Iris: An optimized I/O stack for low-latency storage devices

Anastasios Papagiannis[*]
Institute of Computer Science, FORTH (ICS)
Heraklion, Greece
apapag@ics.forth.gr

Giorgos Saloustros
Institute of Computer Science, FORTH (ICS)
Heraklion, Greece
gesalous@ics.forth.gr

Manolis Marazakis
Institute of Computer Science, FORTH (ICS)
Heraklion, Greece
maraz@ics.forth.gr

Angelos Bilas[*]
Institute of Computer Science, FORTH (ICS)
Heraklion, Greece
bilas@ics.forth.gr

ABSTRACT

System software overheads in the I/O path, including VFS and file system code, become more pronounced with emerging low-latency storage devices. Currently, these overheads constitute the main bottleneck in the I/O path and they limit efficiency of modern storage systems. In this paper we present a taxonomy of the current state-of-the-art systems on accelerating accesses to fast storage devices. Furthermore, we present Iris, a new I/O path for applications, that minimizes overheads from system software in the common I/O path. The main idea is the separation of the control and data planes. The control plane consists of an unmodified Linux kernel and is responsible for handling data plane initialization and the normal processing path through the kernel for non-file related operations. The data plane is a lightweight mechanism to provide direct access to storage devices with minimum overheads and without sacrificing strong protection semantics. Iris requires neither hardware support from the storage devices nor changes in user applications. We evaluate our early prototype and we find that it achieves on a single core up to 1.7× and 2.2× better read and write random IOPS, respectively, compared to the *XFS* and *EXT4* file systems. It also scales with the number of cores; using 4 cores Iris achieves 1.84× and 1.96× better read and write random IOPS, respectively. In sequential reads we provide similar performance and in sequential writes we are about 20% better compared to other file systems.

Keywords

NVM, I/O, storage systems, low latency, protection

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

1. INTRODUCTION

Emerging flash-based storage devices provide access latency in the order of a few μs. Existing devices [16] provide read and write latencies in the order of 68 and 15 μs respectively, and these numbers are projected to become significantly lower in next-generation devices. Phase Change Memories (PCM) [25], STT-RAM [13], and memristors [17] may provide even lower access latency, at the scale of hundreds or tens of nanoseconds [10].

Given these trends, the software overhead of the host I/O path in modern servers is becoming the main bottleneck for achieving μs-level response times application I/O operations. Instead of storage device technology setting the limit in increasing the number of I/O operations per second (IOPS), as was the case until recently, we now have to deal with limitations on the rate of serving I/O operations, per core, due to software overhead in the I/O path. Therefore, in this new landscape, it becomes imperative to re-design the I/O path in a manner that it will be able to keep up with shrinking storage and network device latencies.

In this paper we explore the design of a storage I/O stack that is placed in user-space and in the largest part within the address space of the application itself and we compare our approach to similar state-of-the-art systems by providing a taxonomy of them. An important design aspect is the separation of the control and data planes [24, 6]. This idea comes from the area of networking and several frameworks designed in order to take advantage of fast network devices [14]. The control plane is responsible for taking decisions regarding resource allocation and routing, while the data plane, also termed as the forwarding plane, forwards network packets to the correct destination according to control plane logic. In our storage I/O context, the control plane should decide if an I/O operation should be accelerated by our framework or it should go through the standard I/O path in the Linux kernel. In a modern server a variety of devices exist and there is no need to accelerate I/O operations to HDDs and slow SSDs. More specifically, our control plane consists of an unmodified Linux kernel which is responsible for normal processing for non-file related operations and the configuration of several independent data planes. Our data plane provides a lightweight mechanism to enable direct access to the storage devices without sacrificing strong protection semantics. We use traps in the data plane for protection rather than using a separate trusted server process [33] for enforcing protection. Our approach has the advantage that it does not require any context switches or network messages in the common I/O path. The premise behind our design is to allow the application to operate as close as possible to locally-attached storage devices. The key features of our design are as follows:

1. We intercept file-related calls from applications at the runtime level and convert them to key-value store requests.

2. We serve block operations from a key-value store. The key-value store in our current prototype is build directly over memory-mapped devices and makes extensive use of copy-on-write for failure atomicity, concurrency, and relaxed-update semantics.

3. We rely on virtualization support in modern processors (Intel's VT-x [31] and AMD's SVM [2]) to provide strong protection between different processes that access the same storage devices. These technologies have already been used to improve the performance of virtual machines. In this paper we use them for providing protected, shared access to our key-value store from multiple applications in each server.

4. Finally, we use a kernel-space module for initialization and coarse-grain file operations that do not affect the common I/O path.

We present a proof-of-concept prototype, Iris, for Linux servers and provide preliminary performance results. For our experiments we use PMBD [18, 10], a custom block device that emulates PCM latencies. We show that, per-core, our approach achieves a 1.7× improvement in random read IOPS, and 2.2× in random write IOPS. We also show that our design scales well, providing up to 1.84× and 1.96× improvement for random read and write IOPS respectively using 4 cores. Furthermore in the case of sequential reads both our approach and other state-of-the-art file systems achieve similar throughput. On the other hand, in sequential writes, our approach is around 20% as we provide a more lighweight write path. We compare Iris with the state-of-art Linux kernel file systems, *XFS* and *EXT4*.

The rest of this paper is organized as follows. Section 2 provides background on hardware virtualization support by modern processors and devices. Section 3 presents a taxonomy of the current state-of-the-art systems that try to improve access to fast storage devices. In Section 4 we present the design of Iris, and in Section 5 a preliminary evaluation. Section 6 concludes the paper and outlines future work.

2. BACKGROUND ON PROCESSOR AND I/O DEVICE VIRTUALIZATION

In this section we provide a survey of hardware virtualization features found in modern processors and I/O devices provide to simplify the design and enhance the performance of hypervisors. We focus on the Intel VT-x [31] extensions that we have used in our prototype implementation. AMD [2] and ARM [32] also support similar features, not surveyed in this section.

2.1 Intel VT-x

2.1.1 CPU Virtualization

The baseline design for running virtual machines is to ensure that privileged instructions trigger processor traps and then execute their handlers inside the hypervisor in a protected manner. The downside of this approach is that traps incur significant overheads in the execution. VT-x, a set of hardware features provided by the CPU, proposes a design that the CPU has two operating modes, the VMX root and VMX non-root mode. VMX root is similar to the normal execution of CPUs. The intention is to run the host OS and the hypervisor on this mode. VMX non-root mode is designed for running guest operating systems. In this mode, CPUs have some limitations in order to access hardware resources in a protected manner.

CPUs provide instructions to change their mode. Executing VMLAUNCH or VMRESUME from the VMX root mode, change the mode to VMX non-root, and starts executing guest code. This transition is named VM entry. The opposite transition is needed when a privileged instruction should be handled from the hypervisor and is named VM exit. VM exits occur upon events predefined by the hypervisor. These events and several other configuration options are stored in a memory buffer named VM Control Structure (VMCS) and each CPU has its own VMCS. Except from the predefined events (privileged instructions) the guest can generate a VM exit explicitly by issuing a VMCALL instruction. In both VM entries and VM exits hardware handles the save and restore of architectural state. This information is also stored inside the VMCS.

2.1.2 Extended Page Tables

Another responsibility of the hypervisor is to protect host physical memory, by managing virtual address space translations. This functionality is a significant source of complexity and overhead in the execution of virtual machines. Virtual address space translation is necessary because multiple guest operating systems have to share the same physical memory. In older implementations of hypervisors this is done by a technique named page-table shadowing that is implemented in software. This technique adds a significant amount of overheads in the virtualization because of the large number of VM exits. These exits should be done for page faults, TLB invalidation and manipulation of the CR3 register. A hardware mechanism, part of VT-x extensions, named Extended Page Table (EPT) is designed in order to reduce these overheads. This is a new page-table structure that handles translations between guest physical to host physical addresses and is under the control of the hypervisor. In the host OS there is the EPT base pointer that works in a similar way to the CR3 base pointer in the normal page table. This mechanism activated on VM entry and can be deactivated on VM exit. Using this the guest OS has full control over its own page tables and thus VM exits on page faults, TLB invalidation and manipulation of the CR3 register are not required and related overheads are removed.

2.2 Intel VT-d

Intel Virtualization Technology for Direct I/O (VT-d) enables direct access to I/O devices for the guest operating systems. This include three key features. It provides I/O device assignment for an I/O device to a virtual machines. It enables DMA remapping, which supports address translations for device DMA data transfers. This can be achieved with the help of Input/Output Memory Management Unit (IOMMU) [20]. It also provides interrupt remapping, which provides virtual machine routing and isolation of device interrupts. The idea behind this is to dedicate hardware resources to guest OSes, by removing the need to emulate hardware devices and thus removing associated overheads. Although originally targeting high-speed network devices, this direct device-to-VM assignment capability has become desirable for fast storage devices as well (such as PCIe-based NVMe drives).

2.3 Intel VT-c

Intel Virtualization Technology for Connectivity (VT-c) is a combination of three enabling capabilities. First, Intel I/O Acceleration Technology (I/OAT) is a DMA engine, implemented in the chipset, that offloads memory copies from the main processor. This is mainly used to optimize networking, but it supports other data copies as well. Second, Virtual Machine Device Queues (VMDQ) [1] improve traffic management within the server by offloading traffic sorting and routing from the hypervisor's virtual switch to the Ethernet Controller. Specifically, a network device provides queues for software based NIC sharing. Finally, Single Root I/O Virtualization (SR-IOV) [21] allows partitioning of a single physical network device into virtual devices. It is defined in the PCI Express (PCIe) specification and standardized by PCI-SIG. A SR-IOV network device is presented on PCIe as a single "physical function" and allows the creation of multiple "virtual functions" dynamically. This can be done if there is access to the "physical function" (by the hypervisor in the case of virtualized systems). It also enables the configuration of filters in the SR-IOV adapter to multiplex and demultiplex network packets from virtual to physical functions and the opposite. Each "virtual function" can be configured and used as an independent device. The idea behind this is to map each "virtual function" directly to guest OSes. This combination of capabilities enables lower CPU utilization, reduced system latency, and improved networking throughput. The problem here is that these technologies are designed for network devices and are not applicable in storage devices. Partitioning of hardware queues in networking is more straightforward than partitioning of storage devices. Storage is inherently shared, requiring orchestration of accesses from different users. These technologies does not provide any benefits towards this direction.

3. TAXONOMY OF PROPOSED SYSTEMS

In this work we are aiming to minimize overheads related to the operating system kernel, when accessing low-latency storage devices. Inside the kernel there are several layers that a I/O request should pass before being served by the actual device. One approach is to optimize or remove some of these layers [12, 15, 36, 35, 9]. Another approach is to bypass the kernel and access storage devices directly from user applications [33, 6, 24]. Accessing hardware resources directly from the user applications can introduce several issues. The most important is *protection*, as we cannot assume that a user's applications can be trusted to access a shared storage device without interfering with others. For this purpose all hardware resources can be accessed only via the operating system kernel. To enable this strict separation on what the kernel and the user applications can access, CPUs provide a way to enable the execution of the kernel in a more privileged domain, via a feature called *protection rings*. On mainstream x86 server processors, we have four different protection rings (0 up to 3) in decreasing levels of capabilities, where ring 0 is the most privileged and ring 3 the least privileged. Thus, kernel code executes on ring 0 whereas the code of user applications executes on ring 3. During normal execution, a user application runs on ring 3. When it needs to access a hardware resource, a system call must be issued. This moves the execution inside the kernel, where the first step is to check if the user application has sufficient permissions to access the specified hardware resource. With the system call, code execution moves from ring 3 to ring 0, where the kernel runs. This model has worked well for many years. Hardware devices in general have significant latencies, thus the software overheads are relatively small in comparison. With recent hardware devices, this trend starts to change [8]. In the literature [26, 19], there are already several works that target network devices. In this work we focus on storage devices.

With the introduction of emerging low-latency storage devices and more specifically with NVMe connectivity, the latencies of the storage devices have dropped significantly; as a consequence, system software latency is becoming comparable with the latency of the actual hardware devices, creating the pressing need for optimizations to address this radical change in the balance between hardware and software overheads [8, 9]. In the following paragraphs, we provide a taxonomy where we analyze the current state-of-the-art and alternative approaches addressing this problem. Both network and storage have similar characteristics on accessing them from user space, so we can use previous work from the networking field; however, the situation is not entirely the same, as storage and network devices have different characteristics, necessitating a major redesign of the I/O path. One of the most important differences is the strong requirement to share storage. The idea behind user-space networking is to provide a virtual set of send/recv queues for each process and leave to hardware the multiplexing and demultiplexing of packets. Storage is inherently a shared resource, making the options to provide virtual devices or static partitioning not acceptable.

For networking, apart from the common send/recv API, alternative event-driven APIs has often been proposed. However, an event-driven API is not a good fit for storage devices. Most user applications use the widely-accepted POSIX read/write API. Alternative storage APIs have been proposed, aiming for more relaxed access semantics compared to POSIX. Alternatives include the key-value API and the object API, both based on similar ideas. Changing user applications to work on key-value API can be challenging, requiring a lot of software engineering effort. Therefore, maintaining the common read/write API seems to be a necessity at this point in time. Moreover, storage frameworks that support more that one storage APIs would be a good idea, as different user applications could use different APIs based on their needs.

All of the points made above motivate innovations in how to access storage devices with reduced system overheads. Ideas from networking can be re-used but also many additional challenges particular to storage should be addressed. We have created a taxonomy of current state-of-art storage systems that target fast storage devices, and we compare

Table 1: Related work on optimizing the I/O path.

Title	Citations	Key Idea(s)	Key Differences(s)
`User-space networking`	[26, 19]	Access hardware directly from user-space and optimize software to minimize overheads.	Iris provides enhanced protection and sharing to storage devices.
`Arrakis`	[24, 23]	Uses SR-IOV to assign a separate "virtual function" to each process. Applicable only for network devices.	Iris targets storage devices and doesn't require any hardware support from them.
`IX`	[6]	Separation of control and data plane. Processor virtualization to provide protected accesses from data plane to network devices. Also provides an event-driven API.	Iris targets storage devices and provides file based API. Optimized for file sharing that is not applicable on networking.
`Moneta, Moneta-D`	[8, 9]	Enhanced hardware to enforce protection and modified *XFS* filesystem.	Iris doesn't require specialized hardware and it also bypasses VFS and other layers related to I/O path.
`Aerie`	[33]	Move functionality to user-space in order to accelerate accesses to NVM.	Iris doesn't use a centralized process for protection that can limit scalability.
`Mnemosyne, NV-Heaps`	[34, 11]	Provide transactional semantics over byte-addressable NVM.	Iris provides a file-based API and does not require byte-addressable NVM.
`BPFS, PMFS, NOVA, and SCMFS`	[12, 15, 36, 35]	Optimizing in-kernel file systems to access fast storage devices.	Iris bypasses VFS and other layers related to I/O path.

them to our proposed approach. Table 1 summarizes our findings. In the following paragraphs we present this taxonomy in detail.

3.1 Direct and protected device access

Recent papers have addressed the issue of how to optimize accesses to fast I/O devices. The Arrakis [24] [23] and IX [6] operating systems are based on the concept of separating the control and data planes. The control plane is responsible of managing the hardware resources in a protected and isolated manner, while the data plane is a low-overhead mechanism that allows direct but safe access to the hardware resources, specifically I/O devices.

Arrakis, which is based on Barrefish [4], achieve this by relying on SR-IOV [21] hardware features. SR-IOV allows a single physical PCI-Express device to export several virtual devices that are isolated from one another. Although they present the idea on both network and storage devices, their evaluation is mainly for network devices. Currently, SR-IOV support is not available for storage controllers, although it is commonly available in server network adapters. The current SR-IOV support for storage controllers/devices has many limitations and is not practical to use yet. In Arrakis they also do not handle the case of data sharing, which is a fundamental design issue in storage hierarchies. In [23] the authors present the key concepts of Arrakis but with emphasis on the storage path. They claim that the current storage path suffers from many sources of overheads because of the very broad-scope requirement to provide a common set of I/O operations for a wide variety of different user applications. They propose a custom specialized storage path for different kinds of applications, with direct access to storage devices. Similarly to Arrakis, they require hardware virtualization support from storage devices (SR-IOV), which however is not practical today.

Compared to Arrakis, we only require hardware virtualization support from the processor (e.g. Intel's VT-x in our prototype), but not from the I/O devices. We also use an unmodified Linux kernel, thus we still support user applications that do not require I/O acceleration. The operations that our custom data plane cannot handle (e.g. network accesses) still go through the normal path inside the kernel.

IX uses the unmodified Linux kernel as the control plane and implement a lightweight OS abstraction for the data plane. It uses Dune [5] to provide privilege separation between the control plane, the data plane and the normal processes, to provide safe access to the hardware devices. They do not require SR-IOV virtualization support, but they propose a solution and evaluation only for network devices. Authors provide an event-driven API (libIX) that provides run to completion with adaptive batching, zero-copy API and synchronization free processing. These optimizations targeting throughput and the new event-driven API require changes to the applications.

Our system is based on the idea of IX for protection. But a radical redesign is needed in order to be suitable for storage. We also use the same notation of the separation of control and data plane. We use as control plane an unmodified Linux kernel. The data plane is a way to access storage devices bypassing the Linux kernel I/O stack. But we cannot enable direct access from several user applications to a shared hardware resource. A layer that enables protection and synchronization of the accesses is required. This paper presents the design and implementation of this layer named Iris. In order to run in a different protection domain compared to the Linux kernel we use hardware virtualization features. Thus unmodified Linux kernel runs on VMX root mode and Iris and user applications runs on VMX non-root mode. More specifically Iris runs on VMX non-root ring 0 and user applications run on VMX non-root ring 3. There is an analogy here with virtual machines. Iris can be assumed to be the guest operating system. But there is no need to be so complex as there is also a full operating system in the control plane. The primary purpose of Iris is to filter I/O requests and serve them without interacting with the Linux kernel. These I/O requests arrive in Iris as system calls. All the other system calls that are not related to storage I/O can be forwarded to the Linux kernel and served from this. For the system calls that are related to I/O we still require kernel crossing. In [24] authors claim that *syscall* and *return* are about 6% of the total overhead in the I/O path. The other 94% of overheads comes from the software, where we provide a lightweight I/O stack compared to what Linux kernel provides. There are works showing that we can remove the system call overheads [29] as well, but we leave this optimizations as a future work. Another important issue that Iris should handle is the management of the virtual addresses translation and the EPT can be used in order to accelerate this procedure. Finally it has to handle signal delivery to applications. For the purpose of initialization and running Iris we use Dune. This provides the ability to run guest code in VMX non-root mode, initialize EPT and other important features. More generally, we could also built our work on top of either the KVM or Xen hypervisors for Linux.

First of all Iris executes permission checks (i.e. check if the user can read or modify the specific file). Then in order to serve the I/O requests an indexing data structure is needed. In this case we use Tucana [22], a persistent, write optimized key-value store. This means that a translation from read/write API to get/put API is needed inside Iris. By providing read/write API we do not require any changes to user applications nor recompilation of them as we can use the LD_PRELOAD hack in order to add a priority of calling our functions compared to libc functions.

3.2 Device-level hardware support

Moneta-D [9] uses specialized hardware for fast access to I/O storage devices with strong protection semantics. All the metadata operations still go through the normal I/O path in the Linux kernel. They optimize read/write operations in a way that does not require crossing the kernel for permission checks. Moneta-D provides a private, virtualized interface for each process and moves file system protection checks into hardware. As a result applications can access file data without operating system intervention, eliminating OS and file system costs entirely for most accesses. In our work, we only require virtualization support in the processor, rather than in the interface to storage devices.

3.3 Process-level protection

Another approach to access fast storage devices appeared in Aerie [33]. This work assumes byte-addressable NVM placed on the memory bus. The key idea in this work is that the NVM is directly mapped in the user's address space. Using this approach, user applications can read/write data and read metadata directly; however, the metadata updates have to be performed by a separate trusted process, the *Trusted FS Process*. This approach has the disadvantage that metadata updates, which are done by a centralized process, can limit scalability especially in highly multithreaded servers. Our approach is not subject to this limitation as multiple applications can update their metadata concurrently in different databases (more details in Section 4.1). Furthermore, we do not assume byte-addressable NVM placed on the memory bus. Aerie would require a different design to leverage PCIe based devices. On the other hand, Iris works for both cases without the need for major modifications.

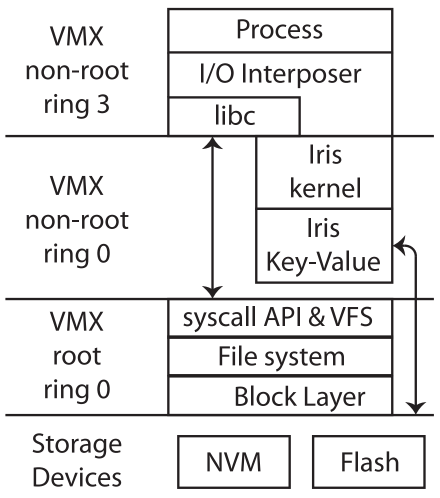

Figure 1: Top-level architecture of Iris.

3.4 Managing byte-addressable NVM

In Mnemosyne [34] and NV-Heaps [11] the authors propose ideas on how to use NVM for a persistent replacement to volatile memory that user applications can use, i.e. applications can rely on in-memory data-structures that can survive system crashes. Mnemosyne and NV-Heaps provide an API for NVM allocation and deallocation, with failure handling provisions. They also implement persistent data structures and atomic semantics (transactions) to leverage NVM from user applications. These works are orthogonal to our approach. In principle, we can apply these techniques to optimize access to NVM from our key-value store.

3.5 Optimizing kernel-level file-systems

Other works like BPFS [12], PMFS [15], NOVA [36] and SCMFS [35] try to optimize in-kernel file systems. They use the standard VFS layer, and try to optimize the file system data structures to access NVM. We don't compare with these approaches as we propose an alternative way to access NVM, different from the common system call and VFS layer approach.

4. Iris DESIGN

We implement a custom I/O path over fast persistent devices that removes most of the overheads from the Linux kernel I/O path. Figure 1 shows the top-level architecture of our system. Iris consists of three main parts:

- the key-value store, responsible for storing file blocks, providing atomic semantics, and handling failure scenarios (e.g. system crashes),
- the Iris kernel, which handles accesses to the key-value store and performs permission checks, and
- the I/O interposer which handles I/O processing at the user-space and generate key-value requests.

4.1 Key-Value Store

Our key-value store is designed primarily for fast storage devices, and is mainly based on Tucana [22]. Its API provides methods for inserting a <key, value> pair and for retrieving a <value> based on a <key>. At its core, our key-value store implements a variant of B$^\epsilon$–tree [7], a write-optimized indexing data structure. It supports multiple databases over a single or multiple devices. Since it operates at the device level, it implements its own allocation mechanism for space management over storage volumes. It maps the underlying devices in memory, and access them as memory regions.

Our persistence mechanism is based solely on the Copy-On-Write (COW) mechanism [27]. Common key-value stores use journaling for consistency purposes. In this case, for each update the mutation is first appended in a log and then updated in-place in the primary storage space. Our store operates differently: It creates a copy of the new value and subsequently modifies it. More specifically, each modification to the tree data structure requires the update of a set of nodes. Instead of updating them in-place, we create a copy of the old nodes and updates only the copy. This procedure begins from a leaf node, where a new <key, value> inserted, and goes recursively up to the root of the tree. At any point in time, there are two root nodes: one is read-only, while the other is where all data updates occur.

Our system is capable of batching a series of updates which subsequently are written to the device in an atomic manner, thus reducing actual I/O operations. After a period of time has elapsed or the application explicitly instructs to make its changes persistent, the key-value store with an atomic operation will update the read-only root to be the new persistent view of the database. Finally, keeping versions of the database is supported by keeping pointers to previous versions of the tree-structured index.

We keep both file blocks and file metadata in the key-value store. To distinguish different files, we use the persistent and unique inode number provided by VFS for each file. The key for accessing a file block in the key-value store is formed by the concatenation of the file's inode number and the requested block number. This format of keys enable keys for the same file to be contiguous in the device, and this does not hurt the performance of sequential accesses. Let's consider the case where we want to read the contents of a large file in its entirety. We need to issue a point query (*get()*) for every data block. Using range queries, we can find the first block of the file and get the subsequent blocks by using the *next()* function. This reduces overhead compared to a point query as we do not need to traverse the tree from the root to a leaf node. In the current version we haven't implemented this optimization, and all read/write requests translate to point queries. In our current implementation, we use a block size of 4KB, but this is a parameter configurable by the system administrator. The value returned by the key-value store is a block of the actual data of the file. We also keep persistent metadata for each file that is present in the key-value store. These include the inode number, the file path and the name of the file, a *struct stat* that also contains the size of the file, and the file ownership and permissions information.

We rely on the key-value store to provide data and metadata consistency upon failures. By guaranteeing a series of update operations to be atomic, we ensure that file data and metadata will not be in an inconsistent state after a failure. Current state-of-art file systems use a journaling mechanism to provide data integrity after a failure. Each write has to be done first on the journal device and then on the primary device. When a failure occurs, the file system has to replay the log. We use a different approach for failure handling. By using the copy-on-write technique, we remove the overhead to perform a write on both the journal device and then to the primary device. After a failure, only the last consistent view of our key-value store is visible to applications. BTRFS [28] file system also uses copy-on-write for metadata integrity.

Our key-value store is designed to be mapped to multiple applications, allowing shared storage. Therefore, it has to support concurrent *get* and *put* requests. Tucana [22] in its current version supports single writer per database and multiple concurrent readers. In order to avoid the bottleneck in the write path we leverage Tucana's support for multiple databases and we store different files in different databases using a hash based distribution. This allows data and metadata updates on different files to be concurrent. To maintain POSIX semantics, for each file the results of the last write must be returned to any subsequent read operation. Instead of simple coarse-grain locking, we have implemented a more sophisticated locking protocol to support concurrent reads and writes for different files.

Finally, in order to fully bypass the Linux kernel we plan to map PCIe storage devices directly to Iris using Intel VT-d virtualization features. In this case we will be able to access these device using a user-level NVMe driver framework like SPDK [30]. Currently Tucana key-value store requires a block device as an underlying storage device. We plan to extend Tucana in order to access NVMe devices directly through SPDK. In Iris we also don't require specialized hardware support as the only hardware support we need is the virtualization extension, that all modern processors and motherboards provide.

4.2 Iris Kernel

The Iris kernel is the heart of the system. It maps a fast storage device to the application process address space. Therefore, in the common path Iris avoids the overheads of system call processing, VFS, and in-kernel file system processing. The main drawback of moving all I/O processing into user space is the lack of protection that Linux kernel provides. To address this concern, we rely on processor virtualization virtualization features. Intel VT-x [31] virtualization technology provides two different privilege domains: VMX-root and VMX non-root. Each of them supports the standard privilege rings (0 to 3). The purpose of this separation is to better support hypervisors. Normally, the hypervisor runs on VMX-root, ring 0, while the guest OS of each virtual machines runs on VMX non-root, ring 0, and guest processes on VMX non-root, ring 3. In our work, we use this privilege separation for a different purpose, following the idea behind the Dune [5] prototype. The Linux kernel runs on VMX-root, ring 0, the protected I/O path code runs on VMX non-root ring 0, and user processes (issuing I/O requests) run on VMX non-root ring 3. By using this privilege separation we provide strong protection semantics to access shared storage devices, similar to the unmodified Linux kernel.

The Iris kernel runs on VMX non-root ring 0, thus it is protected from user processes that run on VMX non-root ring 3. When I/O interposer issues a *get* or *put* request,

	EXT4	*XFS*	Iris
read	269	261	445
write	203	199	439

Table 2: Single thread random IOPS (thousands).

it checks if the specified process has sufficient privileges to access the file with the specific inode number. If not, an error is returned to the interposer and then to the user.

4.3 I/O Interposer

The purpose of this part is to intercept I/O system calls to libc. We provide our own dynamically linked library that replaces these libc calls and ensure that our library gets priority over libc (via *LD_PRELOAD*). Therefore, applications run unmodified, while our I/O interposer handles all open file descriptors and translates I/O requests to key-value requests: *get* and *put*. For each open file, we maintain state related to the file, which allows us to handle *ftruncate*, *fallocate*, *stat*, *lseek* and their variants. For correctness, each metadata operation to a file stored in Iris should be handled by our system, because normal Linux kernel does not have correct information for these files. We also support all calls needed for normal Linux kernel tools, like *reaadir* call for *ls* application.

In addition to the persistent file metadata stored in the key-value store, the interposer also maintains in-memory (not persistent) metadata. These metadata include open file descriptors and the current read/write offset in each file. These metadata are also not persistent in the case of the unmodified Linux kernel. After a failure, applications do not expect to have the files descriptors that are available before a failure. We also maintain an in-memory cache of persistent metadata, to accelerate metadata operations without sacrificing correctness.

5. EVALUATION

In this section we provide a preliminary evaluation of Iris. Our testbed consists of two Intel Xeon E5620 processors running at 2.40GHz and 24 GBytes of DDR3/1333 DRAM organized in 2 NUMA nodes, each of them with 12 GBytes of DDR3 DRAM. In our experiments we pin the benchmark threads on a single NUMA node in order to remove NUMA-related effects. We run experiments with *FIO* [3] to measure random-access read and write IOPS, with a block size of 512 bytes, and sequential-access read and write throughput, with a block size of 4096 bytes. In both case we set the device queue depth equal to 1, and direct I/O to bypass the page cache. We vary the number of I/O issuing threads from 1 to 4. Each thread performs I/O on a separate file of size equals to 128MB. We use the PMBD [18, 10] block device driver to emulate the access latencies of a PCM memory device over DRAM. We dedicate 8GBytes of the testbed's DRAM for use as PMBD's storage space. We compare Iris with the current state-of-art file systems provided by the Linux kernel, *EXT4* and *XFS*. For both of these file systems, we also use PMBD as the underlying block device. In all cases, the results obtained from Iris have very small variance between runs.

Table 2 shows the number of random IOPS for both reads and writes using a single thread. Regarding random read IOPS, Iris provides 1.65× and 1.7× higher number of IOPS compared with *EXT4* and *XFS*, respectively. For random write IOPS the improvement is 2.16× and 2.2×, respectively.

Figure 2 shows how random IOPS scale while increasing the number of threads from 1 to 4, compared to *EXT4* and *XFS*. Using 4 threads, Iris provides 1.84× and 1.82× for reads and 1.96× and 1.8× for writes higher number of IOPS respectively. These results show that while we increase the number of threads the performance improvements remains almost the same. With Iris, we serve around 400 KIOPS per thread (i.e. processor core in this evaluation experiment), almost 2× more than what is achievable with *EXT4* and *XFS*, without sacrificing protection guarantees and failure resilience.

Table 3 shows the sequential throughput for both reads and writes using a single thread. Regarding sequential read throughput, Iris provides 1% and 5% higher MB/s compared with *EXT4* and *XFS*, respectively. We measure the raw performance of the device using the *dd* tool. We use block size 4KB and direct accesses to the device. Using one instance of *dd* (i.e. 1 I/O issuing thread) we get around 1.8GB/s for sequential reads. With Iris, *EXT4* and *XFS* we get about one half of this throughput, because one additional data copy is needed compared to direct device access without a file system. Current state-of-the-art file systems do a good job in sequential reads as they try to store large files sequentially in the device. For sequential write throughput the improvement is 35% and 14%, respectively. Using again the *dd* tool, with the same configuration, we get around 1.3GB/s for sequential writes, so we achieve about the half of the peak device throughput. We are about 20% better on average compared to the other file systems, and this comes from the more lightweight write path.

Figure 3 shows how sequential throughput scales while increasing the number of threads from 1 to 4, compared to *EXT4* and *XFS*. Using 4 threads, Iris provides 5% and 14% for reads and 27% and 7% for writes higher throughput in terms of MB/s respectively. These results show that while we increase the number of threads the performance improvements remains almost the same. With Iris we achieve

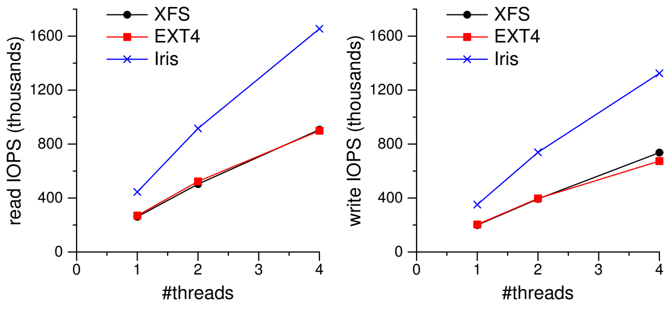

Figure 2: Random read/write IOPS scaling.

	EXT4	*XFS*	Iris
read	927	882	933
write	432	510	584

Table 3: Single thread throughput (MB/s).

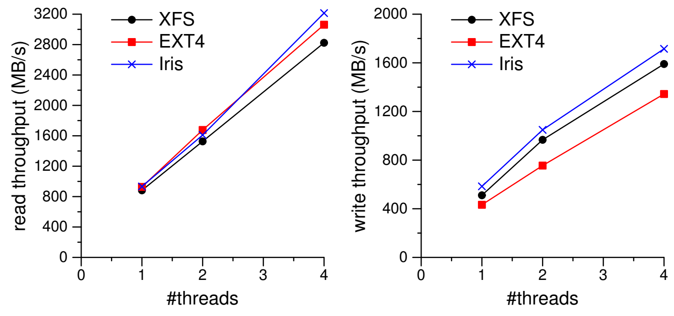

Figure 3: Sequential read/write throughput scaling.

almost the same throughput in sequential reads with the state-of-the-art file systems as the bottleneck is the device. In the case of sequential writes, we achieve about 20% better throughput than what is achievable with *EXT4* and *XFS*, because we have a more lightweight path for write.

In this work, we have focused the evaluation on small random read/write accesses, to better highlight overheads and the improvements achievable with Iris. In random patterns the overheads of the software stack are more pronounced compared to the sequential pattern. Optimizations focusing on throughput, especially for sequential accesses, are outside the scope of this paper, but we expect significant improvements for such access patterns as well. These improvements are a consequence of the design decision to build out key-value store on top of a B^{ϵ}–tree, rather than more commonly used hash-based data structures. As a specific example, to serve sequential accesses, Iris could issue range queries to its underlying key-value store, which then returns the requested blocks in sorted-by-key order. This helps Iris to accelerate sequential accesses. We leave this optimization and its evaluation to future work. Our results for sequential file accesses shows that our approach already perform up to 20% better than state-of-the-art file systems. Using range queries in sequential file accesses would further improve the performance in our case.

6. CONCLUSIONS & FUTURE WORK

In this paper we present Iris, a custom storage system for providing direct access to fast storage devices and minimize system software overheads without sacrificing strong protection semantics. We have implemented a key-value store (Tucana) for storing file data and metadata, and guarantee both atomicity and recoverability. We use processor virtualization features to provide a fast path for protected accesses to the key-value store. We are currently extending our key-value store to support scale-out configurations, so that Iris can utilize fast storage devices at multiple nodes.

In our preliminary evaluation, we have shown improvements up to 1.7× for random read IOPS and 2.2× for random write IOPS as compared with state-of-art Linux kernel file systems using a single core. Performance scales with the number of cores, with up to 1.84× and 1.96× improvement for random read and write IOPS, respectively, using 4 cores. In sequential reads we provide similar performance and in sequential writes we are about 20% better compared to other file systems but with several optimization opportunities. Our future work includes the full implementation of Iris and its extensive evaluation using real applications, including On-Line Transaction Processing (OLTP) and On-Line Analytical Processing (OLAP) workloads.

Acknowledgments

We thankfully acknowledge the support of the European Commission under the Horizon 2020 Framework Programme for Research and Innovation through the ExaNeSt project (Grant Agreement 671553).

7. REFERENCES

[1] Intel VMDq Technology: An Overview, 2008.

[2] AMD. Secure Virtual Machine Architecture Reference Manual.

[3] J. Axboe. Flexible I/O Tester. https://github.com/axboe, 2005.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[5] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. 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)*, pages 335–348, Hollywood, CA, 2012. USENIX.

[6] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 49–65, Broomfield, CO, Oct. 2014. USENIX Association.

[7] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[8] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, and S. Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 385–395, Washington, DC, USA, 2010. IEEE Computer Society.

[9] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[10] F. Chen, M. Mesnier, and S. Hahn. A protected block device for persistent memory. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–12, June 2014.

[11] 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 the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 105–118, New York, NY, USA, 2011. ACM.

[12] 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 the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[13] B. Dieny, R. Sousa, G. Prenat, and U. Ebels. Spin-dependent phenomena and their implementation in spintronic devices. In *VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on*, pages 70–71, April 2008.

[14] DPDK. Data plane development kit. http://dpdk.org/, 2016.

[15] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[16] FusioIO. ioDrive2/ioDrive2 Duo Datasheet. http://www.fusionio.com/load/-media-/2rezss/docsLibrary/FIO_DS_ioDrive2.pdf, 2014.

[17] Y. Ho, G. Huang, and P. Li. Nonvolatile memristor memory: Device characteristics and design implications. In *Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on*, pages 485–490, Nov 2009.

[18] Intel. Persistent memory block driver (pmbd) v0.9. https://github.com/linux-pmbd/pmbd, 2013.

[19] E. Jeong, S. Wood, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park. mtcp: a highly scalable user-level tcp stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, Apr. 2014. USENIX Association.

[20] A. Kegel, P. Blinzer, A. Basu, and M. Chan. IOMMU: Virtualizing IO through IO Memory Management Unit (IOMMU). AMD Tutorial on ASPLOS 2016.

[21] P. Kutch. PCI-SIG SR-IOV primer: An introduction to SR-IOV technology, 2011. Intel application note, 321211-002.

[22] A. Papagiannis, G. Saloustros, P. González-Férez, and A. Bilas. Tucana: Design and implementation of a fast and efficient scale-up key-value store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[23] S. Peter, J. Li, I. Zhang, D. R. K. Ports, T. Anderson, A. Krishnamurthy, M. Zbikowski, and D. Woos. Towards high-performance application-level storage management. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Philadelphia, PA, June 2014. USENIX Association.

[24] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. Anderson, and T. Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, Broomfield, CO, Oct. 2014. USENIX Association.

[25] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. Lung, and C. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, July 2008.

[26] L. Rizzo. netmap: A novel framework for fast packet i/o. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association.

[27] O. Rodeh. B-trees, shadowing, and clones. *ACM Trans. Storage*, 3(4):2:1–2:27, Feb. 2008.

[28] O. Rodeh, J. Bacik, and C. Mason. Btrfs: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

[29] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 33–46, Berkeley, CA, USA, 2010. USENIX Association.

[30] SPDK. Storage performance development kit. http://www.spdk.io/, 2016.

[31] R. Uhlig, G. Neiger, D. Rodgers, A. Santoni, F. Martins, A. Anderson, S. Bennett, A. Kagi, F. Leung, and L. Smith. Intel virtualization technology. *Computer*, 38(5):48–56, May 2005.

[32] P. Varanasi and G. Heiser. Hardware-supported virtualization on arm. In *Proceedings of the Second Asia-Pacific Workshop on Systems*, APSys '11, pages 11:1–11:5, New York, NY, USA, 2011. ACM.

[33] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible file-system interfaces to storage-class memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[34] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, New York, NY, USA, 2011. ACM.

[35] X. Wu and A. L. N. Reddy. Scmfs: A file system for storage class memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[36] J. Xu and S. Swanson. Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, Feb. 2016. USENIX Association.