Twizzler: An Operating System for Next-Generation Memory Hierarchies

Technical Report

UCSC-CRSS-17-01
UCSC-SSRC-17-01

December 5th, 2017

Daniel Bittman dbittman@ucsc.edu
Matt Bryson mbryson@ucsc.edu
Yuanjiang Ni yuanjiang@ucsc.edu
Arjun Govindjee agovindj@ucsc.edu
Isaak Cherdak icherdak@ucsc.edu
Pankaj Mehra pankaj.mehra@ieee.org
Darrell Long darrell@ucsc.edu
Ethan Miller elm@ucsc.edu

Center for Research in Storage Systems
Storage Systems Research Center

Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064

http://crss.ucsc.edu/
http://ssrc.ucsc.edu/

Abstract

The introduction of NVDIMMs (truly non-volatile and directly accessible) requires us to rethink all levels of the system stack, from processor features to applications. Operating systems, too, must evolve to support new I/O models for applications accessing persistent data. We are developing Twizzler, an operating system for next-generation memory hierarchies. Twizzler is designed to provide applications with direct access to persistent storage while providing mechanisms for cross-object pointers, removing itself from the common access path to persistent data, and providing fine-grained security and recoverability. The design of Twizzler is a better fit for low-latency and byte-addressable persistent storage than prior I/O models of contemporary operating systems. We are in the process of building a prototype inside FreeBSD, allowing us to leverage and explore the ideas presented in this report without investing the time to build an operating system from scratch first.

The purpose of this report is to provide an overview of Twizzler and the design behind it while giving a direction for our upcoming work. We intend to further develop our ideas by extending our FreeBSD prototype, followed by designing a kernel and hardware extensions to make better use of some of our designs (resulting in improved security, simplicity, and improved support for persistent kernel state).

1 Introduction

We are on the cusp of a fundamental shift in the way that we program computers. For seven decades, programmers have written programs that are loaded, initialized, executed, and terminated. In the very near future, multiple vendors will introduce persistent memory that will attach to the memory bus. Not since the days of small magnetic core memories fifty years ago have we had mutable memory from which our programs execute directly and still survive power cycling. Unlike flash and PCIe based memories, these new memories will be byte addressable. Consequently, we are entering a new era, and it is essential that the operating system and other system software make the best possible use of this new technology. However, leaving the design choices solely to operating system vendors risks incremental change and the inability to benefit from the dramatic improvements in performance, programming models, and security that this may facilitate.

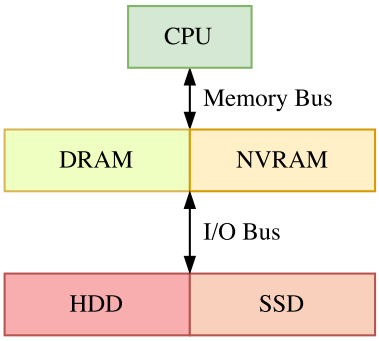

Figure 1: Expanded form of the memory hierarchy. We expect non-volatile memory to have asymmetric access time with latency similar to DRAM [17]. We expect NVM presence to range from NVM-only systems (IoT devices) to a mix between DRAM and NVM.

Given a new memory hierarchy including byte-addressable non-volatile memory (shown in Figure 1), we have a chance to revisit operating system design. Existing systems maintain separate management domains for volatile, high-speed memory, on which computation operates, and persistent high-latency storage, which cannot be operated on directly. Operating systems designed for such systems reflect this structure and provide applications with interfaces to interact with the memory hierarchy. System calls such as `write()` and `read()` reflect the basic structure of using DRAM as a cache for disk, while `mmap()` uses virtual memory to provide the illusion of direct persistent object access. Both cases are fundamentally the same, with the operating system interposing itself between the application and its desired actions of persistent data access, with the need to manage ephemeral copies of persistent data in volatile memory. Moreover, the existing approach to persistence via block storage requires data serialization, exacting a penalty that is becoming increasingly larger as NVM performance increases [62]. Applications that use NVM-optimized techniques for persistence [10, 34, 64] avoid this overhead, but the problem of managing persistent pointers in a very large address space remains.

To address these challenges, our lab has been developing Twizzler, an operating system designed to support persistent, byte-addressable memory by "getting out of the way" of applications as much as possible as they operate on persistent data. We are focusing on the following items that we believe to be key requirements and challenges of an NVM-aware system:

Persistent Object Access The I/O model that the operating system provides must allow applications direct access to persistent data with minimal kernel involvement while also providing a rich enough model for cross-object data references and a framework for object naming and late-binding. Twizzler will maintain a namespace of data objects, each with a unique 128-bit ID. These objects may contain code, data, or both. Much like the UNIX and Plan-9 "everything is a file" approach, we plan to have every piece of data on which applications operate be one of these objects. Since these persistent objects are directly accessible from the processor, and because we can uniquely identify objects by ID, we can have "cross-object pointers". Such pointers refer to data in other objects, and an application may use these pointers to read and operate on data in other objects. Cross-object pointers allow us to support more natural programming semantics for non-volatile memory without the manual pointer-loading and swizzling (pointer translation) [39] required by applications on current operating systems [14].

Security of Persistent Objects Direct access to persistent objects increases the need for richer security models. Applications must be able to access sensitive or protected data while protecting that data from untrusted libraries or unverified subcomponents. In Twizzler, we separate components of applications into *security contexts* in order to provide isolation and access control. Security contexts allow us to protect applications at finer granularity than current systems, thereby reducing the risk of allowing direct access to persistent data.

Persistent Kernel State The kernel must be designed with *persistent* kernel state in mind. The kernel uses persistent objects accessible from user-space to determine the internal kernel state (such as thread state and address spaces), thereby allowing the operating system state to be recovered easily after a power failure. The kernel state is then a cache of *kernel state objects* which can be reconstructed after a power failure. Kernel state objects have the added benefit of reducing system calls and simplifying the application-kernel boundary.

Hardware Support for Persistent Objects While we can implement our design on existing hardware, extending processors with new primitives will allow applications to be simpler and have higher performance. We propose to increase the virtual address space size and reintroduce segmentation support. While existing processors require us to emit additional pointer swizzling and dereference instructions, hardware support to enable direct support for our cross-object pointer design and security features such as protection lookaside buffers could improve performance and simplicity of Twizzler as well.

While our eventual goal is to provide this support by creating a new operating system (including a new kernel), such an "all-or-nothing" approach would require too much initial effort before being able to demonstrate the efficacy of the approach. Thus, we are currently implementing these techniques inside FreeBSD, allowing us to evaluate them without the need to build an entire operating system from scratch. This incremental approach has a second advantage: it allows system designers to immediately leverage the techniques, which we will open-source, without the need to adopt a new operating system, thus increasing the impact of the research. Our prototype Twizzler system implemented in FreeBSD will provide applications with a full Twizzler framework and interface, allowing us to later implement a kernel ourselves to better demonstrate some of our ideas.

2 Persistent Objects

Twizzler's I/O model follows from our expectations of upcoming memory hierarchies designs. Current I/O models that use calls such as `read()` and `write()` are designed for indirect, high-latency access to persistent storage. These archaic models involve significant operating system overhead that, while acceptable in a two-tier memory hierarchy model, is unacceptable in a memory hierarchy with low-latency access to persistent storage. Since persistent data is directly accessible, copying data into temporary buffers, as is common when using `read()` and `write()`, makes little sense. Instead, we can directly map the same persistent data into multiple address spaces without wasteful copying. Zero-copy I/O [45, 52, 61] alleviates these problems to some extent, but still has heavy kernel involvement and is still designed for a model with an explicit device I/O operation to persist and acquire data. While memory mapping (`mmap()`) provides some relief, it places a heavy burden on application programmers who wish to share mapped regions across processes because the address space layouts differ across programs and so shared data must be translated in some way or be location-dependent.

Reducing copies means that each address space that maps the same data will see the same contents. If that data contains pointers, they must have a form which allows any process to dereference the pointer regardless of the mapped location. Techniques such as explicit serialization are well studied [62], but fall short here—explicit serialization would be wasteful since data is already stored in persistent memory, and slow compared to the device latency. It makes more sense to keep the data in a format that does not require an explicit serialization step. Because concurrent access to objects (without copies) from different address spaces will be common, pointers must be kept in a format that allows dereferencing from anywhere in *any* virtual address space. We will discuss how we achieve this in theory (below) and in practice (section 6).

Twizzler organizes logical units of storage into *objects*, similar to files in UNIX, allowing users to coalesce data into meaningful units for ease of identification, storage organization, and access control. An object is uniquely identified with a 128-bit ID (though larger IDs would be possible). Twizzler provides applications with a logical view of the object address space, allowing them to map objects contiguously into their virtual address spaces. Since we are already storing pointers in a universally accessible format, we allow for this format to refer to data within external objects. These cross-object pointers are a natural programming model that arises from the persistence of data in main memory, and have been present in earlier operating system designs [9, 13].

The use of cross-object references allows the construction of applications where the operating system code (and more specifically, kernel code) is rarely invoked during persistent data access. Instead, applications are capable of doing most of their work without having to involve the kernel for I/O. This improves performance by reducing kernel boundary crossings, boosts security by simplifying and shrinking the kernel, and provides middleware with more flexibility so applications can make the best decisions for how to operate on data.

Invariant Pointers To support cross-object pointers, Twizzler stores pointers in the form `object-id:offset`. While this has been studied before [13, 54, 55], we differ by adding a level of indirection per object. Each object has a *Foreign Object Table* (FOT)—a table of entries containing object IDs along with additional information such as permissions. A pointer is then stored in a `FOT-entry:offset` format. When it is dereferenced, the appropriate entry in the FOT is selected according to the `FOT-entry` field of the pointer, where a value of 0 indicates an intra-object (local) pointer. The target object ID entry is read from the FOT and used to dereference the pointer, as shown in Figure 2. The FOT is a per-object data structure, so the combination of an object's FOT and a `FOT-entry:offset` pointer within the object corresponds logically with a pointer of the form `object:offset`.

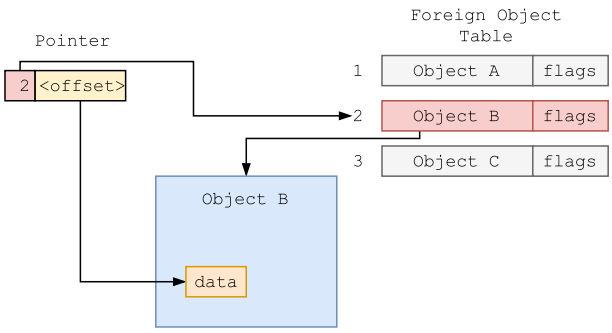

Figure 2: Translation of a pointer through the FOT. The pointer and the FOT are contained within some other object that is not shown.

Using a per-object FOT for indirection of pointers provides a number of benefits:

- While an object may refer to relatively few external objects, we expect each object will have a different set of referenced external objects, making a system-wide FOT infeasible.
- We have elected to keep pointers 64 bits in order to avoid increasing the memory usage and hardware costs of storing and using pointers, and to improve compatibility with current processors. This means that the `FOT-entry` and `offset` fields must share a limited space. In order to allow objects to grow to large sizes, the `FOT-entry` field is limited, likely to 16–24 bits. Since this is not sufficient for globally unique identifiers (GUIDs) for objects, we must use an indirection table.
- By storing the larger object IDs in a table, we can save memory when we have a large number of pointers refer to the same object.
- We can support late-binding in the FOT by allowing *names* in addition to fixed object IDs. These names can then be resolved into an object ID by a *name resolver*. Late-binding [13]

lets us use human-readable names and allows pointers to refer to different objects over time (for example, *current* account information).

- We can associate additional information with FOT entries, and in a single operation, change that data for a large number of pointers. This works with names or object IDs, allowing us to rebind groups of pointers easily at runtime.

The result of using an FOT is that a pointer within an object refers not to a virtual memory address, but instead to a particular location inside a particular object. While this solves the problem we introduced of supporting cross-object pointers, it affords us even more flexibility when making our system distributed. Invariant pointers have a universal form on any machine, making them a natural fit for distributed computation. While we are not directly considering distributed computation in Twizzler yet, it is a valuable feature-set to keep in mind for future work.

The Twizzler Object Model is heavily influenced by MULTICS [5, 12, 13] and single address space operating systems [9, 31, 53, 59]. The complexity required to properly support persistent segmented data structures was difficult for the hardware and software at the time of MULTICS, but now the cost of silicon has dropped significantly enough that we can build support for efficient address translations for `object:offset` pointers. Segmentation support has fallen by the wayside in modern processors, but we believe that it is a natural model for persistent object access. The Twizzler I/O Model fits naturally with it, but does not require it. We have solutions for support on contemporary hardware, as well as a vision for future support to improve performance and simplicity (section 6).

3 Security and Access Control

Because our design removes the operating system from the path of the I/O accesses of applications, Twizzler must ensure that access control to objects is correctly specified in page-tables so that access control can be enforced by the hardware. Additionally, the kernel must be able to check and assign permissions quickly because the points at which checks must occur may happen frequently. Twizzler meets these goals by running threads in *security contexts*, which contain the necessary information for the kernel to determine the permissions for any object mapping in an application's address space. Security contexts also allow the kernel to transparently separate trusted and untrusted components of a process, partitioning the application's address space to protect sensitive data. By coalescing access control information into security contexts, the kernel can cache permission bits and can ensure separation of privilege for threads running in different security contexts but in the same virtual address space.

We separate the notion of access control from the *protections* of a virtual address space in order to facilitate sharing of page-table structures while providing applications with a way to freely define protections. This separation allows the system to restrict processes' access rights without needing to check the protections defined by a process. In section 6 we discuss how this is implemented on Intel x86-64 processors using two-level address translation.

Security Contexts A thread may be associated with any number of security contexts, each of which may have many threads running inside it. However, a thread may only have one *active* security context at any given time, and the thread runs with the permissions defined by its active context. If a thread attempts a memory access that is invalid in its active security context, Twizzler looks for a security context with which the thread is associated and in which the attempted access is valid. If it finds one, it switches the thread to it automatically. Security contexts can thus be used to isolate components of programs from each other or from untrusted libraries. An example of this is shown in figure 3. The thread is associated with both contexts, but only one is active. Should the thread attempt an access in one context that is not valid, the kernel can switch it to the other context. This prevents the untrusted code from corrupting the protected data while still letting the program use untrusted services if necessary. We can also use this technique to limit the access of a particular program or library to outside communication.

Although the kernel could do a linear search through security contexts looking for one in which a particular access is valid, such an approach may be slow if the kernel must search though a large number of them; therefore a mechanism for optimizing the method the kernel uses for searching would have potential performance benefits. Possible solutions involve organizing the list of security contexts a thread is associated with by most-recently-used, or prioritizing certain security contexts over others. The kernel's thread scheduler could make use of active security contexts in its scheduling decisions in

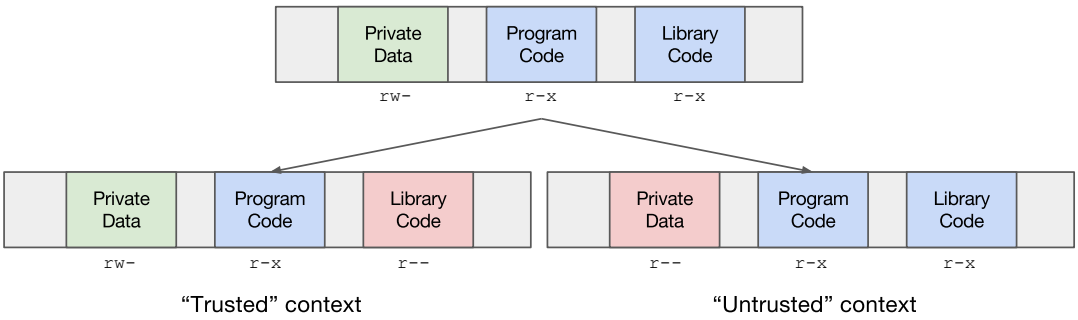

Figure 3: Two security contexts derived from a process's virtual address space, designed to protect a "private data" object from an untrusted library.

order to improve performance, however we plan to explore methods of making a security context switch lightweight with improved hardware support such as support for protection lookaside buffers [66] (see section 6).

Capabilities and Delegations Security contexts present to the kernel a set of capabilities and delegations which together provide the access rights to objects for threads in that security context. The capabilities and delegations are signed with a private key (per-object or per-user), and are therefore unforgeable. Whereas previous systems often require kernel support to create capabilities [28, 47, 57], signed capabilities would allow users to create them [47], and the kernel need only verify them without needing to hide them from users or maintain a protected set of them or verification data internally. Furthermore, in a distributed environment, capabilities could prove access rights across systems without needing heavy interaction between machines or a centralized authorization system.

Each object references a public and a private key with which its capabilities or delegations are signed. The kernel needs only the public key to verify a given capability or delegation, so we can keep private keys protected either by not distributing them (for objects on other machines) or by encrypting them. Since the capabilities are signed, the natural semantics for creating capabilities is that anyone who can read an object's private key (and optionally decrypt it) can create capabilities and delegations with that key. Since a private key is itself an object, we can apply our existing access control mechanisms to it.

4 Persistent Kernel State

In a memory hierarchy where persistent memory is directly accessible from the processor, a power cycle should result in only a minimal loss of system state: CPU state and perhaps cache. Applications can tolerate power cycling and state loss when operating on persistent data through various existing techniques [10, 11, 30, 55, 64, 65], but the operating system is more complex and needs to manage low-level hardware. Upon resume the operating system must re-instantiate threads and address spaces so that the system state is consistent once power is restored, after which applications may resume using their choice of mechanism for consistency across power cycling. Current operating system design handles this poorly because it is built around the idea of the kernel holding a large amount of system state internally in a volatile manner with that state mutating (primarily) through system calls.

In contrast, Twizzler is designed to keep kernel state minimal and in persistent *kernel state objects* (KSOs). An example of a KSO is a security context, as described above. The persistent security context KSO contains capabilities and delegations needed for the kernel to correctly map objects, which the kernel reads when needed. Another example of a KSO is a thread object, which contains thread-local data and thread control and state information for the kernel. The information that the kernel reads is then cached internally to improve future lookup performance, and changes to that state are propagated to the KSOs as necessary. Since all kernel state is represented in KSOs, the kernel's internal state is then just a cache of the information present in the control objects in the system. When a power cycle occurs, the state of the operating system can be recovered easily by loading the appropri-

ate KSOs. Furthermore, the state can be recovered *quickly* (since we no longer have to manually setup the system state through numerous system calls), making it possible to shutdown the system with smaller interruptions in service. Using the Twizzler model, small devices that only occasionally need to do work can save significant power by both using only NVDIMMs (which save power over DRAM's need for constant refresh [19]) and spending most of the time powered off, only to wake up occasionally to service a request before shutting down again.

When a program updates a kernel state object, it must notify the kernel through an invalidation system call. The kernel will then invalidate any internally cached information and reread the appropriate part of a KSO if necessary. Also note that not all updates to KSOs require an invalidation system call. For example, adding a capability to a security context does not require an invalidation, since the kernel has no information for that capability to cache in the first place. This has the added benefit of reducing the number of system calls required to mutate kernel state.

The security of KSOs is a paramount issue; however, they are normal persistent objects, so we can apply the standard access control system introduced above. Additionally, the format of KSOs must help the kernel protect itself from possibly malicious input. To facilitate this, KSO data may not contain external pointers, and the kernel must sanitize all data read from KSOs. Giving super-users access to KSOs in Twizzler is not unlike the unrestricted powers of root on Unix today; however in Twizzler we can apply much more fine-grained access control to better protect the system against mistakes and external attacks.

Of course, protection from kernel bugs is also necessary. Whereas the go-to solution in the past for program state corruption was to reboot the machine, persistent memory would not reset errors. We must have a mechanism in the kernel to prevent erroneous writes and to recover from them if they occur. While programs must also be designed for these concerns, the kernel is of more importance. For one, the operating system will be involved in resetting application state if necessary, and two, erroneous operating system state can be more devastating. The operating system would keep a set of "clean" kernel state objects which are not mapped into memory unless necessary to recover the system. Additionally, we can take advantages of techniques described by PMFS [18] to protect persistent memory from the kernel, and we can go further by leveraging hardware extensions (such as those described in section 6) to protect data from the kernel.

On system startup, including first-time initialization or after a power cycle, the initialization process attaches KSOs to the kernel, which then allows the kernel to resume threads that were running. Once this is done, the kernel will have returned to the state it was before power was lost. However, applications that were in the middle of an operation may have lost updates or have partially completed work. We expect applications to be programmed such that they can be resumed safely and continue near where they left off [10, 55, 64]. If an application is not programmed this way (*e.g.*, a traditional Unix program), it can be optionally reset upon system start up to emulate the lost-process semantics of current operating systems. Some programs may need to be informed that the power was cycled, and can optionally receive a signal informing them before they resume. After receiving such a signal, the recovery logic of the NVM support library can be invoked, followed by applications specific reinitialization logic. In this way, applications that use durable transactions to update data structures can resume safely after a power interruption.

For most types of KSOs, handling interrupted updates is simple: security contexts and virtual address space definitions are only read by the kernel and only affect Twizzler processes. However, KSOs that are updated by the kernel (for example, thread control objects) require additional work. Both user-space and the kernel must agree on how KSOs are to be updated in a durable manner. Part of our research into KSOs is determining how to safely update state in KSOs.

For threads in particular, there are two options to handle checkpointing and resuming. Threads could register a signal handler for a "power-resume" signal which is invoked when the system resumes. Durable transactions that the thread performed on persistent data maintain transaction semantics, allowing the thread to make decisions on how to resume. This had the advantage of not needing to store thread control block data inside the persistent object in order to resume it. Instead, the thread is resumed by allowing it to return to a checkpoint and continue, or choose an alternate approach. Alternatively, the kernel could be modified to store thread control block data inside a KSO. This would allow a thread to be resumed close to where it left off, but not necessarily exactly where it left off. In order to handle interruption of execution inside the kernel, the current system call number for the thread could

be stored in a KSO upon entry and cleared upon return to user-space. Then, when power resumes, the kernel can restart the system call. Since the primary system call is invalidation, there is little penalty to running the system call multiple times. There is an additional concern where a power-failure happens while entering a system call. Should this occur, we can support atomic transaction-like updates of the thread control block in persistent storage.

5 FreeBSD Prototype

We are building a prototype of Twizzler inside FreeBSD 11 for use on Intel x86-64 processors, with plans to add support for RISC-V in order to explore processor extensions (discussed below). Our early-stage prototype runs Twizzler programs inside of a FreeBSD process as threads within security contexts and "views" into the object address space (see Section 6). It currently provides support for these address spaces and Twizzler threads. The threads have access to the functionality required for the Twizzler Object Model along with the ability to continue to use existing FreeBSD services. We are currently implementing a layer which allows us to handle POSIX system calls within Twizzler, falling back to calls to FreeBSD if necessary. This shim layer allows us to run existing software on Twizzler with minimal porting effort.

Implementing a prototype inside FreeBSD allows us to explore our proposed ideas without committing to the effort of writing a kernel from scratch. Even though Twizzler programs can continue to use FreeBSD services, we consider Twizzler processes operating together to be a separate operating system that runs alongside FreeBSD. This is reflected not only in the access control policies of Twizzler processes, but also in the core sets of system calls, user-space libraries, process listings, and programming models. Our prototype further gives us the ability to use parts of the Twizzler design inside existing programs. For example, we can use the security contexts framework on existing POSIX programs to provide isolation of sub-components of a process. Prior work has established the utility of such a feature [7, 24, 43] in applications such as `ssh` servers or data management programs for sensitive data.

6 Hardware Support

While our designs are implementable on modern hardware, extensions to current processor features could improve performance and yield a simpler implementation. We have been implementing our prototype for Intel x86-64 processors, but doing so efficiently requires us to use certain components of the hardware in new ways. A future target for support is the extensible RISC-V architecture [3], and we will design modifications to RISC-V to better support the Twizzler model. Extensions to processor architecture are already required in order to support consistency across power interruptions with NVDIMMs [11, 49], so we will take this opportunity to motivate additional features to improve persistent object access as well.

Segmentation and Large Address Spaces Many current processors limit their virtual address spaces to less than 64 bits. We propose to extend the virtual address space to a full 64 bits. Reintroducing segment registers would pair nicely with this change by allowing us to directly use `object-id:offset` type pointers by mapping the `object-id` field to a segment register and dereferencing using the `offset` field with the segment register, avoiding the need for pointer swizzling [39] on external object accesses. Each segment has a page-table to map (possibly large) pages to physical memory. This allows us to easily move object data between components in a heterogeneous memory system while still providing similar features as current virtual memory systems (such as demand paging and "unlimited" address space). Note that increasing the virtual address space size to 64 bits may require deeper page-table structures if page sizes are kept the same. However, since we map objects contiguously at the thread level and objects are less likely to be evicted from NVDIMMs, we can afford to map larger contiguous regions of objects, thereby reducing the page-table depth to balance out the effect of the increased address space.

A possible extension to RISC-V would be to add a small per-thread segment table along with instructions for dereferencing with `segment:offset` arguments. The segment table would provide a finite number of valid segments that can be indexed via a small integer, which would allow better hardware support for the FOT. Each entry can contain an object ID and a pointer to a page-table. User-space code would then be able to add objects to the segment table as needed, and the processor can trap into the kernel in order to update references to page-tables. We could then collapse the idea of virtual address space protections and security context access control using either a *protection lookaside buffer* [66], which can be set per-thread to con-

trol the access to objects in the segment tables, or fields in the segment tables to handle protections with permission bits in page-tables to handle access control.

The extension of a protection lookaside buffer (of which some features are provided in Skylake server x86 processors) would allow us to reduce the number of page-table structures we use by decoupling permissions from mappings. This is useful here because the page-tables are used to map *objects* into physical memory, not define address spaces. Therefore, we would be able to keep the number of page-tables to just those objects which are in physical memory space. This is advantageous since those structures need to be stored per object regardless in order to keep track of page allocation for object data.

Intel x86-64 Support The solution we use on x86-64 to support the mapping power of virtual segments is to use the Extended Page Tables (EPT) hardware provided in the VT-x extensions for two levels of mapping. The EPT allows us to map a virtual address to a logical address before it is mapped with another set of page-tables to a physical address. We refer to the mid-level address space, typically called the guest physical address space, as the *object physical* space (as shown in Figure 4). Here, objects are contiguous in (logical) memory and are given permissions associated with the current security context of a thread. A thread's virtual address space also has objects contiguous in memory and is broken up into "slots". A virtual slot is mapped to an object logical slot with protections defined by an application. This allows us to easily page objects in and out of memory or move them in physical memory by only changing the object-logical mappings. Additionally, two-levels of mapping allow us to reduce the number of page-tables required by the system. Since threads may define their requested protections of an object while their active security context defines the limitations of that thread, the number of page-table structures would be $O(nm)$ in a single-level mapping system, where n is the number of running programs and m is the number of security contexts. Instead, our two-level mapping approach has $O(n + m)$ page-table structures because they can be selected and exchanged independently while the hardware applies the correct permissions as a bitwise-and between the two levels. As before, we need to store structures which record the page allocations of objects anyway, for which we can use the page-table structures in the object physical layer.

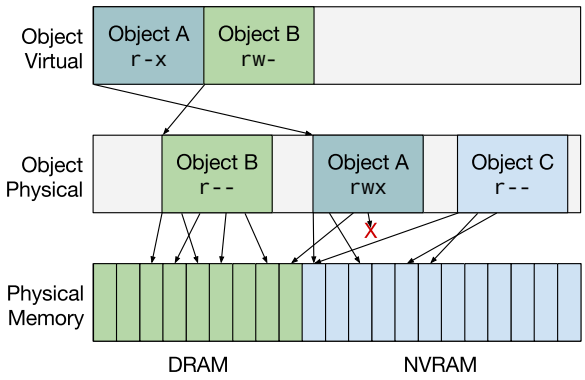

Figure 4: The use of two-level address translation in Twizzler for x86-64. Protections are assigned in the object-virtual address space and permissions (security context data) are assigned in the object-physical address space. The final access control of a mapping is the bitwise-`and` between the protections of the two levels. Objects may be moved around in physical memory without affecting object-virtual address spaces, and the separation of security contexts and virtual address spaces reduces the number of required mapping structures.

7 Name Resolution

With all data and code stored as objects, and objects residing in a single level store, our prototype operating system manages objects more like a key-value store than a file system, with no clear, built-in organization of data. This presents an opportunity to implement a more flexible method of organizing and accessing objects. Our object name resolution system, *Nomenclature*, is the mechanism Twizzler will use to translate names in the FOT into actual object GUIDs.

By including a pointer to a name resolver along with the object identification data in the FOT, Nomenclature allows the use of a wide range of name resolution mechanisms within a single system stack. Objects could have names in multiple resolvers; for example, the same objects could be organized by both a relational database resolver and a hierarchical resolver similar to a conventional filesystem. The implementation of non-hierarchical file systems is well studied [2, 27, 29, 50, 51], but these systems do not easily cooperate with one another to organize the same files or objects. By using a key-value store for the underlying objects, Nomenclature separates name translation from object retrieval, providing flexibility and extensibility for object naming.

8 Related Work

The introduction of byte-addressable non-volatile memory presents an outstanding opportunity to rethink the way that operating systems handle memory and storage. When working with new technologies it is often best to turn to the past—ideas that were only theoretically possible in the past may have become more viable with the change in time. Such is the case with the Twizzler project. Our design is shaped by fundamental operating systems papers, including Multics [12], Opal [9], IBM's K42 [40], and exokernel-style approaches [21, 22, 23, 37]. While there has been a large amount of recent research on building data structures in non-volatile memory [10, 11, 16, 44, 64], most of it has focused on how applications can provide persistent versions of existing data structures in the face of crashes that may happen at any time. In contrast, we propose to explore how persistent NVM can simplify operating systems and perhaps application programming while improving performance.

Our approach will leverage non-volatile memory characteristics to greatly reduce the need for an operating system stack and provide full system resumability, we draw on prior object-oriented operating system design work to simplify the kernel using techniques developed in systems such as K42 [40] and Exokernel [23, 38]. However, we also draw from more recent work on providing operating systems support for byte-addressable non-volatile memory systems [8], integrating the two to provide high-performance operating system support for applications running on a single-level store.

8.1 Memory Model

MULTICS was one of the first systems to use segments as a technique for partitioning memory and supporting relocation [5, 13]. The MULTICS system used segments to support location independence, but still maintained them in a file system, requiring manual linkage rather than the automated linkage we propose. Nonetheless, MULTICS demonstrated that the use of segmenting for memory management can be a viable approach.

The core of our design of the Twizzler address space is similar to concepts introduced by Opal [9], which used a single 64 bit virtual address space mapping for all processes on a system. This design made it easier to share data between applications since they use the same virtual address space, and prevented naming conflicts when importing virtual addresses for devices that may be [dis]connected during runtime. Opal's design resulted in significant speedup of data transfer/sharing as well as interfacing of devices, but it did not address the issue of file storage and resolution of names. Opal still required a file system, since there was no way to have a pointer refer to an object whose identity may change over time. In contrast, our approach adds the ability for late resolution of object pointers, removing the need for an explicit file system. Additionally, our use of hardware virtualization support will provide much stronger security guarantees than Opal. Other single-address space operating systems, such as Mungi [31] and Sombrero [60], show that the single address space approach has merit, but, like Opal, did not consider how the use of NVM would alter their design choices.

Single-level stores [56, 58] have been known for some time, though "relatively little has appeared about them in the public literature" [56], even in the decade since the EROS paper was written. Single-level stores eliminate the distinction between memory and persistent storage, using a single model for data at all levels of the hierarchy. The Clouds system [15] implemented a distributed object store in which individual objects contained code, persistent data, and both volatile and persistent heaps. Our approach proposes lighter-weight objects, allowing direct access to an object from outside, unlike Clouds. Another example of such a system is software persistent memory [30], which was designed to operate within the constraints of existing operating systems. Our proposal is to build a *usable* single level store, developing pointer mechanisms and name resolution that can leverage the built-in persistence of NVM. This idea was described by Meza [46], who suggested that hardware manage a hybrid persistent-volatile memory store with fine-grained data movement to and from persistent storage. This is contrast to existing single-level stores that must migrate data to and from disk in large chunks, making it more difficult to provide fine-grained persistence. Moreover, because most persistence in our system is to NVM, which has low access latency, our operating system need not interpose on most data movement between persistent storage and volatile temporary memory, instead simply managing mappings of persistent objects into memory, thus reducing operating system overhead.

Recently, several projects have begun considering the impact of non-volatile memories on operating system structure. For example, Bailey, *et al.* [4] suggest the use of a single-level store and the use

of NVM to ship an application as a checkpoint of a running process. Our proposed research will explore this approach and its implications. The Moneta project [8] noted that, in their prototype, eliminating the heavyweight operating system stack dramatically improved performance by reducing overhead. While this work was focused on I/O performance, not on rethinking the system stack, we will leverage their techniques to reduce operating system overhead as much as possible, even when the operating system must intervene. Lee and Won [41] considered the impact of byte-addressable 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. Kernel state objects are one of our solutions to problems of power-failure recovery, also discussed by Narayanan [48]. However, Twizzler allows for systems which are not designed to serialize CPU state on power-failure. Additionally, kernel state object have additional performance, networking, and simplicity benefits.

8.2 Object Model

IBM's K42 operating system [40] provided inspiration for the high-level design principles of Twizzler. The object-oriented approach to designing a micro- or exokernel used in K42 forms an efficient design for Twizzler to implement operating system components simply and modularly. Like K42, Twizzler will lazily load or map in only the resources that an application needs to execute. Similar techniques for faulting-in objects at run-time have been studied [33]. Communication between objects in Twizzler will be implemented as protected procedure calls, similar to K42, but we will develop new methods to implement them on modern hardware.

Prior work such as Emerald [35, 36] and Mesos [32] implemented object mobility over the network, which Twizzler could support through its object model. Emerald implemented a kernel, language, and compiler to allow objects to become mobile over the network, using wrapper data structures to track metadata for each object and presenting these objects in a object-oriented language. This approach adds layers of indirection for even simple operations like integer addition, causing a performance hit. Twizzler will improve on this implementation by using modern CPU hardware support to allow even lower overhead for cross-object references.

The Twizzler persistent object model was also shaped by the design of NV-heaps [10]. NV-heaps provide memory-safe persistent objects suitable for non-volatile memory systems. The paper describes memory safety pitfalls in providing direct access to NVM for application programmers such as leaks and invalid pointers. Language-level primitives are introduced, which enable a program to actively utilize persistent structures and techniques. Instead of imposing limitations, Twizzler aims to present the benefits of persistent objects to the programmer with fewer restrictions, more akin to Mnemosyne [64], to allow more powerful applications to be constructed. While NV-heaps manage different kinds of pointers with the restriction that pointers in an NV-heap can only refer to memory within the same heap, Twizzler has no such restriction, allowing for executable code to reside in an object and refer to different data structure objects or library objects by a direct pointer. We believe that some languages may choose to provide the restrictions presented in NV-heaps, but the Twizzler system will provide for more powerful usage of NVM. Additional projects such as PMFS [18] and NOVA [67] provide a filesystem for NVM devices. Twizzler, in contrast, provides direct NVM access atop of a key-value interface for objects. Although Twizzler does not direct supply a filesystem, one can be easily built atop it (as is one of the projects in involving Nomenclature in Twizzler). Furthermore, while NOVA and PMFS provide mechanisms for direct access to NVM, NOVA adds an indirection layer with copies and both use `mmap()` interfaces (which fall short as discussed above) and, unlike Twizzler, require significant kernel interaction when using persistent memory.

8.3 Security Model

Our use of a small kernel that "gets out of the way" is influenced by systems such as Exokernel [23] and SPIN [6], both of which drew on the Mach microkernel design [1]. In Exokernel, much of the operating system functionality is implemented in user space, with the kernel providing only resource protection. Our approach is similar in some respects, but goes much further in providing a single unified logical name space for all objects, making it simpler to develop applications that can directly leverage non-volatile memory 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.

Mungi proposed a system which operates similarly to the proposed Twizzler design, namely a microkernel with page fault handling to allow an object-based approach with protected procedure calls [31]. Mungi proposed a practical implementation based on the L4 microkernel on an MIPS R4600 processor. Protected procedure calls were implemented in Mungi by temporarily replacing the threads protection domain by the union of selected subsets of the caller and callees domains. We call these protection domains security contexts in Twizzler, and propose a novel implementation that takes advantage of modern x86 virtualization hardware to improve performance. Mungi recognized that this object model would be highly suitable for persistent objects, but did not propose any method of pointer swizzling [39], instead opting to limit objects to a single address space. Twizzler's x86 implementation allows efficient swizzling and even the transmission of objects over the network.

Isolating components of a process has been studied before, but previous efforts have fallen short in certain aspects. Wedge [7], for example, separates privilege via separate Linux processes, resulting in additional scheduling and context-switching overhead. In our implementation of security contexts, scheduling processes and threads has minimal additional overhead. Lightweight-contexts [43] improve in this regard by making their protection contexts orthogonal to threads with first-class support in the kernel. While we also follow this approach, both Wedge and Light-weight Contexts require applications to be refactored to make use of protection domains. Our approach allows unmodified software to undergo privilege separation. Previous work has also been explored on threads switching entire address spaces, whether as a performance optimization (as in the case of Mach [25]) or as a method of address space extension and data sharing (as in the case of SpaceJMP [20]). Address space switching of threads is also discussed by Lindstrom *et al.* [42]. While the primary goal of these systems is not to provide privilege separation and security, our system shares the thread migration nature of these systems, and allows the kernel to automatically migrate threads to the correct context as necessary.

While there are numerous existing operating systems which use capabilities as their primary form of access control [26, 28, 47, 57, 63], these systems either often keep and track capabilities in the kernel or require the kernel to distribute or create them. We plan to model our system around signed capabilities, requiring only a public key to verify a capability. The capabilities can prove their authenticity to the kernel without it needing to create or protect them in a special way.

9 Future Work

Beyond extending our FreeBSD prototype, implementing and testing kernel state object and security contexts, and exploring the design of a userspace built around our I/O model, there are three additional significant aspects to this work that we will pursue:

1. Current toolchains do not emit the code necessary to do the translation of pointers into a usable form upon load or dereference. We will modify the toolchain to automatically emit the correct code by adding a pass to the LLVM compiler. In the meantime, code we write will need to include calls to functions to handle the transformations of pointers. We also plan to extend the compilers to include language support for defining cross-object pointers.

2. Building our own kernel will allow us to explore a kernel design based solely off of kernel state objects. We will use that to develop an operating system that is capable of handling unexpected power cycles with resumability for applications (up to their ability to checkpoint and resume themselves).

3. We believe our object model and operating system design is well equipped to handle a networked environment. We plan to explore the interplay between the object model, content-addressable networks, and a capability-based security model. We expect to build a distributed variant of Twizzler which will allow us to cheaply move data and computation around the system automatically.

10 Conclusion

Operating systems must evolve to support future trends in memory hierarchy organization. Failing to evolve will not only relegate new technology to outdated access models thereby not allowing it to reach its potential, but it will also make it more difficult for operating systems to evolve in the future. We must not pass up a chance to provide applications with a better data access model going forward than current modes designed for tape and disk. We believe that Twizzler will show a way forward with an object model designed around NVDIMMs and

an operating system that provides new semantics for direct access to persistent memory with fine-grained security and recoverability. The invariant pointers we defined will allow us to support cross-object pointers with low-overhead. The operating system kernel itself will support persistent kernel state and allow it to recover from power cycling and crashes. It will remove itself from path of applications in their accesses to persistent data, thereby providing middleware with improved flexibility and performance. Twizzler will give us a system from which we can build our a persistent-memory based operating system and explore the future of applications, operating systems, and processor design on a new memory hierarchy.

Acknowledgements

This research was supported in part by the National Science foundation grant number IIP-1266400, by a grant from Intel corporation, and by the industrial members 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.

References

[1] 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.

[2] 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.

[3] Krste Asanović and David A Patterson. Instruction sets should be free: The case for RISC-V. Technical Report EECS-2014-146, University of California at Berkeley, 2014.

[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-XIII)*, May 2011.

[5] 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.

[6] 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.

[7] 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.

[8] 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*, 2010.

[9] 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.

[10] 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 2011)*, March 2011.

[11] 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.

[12] 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.

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

[14] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y A Khalidi, and C. J. Wilkenloh. Design and implementation of the clouds distributed operating system. *Computing systems*, 3(1):11–46, 12 1990.

[15] Partha Dasgupta, Richard J. LeBlanc, Jr., Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operating system. *IEEE Computer*, November 1991.

[16] 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.

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

[18] 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.

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

[20] 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.

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

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

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

[24] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306. Boston, MA, 2008.

[25] Bryan Ford and Jay Lepreau. Evolving mach 3.0 to a migrating thread model. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 9–9, Berkeley, CA, USA, 1994. USENIX Association.

[26] Bill Frantz, Norman Hardy, Jay Jonekait, and Charles R Landau. Gnosis: A prototype operating system for the 1990's. In *Proceedings of SHARE*, volume 52, pages 3–17, 1979.

[27] 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.

[28] Li Gong. A secure identity-based capability system. In *In Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 56–63, 1989.

[29] 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)*, pages 265–278, February 1999.

[30] 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.

[31] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: a dis-

tributed single address-space operating system. Technical Report 9314, School of Computer Science and Engineering, University of New South Wales, November 1993.

[32] 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 Association.

[33] 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.

[34] Qingda Hu, Jinglei Ren, Anirudh Badam, and Thomas Moscibrod. Log-structured non-volatile main memory. In *Proceedings of the 2017 Usenix Annual Technical Conference*, June 2017.

[35] 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.

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

[37] 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.

[38] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceno, Russell Hunt, David Mazieres, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *ACM SIGOPS Operating Systems Review*, volume 31, pages 52–65. ACM, 1997.

[39] Alfons Kemper and Donald Kossman. Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis. *VLDB Journal*, 4:519–567, 1995.

[40] 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.

[41] 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.

[42] A. Lindstrom, J. Rosenberg, and A. Dearle. The grand unified theory of address spaces. In *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, HOTOS '95, pages 66–, Washington, DC, USA, 1995. IEEE Computer Society.

[43] 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 *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 49–64, GA, 2016. USENIX Association.

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

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

[46] 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 2013)*, June 2013.

[47] Sape J Mullender, Andrew S Tanenbaum, and Robbert van Renesse. Using sparse capabilities in a distributed operating system. In *Proceed-*

ings of the 6th IEEE conference on Distributed Computing Systems, 1986.

[48] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. *SIGPLAN Not.*, 47(4):401–410, March 2012.

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

[50] 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.

[51] 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 2014)*, June 2014.

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

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

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

[55] Andy Rudoff et al. Persistent memory programming library. http://pmem.io/nvml/.

[56] 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.

[57] 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.

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

[59] 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.

[60] 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.

[61] Moti N. Thadani and Yousef A. Khalidi. An efficient zero-copy i/o framework for unix. Technical report, Sun Microsystems Laboratories, Inc., may 1995.

[62] 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.

[63] J. Vochteloo, S. Russell, and G. Heiser. Capability-based protection in the mungi operating system. In *Proceedings Third International Workshop on Object Orientation in Operating Systems*, pages 108–115, Dec 1993.

[64] 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 2011)*, March 2011.

[65] Zhaoguo Wang, Han Yi, Ran Liu, Mingkai Dong, and Haibo Chen. Persistent transactional memory. *IEEE Computer Architecture Letters*, 14(1), 2015.

[66] John Wilkes and Bart Sears. A comparison of protection lookaside buffers and the PA-RISC protection architecture. Technical Report HPL-92-55, HP Laboratories, March 1992.

[67] 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.