Twizzler: a *Data-Centric* OS for Non-Volatile Memory

Daniel Bittman and Peter Alvaro, *UC Santa Cruz;* Pankaj Mehra, *IEEE Member;* Darrell D. E. Long, *UC Santa Cruz;* Ethan L. Miller, *UC Santa Cruz / Pure Storage*

https://www.usenix.org/conference/atc20/presentation/bittman

This paper is included in the Proceedings of the
2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

Twizzler: a *Data-Centric* OS for Non-Volatile Memory

Daniel Bittman
UC Santa Cruz

Peter Alvaro
UC Santa Cruz

Pankaj Mehra
IEEE Member

Darrell D. E. Long
UC Santa Cruz

Ethan L. Miller
UC Santa Cruz
Pure Storage

Abstract

Byte-addressable, non-volatile memory (NVM) presents an opportunity to rethink the entire system stack. We present Twizzler, an operating system redesign for this near-future. Twizzler removes the kernel from the I/O path, provides programs with memory-style access to persistent data using small (64 bit), object-relative cross-object pointers, and enables simple and efficient long-term sharing of data both between applications and between runs of an application. Twizzler provides a clean-slate programming model for persistent data, realizing the vision of Unix in a world of persistent RAM.

We show that Twizzler is simpler, more extensible, and more secure than existing I/O models and implementations by building software for Twizzler and evaluating it on NVM DIMMs. Most persistent pointer operations in Twizzler impose less than 0.5 ns added latency. Twizzler operations are up to 13× faster than Unix, and SQLite queries are up to 4.2× faster than on PMDK. YCSB workloads ran 1.1–2.9× faster on Twizzler than on native and NVM-optimized SQLite backends.

1 Introduction

Byte-addressable non-volatile memory (NVM) on the memory bus with DRAM-like latency [23, 38] will fundamentally shift the way that we program computers. The two-tier memory hierarchy split between high-latency persistent storage and low latency volatile memory may evolve into a single level of large, low latency, and directly-addressable persistent memory. Mere incremental change will leave dramatic improvements in programmability, performance, and simplicity on the table. It is essential that operating systems and system software evolve to make the best use of this new technology.

These opportunities motivate us to revisit how programs operate on persistent data. The separation of volatile memory and high-latency persistent storage at the core of OS design requires the OS to manage ephemeral copies of data and interpose itself on persistence operations, a penalty that will consume an increasing fraction of time as NVM performance increases [64]. The direct-access nature of NVM invites the use of load and store instructions to directly access persistent data, simplifying applications by enabling persistent data manipulation without the need to transform it between in-memory and on-storage data formats. Thus, the model that best exploits the low latency nature of NVM is one in which persistent data is maintained as in-memory data structures and not serialized or explicitly loaded or unloaded. To avoid serialization, this model must support *persistent pointers* that are valid in *any* execution context, not just the one in which they were created.

Trying to mold NVM into existing models will not enable its fullest potential, just as SSDs did not reach their full potential until they transcended the disk paradigm. To explore a "clean-slate" approach, we are building Twizzler, an OS designed to take full advantage of this new technology by rethinking the abstractions OSes provide in the context of NVM. Twizzler divides NVM into *objects* within a global object space, and pointers are interpreted in the context of the object in which they reside. This decouples pointers from the address space of an individual thread, providing a data-centric programming model rather than a process-centric one. The result is a vastly simpler environment in which the OS's primary function is to support manipulating, sharing, and protecting persistent data using few kernel interpositions.

We implemented a simple, standalone kernel that supports a userspace for NVM-based applications, with compatibility layers for legacy programs. We wrote a set of libraries and portability layers that provide a rich environment for applications to access persistent data that takes into account both semantics (persistent pointers) and safety (building crash-consistent data structures). We then performed a case-study by writing software for Twizzler, taking into account the new flexibility and power gained by our model and evaluating our software for complexity and performance. We ported SQLite to Twizzler, showing how our approach can provide significant performance gains on existing applications as well.

In a world where in-memory data can last forever, the context required to manipulate that data is best coupled with the *data* rather than the process. This key insight manifests itself in the three primary contributions of this paper:

- We discuss (§ 2) our vision for a data-centric OS and the requirements that it must meet to provide low latency memory-style access to NVM with efficient data sharing.
- We present Twizzler (§ 3) and describe its mechanisms to meet those requirements, including decoupling traditionally linked concerns, reducing kernel involvement in address space management, and providing a rich model for constructing in-memory persistent data structures that can be easily shared between programs and machines.
- We evaluate (§ 4) the ease-of-use, security advantages, and programmability offered by our environment, for both new and existing, ported software (SQLite), along with performance improvements (§ 5) on NVM DIMMs.

2 The Data-Centric OS

Operating systems provide abstractions for data access that reflect the hardware for which they were designed. Current I/O interfaces and abstractions reflect the structure of mutually exclusive volatile and persistent domains, the hallmarks of which are heavy kernel involvement for persisting data, a need for data serialization, and complexity in data sharing requiring the overhead of pipes or the management cost of shared virtual memory. However, the introduction of low latency and directly attached NVM into the memory hierarchy requires that we rethink key assumptions such as the use of virtual addresses, the kernel's involvement in persistent I/O, and the way that programs operate on and share persistent data [30].

The first key characteristic of NVM is low latency: only 1.5–8× the latency of DRAM in most cases [38], so the cost of a system call to access NVM dominates the latency of the access itself. The second key characteristic is that the processor can directly access persistent storage using load and store instructions. Direct, low latency access to NVM means that explicit serialization is a poor fit—it adds complexity, as programmers must maintain different data formats and the transformations between them, and the overhead is intolerable due to NVM's low latency. Hence, we should design the semantics of the programming model around *in-memory* persistent data structures, giving programs direct access to them without explicit persistence calls or serialization methods.

These characteristics imply two basic requirements for OSes to most effectively use NVM:

1. **Remove the kernel from the persistence path.** This addresses both characteristics. System calls to persist data are costly; we must provide lightweight, direct, memory-style access for programs to operate on persistent data.
2. **Design for pointers that last forever.** Long-lived data structures can directly reference persistent data, so pointers must have the same lifetime as the data they point to. Virtual memory mappings are, by contrast, ephemeral and so cannot effectively name persistent data. Persistent data is, by definition, accessed by multiple actors, both simultaneously and over time, and thus must be stored in a form that is conducive to sharing without needing the ephemeral context associated with a particular actor.

We call an OS that meets both of these requirements *data-centric*, as opposed to current OSes, which are *process-centric*. Operations on persistent, in-memory data structures are the primary functions of a data-centric OS, which tries to avoid interposing on such operations, preferring instead to intervene only when necessary to ensure properties such as security and isolation. To meet both of these requirements a data-centric OS must provide effective abstractions for identifying data independent of data location, constructing persistent data relationships that do not depend on ephemeral context, and facilitating sharing and protection of persistent data.

2.1 Existing Interfaces

Current OS techniques do not meet these requirements—file `read` and `write` interfaces, designed for sequential media and later expanded for block-based media, require significant kernel involvement and serialization, violating both requirements. While support for these interfaces can be useful for legacy applications, as we will demonstrate, providing the programmer with abstractions designed *for* NVM both reduces complexity and improves performance.

The `mmap` call attempts to hide storage behind a memory interface through hidden data copies. But, with NVM, these copies are wasteful, and `mmap` still has significant kernel involvement and the need for explicit `msync` calls. "Direct Access" (DAX) tries to retrofit `mmap` for NVM by removing the redundant copy, but this fails to address requirement two! Operating on persistent data through `mmap` requires the programmer to use either fixed virtual addresses, which presents an infeasible coordination problem as we scale across machines, or virtual addresses directly, which are ephemeral and require the context of the process that created them.

Attempting to shoehorn NVM programming atop POSIX interfaces (including `mmap`) results in complexity that arises from combining multiple partial solutions. Given some feature desired by an application, the NVM framework can provide an integrated solution that meshes well with the existing support for persistent data structure manipulation and access, or it can fall-back to POSIX resulting in the programmer needing to understand two different "feature namespaces" and their interactions. An example of this is naming, where a programmer may need to turn to the filesystem to manage names in a completely orthogonal way to how the NVM frameworks handles data references. We will discuss another example, security, in our case study (§ 4).

Additionally, models that layer NVM programming atop existing interfaces often fail to facilitate effective persistent data sharing and protection. PMDK, an NVM programming library, makes design choices that limit scalability, since its data objects are not self-contained and do not have a large enough ID space, resulting in the need to coordinate object IDs across machines [10]. For the same reason, although single-address space OSes [12] somewhat address our first requirement, they do not consider both requirements at once, nor do they provide an effective and scalable solution to long-term data references due to that same coordination complexity [9].

2.2 A Data-Centric Approach

We cannot store virtual addresses in persistent data, so we need a new way to name a word of persistent memory: a *persistent pointer*. The persistent pointer encodes a persistent identification of data (§ 3.3) instead of an ephemeral address, allowing any thread to access the desired word of memory regardless of address space. This approach dramatically improves programmability, as programmers need not worry about the complexity of referring to persistent data with ephemeral constructs, improving data sharing across programs and runs of a program. Twizzler still makes use of virtual memory *hardware* to provide isolation and translation, but persistent data structures should not be written in terms of virtual addresses.

The Death of the Process. Processes as a first class OS abstraction are, like virtual addresses, unnecessary; a traditional process couples threads of control to a virtual address space, a security role, and kernel state. However, with the kernel removed from persistent data access, much of that kernel state (*e.g.* file descriptors) is unnecessary, leading to a decoupling of mechanisms: nothing fundamentally connects a virtual address space (*how* threads access data) and a security context (*what* data they may access). Instead, a data-centric OS can replace the process abstraction with security contexts, allowing greater flexibility for how security policy is managed.

The process abstraction is just one example. Persistent data access plays a key role in OS abstraction design, and we need to avoid complexity arising from combining old and new interfaces. Hence, we need to consider the wide-reaching effects of changing the persistence model on *all* aspects of the system, not just I/O interfaces. NVM gives us an opportunity to design an OS around the requirements of the target programming model instead of trying to mold support libraries around existing interfaces. While it is important that we provide support for legacy applications, it is these applications that should be relegated to support libraries; new applications built for the programming model should get first-class OS support.

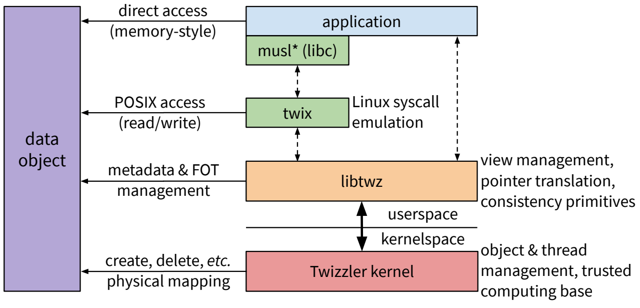
Targeting these Constraints with Twizzler. The consequences of meeting the requirements of these hardware trends define a bounded design space for data-centric OSes. We have chosen a point in that space and built Twizzler, our approach to providing applications with efficient and effective access to NVM. In the following section we will discuss how our four primary abstractions—a low level persistent object model, a persistent pointer design, an address space mechanism called *views*, and a security context mechanism—achieve these goals of removing the kernel from the persistent data access path.

3 The Design of Twizzler

Twizzler is a stand-alone kernel and userspace runtime that provides execution support for programs. It provides, as first-class abstractions, a notion of threads, address spaces, persistent objects, and security contexts. A program typically executes as a number of threads in a single address space (providing backwards compatibility with existing programming models), into which persistent objects are mapped on-demand. Instead of providing a process abstraction, Twizzler provides *views* (§ 3.2) of the object space, which enable a program to map objects for access, and *security contexts* (§ 3.4) which define a thread's access rights to objects in the system. Twizzler provides persistent pointers (§ 3.3) for programs, as well as primitives to ensure crash-consistency (§ 3.5). The thread abstraction is similar to modern OSes; the kernel provides scheduling, synchronization, and management primitives. Figure 1 shows an overview of the system organization and how different parts of the system operate on data objects.

Twizzler's kernel acts much like an Exokernel [28, 41], providing sufficient services for a userspace library OS, called *libtwz*, to provide an execution environment for applications. The primary job of libtwz is to manage mappings of persistent objects into the address space (§ 3.2) and deal with persistent pointers (§ 3.3). Twizzler also exposes a standard library that provides higher level interfaces beyond raw access to memory. For example, software that better fits message-passing semantics can use library routines that implement message-passing atop shared memory. Twizzler's standard library provides additional higher level interfaces, including streams, logging, event notification, and many others. Applications use these to easily build composable tools and pipelines for operating on in-memory data structures without the performance loss and complexity of explicit I/O.

We provide POSIX support with twix, a library that emulates Linux syscalls. We modified musl [1], a C library which all programs link to, replacing invocations of the syscall instruction with calls into twix, which internally tracks UNIX state like file descriptors. This is handled entirely in userspace; calls to read and write often reduce to calls to memcpy.



* modified musl to change linux syscalls into function calls

Figure 1: Twizzler system overview. Applications link to musl (a C library), twix (our Linux syscall emulation library), and libtwz (our standard library).

3.1 Object Management

Twizzler organizes data into *objects*, which may be persistent. Each object is identified by a unique 128 bit object ID (though larger IDs would be possible). Objects provide contiguous regions of memory that organize semantically related data with similar lifetime. Applications access objects via mapping services (discussed in the next section) by mapping each object into a contiguous range in the address space, though the address space itself may be densely or sparsely mapped. Objects can be anywhere from 4 KiB (the size of a page) to 1 GiB; the upper bound on object size is a prototype implementation choice, and not fundamental to the design.

Twizzler uses objects as the unit of access control, building off a read/write/execute permissions model which mirrors that of memory management units in modern processors. This is a direct consequence of avoiding the kernel for persistent data access—it can set policy by programming the MMU, but must leave enforcement up to the hardware which, in-turn, defines what protections are possible.

An object, from the programmer's perspective, is flexible in its contents—for example, it could contain anywhere from a single B-tree node to the entire B-tree. Often, an object would contain the entire tree, since the entire tree is typically subject to the same access semantics by programs, and there are overheads associated with objects that can be amortized over larger spaces. Data and data structures that are too large for one object or require different access permissions can span multiple objects with references between them. We demonstrate the benefits of this flexibility in Section 4.

The kernel provides services for object management, such as creating and deleting objects. Objects are created by the `create` system call, which returns an object ID. A program may also optionally provide an existing object ID to the `create` call, stating that the new object should be a copy of the existing one, for which Twizzler uses copy-on-write. The new ID is a number that is unlikely to collide with existing IDs in the 128 bit ID space, and can be assigned using a technique that supports this requirement (random, hashing, *etc*.). Some forms of ID assignment support a form of access control: a program can only access an object whose ID it knows. Twizzler provides object naming as well, discussed in Section 3.3.

Objects may be be deleted via the `delete` system call. Like Unix's `unlink`, objects are reference counted, where a reference refers to a mapping in an address space. Once the reference count reaches zero, the object may be deleted.

3.2 Address Space Management

Although virtual addresses are the wrong abstraction to use for persistent data access, we do leverage virtual address hardware in modern processors for isolation and protection. Twizzler provides access to persistent objects by mapping them into the virtual address space behind-the-scenes (via `libtwz`). This generates many mapping operations to access persistent data, so requiring system calls would be costly. Additionally, our kernel avoidance necessitates an increased address space management responsibility for userspace. For example, executable loading and mapping is handled largely without the kernel.

To support userspace manipulation of address spaces, the kernel and userspace share an object (called a "view") that defines an address space layout. The view is just a normal object, and so standard access control mechanisms apply to enforce isolation. When applications map objects into their address space, they update the view to specify that a particular object should be addressable at a specific location. The kernel then reads the object and determines the requested layout of the virtual address space. The view object is laid out like a page-table, where each entry in the table corresponds to a slot in the virtual address space. Each table entry contains an object ID and read, write, and execute protection bits to further protect object access (like `PROT_*` in `mmap`).

When a page-fault occurs, the fault handler tries to handle the fault by either doing copy-on-write, checking permissions, or by trying to map an object into a slot if the view object requested one. If it cannot handle the fault (due to a protection error or an empty entry in the view object), it elevates the fault to userspace where `libtwz` handles it, possibly by killing the thread, or possibly by mapping an object if the slot is "on-demand". When the kernel maps an object into a slot, it updates the address space's page-tables appropriately.

When threads add entries to a view object they need not inform the kernel—when a fault occurs, the kernel will read the entry as needed. However, when *changing* or *deleting* an entry, threads must inform the kernel so it can update existing page table entries. We provide two system calls for views. The `set_view` call allows a thread to change to a new view, which might be used to execute a new program or jump across programs to, for example, accomplish a protected task. Twizzler's access control system prevents this from happening arbitrarily. The second system call is `invalidate_view`, which lets a thread inform the kernel of changed or deleted entries.

3.3 Persistent Pointers

Section 2 discussed the needs for references that outlive ephemeral actors. Twizzler provides *cross-object* persistent pointers so that a pointer refers not to a virtual address but to an offset within an object by encoding an `object-id:offset` tuple. This enables a pointer to refer to persistent data, but it also allows objects to have *external* pointers that refer to data in any object in the global object space. We highlight cross-object pointers' power and flexibility by demonstrating their ability to express inter-object relationships in Section 4.

To efficiently encode this tuple, we use indirection through a per-object *foreign object table* (FOT), located at a known offset within each object. The FOT is an array of entries that each stores an object ID (or a name that resolves into an object

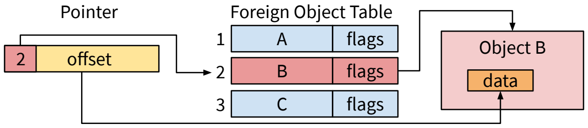


Figure 2: Pointer translation via the FOT. The pointer and the FOT are both contained in the same object (not shown).

ID, as we will see below) and flags. A cross-object pointer is stored as a 64 bit `FOT_idx:offset` value, where the `FOT_idx` is an index into the FOT. This provides us with both large offsets *and* large object IDs, since the IDs are not stored within the pointer itself. If an object wishes to point to data within itself (an *intra-object* pointer), it stores 0 in `FOT_idx`. When dereferencing, Twizzler uses the `FOT_idx` part of the pointer as an index into the FOT, retrieving an object ID. The combination of a FOT and a cross-object pointer logically forms an `object-id:offset` pair, as shown in Figure 2.

Our design (discussed in prior work [9, 10]) differs from existing frameworks [6, 13, 18, 19, 57, 58] because of the indirection. Frameworks like PMDK store entire object IDs within pointers, increasing pointer size and reducing flexibility by removing the possibility of late-binding (discussed below). Additionally, Twizzler extends the namespace of data objects beyond one machine, as machine-independent data references are a natural consequence of cross-object pointers. Existing solutions are limited in this scalability. They either limit the ID space (necessary for storing IDs in pointers) and thus resort to complex coordination or serialization when sharing, or they require additional state (*e.g.* per-process or per-machine ID tables) that must be shared along with the data, forcing the receiving machine to "fix-up" references. Worse still, the fix-up is application-specific, since the object IDs are within any pointer, not in a generically known location. Our per-object FOT results in self-contained objects that are easier to share, thus interacting better with remote shared memory systems.

Part of our motivation for this FOT indirection was to allow a large ID space without increasing pointer size. PMDK, by contrast, increases pointer size to 128 bits for each pointer. Twizzler has no additional space overhead per-pointer, instead adding a 32-byte overhead per FOT entry. The number of FOT entries, however, is typically much smaller than the number of pointers since pointers to the same external object can all use the same FOT entry. As we will see in Section 5, this has a dramatic benefit to performance.

FOT Entries and Late-Binding. The FOT entry's `flags` field has bits for read, write, and execute protections. The protections are *requests*; Twizzler implements separate access control on objects. This allows some pointers to refer to data with a read-only reference while others can be used for writing, reducing stray writes (a single ID can repeat in the FOT with different protections). The FOT entries also enable atomic updates that apply to all pointers using that FOT entry.

Instead of *requiring* programmers to refer to objects via IDs only, we allow names in FOT entries. These entries may contain a pointer to an in-object string table that contains a name. Names enable late-binding [19], a vital aspect of systems, allowing references to objects which change over time, *e.g.* shared library versions. Names are passed to a *resolving* function (specified in the FOT entry). Allowing a program to specify how its names are resolved increases the flexibility of the system beyond supporting UNIX paths. Twizzler provides a default name resolver that uses UNIX-like paths.

The implementation of naming is orthogonal to Twizzler's design. We allow a range of name resolution methods within the system stack and allow objects to specify their own name resolution functions for flexibility. For example, objects could be organized by both a relational database and a hierarchical namer similar to conventional file systems. Non-hierarchical file systems are well studied [3,31,32,54,55], but these systems do not easily cooperate atop a single data space. Since Twizzler uses a flat namespace as its "native" object naming scheme, it enables the required cooperation.

Pointer Translation. Current processors provide only a virtual memory abstraction, so applications must do some extra work to dereference a pointer, *translating* a pointer from its persistent form into a virtual address. This does not affect the *stored* pointer, which is still persistent and independent of any translation or address space. Thus multiple applications, possibly with different address space layouts, can translate the same pointer at the same time without coordination.

Pointer translation occurs with the help of two `libtwz` functions: `ptr_lea` (load effective address) and `ptr_store`. When a program dereferences a pointer, it first calls `ptr_lea`. The pointer is resolved into an object-ID and offset pair through a lookup in the FOT, after which `libtwz` determines if the referenced object is already mapped (by maintaining per-view metadata). If not, it picks an empty slot in the view and maps the object there (a cheap operation that does not invoke the kernel). Once mapped, `libtwz` combines the object's temporary virtual base address with the offset, and returns the new pointer. The `ptr_store` function does the opposite of `ptr_lea`—it turns a virtual pointer into a persistent one. While these are done manually in our implementation, we plan to implement compiler support to emit these calls automatically.

FOT management is handled by `libtwz`. While a lookup in the FOT is a simple array-indexing operation, a store may require adding to the FOT. To avoid duplicate entries, `libtwz` walks the FOT looking for a compatible entry. If one is not found, it atomically reserves a new entry and fills it (flushing cache-lines to persist it) before storing the pointer. The `ptr_store` operation is less common than `ptr_load`, and in the future we may include additional caching metadata that would speed-up the FOT walk (such as storing recent IDs).

Translating pointers has a small overhead (§ 5) and the result can be cached. Twizzler improves performance via a per-object cache of prior translations. The common case, intra-object pointers, does not require an external lookup and is implemented as a simple bitwise-or operation.

3.4 Security and Access Control

Twizzler's focus on memory-based objects requires that we design the security model around hardware-based enforcement, where the MMU checks each access. This design is *inevitable* in a data-centric OS, since the kernel is not involved in every memory access. The kernel merely specifies the access rights when mapping an object and then relies on the hardware to enforce those rights with a low overhead.

A key design choice we make is *late-binding on security*. Applications request access to an object with permissions that they desire; if they access the object in only allowed ways (*e.g.*, only reading a read-only object), no fault occurs. This is because when we map an object (via a view), the kernel is not immediately involved, and so cannot check access rights for a particular access at the time the mapping is setup. Performing an access rights check on time of first access does not make sense either, as it associates a specific access (that might be allowed) with a permissions error. For example, if a program reads object A, and that program is allowed to read A, it should be allowed to perform the read even if it requested read-write access to the object. This late-binding enables simpler programs that need not worry about elevating access rights through remapping data objects. Programs can make progress without knowing in advance the permissions of the objects they might access, thus enabling the reuse of the OS's access control mechanism in applications. We will show the flexibility of this in Section 4, wherein we add access control to a program by changing only a few lines of code.

Threads run in a security context [8,25,44], which contains a list of access rights for objects and allows the kernel to determine the access rights of programs. Using these contexts, Twizzler is able to provide analogues to groups and owners in Unix while providing more fine-grained access control if necessary. Unlike past exploration into security contexts, data-centric OSes offer an advantage in simplicity. A security context abstraction in a Unix-like OS needs to maintain access rights to a set of fundamentally different things (such as paths, virtual memory locations, and system calls). Instead, Twizzler's security contexts specify access rights to an object via IDs instead of virtual addresses. This also makes security contexts persistent, allowing us to use them as the primary way we assign security roles to threads.

Security contexts are implemented via virtualization hardware that maps virtual memory to an intermediate "object space" which specifies the access rights, which is then mapped to physical memory [9]. This reduces the number of page-table structures and mappings, as threads in the same security context can share the same page-tables for each object.

3.5 Crash Consistency

Twizzler provides primitives for building crash-consistent data structures. At a low level, it provides a mechanism for writing back cache-lines and appropriate fences. Applications use these primitives today outside of Twizzler to build up larger, more complex support for crash-consistent data structures.

Our goal is to provide low level primitives without restricting programs or prematurely prescribing particular solutions. There is a wealth of research on crash-consistent data structures for NVM [15,16,24,46,50–53,65], but it is still in flux. Of course, Twizzler manages *system* data structures, such as FOT entries, views, *etc.*, in a crash-consistent manner using the aforementioned primitives, locking, and fencing.

Twizzler also provides a transactional-persistent logging mechanism. Programmers can write `TXSTART`–`TXEND` blocks to denote transactions and `TXRECORD` statements to record pre-changed values. This is similar to the mechanism provided by PMDK [58]. If applications need more complex transactions using different logging mechanisms, they can use libraries.

Twizzler provides a mechanism for restarting threads when power is restored following a crash. Since views are persistent objects, all mapped objects during a thread's execution are known across power cycles, and are mapped back in. The thread is then started at a special `_resume` entry point, allowing the program to handle the power failure in an application-specific manner with access to the state of the program (data segment, heap, *etc.*) as it was when power was cut.

3.6 Implementation

Twizzler's kernel is similar to many microkernels, providing a small set of key primitives. It is 5,500 lines of architecture-independent code and 5,700 lines of architecture-dependent CPU driver code. The primary complexity in the system is implemented in userspace, as the design of the programming model greatly simplifies the kernel. Twizzler is open-source; more information can be found at https://twizzler.io.

We also built a prototype of Twizzler by modifying the FreeBSD 11.0 kernel before implementing our standalone kernel. This was done both to more rapidly verify our design and to provide a prototyping environment for developers to write code for Twizzler in a familiar environment. We added Twizzler services to FreeBSD by adding system calls, modifying the fault-handling logic, and distinguishing Twizzler threads from FreeBSD threads. This is also a testament to the simplicity of the kernel in our model, since FreeBSD was relatively easy to modify to support the Twizzler userspace. However, the FreeBSD prototype is limited by its need to coordinate with FreeBSD's Unix services, thus the standalone kernel is more efficient and simpler, and provides a better environment for researching kernel design changes in the face of NVM.

4 Evaluation

Our primary goals for evaluating Twizzler were:

1. Show that Twizzler meets the needs of a data-centric OS in enabling programs to directly access persistent data.

2. Demonstrate that the programming model we defined provides sufficient power to easily and effectively build real applications with NVM in mind.
3. Measure the performance of our system to understand where we gain and lose performance.

We approached these goals two ways: porting existing software (SQLite) and writing new software for Twizzler. The first demonstrates both the generality of the programming environment (legacy software can be easily ported) and the potential performance gains to be had even for legacy software. The second demonstrates the true power of Twizzler's programming model and allows us to explore the consequences of our design choices fully without being constrained by legacy designs.

We built three pieces of new software: a hash-table based key-value store (KVS), a red-black tree data structure, and a logging daemon. Each had different characteristics and goals, and together they demonstrate the flexibility that Twizzler offers in allowing simple implementation, nearly-free access control, and the ability to directly express complex relationships between objects. Using our KVS and red-black tree code, we ported SQLite (a widely used SQL implementation) to Twizzler along with a YCSB [17,29] driver (a common benchmark), allowing us to explore Twizzler's model in a larger, existing program that would let us study the performance of Twizzler in a complex system that stores *and processes* data. We present the performance of SQLite and our new software, along with microbenchmarks, in Section 5.

4.1 Case Study: Key-Value Store

We implemented a multi-threaded hash-table based key-value store (KVS), called `twzkv`, to study cross-object pointers and our late-binding of access control. Our KVS supports insert, lookup, and delete of values by key (both of arbitrary size), and hands out direct pointers to persistent data during lookup. During insert, it copies data into a data region before indexing the inserted key and value. We built `twzkv` in multiple phases to study how our system handles changing requirements.

We built `twzkv` in roughly 250 lines of C. Handing out direct pointers into data was trivial to implement with cross-object pointers, requiring only a call to `ptr_lea` during lookup. The initial implementation maintains two objects, one for data and one for the index. The complexity typically involved when storing both index and data in a single, flat file is not justified in a programming model where we can express inter-object relationships directly at near-zero cost in complexity or performance. In our case, a pointer from the index object to the data object (such as an entry in the hash table) can be written with a single call to `ptr_store`. This, combined with the simple requirements for an in-memory NVM KVS, resulted in a small implementation that was nonetheless a usable KVS.

Extending Requirements. Next, we added functionality to protect values with access control. We wanted to keep handing out direct pointers to data during lookup and to keep `twzkv` a library (as opposed to a service). Meeting these goals on an existing system would be difficult without adding significant complexity, such as reimplementing a lot of Twizzler's pointer framework or implementing manual, redundant access control.

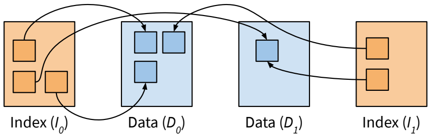


Figure 3: Cross-object pointers in `twzkv`.

In Twizzler, implementing access control in `twzkv` involved having the index refer to data in multiple data objects, assigning those objects different access rights, and allocating from those objects depending on desired access rights. We were able to implement this while preserving the original code due to the transparent nature of Twizzler's cross-object pointers. Now, when inserting, the application indicates the data object into which to copy the data, as shown in Figure 3.

By supporting multiple data objects, `twzkv` can leverage the OS's access control, sidestepping complexity. Unrestricted data can go in D_0 (Figure 3), whereas restricted data can go in D_1. Since each object has distinct access control, a user can set the objects' access rights, then decide where to insert data according to policy. The indexes point to the correct locations regardless of the access restrictions of the data objects, and `twzkv` still hands out direct pointers, but a user that is restricted from accessing data in D_1 will not be able to dereference the pointer. A further extension is to support secondary indices, as shown in Figure 3, enabling alternative lookup methods and limiting data discovery with index object access control. This extension is easy to implement on Twizzler.

Comparison to UNIX Implementation. To compare with existing techniques, we built a similar KVS using only UNIX features (called `unixkv`). It also separates index and data, but it must manually compute and construct pointers. Supporting multiple data objects was complex in `unixkv`, because we had to store and process file paths in the index and store references to paths for pointers, increasing overhead and code complexity by 36%—a lot for an implementation with relatively few pointers—just to reimplement Twizzler's support. The extra complexity also included code to manually open, map, and grow files, much of which Twizzler handles internally. Development time was extended by bugs that were not present when developing `twzkv`, due to the manual pointer processing. While `twzkv` gains transparent access control, `unixkv` does not due to the lack of on-demand object mapping and late-binding of security. Instead, `unixkv` needs to know object permissions before mapping, a restriction that limits the ability to reuse OS access control, something that `twzkv` could leverage through late-binding on security (§ 3.4)[1]. Other frameworks like PMDK that do not integrate access control and late-binding into their models have similar limitations.

[1] `unixkv` could trap segmentation faults to do this, but that would be application-specific, difficult, and would reimplement Twizzler functionality.

4.2 Case Study: Red-Black Tree

To evaluate the process of writing persistent, "pointer-heavy" data structures, we implemented a red-black tree in C using normal pointers (`ramrbt`) in 100 lines of code, and evolved it for persistent memory in two ways: manually writing base+offset style pointers, as current systems require (`unixrbt`), and using Twizzler (`twzrbt`). Porting existing data structure code to persistent memory will be common during the adoption of NVM, and much of the complexity therein comes from dealing with persisting virtual addresses [47].

In developing `unixrbt`, we found 83 locations where we had to perform pointer arithmetic for converting between persistent and virtual addresses. Consider an expression such as `root->left->right = foo`. Inserting calls to translate this directly results in `L(L(root)->left)->right = C(foo)`, where `L` converts to a virtual address and `C` converts back, which is heavily obfuscated and took more development time than writing `ramrbt` in the first place due to debugging.

We built `twzrbt` like `unixrbt`, annotating pointer stores and dereferences. However, `unixrbt` used an application-specific solution for pointer management; if other applications wanted to use the data structures created by `unixrbt`, they would have to know the implementation details of the pointer system (or share the implementation, thus reimplementing much of Twizzler's library). Additionally, due to Twizzler enabling improved system-wide support for cross-object pointers, these transformations can be made automatic.

Unlike `twzrbt`, `unixrbt`'s tree is limited to a single persistent object; a limitation that prevents the tree from growing arbitrarily, does not allow it to directly encode references to data outside the tree object, and does not gain it the benefits of cross-object data references that were discussed above for `twzkv`. Adding support for this to `unixrbt` would require modifying the core data structures to include paths and significantly altering the code, increasing its length by at least a factor of 2, whereas `twzrbt` gets this functionality for free.

Another advantage of `twzrbt` is reduced support code compared to `unixrbt`; `unixrbt` needed code to manage and grow files and mappings, while we implemented `twzrbt` as simple data structure code with Twizzler managing that complexity. The additional error handling code and pointer validity checks in `unixrbt` (handled automatically in Twizzler) increased development time and implementation complexity.

4.3 Porting SQLite

We ported SQLite to Twizzler to demonstrate our support for existing software and to evaluate the performance of a SQLite backend designed for Twizzler. We used our POSIX support framework, a combination of `musl` and our library `twix`, to support much of SQLite's POSIX use. We took a modified version of SQLite called SQLightning that replaced SQLite's storage backend with a memory-mapped KVS called LMDB [14]. We chose this port because LMDB is implemented with `mmap`'d files as the primary access method and hands out direct pointers to data as one would expect from an effectively designed NVM KVS[2]. Since LMDB's SQLightning port already replaces the storage backend with calls to LMDB, we ported SQLite to Twizzler by taking our KVS and red-black tree code and implementing enough of the LMDB interface for SQLite to run using Twizzler as a backend. Outside of the B-tree source file few changes were needed for SQLite to run on Twizzler. We further ported our modified SQLite backend to PMDK to compare directly with a commonly used NVM programming library that supports persistent pointers.

We also ported a C++ YCSB driver [29], which required porting the C++ standard template library (STL). Since we had already ported a standard C library, the C++ STL was easily ported, demonstrating the ease of porting software to Twizzler. We have also ported some existing Unix utilities (such as `bash` and `busybox`), which largely require only recompiling to run on Twizzler. Of course, to gain *all* of the benefits of Twizzler, programs will be need to be written with NVM in mind (but this is true regardless of the target OS).

Our implementation of the LMDB interface corroborated our experience from the KVS case study: much of the complexity in storage interfaces and implementations comes from the separation between storage and memory. This has been studied before (as we will elucidate in Section 6), but the advent of NVM changes the game significantly by allowing programmers to think directly via in-memory data structures. The result is that interfaces like cursors in a KVS become redundant. We implemented to this interface for LMDB, but the functions were largely wrappers around storing a pointer to a B-tree node and traversing the tree directly without separate loads and copies. The result was an extremely simple implementation (500 LoC) that still met the required interface. Future software for NVM can use Twizzler's programming model to more effectively write software that eschews the need for complexity forced by the two-tier storage hierarchy.

4.4 Discussion

Although these implementations were simple, they represent the applications and data structures we expect in a data-centric system. Pointers we can directly use in our programming languages make computing over persistent data almost transparent, allowing simple implementations that are nevertheless easy to evolve as requirements change.

Not only does `twzkv` have access control, but it enables concurrent access via cross-object pointers. Applications can load indexes for multiple databases without needing to worry about address space layout and without writing complex pointer management code that would be required by an implementation using `mmap`. We were able to provide access control without a single line of code in `twzkv` dedicated to checking

[2] These are not persistent pointers, however, unlike Twizzler's.

or enforcing access rights. Instead, we relied on the system's access control, something not possible with other frameworks that do not support late-binding of access rights and do not consider security as part of their programming model. Twizzler thus removes the need for applications to manage their own access control, which increases the security of the system by divesting programmers from the responsibility of getting it right. Similar functionality for current systems would traditionally require separation of the library and application into a client-server model, but that additional overhead is unneeded here and inappropriate on a persistent memory system.

Although `twzrbt` and `twzkv` had different densities of pointer operations, `twzrbt` being "pointer-heavy" and `twzkv` being "pointer-light", Twizzler improved the complexity of both over manual implementation and improved flexibility over existing persistent pointer methods. Using a system-wide standardized approach to pointer translations not only enables better compiler and hardware support, but it also improves interoperability; because they share a common framework, `twzkv` could use the red-black tree code and data with ease, and even interact with the SQLite database even though they were written separately without that goal in mind. The position-independence afforded by this model enables both composability and concurrency, while also simplifying programming on persistent data to a natural expression of data structures.

Non-Shared-Memory Programs. To push the limits of our model and show that Twizzler does not constrain programmers into a shared-memory model, we implemented a logging framework (similar to `syslogd`). The logging daemon, `logboi`, can receive log messages either synchronously or asynchronously. In both cases, the interface is the same, but synchronous logging uses shared-memory abstractions while asynchronous logging relies on message-passing semantics.

For synchronous logging the thread switches security contexts, which is made possible by decoupling address spaces and security. The call to the logging framework then updates the log and returns. An asynchronous logging event sends data to the logging thread via a stream object (a standard API provided by Twizzler) that `logboi` and the application share. The choice of asynchronous or synchronous is left to the programmer; synchronous can have lower latency and predictable behavior while asynchronous offloads processing to `logboi`.

5 Performance

Our evaluation's primary focus is on the benefits of the programming model, showing new functionality with reduced complexity at an acceptable overhead. Nevertheless, there are many cases where we see significant improvement (such as SQLite) because the programming model has less overhead, and our pointer design is space efficient and fast to translate.

We measured the performance of our KVS and red-black tree, performed microbenchmarks, and evaluated the Twizzler port of SQLite against Linux (Ubuntu 19.10) instances of SQLite, SQLightning, and our port of SQLite to PMDK. Tests ran on an Intel Xeon Gold 5218 CPU running at 2.30 GHz with 192 GB of DRAM and 128 GB of Intel Persistent DIMMs. We compiled all tests against the `musl` C library instead of `glibc` because Twizzler uses `musl` to support UNIX programs.

Table 1: Latency of common Twizzler operations.

Pointer Resolution Action	Average Latency (ns)
Uncached FOT translation	27.9 ± 0.1
Cached FOT translation	3.2 ± 0.1
Intra-object translation	0.4 ± 0.1
Mapping object overhead	49.4 ± 0.2

All Linux tests used the NOVA filesystem [69] (a filesystem optimized for NVM) on the NVDIMMs, mounted in DAX mode. This enabled direct access to the persistent memory without a page-cache interposing on accesses.

5.1 Microbenchmarks

Table 1 shows common Twizzler functions' latencies, including pointer translation. The overhead shown for resolving pointers does not include dereferencing the final result, since that is required regardless of how a pointer is resolved. The first row shows the latency for resolving pointers to objects the first time. Twizzler makes a further optimization by caching the results of translations for a given FOT entry. Each successive time that FOT entry is used to resolve a pointer, the result of the original translation is returned immediately, improving the latency as shown on the "cached" row of Table 1. Note that the low latency of these results is expected; the performance critical case of these functions' use is repeated calls, and since these operations are simple, they fit within the processor cache.

Twizzler translates intra-object pointers by first checking if the pointer is internal and, if so, adding the object's base address to it—the same operation required for application-specific persistent pointers. The expanded programming model offered by Twizzler makes this overhead minor relative to the high costs for persistent data access on current systems, which have high-latency for equivalent operations.

We compared our pointer translation to UNIX functions. Resolving an external pointer with an ID corresponds roughly to a call to `open("id")`, which has a latency of 1036 ± 15 ns. The comparison is not exact, of course; the pointer resolution also maps objects, and the call to `open` must handle file system semantics. However, the direct-access nature of NVM results in pointer translation achieving the same goal as opening a file does today. The pointer operations in Twizzler accomplish much of the same functionality as the heavier-weight I/O system calls on UNIX with more utility and less overhead.

A more direct comparison is object mapping, which has low latency compared to `mmap` (658.7 ± 12.7 ns—a 13.3× speedup) though the two have similar functionality. Since map-

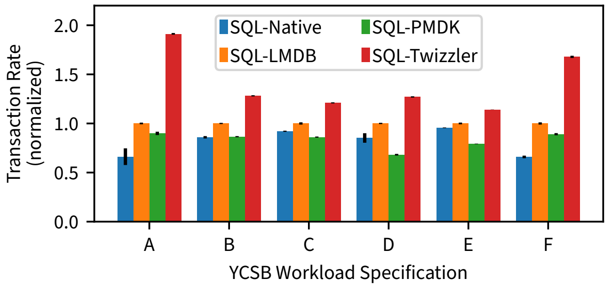


Figure 4: YCSB throughput, normalized (higher is better).

ping occurs entirely in userspace, cache pollution is reduced. While both mmap and Twizzler's mapping require page-faults to occur before the data is actually mapped, this overhead is similar in Twizzler and UNIX, and so is not shown.

5.2 SQLite

We ran four variants of SQLite, three on Linux and one on Twizzler, and compared their performance: "SQL-Native" (unmodified SQLite), "SQL-LMDB" (SQLite using LMDB as the storage backend), "SQL-PMDK" (SQLite using our red-black tree on PMDK), and "SQL-Twizzler" (our port of SQLite running on Twizzler). SQL-Native was run in mmap mode so that both it and SQL-LMDB used mmap to access data. We ran each on the same hardware and normalized the results.

Figure 4 shows the three variants' throughput under standard YCSB workloads. The performance improvement of the LMDB and Twizzler variants over SQL-Native is likely due to handing SQLite direct pointers to data. However, in the Twizzler case we get an additional benefit of operating on data structures directly while LMDB has an abstraction cost.

Figure 5 shows the latency of queries on a one million row table. This is common data processing—loading and then examining data in a variety of ways. We measured the performance of calculating the mean and median, sorting rows, finding a specific row, building an index, and probing the index. SQL-Twizzler had similar performance to SQL-LMDB and SQL-Native despite comparing its extremely simple storage backend to optimized B-tree backends (that benefit from scan operations). As a more direct comparison, SQL-Twizzler significantly out-performed SQL-PMDK in most tests. PMDK's pointer operations are more expensive than Twizzler's, requiring up to two hash table lookups per translation [5]. Additionally, PMDK's pointers are 128 bits, while Twizzler does not increase pointer size. Increased pointer size results in significantly worse cache performance, especially in a pointer-heavy data structure like a persistent red-black tree.

5.3 Key Value Store

We compared twzkv to unixkv by inserting one million distinct key-value pairs, followed by looking up each in-order.

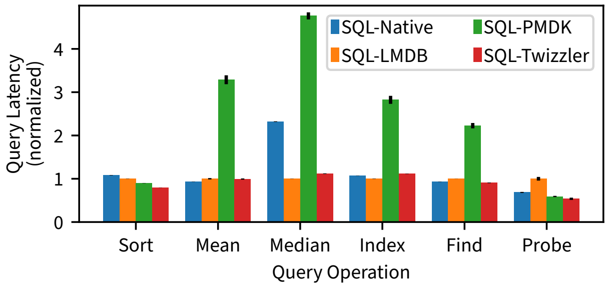


Figure 5: Query latency, normalized (lower is better).

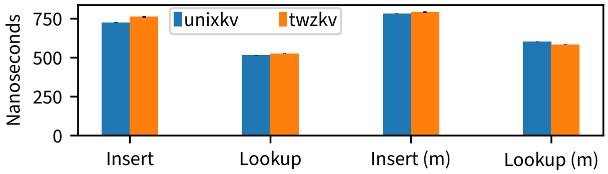


Figure 6: Latency of insert and lookup in twzkv and unixkv. An "(m)" indicates support for multiple data objects.

The inserted items were 32-bit keys and 32-bit values, chosen to reduce the overhead of data copying since we were focusing on pointer translation overhead. Both were compared under two modes, single-data-object and multiple-data-objects. Both KVSes translated between virtual and persistent addresses when storing and retrieving data, but for multiple-data-objects, we allow for storing the data in an arbitrary object.

Figure 6 shows the latency of lookup and insert, demonstrating that not only is the memory-based index and data object structure that can hand out direct data pointers sufficiently low latency to take advantage of NVM, but the additional overhead of cross-object pointers is minimal. Compared to unixkv, twzkv has minimal overhead in the single-object case, and improves lookup performance in the multiple-object case. The minor overhead in other cases comes with improved flexibility, simplicity, and access control support (unixkv does not support access control). Finally, multithreaded access on twzkv and unixkv did not improve performance; despite the pointer translations, they ran at memory bandwidth (for NVM).

5.4 Red-Black Tree

We measured the latency of insert and lookup of 1 million 32-bit integers on both unixrbt and twzrbt. The insert and lookup latency of twzrbt was 528 ± 3 ns and 251.8 ± 0.5 ns, while insert and lookup latency of unixrbt was 515 ± 2 ns and 213 ± 1 ns. The modest overhead comes with significantly improved flexibility, as unixrbt does not support cross-object trees, and less support code (unixrbt manually implements mapping and pointer translations). Note that even though there is lookup overhead in twzrbt, this overhead did not predict the results of a larger program—the SQL-Twizzler port used this red-black tree, and saw performance benefits over block-based implementations.

6 Related Work

Twizzler's design is shaped by fundamental OS research [12, 18, 26–28, 41, 42], which, while approaching similar topics described in Section 2, often did not consider *both* design requirements simultaneously, resulting in an incomplete picture for NVM. Recent research on building NVM data structures [15, 16, 22, 37, 45, 65], often focuses on building data structures that provide failure atomicity and consistency. In contrast, we explore how NVM affects programming models. We draw from recent work on providing OS support for NVM systems [11] and work providing recommendations for NVM systems [48], integrating object-oriented techniques and simplified kernel design to provide high-performance OS support for applications running on a single-level store [4, 61].

Multics was one of the first systems to use segments to partition memory and support relocation [6, 19]. It used segments to support location independence, but still stored them in a file system, requiring manual linkage rather than the automated linkage in Twizzler. Nonetheless, Multics demonstrated that the use of segmenting for memory management can be a viable approach, though its symbolic addresses were slow.

The core of Twizzler's object space design uses concepts from Opal [12], which used a single virtual address space for all processes on a system, making it easier to share data between programs. However, Opal was a single-address space OS, which is insufficient for NVM [9, 10], and it did not address issues of file storage and name resolution. It also required a file system, since there was no way to have a pointer refer to an object with changing identity, whereas our approach removes the need for an explicit file system. Other single-address space OSes, such as Mungi [34], Nemesis [56], and Sombrero [63], show that single address spaces have merit, but, like Opal, did not consider how the use of NVM would alter their design choices; in particular, how the use of fixed addresses results in a great deal of coordination that is unnecessary in our approach. OSes such as HYDRA [68] provide functionality similar to cross-object pointers; however, in Twizzler, we extend their use from procedures-referencing-data to a more general approach. Furthermore, they required heavy kernel involvement, an approach incompatible with our design goals.

Single-level stores [21, 60, 62] remove the memory versus persistent storage distinction, using a single model for data at all levels. While well-known, "little has appeared about them in the public literature" [60], even since the EROS paper. Our work is partially inspired by Grasshopper [21], AS/400, and orthogonal persistence systems, but while these are designed to provide an illusion of persistent memory, Twizzler is built for real NVM and focuses on providing a truly global object space with global references without cross-machine coordination. Clouds [20] implemented a distributed object store in which objects contained code, persistent data, and both volatile and persistent heaps. Our approach uses lighter-weight objects, allowing direct access to objects from outside, unlike Clouds. Software persistent memory [33], designed to operate within the constraints of existing systems, built a persistent pointer system using explicit serialization without cross-object references, in contrast to Twizzler. Meza [49] suggested hardware manage a hybrid persistent-volatile store with fine-grained movement to and from persistent storage. Since persistence in Twizzler is to NVM, we need not interpose on movement between storage and memory, instead simply managing memory mappings of persistent objects, reducing OS overhead.

Recently, several projects have considered the impact of non-volatile memories on OS structure. Bailey, *et al.* [4] suggest a single-level store design. Faraboschi, *et al.* [30] discuss challenges and inevitable system organization arising from large NVM, and we follow many of their recommendations. The Moneta project [11] noted that removing the heavyweight OS stack dramatically improved performance. While Moneta focused on I/O performance, not on rethinking the system stack, we leverage their approach to reduce OS overhead as much as possible, even when the OS must intervene. Lee and Won [43] considered the impact of NVM on system initialization by addressing the issue of system boot as a way to restore the system to a known state; we may need to include similar techniques to address the problem of system corruption.

IBM's K42 [42] inspired the high level design of Twizzler. The object-oriented approach to designing a micro or exokernel used in K42 is an efficient design for implementing modular OS components. Like K42, Twizzler lazily maps in only the resources that an application *needs* to execute. Similar techniques for faulting-in objects at run-time have been studied [36]. Communication between objects in Twizzler is, in part, implemented as protected calls, similar to K42.

Emerald [39, 40] and Mesos [35] implemented networked object mobility, which we can also support. Emerald implemented a kernel, language, and compiler to allow objects mobility using wrapper data structures to track metadata and presenting objects in an object-oriented language, impacting performance via added indirection for even simple operations.

The Twizzler object model was shaped by NV-heaps [15], which provides memory-safe persistent objects suitable for NVM and describes safety pitfalls in providing direct access to NVM. While they have language primitives to enable persistent structures, Twizzler provides a lower-level and uninhibited view of objects like Mnemosyne [65], allowing more powerful programs to be built. Languages and libraries may impose further restrictions on NVM use, but Twizzler itself does not. Furthermore, Twizzler's cross-object pointers allow external data references by code, whereas NV-heap's and DSPM's [59] pointers are only internal. Existing work beyond Multics on external references shows and recommends hardware support [58, 66], but provides a static or per-process view of objects, unlike Twizzler, limiting scalability and flexibility.

Projects such as PMFS [24] and NOVA [69] provide a file system for NVM. Twizzler, in contrast, provides direct NVM access atop of a key-value interface of objects. Although Twizzler does not supply a file system, one can be built atop it. While NOVA and PMFS provide direct access to NVM, NOVA adds indirection with copies. Both use mmap (which falls short as discussed above) and, unlike Twizzler, require significant kernel interaction when using persistent memory.

Our kernel that "gets out of the way" is influenced by systems such as Exokernel [28] and SPIN [7], both of which drew on Mach [2]. In Exokernel, much of the OS is implemented in userspace, with the kernel providing only resource protection. Our approach is similar in some respects, but goes further in providing a single unified namespace for all objects, making it simpler to develop programs that can leverage NVM to make their state persistent. In contrast, SPIN used type-safe languages to provide protection and extensibility; our approach cannot rely upon language-provided type safety since we want to provide a general purpose platform.

7 Future Work

Compiler and Hardware Support. Clean-slate NVM abstraction reopens the possibility of coevolving OSes, compilers and languages, and hardware. Standardized OS support for cross-object pointers enables compiler support more effectively than application-specific solutions [47] or simple libraries [58]. Twizzler's pointer translation functions are simple enough to be automatically emitted by a compiler. Similarly, designing an OS for cross-object pointers allows us to better state our needs to hardware, which can alleviate performance overheads for pointer translation [66, 67].

Security. Although we discussed the Twizzler security model briefly, there is still much to do. The current model provides access control, a basic ability to define and assign roles based on security contexts, and simple sub-process fault isolation through the ability to switch security contexts. We are exploring a *flexible* security model that allows programmers to easily trade-off between security, transparency, and performance using capability-based verification. For example, we are implementing a call-gating mechanism that will allow us to restrict control-flow transfers between application components, improving the security against malicious components and reducing the possibility of memory-corrupting bugs.

Networking and Distributed Twizzler. One of the key principles of Twizzler is to focus the programming model on data and away from ephemeral actors such as processes and nodes. This is enabled by our identity-based references that decouple location from references, and by ensuring all the context necessary to understand these relationships is stored with the data. Because our data relationships are independent of the context of a particular machine, applications can more easily share data. This easy sharing, combined with a large ID address space, motivates a *truly* global object ID space.

We are building a networking stack and support for a distributed object space into Twizzler. Our networking stack is based around extensive use of hardware virtualization in modern NICs. This design, which is in use in existing kernel-bypass strategies, will mesh well with our core OS design of reducing kernel interposition. At a higher level, we are considering how distributed applications change in our model. For example, an increase in data mobility facilitated by our location-independent data references and identities means that we can manifest both data and code where they are needed without complex marshalling, turning distributed computation into a rendezvous problem. We plan to build distributed applications atop Twizzler to demonstrate this approach.

Of course, for compatibility we will provide a traditional sockets-based networking stack. However, we can use existing userspace libraries that, *e.g.*, implement TCP in userspace. Because we implemented our POSIX compatibility library in userspace, applications can gain many benefits afforded by kernel-bypass networking frameworks while still using traditional socket interfaces.

8 Conclusion

Operating systems must evolve to support future trends in memory hierarchy organization. Failing to evolve will relegate new technology to outdated access models, preventing it from reaching full potential, and making it difficult for OSes to evolve in the future. Twizzler shows a way forward: an OS designed around NVM that provides new, efficient, and easy to use semantics for direct access to memory. Cross-object pointers in Twizzler allow programmers to easily build composable and extensible applications with low overhead by removing the kernel from persistent data access paths, thereby improving the flexibility and performance. Our simpler programming model improved performance despite the (small) pointer translation overhead. Even a memory hierarchy with large RAM but without persistent memory benefits from our design by enabling programs to operate on large, shared, in-memory data with ease. Our programming model is easy to work with compared to existing systems, and we were able to both quickly prototype real applications with advanced access control features and port existing software (SQLite). Twizzler will give us a system from which we can build a full NVM-based OS around a data-centric design and explore the future of applications, OSes, and processor design on a new memory hierarchy.

Availability Twizzler is available at twizzler.io.

Acknowledgements

This work was supported in part by the National Science Foundation (grants IIP-1266400, IIP-1841545), a grant from Intel Corporation, and the industrial members of the UCSC Center for Research in Storage Systems. We thank our shepherd, Yu Hua, the anonymous reviewers, and the members of the Storage Systems Research Center for their support and feedback.

References

[1] The musl C library. https://musl.libc.org/.

[2] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the Summer 1986 USENIX Technical Conference*, pages 93–112, Atlanta, GA, 1986. USENIX.

[3] Sasha Ames, Nikhil Bobb, Kevin M. Greenan, Owen S. Hofmann, Mark W. Storer, Carlos Maltzahn, Ethan L. Miller, and Scott A. Brandt. LiFS: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*, College Park, MD, May 2006. IEEE.

[4] 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.

[5] Piotr Balcer. An introduction to pmemobj (part 1) - accessing the persistent memory. https://pmem.io/2015/06/13/accessing-pmem.html, 2015.

[6] 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.

[7] 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.

[8] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)*, pages 309–322, Berkeley, CA, USA, 2008. USENIX Association.

[9] 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.

[10] Daniel Bittman, Peter Alvaro, and Ethan L. Miller. A persistent problem: Managing pointers in NVM. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems (PLOS '19)*, pages 30–37, October 2019.

[11] 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.

[12] 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.

[13] Guoyang Chen, Lei Zhang, Richa Budhiraja, Xipeng Shen, and Youfeng Wu. Efficient support of position independence on non-volatile memory. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '17)*, pages 191–203, New York, NY, USA, 2017. ACM.

[14] Howard Chu and Symas. Lightning memory-mapped database (part of the OpenLDAP project). https://symas.com/lmdb/.

[15] 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.

[16] 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.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*, pages 143–154, New York, NY, USA, 2010. ACM.

[18] 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.

[19] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[20] Partha Dasgupta, Richard J. LeBlanc, Jr., Mustaque Ahamad, and Umakishore Ramachandran. The Clouds

distributed operating system. *IEEE Computer*, November 1991.

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

[22] Biplob Debnath, Sudipta Sengupta, and Jin Li. FlashStore: High throughput persistent key-value store. In *Proceedings of the 36th Conference on Very Large Databases (VLDB '10)*, September 2010.

[23] Xiangyu Dong, Cong Xu, Norm Jouppi, and Yuan Xie. *Emerging Memory Technologies: Design, Architecture, and Applications*, chapter 2, pages 15–50. Springer, 2014.

[24] 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.

[25] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. SpaceJMP: Programming with multiple virtual address spaces. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*, pages 353–368, New York, NY, USA, 2016. ACM.

[26] Dawson R Engler, Sandeep K Gupta, and M Frans Kaashoek. AVM: Application-level virtual memory. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS '95)*, pages 72–77. IEEE, 1995.

[27] Dawson R Engler and M Frans Kaashoek. Exterminate all operating system abstractions. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS '95)*, pages 78–83. IEEE, 1995.

[28] 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.

[29] Hewlett Packard Enterprise. YCSB-C. https://github.com/HewlettPackard/meadowlark/tree/master/extra/YCSB-C https://github.com/basicthinker/YCSB-C, 2018.

[30] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS '15)*, Kartause Ittingen, Switzerland, May 2015. USENIX Association.

[31] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole, Jr. Semantic file systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 16–25. ACM, October 1991.

[32] Burra Gopal and Udi Manber. Integrating content-based access mechanisms with hierarchical file systems. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 265–278, February 1999.

[33] Jorge Guerra, Leonardo Mármol, Daniel Campello, Carlos Crespo, Raju Rangaswami, and Jinpeng Wei. Software persistent memory. In *Proceedings of the 2012 USENIX Annual Technical Conference*, 2012.

[34] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: a distributed single address-space operating system. Technical Report 9314, School of Computer Science and Engineering, University of New South Wales, November 1993.

[35] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI '11)*, pages 295–308, Berkeley, CA, USA, 2011. USENIX.

[36] Antony L. Hosking and J. Eliot B. Moss. Object fault handling for persistent programming languages: A performance evaluation. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA '93)*, pages 288–303, New York, NY, USA, 1993. ACM.

[37] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibrod. Log-structured non-volatile main memory. In *Proceedings of the 2017 USENIX Annual Technical Conference*, pages 703–717, Santa Clara, CA, June 2017.

[38] 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. *arXiv*, abs/1903.05714, 2019.

[39] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[40] Eric Jul and Bjarne Steensgaard. Implementation of distributed objects in Emerald. In *Proceedings of International Workshop on Object Orientation in Operating Systems*, pages 130–132. IEEE, 1991.

[41] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, New York, NY, USA, 1997. ACM.

[42] 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 2006 (EuroSys '06)*, pages 133–145, New York, NY, USA, 2006. ACM.

[43] 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.

[44] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight contexts: An OS abstraction for safety and performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 49–64, GA, 2016. USENIX Association.

[45] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence: Efficient transactions in persistent memory. *ACM Transactions on Storage*, 12(1), January 2016.

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

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

[48] Pankaj Mehra and Samuel Fineberg. Fast and flexible persistence: The magic potion for fault-tolerance, scalability and performance in online data stores. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, January 2004.

[49] Justin Meza, Yixin Luo, Samira Khan, Jishen Zhao, Yuan Xie, and Onur Mutlu. A case for efficient hardware/software cooperative management of storage and memory. In *5th Workshop on Energy-Efficient Design (WEED '13)*, June 2013.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] Yoann Padioleau and Olivier Ridoux. A logic file system. In *Proceedings of the 2003 USENIX Annual Technical Conference*, pages 99–112, San Antonio, TX, June 2003.

[55] Aleatha Parker-Wood, Darrell D. E. Long, Ethan L. Miller, Philippe Rigaux, and Andy Isaacson. A file by any other name: Managing file names with metadata. In *Proceedings of the 7th Annual International Systems and Storage Conference (SYSTOR '14)*, June 2014.

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

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

[58] Andy Rudoff et al. Persistent memory programming library. `http://pmem.io/nvml/`, 2017.

[59] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*, page 323–337, New York, NY, USA, 2017. Association for Computing Machinery.

[60] 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.

[61] 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.

[62] Eugene Shekita and Michael Zwilling. Cricket: A mapped, persistent object store. Technical Report 956, University of Wisconsin, August 1990.

[63] 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.

[64] 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.

[65] 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.

[66] 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.

[67] Robert NM Watson, Jonathan Woodruff, Peter G Neumann, Simon W Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, et al. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37. IEEE, 2015.

[68] William Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, C. Pierson, and Fred Pollack. HYDRA: The kernel of a multiprocessor operating system. *Communications of the ACM*, 17(6):337–345, June 1974.

[69] 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.