. 2012. DI-MMAP: A High Performance Memory-Map Runtime for Data-Intensive Applications. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. 731–735. https://doi.org/10.1109/SC.Companion.2012.99

[20] Facebook. [n.d.]. RocksDB. https://rocksdb.org/. Accessed: October 13, 2022.

[21] Facebook. [n.d.]. RocksDB IO. https://github.com/facebook/rocksdb/wiki/IO. Accessed: October 13, 2022.

[22] Facebook. [n.d.]. RocksDB Tuning Guide. https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide. Accessed: October 13, 2022.

[23] E. R. Giles, K. Doshi, and P. Varman. 2015. SoftWrAP: A lightweight framework for transactional support of storage class memory. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14. https://doi.org/10.1109/MSST.2015.7208276

[24] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 981–992. https://doi.org/10.1145/1376616.1376713

[25] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. 2017. Log-Structured Non-Volatile Main Memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 703–717. https://www.usenix.org/conference/atc17/technical-sessions/presentation/hu

[26] IBM. [n.d.]. Power ISA. Version 2.06 Revision B.

[27] S. Imamura and E. Yoshida. 2019. POSTER: AR-MMAP: Write Performance Improvement of Memory-Mapped File. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques*

(PACT). 493–494.

[28] Intel. [n.d.]. OPTANE SSD DC P4800X SERIES. https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-p4800x-series.html. Accessed: October 13, 2022.

[29] Intel. [n.d.]. PCI-SIG SR-IOV Primer. https://www.intel.com/content/www/us/en/pci-express/pci-sig-sr-iov-primer-sr-iov-technology-paper.html. Accessed: October 13, 2022.

[30] Intel. [n.d.]. Virtualization Technology Specification for the Intel Itanium Architecture (VT-i). Accessed: October 13, 2022.

[31] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714 (2019). arXiv:1903.05714 http://arxiv.org/abs/1903.05714

[32] Jens Axboe. [n.d.]. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf. Accessed: October 13, 2022.

[33] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[34] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M. Voelker, and Amin Vahdat. 2012. Chronos: Predictable Low Latency for Data Center Applications. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. Association for Computing Machinery, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2391229.2391238

[35] Linux kernel. [n.d.]. cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. Accessed: October 13, 2022.

[36] Linux kernel. [n.d.]. Userfaultfd. https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt. Accessed: October 13, 2022.

[37] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. NVMeDirect: A User-space I/O Framework for Application-specific Optimization on NVMe SSDs. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotstorage16/workshop-program/presentation/kim

[38] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash $\approx$ Local Flash. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 345–359. https://doi.org/10.1145/3037697.3037732

[39] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. 2016. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 399–411. https://doi.org/10.1145/2872362.2872381

[40] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast in-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, USA, 429–444.

[41] Linux. [n.d.]. KVM - Nested VMX. https://www.kernel.org/doc/Documentation/virtual/kvm/nested-vmx.txt. Accessed: October 13, 2022.

[42] W. Mauerer. 2008. *Professional Linux Kernel Architecture*. Wiley. https://books.google.gr/books?id=e8BbHxVhzFAC

[43] Micron. [n.d.]. UNVMe - A User Space NVMe Driver. https://github.com/MicronSSD/unvme. Accessed: October 13, 2022.

[44] Microsoft. [n.d.]. Run Hyper-V in a Virtual Machine with Nested Virtualization. https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/user-guide/nested-virtualization. Accessed: October 13, 2022.

[45] MongoDB. [n.d.]. MMAPv1 Storage Engine. https://docs.mongodb.com/v4.0/core/mmapv1/. Accessed: October 13, 2022.

[46] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (June 1996), 351–385. https://doi.org/10.1007/s002360050048

[47] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2016. Tucana: Design and Implementation of a Fast and Efficient Scale-up Key-value Store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 537–550. https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis

[48] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. 2018. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. ACM, New York, NY, USA, 490–502. https://doi.org/10.1145/3267809.3267824

[49] Anastasios Papagiannis, Giorgos Saloustros, Giorgos Xanthakis, Giorgos Kalaentzis, Pilar Gonzalez-Ferez, and Angelos Bilas. 2021. Kreon: An Efficient Memory-Mapped Key-Value Store for Flash Storage. *ACM Trans. Storage* 17, 1, Article 7 (Jan. 2021), 32 pages. https://doi.org/10.1145/3418414

[50] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-mapped I/O for Fast Storage Devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 813–827. https://www.usenix.org/conference/atc20/presentation/papagiannis

[51] I. Peng, M. McFadden, E. Green, K. Iwabuchi, K. Wu, D. Li, R. Pearce, and M. Gokhale. 2019. UMap: Enabling Application-driven Optimizations for Page Management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 71–78.

[52] Persistent Memory Development Kit (PMDK). [n.d.]. https://pmem.io/pmdk/. Accessed: October 13, 2022.

[53] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2014. Arrakis: The Operating System is the Control Plane. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 1–16.

[54] pmem.io: Persistent Memory Programming. [n.d.]. http://pmem.io/. Accessed: October 13, 2022.

[55] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-Scale Networked Tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 325–341. https://doi.org/10.1145/3132747.3132780

[56] Jinglei Ren. 2016. YCSB-C. https://github.com/basicthinker/YCSB-C.

[57] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. ACM, New York, NY, USA, 135–146. https://doi.org/10.1145/2442516.2442530

[58] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. 2016. Efficient Memory-Mapped I/O on Fast Storage Device. *ACM Trans. Storage* 12, 4, Article 19 (May 2016), 27 pages. https://doi.org/10.1145/2846100

[59] SPDK – BlobFS. [n.d.]. https://spdk.io/doc/blobfs.html. Accessed: October 13, 2022.

[60] SPDK – Blobstore. [n.d.]. https://spdk.io/doc/blob.html. Accessed: October 13, 2022.

[61] Storage Performance Development Kit (SPDK). [n.d.]. https://spdk.io/. Accessed: October 13, 2022.

[62] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, and L. Smith. 2005. Intel virtualization technology. *Computer* 38, 5 (May 2005), 48–56. https://doi.org/10.1109/MC.2005.163

[63] Prashant Varanasi and Gernot Heiser. 2011. Hardware-Supported Virtualization on ARM. In *Proceedings of the Second Asia-Pacific Workshop on Systems* (Shanghai, China) *(APSys '11)*. Association for Computing Machinery, New York, NY, USA, Article 11, 5 pages. https://doi.org/10.1145/2103799.2103813

[64] VMware. [n.d.]. Running Nested VMs. https://communities.vmware.com/docs/DOC-8970. Accessed: October 13, 2022.

[65] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, New York, NY, USA, 91–104. https://doi.org/10.1145/1950365.1950379

[66] Xen. [n.d.]. Nested Virtualization in Xen. https://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen. Accessed: October 13, 2022.

[67] Lu Zhang and Steven Swanson. 2019. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 897–912. https://www.usenix.org/conference/atc19/presentation/zhang-lu