User-space I/O for μs-level storage devices

Anastasios Papagiannis[†], Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas[†]

Institute of Computer Science, FORTH (ICS), Greece
{apapag,gesalous,maraz,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 *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.

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

1 Introduction

Emerging flash-based storage devices provide access latency in the order of a few μs. Existing devices [14] 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) [21], STT-RAM [11], and memristors [15] may provide even lower access latency, at the scale of hundreds or tens of nanoseconds [8].

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

[0] [†]Also, with the Department of Computer Science, University of Crete.

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 device and network latencies and to allow applications to benefit from increasingly fast storage devices.

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. An important design aspect is the separation of the control and data planes [20] [5]. This idea comes from the area of networking and several frameworks designed in order to take advantage of fast network devices [12]. 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. 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 the storage devices without sacrificing strong protection semantics. We use traps in the data plane for protection rather than using a separate trusted process [24] or server 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 [23] and AMD's SVM [1]) 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 [16] [8], a custom block device that emulates PCM latencies. We show that, per-core, our approach achieves a 1.7× improvement in read IOPS, and 2.2× in 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. We compare *Iris* with the state-of-art Linux kernel file systems, *xfs* and *ext4*.

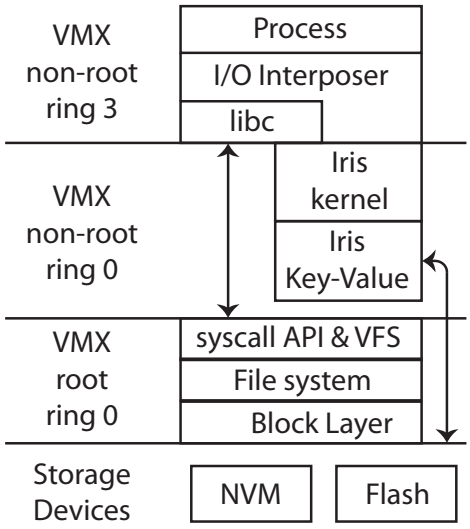

Fig. 1. Top-level architecture of *Iris*.

The rest of this paper is organized as follows. In Section 2 we present the design of *Iris*, and in Section 3 a preliminary evaluation. Section 4 reviews related work. Section 5 concludes the paper and discusses the future work.

2 *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.

2.1 Key-Value Store

Our key-value store is designed primarily for fast storage devices, and is mainly based on Tucana [18]. Its API provides methods for inserting a <key, value> pair and for retrieving a <value> based on a <key>. It also supports range queries which return keys in sorted order. We use range queries in order to

enable better performance for sequential file accesses. At its core, it implements a variant of B^{ϵ}–tree [6], 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.

Its persistence mechanism is based solely on the Copy-On-Write (COW) mechanism [22]. 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: The first one is read-only, while the second one 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. In our 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.

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. To maintain POSIX semantics, for each file the results of the last write must be returned to any subsequent read operation. Although these can be easily implemented using coarse-grain locking, we have implemented a more sophisticated locking protocol to support concurrent reads and writes for different files.

2.2 *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 [23] 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 Virtual Machine Managers (VMMs). Normally, the VMM 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 [4] 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, 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.

2.3 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.

Except from the persistent file metadata that are stored inside the key-value store, the interposer also uses in-memory 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

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

Table 1. Single thread random IOPS (thousands).

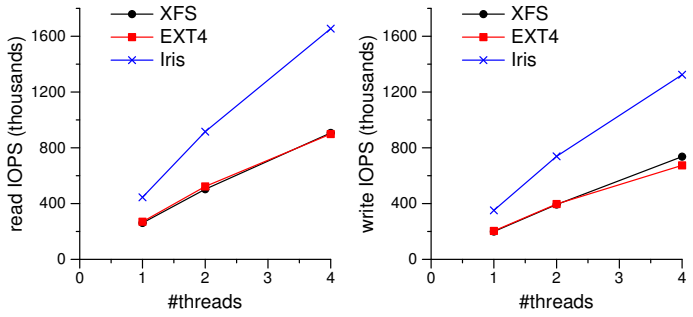

Fig. 2. Random read/write IOPS scaling.

before a failure. We also keep an in-memory copy of persistent metadata, to accelerate metadata operations but without sacrificing correctness.

3 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* [2] to measure random-access read and write IOPS, with a block size of 512 bytes, a 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 [16] [8] 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 filesystems, we also use PMBD as the underlying block device.

Table 1 shows the number of random IOPS for both reads and writes using a single thread. The results obtained from *Iris* have very small variance between the runs. 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.

In this work, we have focused the evaluation on small random read/write accesses, to better highlight overheads and the improvements achievable with *Iris*. 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. To serve sequential accesses, *Iris* issues range queries to its underlying key-value store, which then returns the requested blocks in sorted order. This helps *Iris* to accelerate sequential accesses. We leave this optimization and its evaluation as a future work.

4 Related Work

Recent papers have addressed the issue of how to optimize accesses to fast I/O devices. The Arrakis [20] [19] and IX [5] 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 [3], achieve this by relying on SR-IOV [17] 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 of it 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 [19] 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 [4] 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. We also use Dune for protected accesses to hardware devices but our main contribution is to minimize latency, and we don't require changes to the user applications. Thus IX (i.e. network-specific) optimizations are not suitable for *Iris*, a latency-optimized storage path.

Moneta-D [7] 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.

Another approach to access fast storage devices appeared in Aerie [24]. 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 application 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. We don't have this limitation in our approach, as multiple applications can update their metadata concurrently.

In Mnemosyne [25] and NV-Heaps [9] 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.

Other works like BPFS [10], PMFS [13], NOVA [27] and SCMFS [26] 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.

5 Conclusions & Future Work

In this paper we propose *Iris*, a custom storage system for providing direct access to fast storage devices and minimize system software overheads without sacrificing strong protection semantics. We implement a key-value store for storing file data and metadata, and guarantee both atomicity and recoverability. The key-value store is designed to scale-out by utilizing fast storage devices at several nodes. We use processor virtualization features to provide protected accesses to our key-value store. In the preliminary evaluation, we show 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

Our future work includes the full implementation of *Iris* and its extensive evaluation using real applications, including On-Line Transaction Processing (OLAP) and On-Line Analytical Processing (OLTP) 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).

References

1. AMD: Secure Virtual Machine Architecture Reference Manual
2. Axboe, J.: Flexible I/O Tester. https://github.com/axboe (2005)
3. Baumann, A., Barham, P., Dagand, P.E., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: A new os architecture for scalable multicore systems. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. pp. 29–44. SOSP '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1629575.1629579
4. Belay, A., Bittau, A., Mashtizadeh, A., Terei, D., Mazières, D., Kozyrakis, C.: 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). pp. 335–348. USENIX, Hollywood, CA (2012), https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

5. Belay, A., Prekas, G., Klimovic, A., Grossman, S., Kozyrakis, C., Bugnion, E.: Ix: A protected dataplane operating system for high throughput and low latency. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 49–65. USENIX Association, Broomfield, CO (Oct 2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay
6. Brodal, G.S., Fagerberg, R.: Lower bounds for external memory dictionaries. In: Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 546–554. SODA '03, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA (2003), http://dl.acm.org/citation.cfm?id=644108.644201
7. Caulfield, A.M., Mollov, T.I., Eisner, L.A., De, A., Coburn, J., Swanson, S.: 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. pp. 387–400. ASPLOS XVII, ACM, New York, NY, USA (2012), http://doi.acm.org/10.1145/2150976.2151017
8. Chen, F., Mesnier, M., Hahn, S.: A protected block device for persistent memory. In: Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on. pp. 1–12 (June 2014)
9. Coburn, J., Caulfield, A.M., Akel, A., Grupp, L.M., Gupta, R.K., Jhala, R., Swanson, S.: 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. pp. 105–118. ASPLOS XVI, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1950365.1950380
10. Condit, J., Nightingale, E.B., Frost, C., Ipek, E., Lee, B., Burger, D., Coetzee, D.: Better i/o through byte-addressable, persistent memory. In: Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles. pp. 133–146. SOSP '09, ACM, New York, NY, USA (2009), http://doi.acm.org/10.1145/1629575.1629589
11. Dieny, B., Sousa, R., Prenat, G., Ebels, U.: Spin-dependent phenomena and their implementation in spintronic devices. In: VLSI Technology, Systems and Applications, 2008. VLSI-TSA 2008. International Symposium on. pp. 70–71 (April 2008)
12. DPDK: Data plane development kit. http://dpdk.org/ (2016)
13. Dulloor, S.R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., Jackson, J.: System software for persistent memory. In: Proceedings of the Ninth European Conference on Computer Systems. pp. 15:1–15:15. EuroSys '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2592798.2592814
14. FusioIO: ioDrive2/ioDrive2 Duo Datasheet. http://www.fusionio.com/load/-media-/2rezss/docsLibrary/FIO_DS_ioDrive2.pdf (2014)
15. Ho, Y., Huang, G., Li, P.: Nonvolatile memristor memory: Device characteristics and design implications. In: Computer-Aided Design - Digest of Technical Papers, 2009. ICCAD 2009. IEEE/ACM International Conference on. pp. 485–490 (Nov 2009)
16. Intel: Persistent memory block driver (pmbd) v0.9. https://github.com/linux-pmbd/pmbd (2013)
17. Kutch, P.: PCI-SIG SR-IOV primer: An introduction to SR-IOV technology (2011), Intel application note, 321211-002
18. Papagiannis, A., Saloustros, G., González-Férez, P., Bilas, A.: 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 (Jun 2016), https://www.usenix.org/conference/atc16/technical-sessions/presentation/papagiannis

19. Peter, S., Li, J., Zhang, I., Ports, D.R.K., Anderson, T., Krishnamurthy, A., Zbikowski, M., Woos, D.: Towards high-performance application-level storage management. In: 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14). USENIX Association, Philadelphia, PA (Jun 2014), https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/peter
20. Peter, S., Li, J., Zhang, I., Ports, D.R.K., Woos, D., Krishnamurthy, A., Anderson, T., Roscoe, T.: Arrakis: The operating system is the control plane. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 1–16. USENIX Association, Broomfield, CO (Oct 2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter
21. Raoux, S., Burr, G., Breitwisch, M., Rettner, C., Chen, Y., Shelby, R., Salinga, M., Krebs, D., Chen, S.H., Lung, H., Lam, C.: Phase-change random access memory: A scalable technology. IBM Journal of Research and Development 52(4.5), 465–479 (July 2008)
22. Rodeh, O.: B-trees, shadowing, and clones. Trans. Storage 3(4), 2:1–2:27 (Feb 2008), http://doi.acm.org/10.1145/1326542.1326544
23. Uhlig, R., Neiger, G., Rodgers, D., Santoni, A., Martins, F., Anderson, A., Bennett, S., Kagi, A., Leung, F., Smith, L.: Intel virtualization technology. Computer 38(5), 48–56 (May 2005)
24. Volos, H., Nalli, S., Panneerselvam, S., Varadarajan, V., Saxena, P., Swift, M.M.: Aerie: Flexible file-system interfaces to storage-class memory. In: Proceedings of the Ninth European Conference on Computer Systems. pp. 14:1–14:14. EuroSys '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2592798.2592810
25. Volos, H., Tack, A.J., Swift, M.M.: Mnemosyne: Lightweight persistent memory. In: Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 91–104. ASPLOS XVI, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1950365.1950379
26. Wu, X., Reddy, A.L.N.: Scmfs: A file system for storage class memory. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 39:1–39:11. SC '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2063384.2063436
27. Xu, J., Swanson, S.: Nova: A log-structured file system for hybrid volatile/non-volatile main memories. In: 14th USENIX Conference on File and Storage Technologies (FAST 16). pp. 323–338. USENIX Association, Santa Clara, CA (Feb 2016), https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu