Title

A Tale of Two Abstractions: The Case for Object Space

Authors

Bittman, Daniel

Alvaro, Peter

Long, Darrell DE

et al.

Publication Date

2019-07-08

A Tale of Two Abstractions: The Case for Object Space

Daniel Bittman
UC Santa Cruz
dbittman@ucsc.edu

Peter Alvaro
UC Santa Cruz
palvaro@ucsc.edu

Darrell D. E. Long
UC Santa Cruz
darrell@ucsc.edu

Ethan L. Miller
UC Santa Cruz
elm@ucsc.edu

Abstract

The increasing availability of byte-addressable non-volatile memory on the system bus provides an opportunity to dramatically simplify application interaction with persistent data. However, software and hardware leverage different abstractions: software operating on persistent data structures requires "global" pointers that remain valid after a process terminates, while hardware requires that a diverse set of devices all have the same mappings they need for bulk transfers to and from memory, and that they be able to do so for a potentially heterogeneous memory system. Both abstractions must be implemented in a way that is efficient using existing hardware.

We propose to abstract physical memory into an object space, which maps objects to physical memory, while providing applications with a way to refer to data that may have a lifetime longer than the processes accessing it. This approach reduces the coordination required for access to multiple types of memory while improving hardware security and enabling more hardware autonomy. We describe how we can use existing hardware support to implement these abstractions, both for applications and for the OS and devices, and show that the performance penalty for this approach is minimal.

1 Introduction

Byte-addressable non-volatile memory (BNVM) on the memory bus will soon be commonplace in computer systems [1, 5, 11]. While consistency, data structures, optimization, and file systems have been considered [7, 8, 10, 16, 17, 23, 24, 25, 33, 35, 36], there are two significant avenues of research that have not been as thoroughly explored: first, the effect of persistence on the programming models we currently use, and second, the effect of increasing heterogeneity of the physical address space on OS mediation of increasingly autonomous hardware devices, the software running on them, and their access to physical memory.

The direct access nature of BNVM enables low-latency, memory-style access to persistent data, in which memory-style data structures may have indefinitely long lifetimes. Virtual addresses are the wrong abstraction to facilitate the creation of such structures as programs need to refer to data with a *persistent* pointer that encodes an offset within an *object* (a segment of memory). These objects, identified by a unique ID, contain data with similar access semantics, access control, and lifetime. If these identifiers are never recycled, a persistent pointer of the form $\langle object, offset \rangle$ will *always* remain valid, allowing persistent data structures to maintain pointers that outlive a single process's virtual address space.

In contrast, a hardware view of BNVM has different needs—the hardware need not care about persistence and data relationships. Instead, it needs to coordinate access to objects in physical memory, as requested by applications, while storing and loading to and from memory in an environment where pages of memory could *move* between the different types of physical memory in the physical address space. This coordination problem, arising from the need to move data in physical memory, means hardware and CPUs must operate on a abstract view of memory above physical addresses.

To address these two different, but related, needs, we propose that systems support two new abstractions for object access and mapping: a *global object space* and a *logical object space*. The global object space contains all objects (potentially across multiple systems), allowing persistent pointers to refer to data with long lifetimes and giving software the ability to operate *directly* on persistent structures. The logical object space is an abstraction over physical memory that contains the working set of *actively used* objects local to *one system*. Hardware devices then issues loads and stores to memory by addressing a location within an object instead of physical addresses directly, giving hardware the ability to operate more autonomously with little kernel participation in the system's memory management. We implement these abstractions in part by repurposing virtualization hardware to provide abstraction over physical memory—an ability that is underused outside of full-system virtualization. These two abstractions both allow programmers to easily build applications that manipulate persistent data structures and facilitate OS and hardware management of the underlying memory.

2 BNVM Abstractions

The challenges introduced by BNVM in mapping virtual addresses to physical addresses are the different lifetimes of data with respect to the references to that data and the heterogeneity of physical memory. Software and hardware have different requirements for this mapping, in the following respects:

Latency. Prior persistent data access schemes relied on system calls to operate on persistent storage. BNVM's low-latency can no longer tolerate the cost of those system calls. Instead, programs must have direct access to persistent data via load/store instructions. Similarly, hardware must be able to transfer data to and from BNVM using DMA.

In-memory data structures. Direct access to persistent memory removes the loading and unloading cost, but we must *also* remove the cost of data serialization: since persistent data is located in-memory, programs can operate on data as in-memory data structures using standard programming techniques. This change is not an issue for hardware, which treats data as a "bag of bits".

Data lifetime. If persistent data is stored as in-memory data structures, then applications need a way to refer to data such that references have the same lifetime as the referenced data, a requirement that fundamentally arises from the need for to construct references that encode the relationships between data. Applications can store *persistent pointers* that encode a more persistent name for data than an ephemeral virtual address. Again, hardware has no such requirement, since it neither constructs nor interprets relationships between data. Devices that need to consider data relationships do so in software running atop the hardware.

Mappings. Both software and hardware must have a way of translating a persistent pointer into a physical address. This mapping may change frequently as the OS changes allocation of physical pages to data (*e.g.*, to persist a piece of data). Hardware need only know how to access data in memory for a single operation. In contrast, software must have longer-term mappings, and must be able to support data shared between threads, potentially mapped into the threads in different places.

Memory heterogeneity. Hardware and the operating system must know about different types of memory, *e.g.* DRAM and BNVM. Software need only know how to allocate from the different types, and should not deal with other differences. Hardware, however, will need to know when data is moved between memory types if it wishes to correctly emit loads and stores requested by software.

Our design must support appropriate abstractions for both software and hardware, facilitating the use of *persistent pointers* to long-lived data for applications while providing hardware and the OS with the ability to move data in, out of, and around physical memory, preferably only using existing hardware functionality. We first abstract memory into *objects*, where related data with similar access semantics (*e.g.*, an entire B-tree, where all nodes are subject to the same access control, *etc.*) are placed in a single object and identified by a *globally* unique ID. IDs are formed via hashing, partitioned allocation, or other methods that prevent collisions of IDs across machines with high probability.

We propose two abstractions to provide effective access to objects for both hardware and software: a *global object space*, which facilitates persistent pointers (discussed below), and a *logical object space*, which allows hardware to address data without needing knowledge of the data's physical location. We split these abstractions and consider them different because of the different requirements discussed above; software must worry about reference lifetime, whereas hardware's access to objects is disconnected from persistence. On the other hand, today's hardware must worry about physical location of data, which (with BNVM) is now more likely to change over time.

Note that we are not considering issues of consistency. These have been studied extensively [7, 8, 10, 17, 23, 24, 25, 33], and are orthogonal to the semantic challenges we discuss here. Our work on the global object space does not prescribe a particular solution to consistency, and the logical address space need not be persisted due to its nature as an abstraction over physical space only.

Persistent Pointers Applications running on BNVM must have a way to create pointers that outlast a process's virtual address space and are valid in other virtual address spaces. Previous single address space operating systems (SASOSes) [6, 14, 27, 30] provided this functionality by having a single virtual address space for all processes. However, this approach has several problems. First, the virtual address space must be large, since a global-scale system could require more than a 64-bit address space, especially if the space is allocated sparsely. Second, a SASOS requires a great deal of coordination, since *each* process shares the same virtual address space, even if there is no actual data sharing between processes—a problem that is compounded when coordinating access to data across multiple machines [14, 29].

The use of a global object space constructed of all objects allows a *persistent pointer*, a $\langle object\_id, offset \rangle$ tuple, to refer to any addressable location. An object then need only be mapped into an individual process's virtual address space, and references to that object are simply implemented as $object_{base} + offset$. The virtual address an object is mapped to is per-process, arbitrary, has no bearing on persistent pointers, and exists solely to provide access to the object efficiently with current hardware. This approach, a type of fat pointer [22], is similar to segmentation [3, 9], and can easily be implemented with segment registers, though this feature is being removed from current ISAs. Unlike segmentation in Multics and elsewhere, persistent pointers allow inter-object references to be stored in an object and be usable by another process with no knowledge of the creating process. Persistent pointers also remove the need for different processes to come

to a global agreement on which objects occupy which slots in a process's segment table. As a result, persistent pointers are always valid, assuming that the object to which they point has not been deleted, and facilitate references within a truly global address space that can cross machine boundaries.

Implementing persistent pointers efficiently and flexibly is a key concern for a BNVM system: persistent pointers should be no larger than existing pointers and should be easy to support on current CPUs. A naïve implementation would require very large pointers, since unique object identifiers (OIDs) would be 64–128 bits with an offset of at least 40 bits. To address this, we use an additional level of indirection. Each object X has a *Foreign Object Table* (FOT) listing the objects to which X points, providing support for long object identifiers and flexible naming while allowing all pointers in X to remain 64 bits. A persistent pointer in X is represented as $\langle Z, offset \rangle$, where Z is an index into the FOT. Persistent pointers where $Z = 0$ refer to locations in X, while $Z > 0$ refers to locations in other objects. If *offset* is a 40 bit value, each object can support direct references to $2^{24} - 1$ foreign objects. Since each object has its own FOT, any thread that references an object can easily perform the translation from persistent pointer to a virtual memory reference; preliminary measurements of the performance are discussed in Section 4. Our model, unlike some existing and previous models [15, 26, 28, 34], can be implemented efficiently without additional hardware support, does not increase pointer size, allows a per-object view of the global object space, and yields little change to common, in-memory programming semantics.

Programming Models Made Easier The direct access to persistent data and the memory-like structuring of data make *current* programming tasks easier; programmers no longer need to concern themselves with two programming domains of volatile and persistent data, nor do they need to worry about explicit movement of data and transforming data between a "persistent" form and an "in-memory" form. These access models also invert the traditional systems programming model. Instead of structuring computation in a process-centric manner, persistent data references and objects let us build our programs around *data*, a design goal known to all (programming is all about data) but realized by few, since traditional persistent data access requires significant overhead in terms of programming complexity and OS involvement. Of course, ease of use is key [19]. Providing a standardized, system-wide persistent pointer mechanism is vital, as it presents hardware and compiler vendors with a fixed-target for building persistent pointer support and taking the burden off of programmers.

Hardware and Memory The interface presented to hardware must enable interaction with a *heterogeneous memory system* and support the abstraction of an object space rather than a collection of flat memory spaces. Unlike applications, hardware has no need for creating long-lived pointers to locations within objects. Rather, hardware must have the ability to transfer large chunks of objects to, from, and around memory, and must be able to manage different *types* of memory, preferably in a way that is largely transparent to applications.

Requiring hardware to access a global object space through a $\langle object, offset \rangle$ tuple is unnecessary overhead and complexity—hardware need only ever access the "working set" of objects currently undergoing computation. Thus, our design abstracts the view from hardware of physical memory into a *logical object space*, an address space that maps contiguous objects within it to physical memory pages, managed by the operating system. This solves the heterogeneity problem by allowing hardware to refer to data within objects instead of physical addresses, whose lifetimes are much smaller than the objects, and doesn't require hardware to emit (likely large) addresses for the global address space.

While SASOSes are not a viable solution to the problem of persistent pointers, they are *a* solution to implementing the logical object space. Hardware, and the CPU, are directly connected, reducing the cost of invalidation and coordination of an address space. Additionally, this address space is *intermediate* and hidden from programs—a virtual address is translated to the logical object space, after which the address is translated to a physical location (shown in Figure 1). This translation for applications (after the virtual address translation) and that of hardware is the same, reducing the management complexity for the OS, discussed in Section 3.

Object Space Implementation The logical object space abstraction relies on a single system-wide mapping maintained by the OS which all devices use for accessing memory. The address space is organized by breaking up the flat address space into an array of fixed-size object slots, each of which may be fully or partially occupied by an object. Objects need only be in the map if they are in active use, and the position in the map need not reflect an object's actual identifier nor its location in physical memory (which need not be contiguous).

When an object is needed, it is mapped into a new slot, and hardware can then access the object through its location in the space. Objects may be removed to make room, a process which can occur lazily in the background to remove potential invalidation costs from the critical path. Objects need not be moved in this space—if an object needs to have data moved from one type of physical memory to another, we can update the mappings, and any hardware or software that accesses that object automatically accesses the new physical location. There are further optimizations that can be made to fit the logical object space into a smaller virtual address space.

It is critical that the object space abstraction be implementable on existing hardware—a new model for memory space organization will never be adopted if it requires significant hardware changes. Fortunately, as described in Section 4, the IOMMU and extended page tables originally implemented to support virtualization can be used to support an object space for hardware as well as for software, and can do so more efficiently than existing use of these features.

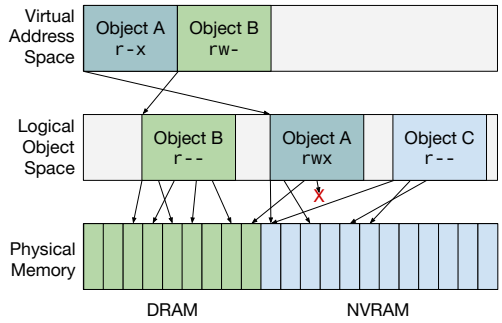

Figure 1: Mapping virtual addresses to "object-based" physical memory using two levels of translation.

3 Implications for OS Design

The operating system's primary role is to manage the two abstractions, providing hardware with an effective means to access objects and providing applications with an effective means to store and access persistent data, while meeting the requirements we discussed earlier. For example, the low-latency requirement discussed in Section 2 requires removing the kernel from the critical path for I/O; thus the kernel need not contain much in the way of persistent data access services and abstractions, and instead relies more heavily on userspace.

The OS in this approach is split, much like an Exokernel [4, 13], with a library operating system that lives in userspace to assist programs with persistent data access—in this case, helping them construct and dereference persistent pointers—while the kernel is responsible for managing the two abstractions discussed above and mediating between hardware and software. Each thread runs in a virtual address space, as usual, but with an additional translation layer underneath (the logical object space), as discussed in Section 2. These spaces are managed by the kernel, where the virtual space is controlled by userspace and the logical object space is managed as needed, adding objects when accessed, and removing them lazily when more space is required.

The logical object space greatly simplifies OS implementation; consider the case of multiple processes using an object that is relocated in physical memory from DRAM and BNVM. Instead of updating each process's address space, and having to do a reverse lookup to find where the object is in each space, the OS need only update the (single) mapping in the logical object space.

Security An immediate consequence of removing the kernel from the data access path is the loss of the traditional access control enforcement. Typically, the OS enforces access control during `open`, giving the process a token which encodes those access control policies (the file descriptor), allowing the OS to enforce the decided rights during calls to `read`, `write`, or `mmap`. Instead, our system allows applications to access data *directly*, and so cannot enforce these decisions in the OS.

Instead, we rely on hardware for enforcement: the mapping hardware we use for the object-space abstraction can also enforce security, if the OS configures mapping tables correctly. In the case of applications, the extended page tables contain the access rights for each object mapped into the space, and the CPU's MMU enforces those restrictions. For hardware, we can use the IOMMU's page-tables to do a similar job. The result is that application and hardware access to data objects is controlled by the same policy and the same basic mechanism, which opens the door to more autonomous operation of hardware. Figure 1 shows permissions of objects in the logical object space along with the requested protections in the virtual address space.

The lack of kernel involvement in persistent data access and the separation of permissions and protections discussed above allows the removal of explicit "open" operations and supports "lazy-binding" of access rights. Traditionally, a process needs to know ahead of time what its access rights are. Consider the case of a program which operates on a file by reading it and only occasionally writing it. It must either open the file read-only and *re-open* it for writing later, handling permission errors, or it must open it read-write and potentially fail to do useful work if it cannot write the file. Instead, if access control is checked and enforced only on *actual access* to the file, the program can do the necessary reads and write when requested, handling a exceptions for permission errors. This model is *much* simpler and more natural than explicitly re-opening the file, and paves the way for more reliance on the operating system for handling access control rather than application-specific solutions. A program can hand out references to data that a recipient may not write, but they need not know that until they try. Moreover, TOCTTOU errors [31] cannot occur, since *each* access is checked by hardware as it is made.

Of course, access control is not static throughout the system; different processes have different access control rights to different objects. We support this by allowing multiple logical object spaces with different access control specified per-object, but with an object occupying the same (virtually-addressed) location across spaces. The OS then maps objects from the logical object space with the appropriate permissions into a process's virtual address space. This does not significantly complicate the management of the logical address space, nor the OS's implementation, because the objects always occupy the same *location*, just with different permissions. The two-level mapping scheme also reduces the number of page-table structures required since we separate protection from security contexts, and so do not need to maintain all combinations of permissions and requested protections.

The result is that the logical address space can also be considered a *security context*, which threads may share. The OS then switches security contexts on context switch. Threads can change security contexts (if allowed) as well, allowing protected non-local control transfers. Similarly, hardware devices can have security roles assigned to them, allowing, for example, a NIC to access only objects associated with its receive and transmit queues.

4 Implementation

We have begun implementing a prototype system with which to evaluate these ideas, including a set of userspace libraries to evaluate our persistent pointer mechanisms and a modified FreeBSD kernel that uses our two-level mapping to abstract physical memory from the point of view of much of the kernel. We are making use of the Extended Page Tables on Intel processors (a similar technology, called Nested Page Tables, exists on AMD).

Since the CPU's addresses are not translated by the IOMMU, but we still want the kernel to operate on the same abstraction as other hardware, we use the virtualization hardware to do two levels of address translation—the first being virtual addresses (which uses the normal address translation hardware) to logical object space, and the second from logical object space to physical memory (which uses the EPT).

The use of the EPT necessitates the use of the rest of the virtualization hardware, since only the guest (operating in VMX-non-root mode) has its addresses translated via the EPT. Thus, we take the approach of having the kernel act as its own hypervisor—during startup, the kernel creates an EPT with an identity map of physical memory, creates a virtual CPU, and launches it. Since the kernel in guest mode still has access to all of physical memory, and we trust the guest implicitly (it is the same kernel), we can avoid nearly every VM exit, with only a few exceptions. More modern processors can take advantage of two additional features to improve performance:

1. The `vmfunc` call allows the guest to switch EPTs efficiently without VM exits. The kernel maintains a list of "known-good" EPTs, and can switch to one of them without an exit.
2. Virtualization exceptions allow the guest to handle EPT violations (similar to page-faults), which correspond to paging operations for objects in our model.

Importantly, our use of virtualization does *not* preclude the possibility of running our system atop a VM or using it as a traditional hypervisor. In the first case, we can make use of nested virtualization[1] or paravirtualization, and in the second, we can start additional virtual CPUs with different protection semantics. We are simply using the virtualization hardware when it is not in use to improve the simplicity of the kernel and align the CPU with what other hardware sees.

Using the IOMMU The IOMMU is capable of providing the same abstraction that we use the EPT for. Existing research on using the IOMMU focuses on either hardware virtualization, hardware protection [18, 20, 21], or partitioning process' access to hardware devices [2, 12]. In contrast, we are using the IOMMU to provide hardware with an abstracted view of data objects instead of physical memory as they access data, improving ease and security when sharing data between devices and enabling more hardware autonomy when accessing physical memory. The performance implications of the IOMMU are well-studied, and most of the performance overhead from the use of the IOMMU comes into play when unmapping DMA buffers (which is costly due to IOMMU invalidation). However, since we are presenting an address space with spaces that can be *lazily* reclaimed, we can avoid much of the invalidation cost. Because all processes on the CPU use the *same* mapping of objects in the logical address space, with multiple spaces only presenting different protections, the number of IOMMU map changes is reduced. Furthermore, the additional management for maintaining DMA buffers and allocations is removed since we are presenting data objects directly to the programs running on the individual hardware components, albeit through the lens of standard access control mechanisms enforced by the translation hardware.

Preliminary Results Since most hypervisors are not structured the way our system is—they need more isolation for the guest OS—we evaluated the cost of memory accesses with virtualization (testing the overhead of the EPT). We read and wrote 256 MiB of data randomly and sequentially under both VMX-non-root mode and as the host, and we found no significant impact to performance, partly because we avoid nearly all possible VM exits.

To evaluate the overhead from our persistent pointers, we repeatedly dereferenced a persistent pointer and measured the overhead. After the first dereference and mapping (which is not discussed here for brevity), we found that dereferencing a pointer to a different object costs 2.4 ± 0.1 ns, while dereferencing a pointer to within the same object costs merely 0.4 ± 0.1 ns, due to an optimized code path. These measurements do not include the cost of the final dereference, since that is required anyway. Note that these pointers allow easy programming for persistent data access that replaces serialization, a much more costly procedure [32].

5 Conclusion

As BNVM becomes commonplace in computer systems, the abstractions governing how software and hardware view memory must change to allow programmers and hardware to efficiently utilize and manage persistent memory. By breaking memory into objects and providing object-based mappings for both applications and devices, software will be able to use a much simpler, more efficient programming model while still providing manageability and security for persistent random access memory. Our preliminary work has shown that this model can provide benefits even on current operating systems, while paving the way for new lightweight operating systems that hold the promise for improved performance and simplicity as computing systems move towards ubiquitous byte-addressable non-volatile memory.

[1] While this has a performance overhead, our system acts more like a hypervisor already, and our use of strong separation semantics and hardware security enforcement means we would see little benefit to virtualizing multiple instances of our system.

Discussion Topics

We have approached our design from the perspective that hardware will grow to be more autonomous (*e.g.*, NICs and special-purpose units operating directly on memory without significant CPU involvement) and that the heterogeneity of systems will continue to increase (*e.g.*, different kinds of memory). We would welcome the storage community's take both on BNVM in the memory hierarchy, and on the use of more independent hardware operating directly on storage.

This paper has proposed that we should treat memory systems built from NVM and DRAM as collections of objects rather than simply as "flat" memory, and that the CPU MMU and IOMMU are the "right" places to manage such abstractions. We seek feedback on whether our design goal—object spaces—is a reasonable one, and whether our approach to the design is likely to improve system security and usability.

The organization of memory into object space allows the system to preserve the semantics of objects (files), and to present it to the CPU and devices directly. This can improve the ability of the CPU and other intelligent devices to manage the memory, since they now are aware of how individual memory pages are associated to make up objects. This approach has equally important benefits for security, since access permissions are specified and propagated on a per-object basis, and can be verified as the memory system is accessed by different devices. Are these benefits sufficient to warrant an object space model moving forward?

The object space model also engenders changes in the programming model, since pointers are now persistent. How much of this should be supported automatically, and how much should be programmer-specified, and how are these implemented? Does the compiler insert code automatically, and what guidance must the programmer provide? There must be some transparency—after all, the programmer would like to take advantage of direct access to BNVM—but there must be some defaults in-place for the complexity this brings.

The model we have proposed works well in the absence of failure, but how do failures of various sizes (*e.g.*, memory cell failures, memory page failures, or simple "object not found") create problems, and how might they be solved? We would encourage feedback on the right failure recovery mechanisms for our approach and the impacts to the system design.

The object space approach has the obvious advantage that, if designed properly, it could span *multiple* individual memory systems using approaches such as those used for remote page faults in a global address space. Is this approach worth the additional complexity that it will require to implement, and would programmers be willing to use such an approach?

Finally, we have been implementing an operating system to make it simple to use an object space-based system. We would enjoy discussion on whether the small number of primitives in the operating system are sufficient, or whether we need additional primitives to create a fully-functional system.

References

[1] Ars Technica. Intel and Micron unveil 3D XPoint, a brand new memory technology, 2015.

[2] Muli Ben-Yehuda, Omer Peleg, Orna Agmon Ben-Yehuda, Igor Smolyar, and Dan Tsafrir. The nonkernel: A kernel designed for the cloud. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*, page 4. ACM, 2013.

[3] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory: Concepts and design. In *Proceedings of the Second ACM Symposium on Operating Systems Principles (SOSP '69)*, 1969.

[4] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, safety, and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, December 1995.

[5] Geoffrey W Burr, Bülent N Kurdi, J Campbell Scott, Chung Hon Lam, Kailash Gopalakrishnan, and Rohit S Shenoy. Overview of candidate device technologies for storage-class memory. *IBM Journal of Research and Development*, 52:449–464, 2008.

[6] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Transactions on Computer Systems*, 12(4):271–307, November 1994.

[7] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2011.

[8] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pages 133–146, Big Sky, MT, October 2009.

[9] Fernando J. Corbató and Victor A. Vyssotsky. Introduction and overview of the Multics system. In *Proceedings of the November 30 — December 1, 1965, fall joint computer conference, part I*, pages 185–196. ACM, 1965.

[10] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. April 2014.

[11] Sean Eilert, Mark Leinwander, and Giuseppe Crisenza. Phase change memory: A new memory enables new memory usage models. In *Memory Workshop, 2009. IMW'09. IEEE International*, pages 1–2. IEEE, 2009.

[12] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. I/O is faster than the CPU – let's partition resources and eliminate (most) OS abstractions. In *Proceedings of the 17th Workshop on Hot Topics in Operating Systems (HOTOS '19)*. ACM.

[13] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, December 1995.

[14] Germont Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The Mungi single-address-space operating system. *Software Practice and Expererience*, 28(9):901–928, July 1998.

[15] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *Proceedings of Eurosys '06*, pages 133–145, 2006.

[16] Dokeun Lee and Youjip Won. Bootless boot: Reducing device boot latency with byte addressable NVRAM. In *2013 International Conference on High Performance Computing*, November 2013.

[17] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. Loose-ordering consistency for persistent memory. In *32nd IEEE International Conference on Computer Design (ICCD 14)*, pages 216–223. IEEE, 2014.

[18] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, volume 50, pages 355–368, New York, NY, USA, March 2015. ACM.

[19] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. Persistent memcached: Bringing legacy code to byte-addressable persistent memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, 2017. USENIX Association.

[20] Alex Markuze, Adam Morrison, and Dan Tsafrir. True IOMMU protection from DMA attacks: When copy is faster than zero copy. *ACM SIGOPS Operating Systems Review*, 50(2):249–262, 2016.

[21] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. Damn: Overhead-free iommu protection for networking. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 301–315, New York, NY, USA, 2018. ACM.

[22] Leonardo Mármol, Mohammad Chowdhury, and Raju Rangaswami. Libpm: Simplifying application usage of persistent memory. *ACM Trans. Storage*, 14(4):34:1–34:18, 2018.

[23] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 401–410, New York, NY, USA, 2012. Association for Computing Machinery.

[24] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. Reducing NVM writes with optimized shadow paging. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association.

[25] Matheus Ogleari, Jishen Zhao, and Ethan L. Miller. Efficient hardware-based undo+redo logging for persistent memory systems. In *Proceedings of the 24th IEEE International Symposium on High-Performance Computer Architecture (HPCA 2018)*, February 2018.

[26] Ariel Pashtan. Object oriented operating systems: An emerging design methodology. In *Proceedings of the ACM '82 Conference*, ACM '82, page 126–131, New York, NY, USA, 1982. Association for Computing Machinery.

[27] Timothy Roscoe. Linkage in the Nemesis single address space operating system. *Operating Systems Review*, 28(4):48–55, October 1994.

[28] Andy Rudoff. Persistent memory programming. In *;Login: The Usenix Magazine*, volume 42, pages 34–40. USENIX Association, 2015.

[29] Alan Skousen and Donald Miller. Using a single address space operating system for distributed computing and high performance. In *Proceedings of the 18th IEEE International Performance, Computing and Communications Conference (IPCCC '99)*, pages 8–14, February 1999.

[30] Frank G. Soltis. *Inside the AS/400, Second Edition*. Duke Communications International, Loveland, CO, 1996.

[31] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably solving file TOCTTOU races with hardness amplification. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST '08, pages 13:1–13:18, Berkeley, CA, USA, 2008. USENIX Association.

[32] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. Morpheus: Creating application objects efficiently for heterogenous computing. In *2016 ACM/IEEE 43rd Annual Intenational Symposium on Computer Architecture*, 2016.

[33] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2011.

[34] Tiancong Wang, Sakthikumaran Sambasivam, Yan Solihin, and James Tuck. Hardware supported persistent object address translation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*, pages 800–812, New York, NY, USA, 2017. ACM.

[35] Jian Xu and Steven Swanson. NOVA: a log-structured file system for hybrid volatile/non-volatile main memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST)*, February 2016.

[36] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: a fault-tolerant non-volatile main memory file system. pages 478–489, October 2017.