A Persistent Problem: Managing Pointers in NVM

Daniel Bittman
UC Santa Cruz
dbittman@ucsc.edu

Peter Alvaro
UC Santa Cruz
palvaro@ucsc.edu

Ethan L. Miller
UC Santa Cruz & Pure Storage
elm@ucsc.edu

Abstract

Byte-addressable non-volatile memory (NVM) placed alongside DRAM promises a fundamental shift in software abstractions, yet many approaches to using NVM promise merely incremental improvement by relying on old interfaces and archaic abstractions. We assert that redesigning the core programming model presented by the operating system is vital to best exploiting this technology. We are developing Twizzler, an OS that presents an effective programming model for NVM sufficient to construct persistent data structures that can be easily and globally shared without serialization costs. We consider and evolve a key-value store that runs on Twizzler, and demonstrate how our programming model improves programmability with early experiments indicating performance need not be lost and may be improved.

1 Introduction

The last decade has seen a surge in research related to byte-addressable non-volatile memory (NVM), yet much of this research focuses narrowly on consistency models or fitting existing interfaces to NVM. Many key issues, including securely and easily sharing persistent data containing persistent references, remain unaddressed. These issues arise from the broader implications of adopting NVM into the memory hierarchy [2]: NVM will fundamentally change the way hardware interacts, the way operating systems are designed, and the way applications operate on data.

The programming model that operating systems provide to software is necessarily influenced by the hardware on which it runs. Current models, such as POSIX, provide an effective programming model for systems with relatively high-latency persistent storage: even an SSD with 50 µs latency suffers only a 2% overhead from a 1 µs system call. However, the advent of NVM as sub-microsecond persistent storage compels us to step back and consider how an operating system and programming environment would look if we built it from the ground up around NVM.

Low-latency persistent storage enables the adoption of a single-level store model such as that used in the IBM i [17], in which all system memory is managed as a single address space, typically by dividing it into variable-sized objects or segments. In such a system, threads must be able to create and store persistent pointers in such a way that other threads can access them directly. But this model introduces other issues, such as ensuring that security is enforced on *each* pointer access, allowing this approach to work across systems (with the attendant impact on a global name space), and exploring the implications of a single-level name space.

While there is some work towards addressing the challenges of managing persistent pointers in a large, persistent, and global address space, these approaches often fall short. PMDK [27], a popular NVM programming framework, cannot achieve these goals because it does not provide a scalable data sharing solution nor does it integrate security with its design, instead outsourcing these key concerns to file and block-based interfaces (such as POSIX). An effective NVM programming model will instead internalize the lessons learned from single-address space OS and single-level store research, and will reconsider the *entire* system stack since the solution to these challenges will have ramifications to how we build secure applications and share persistent data. Current operating systems' abstractions no longer align well with the model the hardware provides.

Our earlier work [5] discusses the abstractions necessary for software and hardware to effectively use NVM. To address these challenges and needs, we are in the process of building *Twizzler*, a clean-slate OS design for NVM that nonetheless can provide backwards compatibility existing programs. This paper extends our previous work by considering the effects of security and data sharing in a global address space for our proposed software abstractions and design choices; in particular, we provide a programming model with a scalable data sharing solution, support for late-binding in both naming and access control, and a security model that checks each data access with little kernel involvement. In this paper, we discuss four aspects of the Twizzler programming model:

1. A programming model that stores context necessary to access data (*i.e.* object IDs, names, access rights, etc) with the data itself and not as ephemeral state.
2. Data references which are not only persistent but *globally* valid without the need for complex coordination.

PLOS '19, October 27, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-7017-2/19/10…$15.00
https://doi.org/10.1145/3365137.3365397

3. A security model that simplifies and combines access control and data access, avoids TOCTTOU [33] errors, and enables "late-binding" on access rights by checking rights at access time, thus reducing the complexity of implementing access control for NVM programs.
4. A naming system that eliminates the distinction between memory and storage, gives programs flexibility in how names are resolved, and provides late-binding between user-friendly names and unique IDs—an often missing feature in NVM programming frameworks.

2 Background and Motivation

Our target NVM technologies are both byte-addressable and directly accessible with a latency similar to that of DRAM [18]. Our prior work discusses the implications of these characteristics on the abstractions for accessing persistent data that should be offered to hardware and software [5]. The insights into how this affects programming models were:

1. Abstractions must enable low-latency access. The high latency of a system call compared to NVM device latency means that the kernel *cannot* be involved in the majority of persistent data access [7, 34], and *every* kernel interposition must be accounted for.
2. Serialization when persisting data is unnecessary for NVM—storing persistent data as in-memory data structures leads to a simpler programming model where applications are primarily an expression of data structures. Additionally, serialization is expensive. In a two-tier memory hierarchy that requires data movement this cost can be tolerated, but that cost *cannot* be tolerated with NVM since it becomes a significant overhead when accessing data.
3. Persistent pointers, which implement references via *identity* and not ephemeral *location*, are vital for software to construct data structures in a world of persistent memory. The implementation details of persistent pointers are critical, as increasing their size has negative effects on performance and memory use. In Twizzler, persistent pointers are 64 bits. Virtual addresses are insufficient as they are ephemeral and cannot encode identity-based data references for objects in a distributed, global address space without dramatically increasing pointer size or incurring coordination costs.

We explicitly avoid providing a specific NVM crash consistency model or mechanism. While crash consistency is important in the context of NVM, and well-studied [9, 10, 12, 20, 22–25, 35], our work does not have any inherent requirements for a particular approach to consistency for applications. Moreover, we do not wish to restrict ourselves to a particular approach while the research is still in flux; instead, we choose to leave the choice up to the programmer.

Incremental Approaches are Insufficient Our design is a significant departure from current storage abstractions; however, we firmly believe that such a sweeping change is necessary to take full advantage of NVM. POSIX's file access model is commonly bypassed by new frameworks [1] and it requires increasing effort to fit new technologies and techniques into it. NVM's characteristics demand that interfaces such as `read` and `write` be bypassed. While interfaces like `mmap` improve the situation somewhat when combined with direct-access to NVM (DAX) [37], `mmap` still cannot hand out persistent, identity-based references, and it relies on file descriptors thus complicating the programming model by requiring the use of POSIX along-side a persistent memory API. These interfaces also fall short compared to our security model, requiring a separation of access control mechanisms from data access mechanisms, leading to more work for programs as discussed in Section 3.3.

A clean-slate approach not only reduces the complexity of the system by dropping the need to support outdated interfaces, but it also enables redesigning interfaces and the OS for direct access to byte-addressable memory instead of spending effort extending APIs indefinitely. Removing unnecessary layers of software and designing the API for modern hardware with an eye towards the future has the potential to enable vast improvements in programmability, data sharing, and security—but this will only be possible if our design decisions permeate all levels of the software stack. While these are our primary goals, we also expect an improvement in performance to arise from the removal of serialization and simplification of software layers.

Related Work Our design for object-based and segment-style memory derives from fundamental and ongoing OS research [3, 4, 6, 8, 14, 16, 19], though we differ by considering how persistent memory affects a global address space. The emphasis on userspace-centered runtimes is based on Exokernel and libOS [4, 6, 14] work. Though we avoid a single, virtual address space solution (it is insufficient for persistent pointers [5]), we take lessons such as linkage and sharing complexity from single-address space OSes [8, 16, 26, 30]. Single-level stores [28, 29] form a backbone of our design, which treats *all* "storage" as a single level of persistent memory, using common techniques such as page faults to move data to and from block or object-oriented devices such as SSDs. Some of these approaches, in particular issues around orthogonal persistence [11], are relevant to NVM, while page-faulting is less so. NVM research that focuses on programming models [9, 35] provides abstractions for persistent structures, but often focuses on memory safety and consistency [10, 24], which, while vital, does not address the issues presented here. While some frameworks and OSes enable more flexible references [27] without the single-address space approach, these too have limitations (see Section 4). NVM filesystems [13, 36] provide vital insights for organizing data on NVM; however they are often tied to POSIX which we try to avoid, thereby often needing interfaces such

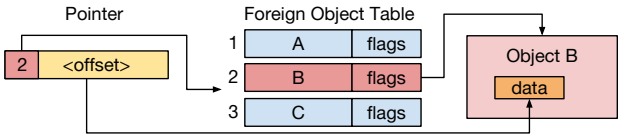

Figure 1. A persistent pointer and the FOT. The pointer indexes into the FOT to retrieve the object ID of the target.

as `mmap` (with DAX) that do not fit our model of in-memory data structures without virtual addresses, and conflate naming data with storing and organizing data (which we also wish to split up in our model).

Case Study: A Key-Value Store To better understand how software is written in our model, consider an example application: a key-value store (KVS). The following section will discuss our programming model in more detail, the benefits of our design choices, and the concrete implications these have on the features and design of the KVS. A KVS that takes best advantage of NVM will both be low-latency and avoid unnecessary copying. Since the keys and values are stored in persistent memory that is directly addressable, a lookup operation should be able to return a direct pointer to the persistent value that the application can copy if it wishes.

Our KVS, `twzkvs`, was built in roughly 250 lines of C code, is low-latency (see Section 5), and can hand out direct pointers. It supports a `put` operation (`kvs_put(K k, V v)`), a get operation (`kvs_get(K k) -> &V`), and a `delete` operation (`kvs_del(K k)`). We also built a similar KVS that ran on a standard Unix system (`unixkvs`).

3 NVM Programming Model

Twizzler partitions data into *objects* which represent units of semantically similar data (*e.g.* similar access control, identification, or meaning). For example, a B-tree could be one object, or multiple objects (if the programmer would like to protect parts of the tree differently than others). Each object is identified with a globally unique 128 bit ID, generated by a method that produces an extremely low chance of collision without needing coordination, such as random generation, machine-specific prefix or (for immutable objects) hashing.

Persistent pointers have the form $\langle object\_id, offset \rangle$, similar to PMDK [27] and other systems. Unlike other systems, however, Twizzler uses indirection to avoid storing the (large) object ID within the pointer. The object ID is stored in a per-object indirection table [5], called the *Foreign Object Table* (FOT), which enables external reference IDs to be encapsulated within the source object while keeping pointers small. A pointer encodes an index into the FOT and an offset within the target object as $\langle fot\_entry, offset \rangle$, as show in Figure 1. The pointer can then be read from the source object and used to lookup the corresponding entry in the source object's FOT to resolve the identity of the target object. Our approach enables late-binding through this indirection, both for security and for naming, as discussed in the next section.

3.1 Cross-object Data References

Since Twizzler's pointers are formed by an object ID and an offset, data in one object can refer to data in another. Data relationships between semantically separate pieces of data are common in systems, such as the myriad of interrelated configuration files in `/etc`. Relationships are often either implicit (the contents of `/etc/passwd` and `/etc/shadow` depend heavily on each other, yet there is no explicit reference), leading to debugging difficulty, or clumsily explicit (*e.g.*, storing an entire path and needing to parse it).

In contrast, Twizzler's *cross-object pointers* enable the programmer to explicitly encode data relationships, allowing the programmer to express their application as operating on data structures with each per-object FOT providing sufficient context for the system to follow references. In `twzkvs`, the index and the data are stored in separate objects. The index object, I, has pointers to data in the data object D. When a program performs a lookup, the KVS uses the index to locate a pointer to the data in object D, which it then hands back to the caller. Since the pointer is stored in object I, the caller can dereference the pointer by using object I's FOT. We can also have multiple data objects for the KVS if we want to use different access control on each, described in Section 3.3, and multiple indexes sharing different data objects to limit discoverability or index the same data differently. Both of these strategies are no more difficult to implement than one index and one data object; in fact, it is the same code since Twizzler makes cross-object pointers a first-class abstraction.

Because objects can easily store cross-object pointers, we gain a significant simplicity over current programming models. Traditionally, the data managed by a KVS in Unix would be stored in a single file, adding the complexity of managing two different namespaces of data in a single, linear address space. When implementing `unixkvs`, we chose to mirror the `twzkvs` implementation by separating the index and data into two files. However, either we need to store full paths in the index file, leading to increased complexity for parsing, opening, and reading during lookup, or the two have an implicit relationship and need to be kept together manually.

Another benefit of the FOT is that objects are self-contained. Any thread can pull in the index object and understand it without needing per-process context from the KVS's creator. This is made possible by our choice of persistent pointers *and* the choice to store the context needed to access data with the data—in this case, the necessary context for all external references from object O are stored within object O itself.

Note that we do not reduce flexibility compared to the example above of storing paths. This is a form of *late-binding*, where the binding of a name to data is resolved at access time. We provide a similar ability; instead of storing an object ID in an FOT entry, an object can instead store a name along with a pointer to a *name resolver* function. The name is passed to the name resolver (for which Twizzler provides a default if

none is specified), which then returns an object ID. Names provide flexibility for programmers, both because they can be human-readable (like a Unix path) and because the ID to which a name refers to can change over time. For example, an object might have a pointer to a piece of data that is information about today's weather. Instead of needing to update FOT entries every day, the object stores a name that resolves to the appropriate ID, allowing objects to easily reference data whose exact identity might change over time.

In addition to enabling late-binding, the consolidation of object IDs for external references into a single, known location per-object allows the construction of system utilities which can operate on the FOT. Given some objects, and a set of references between them, a generic utility can be written to change the object to which a pointer refers. For example, if object A references object B at an offset x, we can change the reference from B to C without needing to know where the pointer is stored in A or any other semantic information about the contents of A. If A has multiple pointers to B and we wish to update all of them, we can do so in a single operation, thereby reducing the possibility of leaving A in an inconsistent state.

Of course, generic system software cannot be written if *offsets* need to change as they are based on semantic information that generic system software may not have access to. This is also a potential problem with using late-binding. However, applications can anticipate changing object formats. For example, shared libraries are a common form of late-binding; binaries refer to the latest version of a particular library. In Twizzler, we allow this too; however, directly referencing a library function from the caller's code is brittle as the location could change with each version. Instead, Twizzler binaries store a pointer to a jump table within the library. Each library can standardize its table, similar to symbol tables and procedure linkage tables in ELF executables, but with less runtime processing and overhead.

3.2 A Global Object Space

Our vision for an NVM programming model is not limited to a single machine. The design choices in Twizzler naturally extend to sharing object across machines. Object IDs, being 128 bits, are large enough for a truly global address space, where *all* objects across *all* machines can be named by a unique ID. Object IDs can be formed via random number generation (like UUIDs) or via concatenating machine-specific identifiers with machine-local unique IDs (*e.g.* using a MAC address). More importantly, if 128 bits proves insufficient, we can easily increase the size of an ID through only modifying FOT entries with generic software, as described above.

The original requirement for persistent pointers is to be "correct" regardless of ephemeral mapping location. Since object IDs are also globally unique, persistent pointer correctness is independent of the machine the object is on. Of course, a given machine may not be able to *access* a given object, but moving an object from one machine to another via shallow-copy will at least not change the meaning of the pointers in the object. This is a vital property for avoiding expensive serialization and solves the problems encountered by single-address space OS approaches when dealing with coordination across machines [16, 31].

Another factor that improves data sharing across a network is that objects are self-contained: objects can be moved to another machine after creation without coordination on data references or copying additional state. The combination of globally unique object IDs and self-contained objects is powerful and greatly simplifies data movement through a network. Consider an application creating a index and data object using `twzkvs`. The index object could be shipped over to another machine without the data object—perhaps the other machine wants to check for the presence of a key. However, if the other machine wishes to *access* the data, it needs to dereference the pointer. If it does not have the data object, it could send out a request to get it. Managing multiple copies of objects, whether they function as caches to reduce latency or replicas to provide fault tolerance, opens up a large design space. As with crash consistency, our goal is to keep the design of Twizzler orthogonal to coherency concerns so as to avoid constraining possible designs.

The other machine need not worry about translating object IDs and state from the source machine. Moreover, if the machine *cannot* get access to the data object (perhaps for security reasons), it will fail to dereference the pointer. It would be *disastrous* if object IDs were not globally unique, as the machine would have a chance of "successfully" dereferencing the pointer into some other data object, leading to extremely difficult-to-track bugs.

3.3 Security for NVM

Our prior work discussed some of the design constraints on a security system for enabling effective hardware cooperation with other hardware and software [5]; here we discuss the implications for software, where we need to protect against unwanted accesses without significant kernel involvement, leading to enforcement via hardware checking at access time.

A primary goal of our security model is to allow software to set access control policy through persistent objects without needing to use a separate API like POSIX files, thus unifying the way software accesses and controls access to persistent data. Twizzler defines access rights to objects through cryptographically signed capabilities [15, 32] that encode which objects can access other objects (for brevity, we elide some details, *e.g.*, objects can have default permissions). Since access rights are defined for persistent objects the kernel must understand the persistent object abstraction. If this were *not* the case, additional context would be needed to map between object IDs and something the kernel understands, such as a path.

Enforcement and Late-Binding Taking the kernel out of the persistent data access path requires reliance on hardware to enforce access control on data accesses. Unix enforces access control at open, resulting in a token that encodes access control as a snapshot of permissions when the token was granted (*e.g.* file descriptors in Unix). However, our model does not *have* explicit open operations, instead replacing the act of opening a file with dereferencing a pointer for which we must rely on the MMU (with help from the OS as needed) to enforce access control policy. This means that checking access rights at access time is a *requirement* resulting from our reduced kernel involvement and reliance on hardware.

In practice, the on-open access control check provided by POSIX is often too rigid. Consider `twzkvs`: a program that needs only read access to the KVS data most of the time should *not* need to `mmap` it read-write every time it simply reads data, or remap it if it needs to write. In POSIX, such a program would need to either re-open the file as writable, or open it read-write from the start, potentially failing to make progress if the program does not have write access.

In Twizzler, we propose another use of late-binding: only bind access permissions as they are needed. This allows a program to *request* access to objects with rights they may not have. In the above example, the program operating on `twzkvs` can request read-write access to the data object by marking the FOT entry with "read-write". If the program does not have write access to the data object, but never writes to it, it is allowed to perform its task. Implementing similar functionality in `unixkvs` was much more complex and required a significant amount of support code.

A consequence of this model is that applications can rely on the system-wide access control to provide a more fine-grained access control in their domain. In `twzkvs`, the index can point to multiple data objects, each with different access rights. When inserting a key-value pair, a program can choose which security level it wishes that pair to have. Later, a program can look up a value given a key, but only read it if the object's permissions allow. We were able to leverage Twizzler's built-in access control mechanism without a single line of code in `twzkvs` dedicated to access control enforcement, while maintaining its status as a library *and* still handing out direct pointers to data.

The Role of the Kernel With the model discussed thus far, the kernel is rarely involved in data access. The only time the kernel is involved is checking access control policy for objects on first access (handling a page fault) and setting up the MMU to enforce policy. The kernel must also invalidate existing MMU access controls on an object whenever rights on an object are changed; subsequent accesses to that object must go through a rights check again. This simple approach reduces the risk of using stale permissions and prevents TOCTTOU errors [33], since rights are checked on each access and invalidated immediately when they are modified.

4 The State of the Art

This section compares our choices to those of existing persistent pointer frameworks with a particular eye towards PMDK [27], since it is arguably the leader in this space. Again, we are not concerned with consistency and coherence, instead focusing on programmability, flexibility, and security.

Programmability Like Twizzler, PMDK stores pointers as $\langle object\_ID, offset \rangle$. However, where Twizzler uses indirection through the FOT, PMDK stores the object ID directly with the offset, as a type of fat pointer [21] that doubles pointer size. This eliminates the benefits enabled by our use of the FOT as an indirection table: not only does PMDK not support late-binding, a vital feature to avoid constantly rewriting references when data is updated, but it becomes impossible to change the ABI in a backwards-compatible way. Twizzler can increase ID size with minimal and controlled effect on object format since such software can be written generically in Twizzler and it only affects the FOT, not the pointers themselves. In PMDK, pointer *length* would would be affected, limiting the ability to update them.

As discussed earlier, Twizzler can have generic software that operates on FOT entries. PMDK's choice to store object IDs in pointers limits its ability to do this; programmers need to provide application-specific solutions for all programs to update object IDs. This is worsened by the lack of late-binding; whereas in Twizzler updating an object ID or name might be a rare operation, PMDK either makes this operation much more common or requires an application-specific late-binding solution, both of which dramatically add to complexity. Furthermore, if Twizzler *does* wish to update all pointers within an object O that refer to object A to instead refer to object B, it can do so in *one* operation, as described in Section 3.1. Other persistent pointer frameworks typically require multiple operations, one per pointer, to accomplish the same goal, with the risk of leaving an object in an inconsistent state after a power loss.

Security Most persistent memory programming libraries do not attempt to handle security, which results in more complexity for programmers and leaves our late-binding benefits on the table. Not only do such libraries need to rely on existing security models (which are flawed for the reasons discussed above), but they also fail to unify the language of access control with the language of persistent data. This means if a programmer wishes to define and enforce access control on a PMDK object, they must do it through the POSIX interface of the system, a *completely* separate interface than the one with which they are programming their application.

Object ID and Pointer Size A significant motivation behind the FOT was to allow Twizzler to leverage the advantages of both large object IDs and narrow pointers. *Small object IDs* have a negative impact on sharing data: PMDK stores 64 bit object IDs that cannot easily be made globally

unique, instead requiring either complex coordination to manage the ID space or limiting object IDs to a per-machine space, limiting PMDK's scalability. *Wide pointers* negatively affect both performance and memory use, and make it more difficult for processors to operate atomically on them. In contrast, Twizzler uses indirection through the FOT to support large object IDs (128+ bits) in narrow, 64-bit pointers. While PMDK's approach has an overhead of 8 bytes *per pointer*, the FOT only adds a 32-byte overhead *per entry*, and entries can be reused. Consider `twzkvs`, where an index object has numerous pointers to a data object. An index with 50 key-value pairs has 100 pointers into an external data object, so the FOT approach adds an overhead of 4% per pointer while PMDK has a 2× overhead (3× if PMDK used 128 bit IDs).

Per-machine object IDs further limits usability in a networked environment. Should a program wish to share an object in Twizzler, it can ship the object alone and all pointers will be correct. In a per-machine object ID space, sharing an object necessitates swizzling the pointers from the source machine to work on the target machine, updating pointers as they are dereferenced, a complex, slow, and error-prone process whose semantics leak to the application programmer.

Moving Forward As discussed above, we believe PMDK to be an incremental improvement to programming persistent memory, as it does not address issues of OS design, sharing, and security. However, PMDK offers a useful position in bridging the gap between the model of Unix and Twizzler. Applications for PMDK can be easily ported to Twizzler, since many of the interfaces are similar.

While we wish to replace the POSIX file model for NVM, much of POSIX does not need to be attacked for an effective NVM programming model, and Twizzler provides a POSIX interface for porting existing programs (in which `read` and `write` reduce to `memcpy`). However, the effects of replacing the file access aspects of POSIX are more pervasive than may appear. In replacing the file access model, we largely do away with file descriptors. Much of the importance of a process is tied to file descriptors, so we no longer need the vast amount of internal kernel state that defines a process. Part of our work on Twizzler will involve exploring the implications of this reduction in internal kernel state for processes.

5 Performance Results

While Twizzler is still in its infancy, we implemented `twzkvs` on it and measured its performance. We stress that the goals of our work are to improve programmability and security in NVM at an acceptable performance overhead, with any performance improvements being a nice side-effect. The Twizzler prototype was implemented by modifying a FreeBSD kernel to provide a new set of OS interfaces that Twizzler threads can use exclusively (without needing to use FreeBSD services), as described in our prior work [5].

Table 1 shows the insert and lookup latency for both `unixkvs` and `twzkvs`. We find a small performance improvement in `twzkvs`, likely because `unixkvs` occasionally parses paths (though we did allow it to cache them) whereas Twizzler avoids such serialization. This shows promise that the Twizzler approach improves programmability and security without a significant performance cost.

Table 1. Insert and lookup latency for `twzkvs` and `unixkvs`, measured on an Intel Core i5-7600K CPU at 3.8 GHz.

KVS	Insert Latency (ns)	Lookup Latency (ns)
`twzkvs`	179.5 ± 2.5	63.6 ± 1.8
`unixkvs`	180.3 ± 3.3	66.0 ± 3.0

6 Conclusions & Future Work

To take full advantage of byte-addressable NVM, we must consider approaches to NVM programming and system management that require redesign of the full system stack, removing the operating system from the data access path while simplifying programming abstractions. Incremental research allows us to learn important implementation details for properly interacting with NVM technologies and to quickly deploy NVM on existing systems, but leaves applications with problems such as serialization and security management that clean-slate, NVM-focused systems such as Twizzler can solve in much simpler ways than existing approaches. When rethinking the design of a system around a particular motivating trend, we must be careful to not turn a blind eye towards other concerns and, in particular, how these concerns interact. Twizzler, unlike some existing approaches, provides mechanisms for a new security model, late-binding and flexibility, generic evolution of the persistent pointer format, and inter-machine data sharing because those concerns couple tightly with persistent data structures and *must* not be an afterthought when designing effective programming models for new technologies.

We are continuing to develop Twizzler, and are prototyping it as a standalone OS with support for networking and key-value SSDs as a longer-term, lower-performance storage medium allowing unused objects to be paged out to lower-cost media. With this standalone OS, we will be able to further demonstrate the utility of the approach we are taking, and will further explore the implications of running Twizzler in a distributed environment.

Acknowledgements

This work was supported in part by NSF grant number IIP-1266400 and by the industrial partners of the Center for Research in Storage Systems. The authors additionally thank the members of the Storage Systems Research Center for their support and feedback. We would like to extend our gratitude to our paper shepherd, Stephen Kell, and the anonymous reviewers for their feedback and assistance.

References

[1] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason Nieh. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 19. ACM, 2016.

[2] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th Workshop on Hot Topics in Operating Systems (HotOS '11)*, May 2011.

[3] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory: Concepts and design. In *Proceedings of the 2nd 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] Daniel Bittman, Peter Alvaro, Darrell D. E. Long, and Ethan L. Miller. A tale of two abstractions: The case for object space. In *Proceedings of HotStorage '19*, July 2019.

[6] Dan Schatzbergand James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. EbbRT: a framework for building per-application library operating systems. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 671–688, November 2016.

[7] Adrian M. Caulfield, Arup De, Joel Coburn, Todor Mollov, Rajesh Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of The 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*, pages 385–395, 2010.

[8] 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.

[9] 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 '11)*, pages 105–118, March 2011.

[10] 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.

[11] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. Grasshopper: An orthogonally persistent operating system. *Computing Systems*, 7(3):289–312, June 1994.

[12] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. System software for persistent memory. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys '14)*, April 2014.

[13] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff 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.

[14] 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)*, pages 251–266, December 1995.

[15] Li Gong. On security in capability-based systems. *ACM SIGOPS Operating Systems Review*, 23(2):56–60, April 1989.

[16] 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.

[17] IBM. IBM i: a platform for innovators, by innovators. an executive guide to the strategy and roadmap for the IBM i integrated operating environment for power systems. https://www.ibm.com/downloads/cas/AD9PJJQ2. Accessed 2019-07-30.

[18] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic performance measurements of the Intel Optane DC persistent memory module. *CoRR*, abs/1903.05714, 2019.

[19] 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 the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '06)*, pages 133–145, New York, NY, USA, 2006. ACM.

[20] 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.

[21] Leonardo Marmol, Mohammad Chowdhury, and Raju Rangaswami. LibPM: Simplifying application usage of persistent memory. *ACM Transactions on Storage*, 14(4), December 2018.

[22] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '12)*, pages 401–500, March 2012.

[23] Yuanjiang Ni, Jishen Zhao, Daniel Bittman, and Ethan Miller. Reducing NVM writes with optimized shadow paging. In *Proceedings of the 10th Workshop on Hot Topics in Storage and File Systems (HotStorage '18)*, July 2018.

[24] Yuanjiang Ni, Jishen Zhao, Heiner Litz, Daniel Bittman, and Ethan L. Miller. SSP: Eliminating redundant writes in failure-atomic NVRAMs via shadow sub-paging. In *Proceedings of the 52nd IEEE/ACM International Symposium on Microarchitecture*, October 2019.

[25] Matheus Ogleari, Ethan L. Miller, and Jishen Zhao. Steal but no force: Efficient hardware-driven undo+redo logging for persistent memory systems. In *Proceedings of the 24th International Symposium on High-Performance Computer Architecture (HPCA 2018)*, February 2018.

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

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

[28] Jonathan S. Shapiro and Jonathan Adams. Design evolution of the EROS single-level store. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 59–72, Monterey, CA, June 2002. USENIX.

[29] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: A fast capability system. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM.

[30] Alan Skousen and Donald Miller. Operating system structure and processor architecture for a large distributed single address space. In *Proceedings of the 1998 Conference on Parallel Distributed Computing and Systems (PDCS '98)*, 1998.

[31] 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.

[32] Andrew S. Tanenbaum and Sape J. Mullender. An overview of the Amoeba distributed operating system. *ACM SIGOPS Operating Systems*

Review, 15(3):51–64, July 1981.

[33] 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.

[34] 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.

[35] 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 '11)*, March 2011.

[36] 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'16, pages 323–338, Berkeley, CA, USA, 2016. USENIX Association.

[37] 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. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, pages 478–489, October 2017.