ABSTRACT

Title of Dissertation:	Composing and Decomposing OS Abstractions
	James Benjamin Litton Doctor of Philosophy, 2021
Dissertation Co-directed by:	Professor Bobby Bhattacharjee Professor Peter Druschel Department of Computer Science

Operating systems (OSes) provide a set of abstractions through which hardware resources are accessed. Abstractions that are closer to hardware offer the greatest opportunity for performance, whereas higher-level abstractions may sacrifice performance but are typically more portable and potentially more secure and robust. The abstractions chosen by OS designs impose a set of trade-offs that will not be well-suited for all applications.

In this dissertation, we argue the following thesis: *Supporting novel hardware such as non-volatile RAM (NVRAM) and new abstractions like fine-grained isolation while maintaining efficiency, usability, and security goals, requires simultaneous access to both high-level OS abstractions and compatible access to their low-level decompositions.* We support this thesis by offering two new abstractions, PTx and light-weight-contexts (*lwC*s), as well as the null-Kernel, a new OS architecture. PTx is a new high-level abstraction for persistence built on top of NVRAM, a new form of persistent byte addressable memory, whereas *lwC*s are a new OS abstraction

that enables fine-grained intra-process isolation, snapshots and reference monitoring. Due to the efficiency requirements of both PTx and *lwC*s, both abstractions required access to low-level decompositions of higher-level abstractions, while interoperability requirements dictated that both low and high-level abstractions were exposed simultaneously. The null-Kernel is an OS architecture that enabled the simultaneous exposure of multiple abstractions for the same underlying hardware in a safe way, which, if adopted, would accelerate the development and deployment of abstractions such as PTx and *lwC*s.

Composing and Decomposing OS Abstractions

by

James Benjamin Litton

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2021

Acknowledgments

The work in this dissertation was not a lone effort, but was instead aided by too many to mention explicitly. All the same, I would like to acknowledge the role a few individuals have played in this work and my life more broadly over this period.

First, I must thank the staff at the University of Maryland and the Max Planck Institute for Software Systems. To Brigitta, Claudia, Tom, Maria-Louise, Annika, and many others, thank you for your patience when dealing with my administrative incompetence and shepherding me through various bureaucratic and logistic obstacles that I ran into along the way.

I must also thank my advisors, Bobby Bhattacharjee and Peter Druschel. Bobby plucked me out of one of his courses and urged me to quit my inane job and start down this journey. All of this is his fault. It was a privilege to work with someone who understood where my mind was heading, even when I wasn't quite prepared to articulate it clearly. Peter, on the other hand, always offered a steady hand that kept my more fanciful ideas grounded and offered expert advice on framing and improving our work. Both contributed to my personal and professional development and this dissertation would not have been possible without their guidance. It has been an honor to be advised by both of them.

Beyond my advisors, there are a number of other researchers who helped me on this journey. Matthew Lentz often liked to give life advice (thankfully not all of it followed) and our frequent conversations about culinary pursuits and wine were a welcome distraction. Deepak Garg was a frequent collaborator who brought a different perspective to our work, which I appreciate. Dave Levin, who I collaborated on a paper with and had many discussions with, was ceaselessly and infectiously optimistic about our capabilities and is a fountain of good advice. Neil Spring, who was a leader in our group, contributed to the rigor of our group and always had an incisive, interesting and thoughtful perspective when discussing our work or the work of others. I benefited greatly from these varied perspectives and am grateful to have had the opportunity to learn from all of them.

Additionally, I would like to thank my advisors, Neil, Dave, and Mark Shayman, who all served on my committee and improved this dissertation.

I would also like to thank the following colleagues for their role as collaborators and friends. In alphabetical order: Paarijaat Aditya, Mohamed Alzayat, Theophilus Benson, Björn Brandenburg, Frank Cangialosi, Eslam Elnikety, Neal Gupta, Stephen Herwig, Mike Hicks, Pete Keleher, Zhihao Li, Andrew Miller, Alan Mislove, Aastha Mehta, Ramakrishna Padmanabhan, Richie Roberts, Roberta de Vita, Anjo Vahldiek-Oberwagner, and Liang Zhang.

Finally, most importantly, I'd like to thank my northern star, my wife Sarah. When Bobby suggested this path, she didn't hesitate and endorsed my pursuit of the Ph.D., and despite my occasional doubts, she remained my biggest cheerleader and defender. In more ways than I could ever articulate, she is the best thing in my life.

This dissertation was supported by the National Science Foundation (TWC 1314857, NeTS 1526635, and CNS 1840902), the European Research Council (ERC Synergy imPACT 610150), and the German Science Foundation (DFG CRC1223).

This dissertation involved collaborative efforts with the following people:

Chapter 3: My co-authors for PTx were Deepak Garg, Peter Druschel, and Bobby Bhattacharjee. We are appreciative of the feedback we received from from Isaac Sheff, Roberta de Viti, Mohamed Alzayat, and Matthew Lentz.

Chapter 4: My co-authors for Light-Weight Contexts [1] were Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. We offer additional thanks to the following for their helpful feedback: Paarijaat Aditya, Björn Brandenburg, Mike Hicks, Pete Keleher, Matthew Lentz, Dave Levin, Neil Spring, anonymous reviewers, and KyoungSoo Park, who shepherded our conference submission.

Chapter 5: My co-authors for the Composing Abstractions using the null-Kernel [2] were Deepak Garg, Peter Druschel, and Bobby Bhattacharjee. We would like to thank the anonymous reviewers, Pete Keleher, Matt Lentz and Dave Levin.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

An operating system (OS) provides a set of abstractions through which access to hardware resources may be granted or multiplexed (virtualized). The design of this set of abstractions is informed by performance, isolation/security, portability, and ease of use constraints. These constraints are often at tension with one another. For instance, maximizing performance, whether it be throughput or latency, and simultaneously maximizing security is difficult if not impossible. Security is typically provided by isolating actors on the system and their data from one another, yet crossing isolation boundaries has a performance cost, which implies an inherent trade-off between performance and security. Similarly, portability, the feature whereby programs can run with little to no modification on different hardware, OS versions, or even different OSes that maintain compatibility, is also at odds with performance and, in some cases, security. Each significant branch of OS architecture discussed below effectively chooses abstractions that impose different priority orderings of these constraints and impose these trade-offs on users. The question we ask is not which OS architecture offers the best abstractions, but rather whether an OS ought to be constrained to a fixed set of abstractions at all.

OSes are often designed to favor one constraint over others. An OS that favors performance above all else will offer abstractions that give direct or near-direct access to hardware. At the extreme, no OS (or just the hardware interface) is the best OS for performance. The abstractions offered in this case (i.e., the hardware interfaces) impose no overhead or unnecessary abstractions that may inhibit performance, but suffer from three drawbacks: they are non-portable, offer little isolation, and rarely provide a virtualized hardware interface to enable resource sharing. OSes that favor isolation and security will limit the amount of software that runs with elevated privileges and separate as many components as possible into separate domains that may only interact through some supervisory process. In addition to security, isolation may also be used to improve the robustness of the system: the effects of a fault may more easily be contained to the isolated domain, which simplifies recovery. Kernels that prioritize isolation may or may not be portable, but isolation cannot be achieved without some performance cost. Finally, OSes that favor portability necessarily have higher level abstractions, as their abstractions must remain constant as hardware evolves and thus, tend to be represented by high-level models for system resources. This implies both a performance cost, due to mismatches between the hardware and its higher-level abstraction, as well as a cost to security, since the need to maintain abstractions indefinitely hampers improvements in security that may result from improving abstractions to fix deficits or adapt to new hardware and application demands. Exokernels, microkernels, and the monolithic architectures exemplified by UNIX and its descendants tend to prioritize, respectively, performance, isolation, and portability. We discuss these kernels next.

The Exokernel [3] architecture prioritize performance over all other factors.

The interface exokernels provide is often non-portable and very closely matches the hardware, which as argued by Clark et. al [4], enables the widest possible set of application specific optimizations. All other design goals are sacrificed, with ease of use, portability, and isolation sacrificed unless layered with a set of libraries that define high-level abstractions, termed a “library OS,” that applications rely on. At the limit, a library OS that prioritizes design goals other than performance will ultimately recapitulate design choices and trade-offs made by other kernel architectures.

Microkernels [5], on the other hand, preference isolation as the over-riding design principle. Liedtke explicitly argues that an abstraction should only be implemented within the kernel (i.e., have direct access to hardware) if the desired functionality cannot otherwise be achieved, leaving performance only as a secondary consideration [6]. Microkernels offer relatively high-level abstractions that are built as a set of userspace services that access hardware via interprocess communication (IPC) with a small privileged kernel. The isolation and high-level abstractions may hamper performance, but microkernels can be portable if the interface provided by the user space services is preserved. Note that user space services are analogous to library OSes, in that both are theoretically replaceable so long as portability is not a concern.

Monolithic kernels, best exemplified by the descendants of UNIX, favor portability. POSIX [7], a standardization of a common set of OS abstractions and basic utilities, specifies a set of abstractions supported by monolithic kernels that remain largely unchanged for over 25 years. These abstractions, which are meant to support a very broad set of hardware and application needs, tend to offer one-size-fits-all abstractions for hardware and applications. A cost of this portability is less flexibility in meeting new application demands that were not considered when the abstractions were standardized, as well as performance loss due to overhead from the OS abstractions that the applications must use. Monolithic kernels do make compromises to mitigate performance costs. They limit isolation, which they may do without affecting portability, which is a practice that leads to their name: almost the entirety of the kernel runs within a single monolithic address-space at increased privilege. Practical concerns do necessitate occasional violations of portability, such as near-direct access to new hardware through an exokernel like interface (e.g., an interface to an ASIC that accelerates machine learning calculations). These exceptions are ad-hoc and do not always interact well with the other abstractions offered by the OS

Each major OS architecture meets its design goals through a set of abstractions that reflect and impose onto applications the priorities of the OS. Applications that want to maximize performance may choose to run on an OS offering direct hardware access (e.g., an exokernel), whereas applications with strong security requirements might be best deployed on OS with stronger isolation (e.g., a microkernel).

New hardware and changing application requirements have given rise to applications with needs that cannot be met with any current OS architecture. Applications can subvert the restrictions of the OS while preserving compatibility, but traditionally this may only be done if the OS itself is modified. Our own work in developing new abstractions has suggested that the trade-off between constraints is not inherent and can be avoided, not by choosing a single ideal abstraction, but instead by simultaneously exposing multiple abstractions for each available resource.

1.1 Thesis

> *Supporting novel hardware such as NVRAM and new abstractions like fine-grained isolation while maintaining efficiency, usability, and security goals, requires simultaneous access to both high-level OS abstractions and compatible access to their low-level decompositions.*

Next I will describe the terms in my thesis statement and discuss how they affect OS design.

While there is no precise definition of low-level and high-level abstractions, for the purpose of this dissertation we use low-level abstractions to refer to abstractions that are more similar to the hardware than a higher level abstraction. We consider direct hardware access as a low-level abstraction,and consider all high-level abstractions to be composed of one or more low-level abstractions. High-level abstractions are built on top of one or more low-level abstractions and typically provide greater ease of use, increased portability, or stronger isolation guarantees, but may sacrifice performance. A decomposition of a higher-level abstraction is then the set of all lower-level abstractions upon which the higher-level abstraction is built. Simultaneous access to both a high-level abstraction and its lower-level decomposition thus implies that that a high-level abstraction may be exposed, but each of the lower-level abstractions from which it is built should also be exposed and accessible to applications.

In the context of the architectures we discussed above, we then describe exokernels as an OS with low-level abstractions and the monolithic kernels discussed above, on the other extreme, as an OS with high-level abstractions. Neither OS consistently exposes both high-level abstractions and their low-level decompositions. This implies that exokernels may be arbitrarily adaptable, but at the cost of usability due to significant implementation effort and loss of portability. Whereas the monolithic kernels and their high-level abstractions give us portability, but at the cost of efficiency that the end-to-end principle suggests is inherent with high-level abstractions. An OS that exported both low and high-level abstractions accommodates a wider set of applications, but care must be taken to do this safely.

The difficulty in exposing low and high-level abstractions simultaneously is one of interference. With few exceptions, all abstractions have a set of invariants that must be maintained for the abstraction to offer a useful contract with its callers. If multiple high-level abstractions are built on top of the same low-level abstraction (e.g., two separate file systems using the same block device) we need some mechanism to prevent interference between different high-level abstractions. A traditional OS achieves this by specifying how abstractions may be used a priori and requiring a supervisory process (i.e., the kernel) to enforce usage. A priori specifications cannot possibly meet the needs of all applications and must have embedded within them their own design trade-offs.

1.2 Contributions

As part of the work in evaluating the abstractions necessary to accommodate new hardware and changing application demands, we developed two new abstractions, PTx (Chapter 3) and Light-Weight Contexts (Chapter 4. We also introduced a new OS architecture, the null-Kernel (Chapter 5). Both PTx and Light-Weight Contexts (*lwC*s) were designed and implemented on production operating systems (FreeBSD and Linux respectively) and required access to traditional high-level OS abstractions, as well as access to non-traditional low-level abstractions to meet their performance goals. Building on our experiences in designing and implementing *lwC*s and PTx, we propose the null-Kernel architecture, which is an architecture that enables the simultaneous exposure of multiple abstractions for the same underlying resources in a safe way. Under the null-Kernel, the development of new abstractions such as *lwC*s and PTx would be accelerated, improving application performance and security. The dissertation is structured as follows:

Chapter 2: Background and Related Work

We contextualize PTx and *lwC*s in relation to other work that provides similar functionality. In addition to discussing the abstractions and capabilities of existing systems, we note if and when related work relies on or provides different lower-level OS abstractions than PTx and *lwC*s rely upon. We also discuss common OS architectures and kernel extension mechanisms and discuss the problems these systems are meant to solve and discuss the trade-offs inherent in each of these architectures.

Chapter 3: PTx

We describe PTx, which uses a new form of non-volatile memory (NVRAM) to efficiently persist in-core data structures to persistent media. We discuss the consistency and performance challenges one faces when using NVRAM, and how PTx solves these challenges. We evaluate PTx and show that it enables high performance persistence of standard C++ data structures that exceed the performance of data structures explicitly annotated for NVRAM. We also show that PTx can be used to provide persistence to Redis [8], a popular key-value server, with comparable overhead to custom solutions and minimal modification, or provide a compatible server with superior performance within 430 lines of source code.

Chapter 4: Light-weight Contexts

We describe Light-weight contexts, which decouples memory isolation, execution state, and privilege separation from within a process. *lwC*s can be used to provide snapshots, session isolation, reference monitoring, and protected compartments within a process. We evaluate *lwC*s with a series of micro-benchmarks and application scenarios and show that *lwC*s can provide enhanced security with low overhead or improve performance with its snapshot facility.

Chapter 5: The null-Kernel

PTx and *lwC*s both showed that existing OS abstractions were insufficient to deal with new hardware and increased security demands. We found that both systems could be implemented by exposing and then making use of lower-level abstractions, while portability concerns dictated that we preserve existing higher-level abstractions. To generalize the constraints that we observed while building these systems, we developed the null-Kernel, a model for describing existing OS architectures and suggest new OS paradigms to support the simultaneous exposure of low-level and high-level abstractions. We discuss how simultaneous exposure of abstractions provides new opportunities for improving performance and security and place PTx and *lwC*s in the null-Kernel context.

Chapter 6: Conclusion and Future Work

We conclude by revisiting the contributions of this work. We discuss future opportunities for improving PTx, both in terms of optimizations, as well as extensions to functionality. We also discuss combining PTx and *lwC*s to create a form of persistent *lwC*. Finally, we discuss the steps we have taken towards exposing lower-level abstractions within FreeBSD to userspace and thus, bring FreeBSD closer in line to a null-Kernel architecture.

Chapter 2: Background and Related Work

In this chapter we discuss related work for persisting state to NVRAM, decoupling process primitives, such as isolation, and improving OS flexibility to hasten the development of new abstractions for differing application requirements and new hardware.

2.1 Abstractions for NVRAM

The price/performance characteristics of NVRAM make it an attractive new point in the storage hierarchy. Currently available NVRAM can be operated in one of two modes: *memory* and *direct*. In memory-mode, conventional DRAM is used as a cache for data stored in NVRAM, affording several TB of main memory at reasonable cost with performance close to DRAM for workloads with good locality. In this mode, NVRAM is used for its byte addressability and low cost per byte; the memory controller actively defeats persistence in this mode for security reasons, by encrypting data and destroying the keys during a system restart.

In direct mode, NVRAM appears directly in the system's physical address space and can be accessed with conventional load and store operations, albeit at reduced performance compared to DRAM. Note that in this mode, once a NVRAM page is mapped, loads and stores can be completed without OS intervention (i.e., without any abstraction overhead). Current commercially available NVRAM has higher latency (3.7x slower) and lower bandwidth when the bus is saturated (1/3 and 1/6 of DRAM read and write bandwidth respectively) [9].

Operating systems expose direct-mode NVRAM in one of two configurations. In the first configuration, the OS wraps NVRAM in a block device abstraction, which is then accessed through a filesystem interface. Applications access the data through the usual filesystem API, which has no bearing on the application other than increased performance. In the second configuration, which PTx relies on, the OS maps NVRAM directly into an application's address space through the mmap interface and applications modify persistent state through memory operations. This latter form is a lower-level interface, which allows for higher performance.

Directly-mapped NVRAM by itself, however, does not provide immediate nor atomic persistence. NVRAM accesses are subject to the same caching layers as DRAM. Writes to mapped NVRAM are not automatically flushed to NVRAM and thus, not persisted, until evicted from the CPU caches. A process may evict (and thus persist) cache lines as needed, or applications may explicitly push writes to NVRAM either via flush and fence instructions or via special instructions that bypass the cache. In either case, temporally proximate and spatially contiguous writes are combined by the hardware and written to NVRAM with an effective block size of 256 bytes. Regardless of the method chosen, persistence must be programmed carefully to ensure that the persisted structure is in a consistent state, and thus fault-tolerant, at all times, and that the NVRAM accesses are efficient. Failure to do so can lead to very subtle, hard-to-detect and hard-to-recover-from bugs.

While an application can use the lower level direct-mapped NVRAM directly to implement persistent data structures in principle, doing so is challenging both in terms of performance and correctness. The application needs to carefully manage data access locality and write amplification for performance, as well as use explicit barriers and cache flush instructions carefully throughout. Failure to use these instructions appropriately may result in inconsistent persistent states where the program fails in specific states, which is very difficult to debug. Our work will offer a higher-level abstraction that is different from the file system interface and reliant on access to lower-level abstractions.

In the rest of this section, we describe prior work on using NVRAM, both as a filesystem and for providing persistence. We also discuss existing solutions for persistence, and compare them to PTx.

NVRAM file systems Several file systems take advantage of the performance characteristics and byte addressability of NVRAM [10–12]. Just as file systems act as a namespace to provide a handle to disk resources, an NVRAM file system can be used to label NVRAM resources that can be mapped into the process address space. The file system interface is a familiar one for programmers seeking persistence, but it does impose overhead. At minimum, reading and writing to an NVRAM file require system calls that copy data from NVRAM to and from DRAM.

Both NOVA-Fortis [10] and PMFS [11] allow applications to map the NVRAM data pages for a file directly into the application's address space. This feature, which the Linux EXT and XFS file systems also provide, is known as "direct-access," (DAX). DAX allows applications to modify file contents directly through CPU load and store directions, and as such grant lower-level access to the hardware. NVRAM file systems that do not support or are not DAX mapped support memory mapping like traditional file systems: with a buffer cache. The OS pages data onto DRAM from NVRAM on demand and writes all modified DRAM pages to NVRAM either opportunistically or when `msync` is called. Implementing persistent data structures through file system mappings shares the challenges described above. PTx, which offers an alternate high-level abstraction, relies on DAX mappings to read and write to NVRAM internally but does not expose the mappings directly to applications.

NVRAM aware data structures Many prior systems have provided persistence using bespoke NVRAM aware data structures [13], such as customized b+-trees [14, 15], radix trees [16], key value stores [17, 18], hash tables [19, 20], and write-ahead logging [21, 22]. PTx instead enables efficient persistence for legacy data structure implementations without annotation or modification to the data structure source code.

Transactional Memory and Semantics Transactional memory was introduced as a method of concurrency control. In this context, serializability and isolation between threads are key requirements, while durability is irrelevant because transactions are performed on volatile DRAM. Similarly, transactions for concurrency control tend to be short to enable fine-grained concurrency (e.g., add an item to a data structure).

More recently, transactional memory has been used as an abstraction for atomic updates to persistent memory, such as NVRAM. Here, atomicity with respect to failures is the key property: updates to persistent state must be applied in their entirely or not at all. Isolation between threads for concurrency control may or may not be provided. In this context, transactions tend to be larger as they may include updates to multiple data structures that must be applied atomically to maintain invariants application-wide, not just within a single data structure

Many persistent memory systems use a transaction abstraction [23–30] or an *atomic* keyword [31] to delineate state changes that must be persisted atomically. Some of these systems operate much like transactional memory systems for concurrency control, and provide concurrent consistency [23, 25, 27], whereas others [26, 28, 31], like PTx, expect applications to use external concurrency primitives for thread isolation within a process.

Transactional memory systems implement crash atomicity using logging. Systems that use undo logging [25–27] copy to-be-modified state to persistent memory before allowing modifications to occur. Applications then make modifications in place (i.e., directly in NVRAM) within a transaction. The undo log is only used to recover from a fault. Other systems use redo logging [31] and write updates to a redo log, to be applied to the main persistent store upon commit. Within a transaction, reads are redirected so that they read the proper values from the redo log. Pronto [32] wraps operations on data structures and asynchronously logs operations and their arguments that may be replayed against the data structure API in the event of a failure.

Intel's persistent memory toolkit (PMDK) [26] provides a transactional interface that uses a combination of undo and redo logs to provide atomicity. Like PTx, PMDK does not provide isolation between threads, but unlike PTx, PMDK writes directly to NVRAM and requires programmer annotations to indicate the write set.

DudeTM [23] attempts to limit both the overhead caused by modifying NVRAM within a transaction, as well as the overhead incurred from the redo log indirection. Towards this end, DudeTM performs modifications on shadow DRAM pages and writes a redo log in volatile memory. As part of commit, the volatile redo log is flushed to NVRAM and subsequently, the persistent redo log is applied to the persistent data. PTx also makes all modifications to volatile shadow pages, but does not maintain an explicit redo or undo log. Instead, the PTx log structure enables writing a comparatively succinct undo log as a set of updates to the mapping between DRAM and NVRAM addresses.

Ni et al. [28] propose a design that also eliminates explicit undo logs, by atomically updating mapping information upon commit, but their approach relies on hardware modifications. Hu et al. [33] also use a log structure for data in NVRAM. New allocations are appended to the log and reads and writes are intercepted and redirected so that they act directly on the log. PTx does not require interception of reads and writes, and PTx's log does not require periodic cleaning due to fragmentation. Correia et al. [34] present a persistent transactional memory system based on universal constructions. Their work eliminates blocking transactions, but either requires code annotation or flushing all the memory where the data structure resides upon commit. This over-approximation of the write set is prohibitive for transactions over large data structures.

2.2 Decoupling Process Abstractions

Privacy-compromising exploits, such as Heartbleed [35], suggest that existing isolation abstractions, such as processes, either do not meet current isolation requirements, or do not meet them with sufficient expressiveness of performance. *lwC*s take the process abstraction, and decouples the isolation, scheduling, and privilege properties traditionally provided by a process and uses them to offer new abstractions that provides for strong, finer-grained isolation with minimal performance overhead, as well as snapshots and reference monitoring. Other systems also revisit lower-level aspects of the process abstraction and put them to new use, which we discuss now.

Wedge [36] provides privilege separation and isolation among *sthreads*, which are a new unit of encapsulation for an application. The program is split up into a series of sthreads, which by default share little to no state, and regions of computation gated by callgates. Each callgate is associated with an sthread, and whenever another sthread attempts to invoke a callgate (i.e., enter a protected region of computation), the caller thread is blocked and the sthread associated with the callgate is scheduled to execute the gated code, potentially while accessing protected state, and return a value back to the calling thread. Unlike *lwC*s, scheduling and isolation are still coupled in Wedge, but sthreads are a finer unit of isolation than a process and have a similar goal of protecting sensitive parts of a program's execution. *lwC*s are orthogonal to threads and therefore avoid the cost of scheduling when switching contexts. Moreover, *lwC*s can snapshot and resume an execution in any state, while a sthread can only revert to its initial state. Wedge provides a software analysis tool that helps refactor existing applications into multiple isolated compartments. *lwC*s could benefit from a such a tool as well.

Shreds [37] builds on architectural support for *memory domains* in ARM CPUs, a compiler toolchain, and kernel support to provide isolated compartments of code and data within a process. Like *lwC*s, shreds provide isolated contexts within a process. *lwC*s, however, are fully independent of threads, require no compiler support, and rely on page-based hardware protection only. *lwC*s also provide protection rings and snapshots, which shreds do not.

In SpaceJMP [38], address spaces are first-class objects separate from processes, which demonstrates broader utility for lower-level address space abstractions that are decoupled from the process. While both systems can switch address spaces within a process, SpaceJMP and *lwC*s provide different abstractions, capabilities, and are motivated by entirely different applications. SpaceJMP supports applications that wish to use memory larger than the available virtual address bits allow, wish to maintain pointer-based data structures beyond process lifetime, and share large memory objects among processes. A SpaceJMP context switch is not associated with a mandatory control transfer and, therefore, SpaceJMP does not support applications that require isolation or privilege separation within a process. *lwC*s, on the other hand, provide in-process isolated contexts, privilege separation, and snapshots. It should be noted, however, that the SpaceJMP address space abstraction is a low-level abstraction that is similar to the address space abstraction proposed in Section 5.5 for supporting new isolation abstractions in the null-Kernel, and in concert with other abstractions for managing control flow, could be used to provide isolation.

In Trellis [39], code annotations, a compiler, run time, and OS kernel module provide privilege separation within an application. The kernel and runtime ensure that functions can be called and data accessed only by code with the same or higher privilege level. *lwC*s provide privilege separation without language/compiler support, and can switch domains at lower cost. Moreover, *lwC*s additionally support snapshots.

Switching among *lwC*s is similar to migrating threads in Mach [40], where they were implemented to optimize local RPCs. Migration of threads across address spaces is also an element of the model described by Lindström et al. [41] and the COMPOSITE OS [42]. In single address space operating systems (SASOS) like Opal [43] and Mungi [44], all processes as well as persistent storage share a single large (64-bit) address space. Unlike *lwC*s, these systems do not provide privilege separation, isolation, or snapshots *within* a process.

Mondrian Memory Protection (MMP) [45] and Mondrix [46] use hardware extensions to provide protection at fine granularity within processes. The CHERI [47, 48] architecture, compiler, and operating system provides hybrid hardware-software object capabilities for fine-grained compartmentalization within a process. *lwC*s provide in-process isolation at page granularity without specialized hardware or language support.

Resource containers [49] separate the unit of resource accounting from a process and account for all resources associated with an application activity, even if the activity requires processing in multiple processes and the kernel. Resource containers decompose task accounting from processes. *lwC*s are orthogonal to resource containers and as such, do not make use of accounting decomposition.

The Corey [50] OS provides fine-grained control over the sharing of memory regions and kernel resources among CPU cores to minimize contention. These lower-level abstractions are orthogonal to the capabilities of in-process isolation, privilege separation, and snapshots provided by *lwC*s.

Light-weight isolation, privilege separation, and snapshots can be provided also within a programming language. Functional languages like Scheme and ML provide closures through the primitive call/cc, which can be used to record a program state and revert to it later, and to implement co-routines. Typed object-oriented languages like C++ and Java provide *static* isolation and privilege separation through private and protected class fields but do not isolate objects of the same class from each other. *Dynamic* language-based protection, often implemented as object capabilities [51–53], provides fine-grained isolation and privilege separation but has considerable runtime overhead. *lwC*s instead provide in-process isolation, privilege separation, and snapshots at the OS level, independent of a programming language.

In low-level languages like C, isolation and privilege separation can be attained using binary rewriting and compiler-inserted checks as in CFI [54], CPI [55] and secure compilation [56]. All these techniques rely on dynamic checks that have runtime overhead. Techniques such as CPI and secure compilation rely on OS support for the isolation of a reference monitor, which *lwC*s can provide at low cost. ERIM [57] uses binary inspection in concert with Intel's memory protection keys to provide memory isolation between contexts with minimal overhead. While ERIM provides similar virtual memory isolation, it does not provide isolation for file tables or privilege.

Software fault isolation (SFI) [58] and NaCl [59] rely on static checking and instrumentation of binaries to isolate memory within applications running on unmodified operating systems. SFI and NaCl do not selectively protect system calls and file descriptors. *lwC*s instead allow fine-grained control over memory, file descriptors and other process credentials, and provide snapshots as part of an OS abstraction.

Process checkpoint facilities create a linearized snapshot of a process's state [60–63]. The snapshot can be stored persistently and subsequently used to reconstitute the process and resume its execution on the same or a different machine. Checkpoint facilities are used for fault-tolerance and load balancing. *lwC*s instead provide very fast in-memory snapshots of a process's state.

The Determinator OS [64] relies on a private workspace model for concurrency control, which enables deterministic execution on multi-core platforms. To support the model, Determinator provides *spaces*, in which programs mutate private copies of shared objects. Like *lwC*s, spaces provide isolated address spaces. Unlike a *lwC*, however, a space is tied to one thread, does not have access to I/O or shared memory, and can interact only with its parent and only in limited ways.

Intel's Software Guard Extensions (SGX) [65] provide ISA support to isolate code and data in *enclaves* within a process. By mapping contexts to enclaves, SGX could be used to harden *lwC*s against a stronger threat model (untrusted OS) and to provide hardware attestation of contexts. However, enclaves have no access to OS services, so some *lwC* applications would need considerable re-architecting to run on SGX.

NOVA [66] provides protection domains (separate address spaces) and execution contexts (an abstraction similar to threads) in a micro hypervisor. NOVA's goal is to isolate VMMs and VMs from the core hypervisor, which is different from *lwC*'s goal of providing isolation, privilege separation, and snapshots within processes.

Dune [67] provides a kernel module and API that export the Intel VT-x architectural virtualization support safely to Linux processes. This is akin to a form of layer bypassing supported by null-Kernel that allows low-level access to hardware, but this access is virtualized. The low-level interface granted is not a decomposition of abstractions used by higher-level abstractions, but is instead disjoint from the higher-level abstractions. Consequently, while privilege separation, reference monitors, and isolated compartments can be implemented within a process using Dune, these abstractions cannot be seamlessly integrated with the kernel's existing abstractions. *lwC*s, by contrast, instead provide a unified abstraction and API for these capabilities, and their implementation does not rely on virtualization hardware, the use of which could interfere with execution on a virtualized platform.

2.3 Improving OS Flexibility

VINO [68] and SPIN [69] offer mechanisms to safely extend monolithic kernels. Both systems require extensions to be written against a restricted, internal interface that maintains kernel invariants. These systems can be thought of as a limited instantiation of the hybrid system presented earlier, but access to the internal interface cannot be shared in a structured manner. This limits how extensions (AMs) relate:for instance, layer bypassing is not possible in either.

Microkernels [6] seL4 [70] and Barrelfish [71] export kernel objects to user space as capabilities. Capability types exported by the system are static. As a result, layer bypassing via delegated capabilities is not supported.

Exokernels [3] provide a minimal, non-portable hardware-like interface. Exokernel abstractions allow for the allocation and revocation of hardware resources in a manner similar to capability allocation, but unlike capabilities, these resources cannot be shared or reduced except by proxying through the resource owner.

EROS [72], derived from KeyKOS [73], is a stateless kernel that maps hardware into a set of capabilities. Applications use the operations permitted by these capabilities to construct higher level abstractions. EROS is equivalent to a specific instantiation of the null-Kernel that only exports a low level AM. HiStar [74] also exposes a limited set of kernel objects to user space, limiting access to those objects by tracking information flow.

The Cal timesharing system [75], Cambridge CAP computer [76] and Fluke [77] all allow an interface's operations to be implemented and over-ridden in a nested manner that is similar to subclasses. This layering is constrained by capabilities. Unlike the null-Kernel, the interface for these interfaces is fixed.

Chapter 3: PTx

In this chapter we discuss PTx, a new abstraction for efficiently persisting in-core data structures to a new form of byte addressable and persistent non-volatile RAM (NVRAM).

3.1 Introduction

Non-volatile RAM (NVRAM) is a newly available memory technology that may have profound impact on both system hardware and software structure. The reason for this impending shift is that NVRAM has a cost/byte and speed between DRAM and Flash memory, is byte-addressable via unprivileged CPU instructions, and *persistent*. Thus NVRAM promises to combine the best features of DRAM (directly addresseable/memory mapped, performance) with those of disk/solid state memory (persistence, relatively low cost per byte).

We focus on the potential for NVRAM to enable persistent data structures. For example, an in-memory database could persist on NVRAM, obviating the need to save modified data on an external storage device for persistence. As a result, NVRAM has the potential to significantly reduce the cost of persisting state, enabling higher performance for a given granularity of transactions, or enabling more fine-grained transactions at a given level of performance.

There are two primary challenges when using NVRAM for persisting process state. First, performance: when saturated, NVRAM writes are ~6x slower than DRAM writes [9], which generally precludes using NVRAM as a direct replacement for DRAM. Instead, NVRAM is used as a backing store for DRAM and writes to NVRAM must be minimized for efficiency. Minimizing extraneous NVRAM writes is simple for some specific data structures, such as an append-only log. In general, however, it requires appropriate techniques for determining which parts of a process's state need to be persisted and which parts of that state were modified as part of a transaction. Performance also requires a design that minimizes NVRAM write amplification, i.e., modified data should have to be written only once per commit in the common case.

The second challenge is consistency: Processes may crash at any point in their execution, and even if all memory writes are saved, the restored state may not be consistent if the failure occurs when invariants on the application's state don't hold. For instance, an application-level transaction may involve modifications to multiple data structures that need to be performed atomically. Restored state must satisfy not only the individual data structures' invariants, but also invariants across the application state. Moreover, without additional information about the failed operation, it may be impossible to return to a consistent state without loss of information.

We introduce PTx, a userspace persistence library specialized for the performance and atomicity needs to persist data structures to NVRAM. PTx enables programs to achieve failure atomicity for arbitrary code sequences simply by bracketing them with primitives to begin and end a transaction. No further annotation is required on code executing inside a transaction; in particular, existing unmodified data structure libraries can be invoked as part of a transaction, thus transparently persisting these data structures. While executing inside a transaction, the program can access data at DRAM speed. When ending the transactions, PTx persists the transaction atomically while minimizing the number of NVRAM writes required.

As discussed in Section 2.1, state-of-the-art, high-performance persistent data structures require manual annotation of individual memory writes, so that these changes can be written to persistent storage efficiently. We introduce several high performance automatic change trackers, which relieve the programmer from having to annotate source code to track changes. PTx trackers are akin to existing compiler-based mechanisms, such as those developed for Software Transactional Memory [78, 79], but provide much higher performance.

Because data accesses within a PTx transaction proceed at DRAM speed, existing data structures not designed for the reduced access speed of NVRAM may be used without penalty and subsequently persisted efficiently. Finally, PTx persists data in NVRAM using a data structure that supports non-destructive writes, avoiding unnecessary and costly NVRAM writes and the need for log cleaning/compaction. The combination of automatic, language independent write-set tracking, DRAM-speed data access within a transaction, and low NVRAM write amplification enables PTx to efficiently persist existing data structure implementations, without modifying their source code and even when transactions are large.

We experiment with persisting the standard C++ STL data structures [80]. With PTx automatic tracking, these structures can be made transparently persistent, and perform close to their native DRAM speeds. This allows PTx to match or exceed the query throughput of custom, hand optimized systems, such as LMDB, Redis, pmem-Redis, and previously developed NVRAM data structures, all of which require manual annotations to track changes.

This chapter is organized as follows: we provide a technical background on NVRAM and discuss related work in Section 2.1. The design of PTx, including its persistent data structures and algorithms is presented in Section 3.2. We discuss implementation specifics in Section 3.3. Section 3.3 contains a comprehensive evaluation of PTx and a comparison to several other persistence libraries and applications.

3.2 PTx design

3.2.1 Requirements

PTx aims to enable persistent in-core data structures for *existing* systems and applications, without requiring extensive changes to code. An application should be able to link to the PTx library, map persistent memory objects into the application's address space using PTx's API, use the PTx memory allocator to allocate additional persistent memory dynamically. The programmer should be able to bracket sequences of operations on state that should be persisted atomically with transaction start and commit primitives as required. Any existing persistence mechanism within the application can be disabled. (Such a mechanism typically serializes and explicitly writes its persistent state to an external storage device, or invokes sync operations on memory-mapped files at appropriate points in its execution.) Due to the greater efficiency afforded by PTx and NVRAM, the application can benefit from increased performance, or persist its state more frequently. To realize this vision, PTx has the following design constraints:

DRAM-speed data access: Existing data structure implementations designed for DRAM may perform poorly on NVRAM. For instance, fine-grained writes get amplified to 256-byte block writes. Therefore, accesses within a transaction should be performed on DRAM.

Automatic write set tracking: Since existing data structure implementations don't explicitly state their write sets, we need to rely on automatic techniques to determine them. These techniques must reliably capture all modified state for correctness, without significantly over-approximating the write set.

Low write amplification: Applications are typically interested in persisting states that correspond to completed application-level transactions, not mutations of individual data structures. Therefore, transactional updates to persistent state are typically larger than memory transactions used for concurrency control. Efficiently supporting such transactions requires low amplification of NVRAM writes, i.e., modified data should be written only once per commit and at the 256 byte granularity of NVRAM.

3.2.2 PTx colors and operations

PTx supports multiple memory pools called *colors*. Each color is backed by an NVRAM file with separate access permissions. An application can map multiple colors for which it holds permissions into its address space. Each color has a separate dynamic memory allocator. PTx transactions operate on a single color, allowing applications to persist data of any one color atomically. Applications may also allocate memory in different colors and transact on them separately.

The PTx library exports four operations that each operate on a color *c*: *ptx_malloc (c)* allocates persistent memory associated with a particular color. *ptx_begin (c)* starts a transaction on a color, *ptx_commit (c)* commits a transaction, and *ptx_restore (c)* rolls back the color to the last committed state. Note that *ptx_restore (c)* can be used at the start of a program execution to reinstate the last committed state of a color, or during execution to abort an uncommitted transaction on a color. We call the data of a given color persisted via a commit a *snapshot*. Later instantiations of the application or other applications may restore the latest snapshot of a given color into their address space by calling *ptx_restore()*.

At runtime, PTx stores persistent data in main memory, which the application may read or write at DRAM speed. The "write set" represents the set of locations within a color that were modified within a transaction. PTx uses "trackers" to determine the write set, as described in Section 3.2.6. When the application invokes *ptx_commit()*, the write set is written to NVRAM atomically.

At any time, the application, a later instantiation of the same application, or a different application with appropriate permissions may restore the last committed state of a memory pool into its address space using *ptx_restore (c)*.

3.2.3 PTx Semantics

PTx transactions are committed atomically with respect to failures of applications and systems. From an application's perspective, a persistent memory pool ("color") is either updated entirely or not at all as part of a commit. Critically, this implies that as long as a memory pool's invariants hold at the time of a `commit` call, those invariants will hold after a system fault/recovery. An application may persist multiple data structures by allocating them using the same color, and different processes may simultaneously map the same color into their address spaces.[1]

PTx provides atomicity, durability, and consistency/isolation between processes (but not threads). The semantics provided by PTx are similar to ACID transactions at the level of processes. However, PTx transactions do not provide concurrency control among different threads of a process. Such synchronization normally occurs at a different granularity as PTx transactions and doesn't require atomic persistence. To synchronize among threads of the same process, an application must use a separate mechanism, such as locks or conventional transactional memory.

[1]Our PTx prototype does not currently support concurrent mappings of a color by multiple processes.

3.2.4 PTx storage

Next we describe the storage components of PTx's design, namely the *data layer*, the *log*, and the *map*.

PTx data layer: The data layer stores persistent data and associated metadata in NVRAM. It must do so efficiently and maintain the ability to recover in the event of system failures. Towards this end, the store performs non-destructive data writes and minimizes write-amplification by allowing modified data to be written only once per transaction.

The data layer is organized as an array of fixed sized blocksets in NVRAM. Each blockset contains space for 41 data blocks of 256 bytes each, plus an additional 256 bytes for metadata. During a commit, PTx writes modified data into available data blocks in the data layer. The unit of allocation is a data block, although PTx seeks to allocate entire blocksets when possible. Note that the blocks of a data structure are typically stored non-contiguously in the data layer. Because NVRAM can sustain random block accesses with little or no performance degradation, there is no need for cleaning or compaction, which would create overhead and increase write amplification.

PTx log: A small, persistent circular log is used to support atomic transactions whose write set does not fit into a single blockset. The log holds metadata about a transaction during a commit; specifically, the transaction's sequence number, and the set of data blocks written as part of the transaction, followed by a hash. In case of a system failure before the hash is written, the information in the undo log is used to free uncommitted data blocks in the data layer and add them to the free list.

PTx map: The map associates virtual addresses of data blocks in an application's address space with addresses in the data layer, where the last committed state of the block is stored. In the NVRAM, the map is distributed over the metadata blocks of the data layer. During execution, a copy of the map, organized as a range tree for efficiency, resides in main memory for efficiency. During a system restart, the in-memory range tree is reconstituted from metadata in the data layer.

PTx free list: The free list is an in-memory data structure that indicates which blocks in the data layer are free. Like the in-memory map, it is reconstituted during a system (re-)start from the data layer metadata.

3.2.5 PTx primitives

Function	Description
BlockLocation ←getFreeBlock()	returns free BlockLocation
⊥ ←freeBlock(BlockLocation)	sets BlockLocation to be empty
⊥ ←copyToBlock(VAddr, BlockLocation)	copies Data from VAddr into BlockLocation
⊥ ←copyToVAddr(BlockLocation, VAddr)	copies Data from BlockLocation into VAddr
⊥ Object.persist()	Flushes calling object to NVRAM

Table 3.1: Auxillary functions used in PTx

Next, we describe the PTx primitives. Basic data types and global structures are defined in Algorithm 1, auxiliary functions are defined in Table 3.1. Here, we assume that a write of an individual data block to the NVRAM (*Object.persist()* in Table 3.1) is atomic with respect to system failures. We discuss in Section 3.2.7

Algorithm 1 PTx Basic Data Types and Global Structures

```
 1: type BlockSet
 2:    Header = {seqNum, VAddr[41]}                          ▷ Seq. #, 41 VAddrs
 3:    BlockData = Data[41]                                  ▷ 41 Data items
 4:
 5: type BlockLocation
 6:    blockID                                               ▷ ID of BlockSet in NVRAM
 7:    blockIndex                                            ▷ index within BlockSet
 8:
 9: type UndoLogEntry
10:    Locations = ¡BlockLocation¿                           ▷ sequence of BlockLocations
11:    SequenceNum                                           ▷ sequence number of commit
12:    Hash                                                  ▷ Hash of commit
13:
14: type CommitEntry
15:    VAddr                                                 ▷ Virtual Address
16:    Data                                                  ▷ Data (256 bytes)
17:
18:
19: Global Structures
20: BlockStore                          ▷ Sequence of BlockSets; stored in NVRAM
21: UndoLog                             ▷ Seq. of UndoLogEntry(ies); stored in NVRAM
22:
23: WriteSet                            ▷ Seq. of CommitEntries; generated by App./tracker
24: InMemoryMap                         ▷ DRAM map of VAddr → BlockLocation
25: SeqNum                              ▷ In-memory copy of last Sequence Number
```

how to generalize the design in case the NVRAM controller does not provide this guarantee.

ptx_commit: The `commit` operation atomically persists the write set of the current transaction on a given color. The pseudocode for `commit` is shown in Algorithm 2.

(i) The transaction's write set is determined using one of the methods described in Section 3.2.6 and passed as an argument to `commit`.

(ii) We write all data in the transaction's write set into currently unused blocks in the data layer, relying on the free list to identify such blocks. To ensure Invariant 1 (see below), we choose blocks such that the current and any previous version of

Algorithm 2 PTx `commit` Schematic

```
 1: function WRITEDATA(WriteSet C)
 2:     bMap ←{}                              ▷ temp. map from BlockLocation → VAddr
 3:     u ←UndoLogEntry.new                   ▷ new empty UndoLog entry
 4:     UndoLog ≪u                            ▷ Append to undoLog
 5:     Cc                                    ▷ iterate over each WriteSet entry
 6:         b ←getFreeBlock
 7:         copyToBlock(c.Vaddr, b)
 8:         u.Locations ≪b
 9:         bMap[b] ←c.Vaddr
10:         b.persist                         ▷ write data to NVRAM
11:     return bMap
12: function UPDATEUNDOLOG
13:     u ←UndoLog.last
14:     u.SequenceNumber ←SeqNum
15:     u.Hash ←Hash(u.Locations)
16:     u.persist                             ▷ flush the undoLog entry to NVRAM
17: function UPDATEMAP(bMap)
18:     U ←UndoLog.last
19:     U.Locationu                           ▷ for BlockLocation in the last undoLog
20:         b ←BlockStore[u.blockID]
21:         vAddr = bMap[b]
22:         b.Header.seqNum ←U.SeqNum
23:         b.Header[blockIndex].VAddr ←vAddr
24:         b.persist
25:         InMemoryMap[vAddr] ←u
26: function COMMIT(WriteSet C)
27:     SeqNum++
28:     bMap ←writeData(C)
29:     updateUndoLog()
30:     updateMap(bMap)
```

Algorithm 3 PTx `restore` Schematic

```
function verifyUndoLog
    u ←UndoLog.last
    return u ? Hash(u.Locations) = u.Hash : true
function verifyLastSync
    u ←UndoLog.last
    u.Locationse
        b ←BlockStore[e.blockID]
        return false if (b.Header.seqNum ≠ u.SequenceNum)
    return true
function rollBack
    u ←UndoLog.last
    u.Locationsb
        freeBlock(b)
        b.persist
    UndoLog.discardLast                    ▷ remove last entry from UndoLog
function restoreDRAM
    for b $\in_{seq}$ BlockStore
                    ▷ traverse by BlockSet sequence numbers, highest first
    b.Header.VAddrvaddr
                    ▷ iterate over virtual addresses in header
        continue if vaddr = $\perp$
        continue if InMemoryMap[vaddr]                    ▷ already restored
        copyToVAddr(vaddr, b.BlockData[vaddr.index])
                    ▷ vaddr.index is index of vaddr in Header.VAddr
        InMemoryMap[vaddr] ←{b.index, vaddr.index}
                    ▷ b.index is index of block b in BlockStore
function Restore
    if ¬ verifyUndoLog
        UndoLog.discardLast
    elsif ¬ verifyLastSync
        rollBack

    SeqNum ←UndoLog.last ? UndoLog.last.SequenceNum : 0
    restoreDRAM
```

the same memory location don't end up in the same blockset. Whenever possible, we use as few data layer blocksets with as many free blocks as possible to reduce overhead and write amplification. At this point, the write set is in NVRAM but the associated blocks are considered unused and their content ignored in case of a failure.

(iii) We append to the undo log the transaction's sequence number, followed by the list of data layer blocks that were written in step (ii).

(iv) We rewrite the headers of blocksets that contain newly written data blocks. The header of each blockset is modified in three ways. First, the sequence number is set to the current transaction's. Second, we inspect all previously allocated blocks in the set to see if they have been superseded by a committed transaction, as indicated by the in-memory map, and then we set the address fields of all obsolete blocks to $\perp$. This ensures that the most recently committed value of a data block resides in a blockset with a higher sequence number than any previous version of that block. Third, the address fields of newly written blocks are changed from $\perp$ to their virtual offset, thereby allocating the blocks and associating them with an offset in the application's address space. At this point, the write set of the uncommitted transaction is allocated in the data layer. This is safe, because the undo log has the information required to deallocate the blocks in case of a failure before the transaction commits.

(v) We append a hash of all information related to the transaction to the undo log and flush the log entry to NVRAM. At this point, the transaction is committed.

(vi) We update the in-memory map and free list with the newly allocated blocks and their mappings.

The implementation of `commit` maintains

Invariant 1: When traversing blocksets in the data store in decreasing order of sequence numbers, the first occurrences of a block's virtual address represents the most recent version of the block (if it was ever written).

System shutdown: During an orderly system shutdown, PTx writes the in-memory map and free list to NVRAM, followed by the latest transaction sequence number and hash.

`ptx_restore`: The `restore` operation executes upon every system restart. The pseudocode for `restore` is shown in Algorithm 3.

(i) We check if the restart follows an orderly system shutdown, indicated by the presence of complete, consistent copies of the map and free list in NVRAM. If so, it loads the map and free list into main memory and skips the remaining steps.

(ii) We find the end of the undo log by searching for the entry with the highest sequence number. Due to the presence of hashes, we can reliably find sequence numbers even though the entries in the undo log have variable length.

(iii) If the last transaction recorded in the log has committed, as indicated by a valid hash at the end of the record, we skip to step (v).

(iv) To roll back an uncommitted transaction, `restore` iterates over the list of block ids in the undo log, finds the associated data layer blocksets, and sets the address fields of blocks written by the uncommitted transaction to $\perp$.

(v) We reconstitute the in-memory map and free list by traversing the data layer blocksets in order of decreasing sequence number. We know that the block with a given address first encountered during this traversal is the last committed block (see Invariant 1); we add it to the map and mark it as allocated in the free list. If we encounter blocks with the same address again during the traversal, we set its address field in the blockset header to $\perp$ and mark it as free in the free list.

There is also an API version of `restore` that can be invoked by applications to map the persistent memory pool of a given color into an application's address space.

Write amplification PTx's data structures and `commit` are optimized the reduce write amplification. During a `commit`, each modified block is written only once to NVRAM, which is optimal since a block is the smallest writable unit. However, the metadata in the data layer as well as the undo log add extra writes that we must consider.

The data layer stores one metadata block for each blockset of 41 data blocks; the block must be written (eventually) whenever one or more data blocks in the set are newly allocated or superseded by a transaction. The undo log requires a sequence number, a hash, and a 9-byte record for each blockset written (30bit blockset id; 41bit block bitmap).

Depending on the average number of blocks per blockset written by a transaction, the amplification can range from 5.2% (41 blocks per set in a large transaction of 1MB) to 313% in the worst case (a tiny one-block transaction). This shows that allocating as few blocksets as possible is important. Since PTx is optimized for medium- to large transactions, the main requirement for low write amplification is that we update as few blocksets as possible with a high average number of blocks written. PTx achieves an average write amplification of 8.4% in our experiments (see Section 3.3.6).

3.2.6 Tracking Write Sets

Next, we briefly review approaches to tracking the write set within a transaction. The write set must be complete for correctness and should not vastly overapproximate the true write set for performance. Approaches broadly fall into three categories:

Source code annotation Data structure source code can be annotated to keep track of modifications. While conceptually simple, the application programmer has to be diligent to mark all possible updates to the data structure, without errors, and ideally, not overestimate the write set.

Compiler/Runtime tracking With appropriate runtime/compiler support, the write set can be automatically generated without programmer input. In our evaluation, we use compiler annotations for Software Transactional Memory (STM) in gcc [81] to automatically trap and record writes within a colored region.

Hardware-assisted tracking MMU-hardware provides page write protections and dirty bits, and either can be used to track writes to a colored region. Operating systems offer APIs to use both methods: write-protect a region and receive notification about the first write to a page; or, reset the dirty bits in a region and inspect the bits as part of the commit[2]. In our implementation, we use trackers based on page faults and on dirty bits.

[2]We use a custom kernel module for dirty-bit tracking that improves upon the kernel provided functionality within Linux.

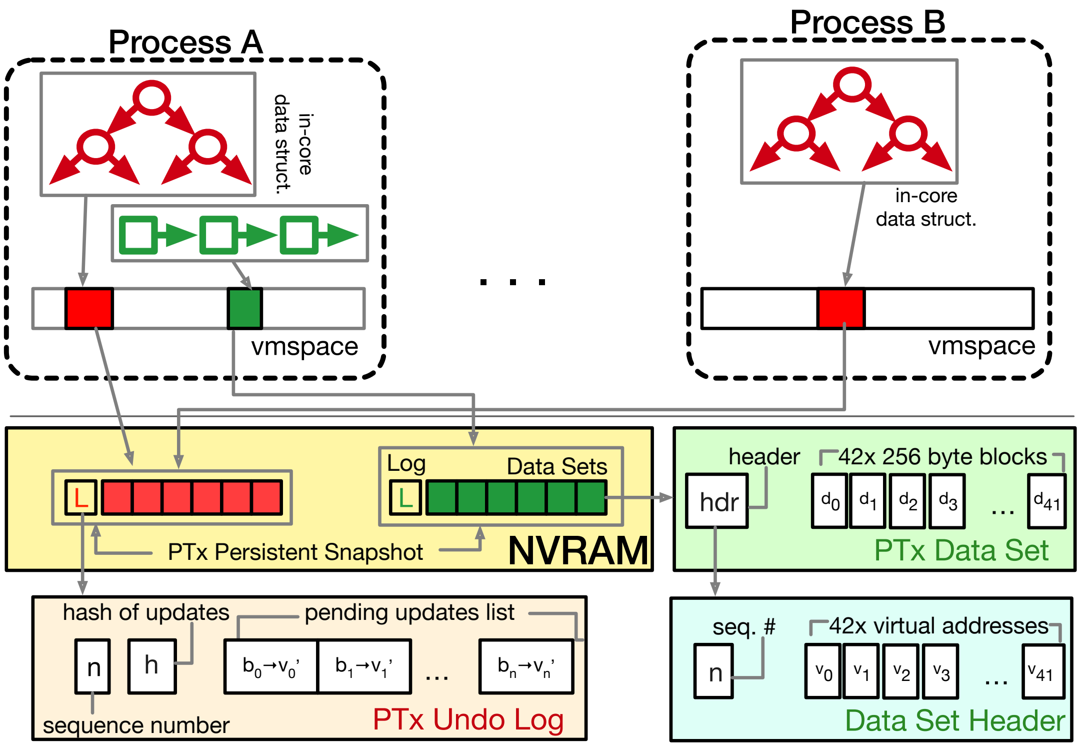

Figure 3.1: Overview of PTx: Applications allocate persistent data structures using “colored” parts of the heap. PTx tracks changes to the colored regions, and atomically updates persistent copies in NVRAM upon `commit`. Details of the NVRAM structure and atomic update are described in the text.

Diffing Because NVRAM writes are slower than NVRAM and DRAM reads, it can make sense to compare the content of a page to be committed with its last committed state in order to narrow down the write set and minimize NVRAM writes. In our evaluation, we use this approach in combination with page-based modification tracking to narrow down which 256-byte blocks within a page were modified. We have found that this optimization often reduces the number of blocks that need to be written by up to 60%.

3.2.7 Non-atomic NVRAM block writes

Our design and prototype assume that writes of individual NVRAM blocks are atomic with respect to failures. Existing software, including Intel's PMDK SDK [26], appears to rely on this assumption. However, we were unable to find any direct confirmation of this guarantee in Intel's documentation of Optane NVRAM memory. In case that Optane, or any future NVRAM product, does not provide atomic block writes, the PTx design can be extended as follows: in the absence of atomic block writes, we have to avoid destructive writes of blockset headers, because a failed write could leave a header in an arbitrary state. Instead, the blockset headers can be placed into a separate circular log, where each block has an additional blockset id and a hash value to be able to verify its integrity. Updated headers are appended to the end of the log. After a restart, the most recent version of a blockset's header can be found by traversing the log in order of decreasing sequence numbers.

3.3 Evaluation of PTx

PTx is implemented as a userspace library and includes the components described in Section 3.2. PTx is written in C++ (with a C compatibility layer) and relies on the NVRAM support provided by the Linux 5.3 kernel. The core of the library itself is comprised of 8629 source lines of code, with an additional 7504 source lines of code for workload generation and testing. The malloc-like allocator provided by PTx is a modified form of Doug Lea's malloc [82] and is not counted in the line count above.

The mechanism provided by the Linux 5.3 kernel to directly map NVRAM pages into the address space is to by configuring NVRAM as a block device, formatting it with ext3 or XFS, and mounting it with "direct-access" enabled. We chose XFS as our file system because it performed better under preliminary evaluation. PTx used it to directly mapped NVRAM into our process's address space.

In Intel Optane NVRAM hardware, writes smaller than 256 bytes are amplified to 256 bytes, and there is no performance advantage to more granular writes. Consequently, PTx is designed to operate largely in terms of 256 byte increments, as we have described in the previous section. PTx uses non-temporal stores, which bypass the write-back cache,to write blocks into NVRAM in 256 byte increments with an AVX instruction (VMOVNTDQ), which bypasses the cache and does not need an additional flush operation (writes to NVRAM are not persistent until evicted from the cache, either implicitly or via a flush call. Non-temporal stores make the flush unnecessary). Since PTx rarely reads from NVRAM (except during `restore` and, for some trackers, `commit`), using non-temporal stores reduces cache pollution.

3.3.1 Experimental setup

We evaluate PTx by measuring the cost of persisting different in-memory data structures, including b-trees, red-black trees, and hash tables. The data structures we use are either the standard C++ library constructs, or the NVRAM specific structures developed for PMDK. (We use the latter for comparing directly with PMDK, as PMDK requires these bespoke implementations.) Along with microbenchmarks over these data structures, we compare PTx persistence against Lightning Memory-Mapped Database (LMDB) [83], which is a database designed for high performance and persistence, to Redis [8], a high performance key-value store, and to Pmem-Redis [84], a custom port of Redis for use with NVRAM.

Our evaluation server has two 2.4 GHz Intel Xeon Platinum 8260 24 core CPUs, 384 gigabytes of RAM, and 3 terabytes of NVRAM. We mounted a 1.5 TB NVRAM backed XFS file system in "direct access" mode, which disabled the page cache and allows NVRAM to be directly mapped into a process address space via the `mmap` call. All of the memory used (NVRAM and DRAM) was local to one CPU, to which we bound all process execution and memory allocation.

There are two critical aspects to the performance of any persistence scheme: the overhead of write set tracking, and the efficiency of the writeback to persistent storage. For tracking, PMDK requires manual annotation of source code, whereas PTx provides automated trackers. In our comparison, we evaluated the four PTx automatic trackers described in Section 3.2.6, and also a manual tracker for PTx. (While not a generic persistence platform, LMDB manually tracks changes for the dictionary data structure it implements).

PTx, PMDK, LMDB all implement their own writeback mechanism, and all provide data structure consistency and persistence. We evaluate all three, and in addition, as a reference, we also compare against a directly mapped NVRAM region. The latter provides persistence but does not provide atomicity or consistency by itself.

When using PTx with the PMDK data structures, we replace the PMDK writeback mechanism with PTx's when the driver application invokes `commit`. The PMDK data structure implementations explicitly state the write set, since that is required by PMDK, but PTx's automated trackers are not provided with this information.

Experiment Design In each experiment, we populate the data structures with 1,000,000 integer keys and 2500 byte values. Insertion operations allocate a 2500 byte array and overwrite it with zeroes. We then insert this value into the data structure. Lookup operations lookup a previously inserted key and read all of the bytes associated with the value. Deletion operations free previously inserted keys and free (but do not read) the value. These operations are performed sequentially by a single process. Our sequence of insertions and deletions ensure that approximately 1,000,000 keys are present in the dictionary at all times. needs to

Parameter Space The performance of all systems depends on the frequency of snapshotting, the read vs. write mixture of the workload, and the access distribution. Towards this end, we evaluate performance while varying all of these parameters. For snapshot frequency, we either performed a `commit` after a certain number of insert or delete operations are performed (10, 500, or 1000 writes) or a period of time has passed (100, 500, or 1000 milliseconds). We also varied the fraction of write operations (0% . . . 50%) during the experiments.

We vary the workload by choosing keys either uniformly at random or from the a Zipf distribution. The code that generates the Zipf distribution and its default parameter are both taken directly from memcached benchmarks [85]. Finally, we also experimented with varying the number of cores that simultaneously try to persist data.

Our full evaluation spanned the Cartesian product of multiple snapshot frequencies, write proportions between `commit` calls, trackers employed, key distributions, thread count, and data structure implementations. In the following, we present representative results from both microbenchmarks and end-to-end application evaluations. We frame the results around a set of high-level questions, which we pose and address in each of the following sections.

3.3.2 PTx versus PDMK

The first set of questions we seek to address is *"How does PTx compare to Intel's PMDK, on data structures designed for PMDK? While taking advantage*

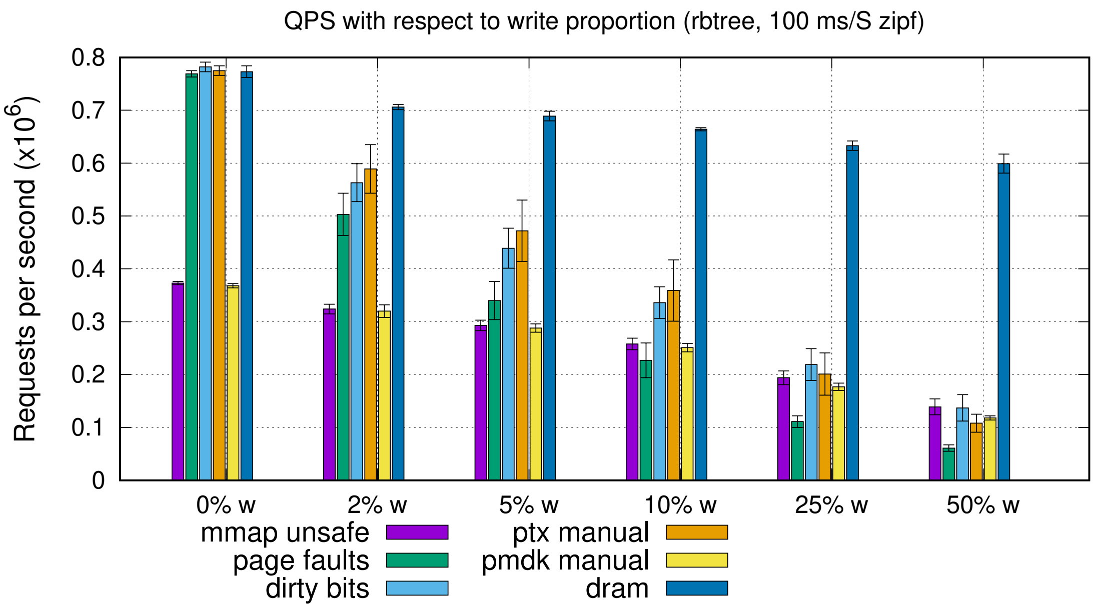

Figure 3.2: Queries per second (in thousands) as a function of write frequency, with the PMDK provided red black tree implementation and Zipf distributed workload,`commit` every 100 ms

of PMDK data structures' explicit write set specifications? While instead using automated write set tracking techniques?" To answer these questions, we perform micro-benchmarks of the NVRAM-specific data structures provided by PMDK.

In the following, the "mmap unsafe" configuration is one where NVRAM is mapped directly into application memory and the `commit` call is mapped to the POSIX `msync` call. This configuration is unsafe as msync does not provide atomic writes, which can lead to inconsistent data if the process faults or the machine crashes during the msync. The "pmdk manual" configuration is stock PMDK, which requires manual annotation of write sets. The PTx configurations replace the PMDK writeback mechanism. 'ptx manual" uses PMDK's manual annotations, whereas the other PTx configurations uses automatic write set trackers.

Figure 3.2 shows the queries per second with the PMDK red black tree while varying the write proportion and choosing keys from a Zipf distribution. Commits are performed every 100 ms. The read-only workload shows the base overhead of using a persistence library and performing periodic snapshots when no data is actually being modified. The DRAM column in the figure shows the throughput of the PMDK red black tree without NVRAM, i.e., the data structure is mapped to DRAM and `commit` is not invoked. (We have performed similar experiments with other PMDK data structures and access distributions: these results are representative.)

For the read-only workload, *all PTx configurations provide 2X or more throughput than either pmdk manual or mmap unsafe*. In fact, query throughput of PTx configurations are on par with DRAM since, by design, PTx performs read operations entirely in DRAM. Both pmdk manual and mmap unsafe map NVRAM directly into application memory, and read-only throughput suffers due to the 3.7X higher read latency of NVRAM. Performance degrades by only 2X (not 3.7X) because caching masks read latency, especially with a skewed workload. The result also shows that invoking `commit` every 100 milliseconds does not appreciably degrade performance when no data is modified.

As the fraction of writes increases, the slower NVRAM writes make the query throughput fall behind the throughput achieved with DRAM. Depending on the workload and application requirements/semantics, however, a persistent data structure can approach the performance of DRAM. While not shown in the figure, e.g., *2% writes and a snapshot once per second performs on par with DRAM*. We also observe that the overhead of some automatic trackers can be significant when transaction

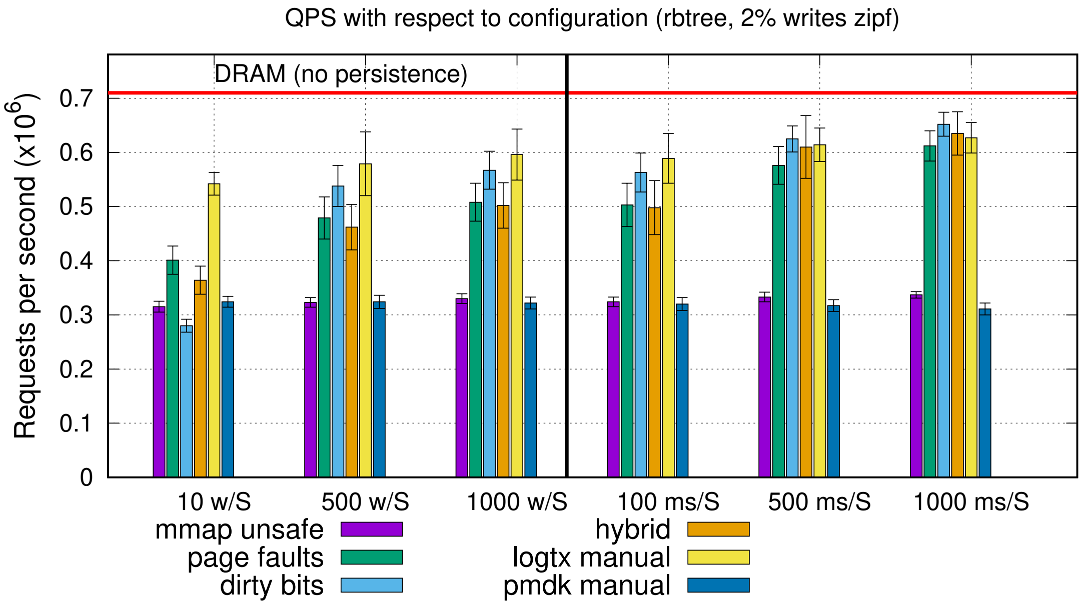

Figure 3.3: Queries per second (in thousands) with the PMDK provided red black tree implementation and Zipf distributed workload with 2% clustered by writes per snapshot or milliseconds per snapshot.

write sets are very small (e.g., dirty bits for 10 writes/snapshot). In particular, while determining the write set, dirty bits iterates over the page tables for the entire allocated region and flushes the TLB, making it unsuitable for this type of workload. In general, less frequent snapshots allow automatic trackers to amortize the cost of finding write sets. With dirty bits, PTx outperforms or matches the performance of PMDK and even the unsafe mmap, and nearly matches the performance of PTx with manual tracking even for small write sets on this workload.

Figure 3.3 and Figure 3.4 show queries per second with the PMDK red black tree while varying snapshot frequency and choosing keys from a Zipf distribution with 2% and 5% writes respectively. With 2% writes, page faults have the higher throughput for very frequent transactions. This is unsurprising. Detecting modifications via dirty bits is a function of the total size of the persistent region and is

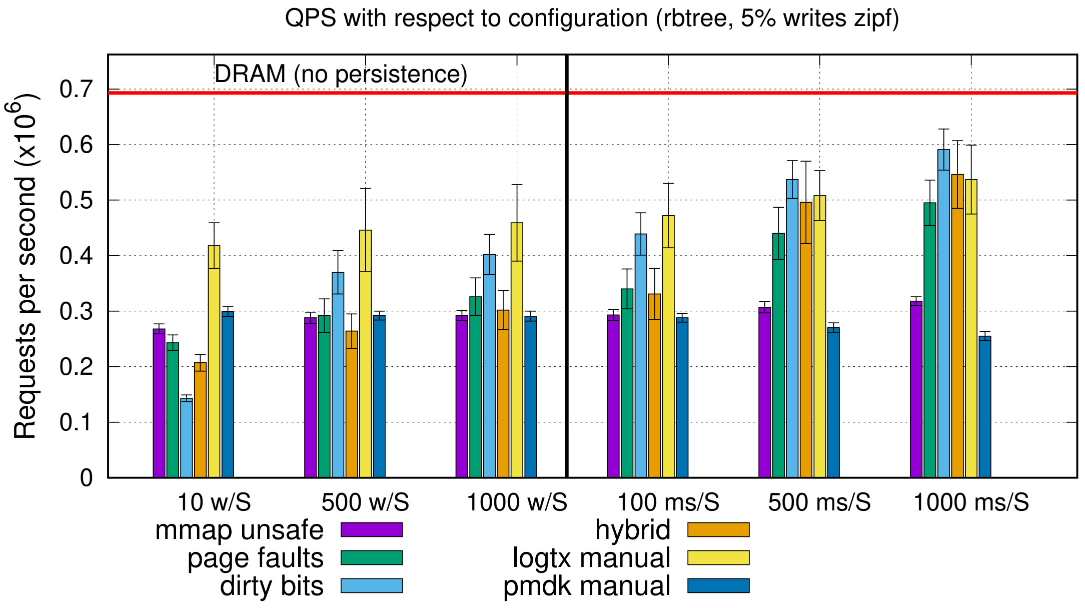

Figure 3.4: Queries per second (in thousands) with the PMDK provided red black tree implementation and Zipf distributed workload with 5% clustered by writes per snapshot or milliseconds per snapshot.

thus, constant. Detecting modifications via page faults is a function of the number of modified pages. This form of detection is expensive (two context switches and system call must be invoked per page written). The implication is that frequent transactions with few writes may have higher throughput when using page faults to detect modifications, but as transaction frequency increases the constant cost of dirty bit detection is amortized and performs better than all other automated forms of tracking.

While PTx performs well in these experiments, perhaps its most enabling feature is its ability to *seamlessly persist any existing data structure without source code change*, including those that were not developed with persistence as a goal. We explore this aspect of PTx next.

3.3.3 PTx on existing data structures

The next question we seek to answer is *"How does PTx perform when used to provide persistence for unmodified data structures from the standard C++ library?"* PTx can atomically persist existing data structures without changing or even necessarily having access to the source code. Any data structure library that takes a memory allocator as input can be persisted without source code change or even re-compilation. The application passes the library an allocator for a colored region, and invokes `commit` as application semantics demand. The C++ standard template library can be parameterized with an allocator, and we use it evaluate PTx using the C++ red black tree and hash tree structures.

In addition to the automatic trackers we evaluated previously, this set of evaluations also include gcc-STM, which is source compatible with the C++ STL. gcc-STM uses GCC's built in STM support modified to supply compiler-generated annotations to PTx.

Figure 3.5 shows the performance of the C++ red black tree with 2% writes and keys chosen from the Zipf distribution described earlier. We varied the commit frequency and corresponding write set sized from 10 to 1000 writes and then from 100ms to 1000ms, respectively. The trends are similar to the earlier results using the PMDK structures but the *absolute query throughput is up to 2.5X higher.* This is likely because the C++ STL data structures are highly optimized, and PTx is able to take advantage of these optimizations without any modification to the source code. We also evaluated the same workload over the C++ hash table with similar

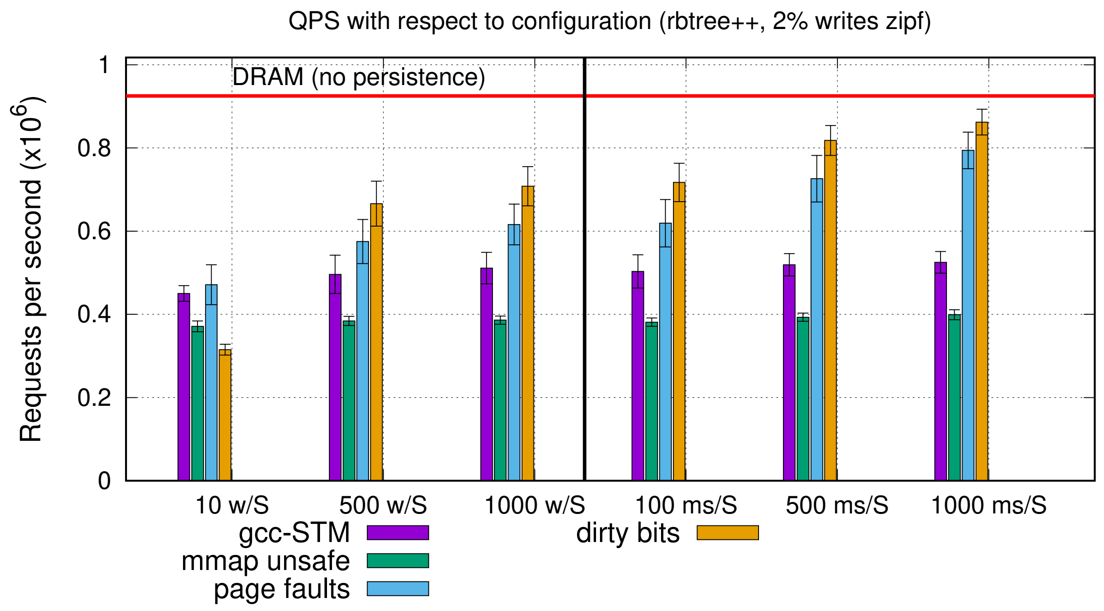

Figure 3.5: Queries per second with the C++ red-black tree implementation from a Zipf distributed workload with 2% writes.

results, shown in Figure 3.6.

Write set tracking based on dirty bits performs best except for very small write sets of 10, where page faults are fastest. gcc-STM performs best for very small transactions but is not competitive otherwise. This is because once a transaction is started, gcc-STM tracks *every* load and store, not just accesses to the colored region. Once again, with larger snapshots (500ms or longer), the cost of PTx automatic marking is amortized, and PTx provides ~90% of DRAM throughput. There is no free lunch however, as write heavy or snapshot heavy workloads expose inherent NVRAM bottlenecks.

In summary, *PTx can provide persistence for existing, unmodified standard data structures with high performance, approaching that of DRAM for large transactions. Moreover, dirty bits outperform other methods of write set tracking, except*

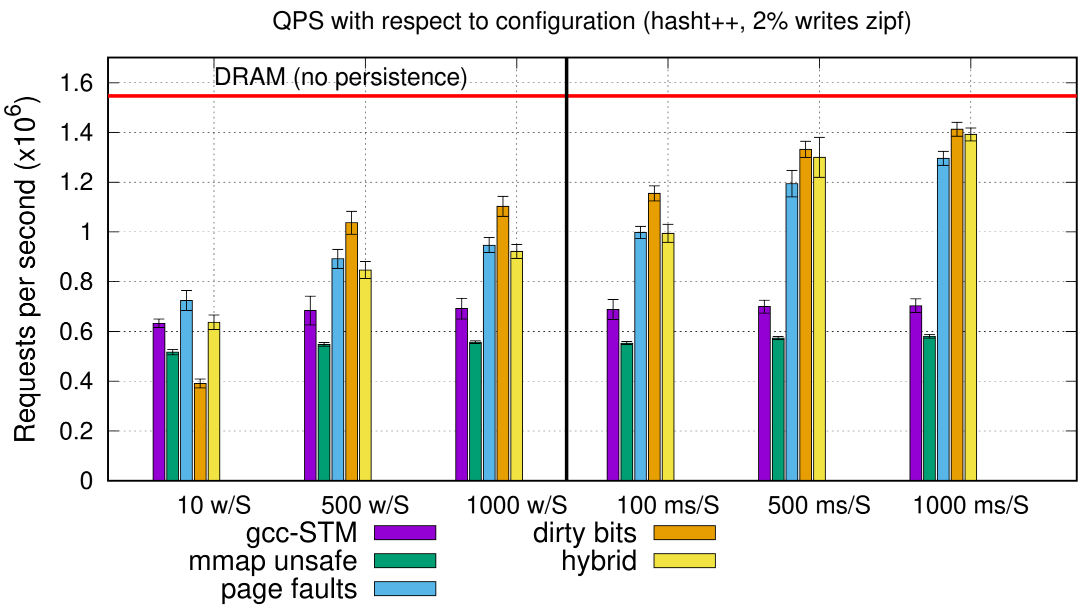

Figure 3.6: Queries per second with the C++ hashtable implementation from a Zipf distributed workload with 2% writes.

for very small transactions, where gcc-STM is best.

3.3.4 Multi-core scalability

We have also performed experiments to evaluate *"How does PTx's performance scale with concurrent threads?"* The single core experiments we have discussed are affected by the longer NVRAM read/write *latency*, but a single core is unable to saturate the NVRAM *bandwidth*. To determine whether or not PTx suffered from any unexpected scaling anomalies, we evaluate the same data structures as before with varying numbers of threads bound to different cores. Each thread executes the same workload and operates independently.

The query performance of PTx and PMDK as the number of cores/threads is increased is linear. When using a 5% mix of writes, and a `commit` every 500 writes, PTx scaled with a normalized co-efficient of 1 for the PMDK provided red-black tree and the C++ red-black tree, regardless of the workload distribution. However, Zipf distributed workloads did have higher absolute throughput by roughly 40%. PMDK also scaled linearly, but with a co-efficient of .91 for a uniformly distributed workload and a co-efficient of .86 for Zipf distributed workloads, with Zipf outperforming PMDK by 40-50%.

There are two key points: *First, PTx shows no unexpected scaling anomalies, with performance increasing essentially linearly up to 24 cores on this workload. Second, PTx scales better than PMDK: we hypothesize this is is due to PMDK's increased interactions with the NVRAM memory bus.* Also, as expected, the Zipf distributions for both PMDK and PTx are able to benefit from DRAM caching, and provide higher absolute performance. (Even though our test machine has 48 cores, we did not scale beyond 24 since each CPU only has 24 cores, and both the NVRAM and DRAM were local to the CPU we ran the tests on.)

3.3.5 Applications with persistent state

Next, we address the question *"How does PTx, when used to provide persistence for a standard C++ library hashtable, compare to a custom PMDK hashtable, and to LMBD, on the same workload? How is the comparison influenced by how often state is persisted?"* Our previous evaluations have focused on microbenchmarks of data structure libraries. We compare PTx's performance to mature application software designed to provide high throughput and native persistence: Lightning

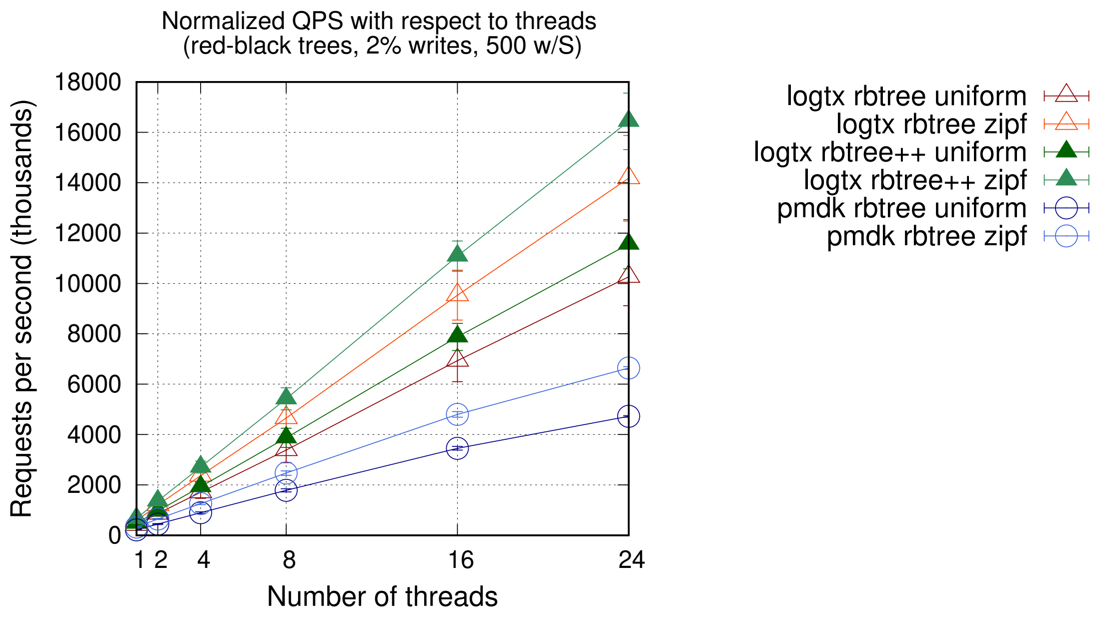

Figure 3.7: Queries per second with the C++ and PMDK-provided red-black tree implementation, 2% write workload, with 500 writes between `commit` calls

Memory-Mapped Database (LMDB) [83], version 1.8. LMDB is a database designed for read-heavy workloads, permitting only a single writer at a time. Internally, LMDB stores data as a memory mapped b-tree. Reads operate directly over this memory mapped tree, whereas writes are propagated to the underlying file in a ACID compliant manner. We ran LMDB on an SSD, on a NVRAM file system with direct mapping, and on a NVRAM filesystem without direct mapping. The last configuration allows LMDB's memory mapped btree to be cached in DRAM, and had the best query throughput, faster than the direct mapped version by 20% and 3X-5X faster than LMDB over SSD (state of the art without NVRAM.)

Figure 3.8 shows the best performing configuration of LMDB and PTx with manual and automatic marking using the C++ hashtable, on a workload with 500 writes per `commit`. The result with PMDK's custom hashtable was included for

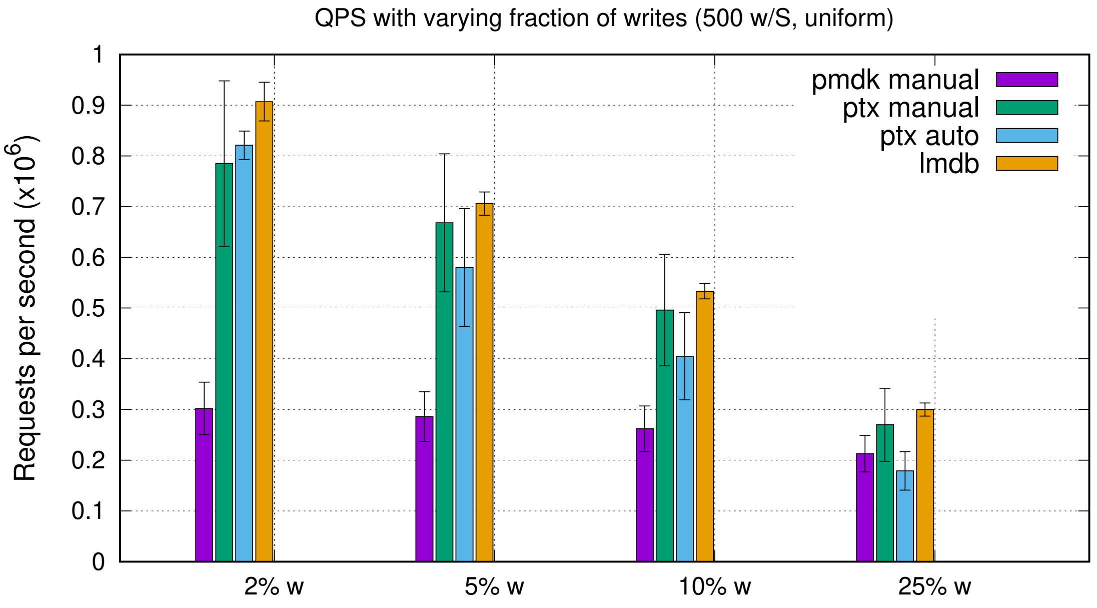

Figure 3.8: Queries per second with 500 writes / `commit` from a uniformly distributed workload varying write proportion.

comparison. We performed these experiments with varying query distributions and write fractions, and the results presented are representative. (In all cases, the test inputs are identical for each system, and the information persisted is also the same for each configuration.) For PMDK and ptx manual, the results show the PMDK hash table data structure as it had the highest throughput for both systems. For automatic tracking, the C++ hash table data structure had highest throughput using the dirty bits marking. LMDB internally uses a btree tree with manual annotation.

LMDB performs well in the low write-mix scenarios it is optimized for (2% writes), outperforming PTx with manual marking by about 10% in both configurations. As the write mix increases, PTx with manual marking performs similar to LMDB. Invoking `commit` every 10 mutations (not shown), with very small modified sets, over a large data structure with 1M keys, represents an extreme worst case for automatic tracking, and it lags manual marking by 20-30%. (PTx automatic tracking remains competitive with PMDK manual marking, however.) With larger write sets, PTx with automatic tracking performs much better, and is comparable to both LMDB and PTx manual marking.

We parameterized the experiment in Figure 3.8 to focus on the absolute worst case scenarios for PTx: no query locality, very small write sets, and very frequent `commit`s. It is interesting that *PTx remains competitive even under these constraints, especially since both PMDK and LMDB have the advantage of implementing manual marking and writeback over a custom-built, optimized data structure, while PTx is a generic persistence library applied in this experiment to a standard C++ data structure not designed for persistence.* The small write sets in these experiments cause a `commit` to be invoked multiple times per *millisecond*, providing very fine-grained persistence at the cost of query throughput. Next we consider how performance is affected when the persistence requirements are less fine-grained.

Figure 3.9 compares the performance of PTx, PMDK, LMDB, to two DRAM-only data structures: Redis' dictionary, and the C++ STL hashtable. Redis is a key value store, but in this experiment we only evaluated its hash table separate from the network front-end and query processor. (We present an evaluation of full end-to-end Redis and variants in the next section.) PTx uses the C++ STL hashtable, with dirty bits automatic tracking. PMDK uses its hashtable, LMDB its btree, and both employ manual marking. The commit interval ranges from 100 ms to 32 seconds.

PTx with fully automatic marking is competitive with the Redis data structure with *no* persistence if we `commit` every two seconds, and far outperforms Redis for

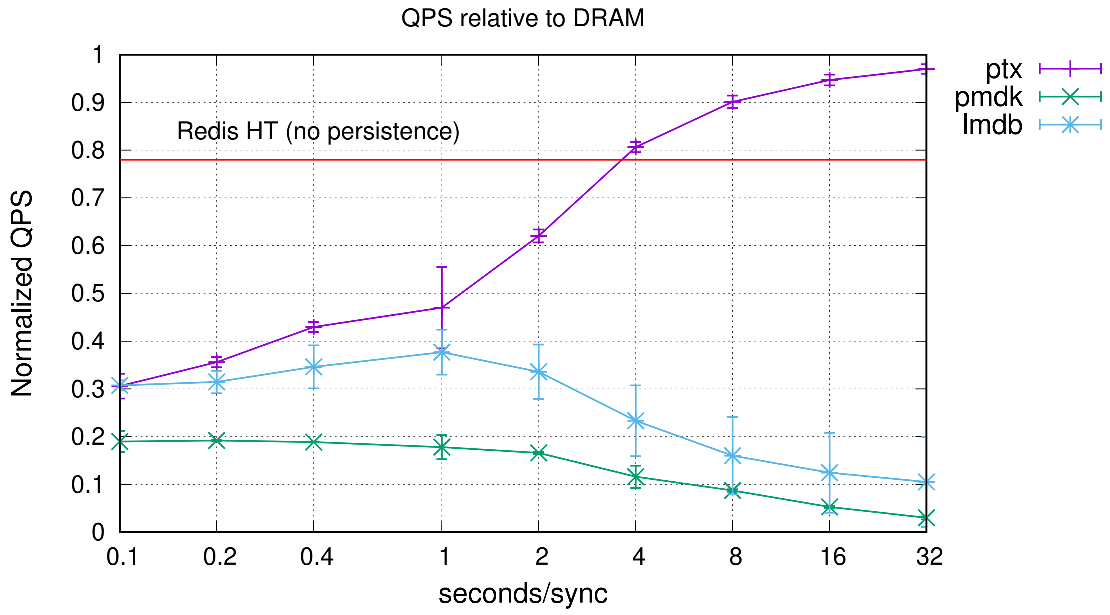

Figure 3.9: Queries per second (relative to DRAM) with the C++ STL hashtable, dirty bits tracker with a 25% write workload generated from uniformly distribution keys. For comparison, we also include the throughput of PMDK and LMDB, both of which require manual annotation, and of the Redis hash table, *without* persistence.

longer `commit` intervals. For all `commit` intervals in the plot, PTx with automatic marking outperforms PMDK and LMDB, both of which are optimized for manual marking. In fact, *if we persist every 8 seconds, PTx with automatic marking, for the same data structure, with no source code changes, provides 90% the query throughput of running directly on DRAM, and achieves over 98% of DRAM throughput with 32 second `commits`.* (As we've shown earlier, PTx can approach DRAM much quicker if the proportion of writes in a transaction is smaller.)

3.3.6 Persistent key-value store performance

Next, we address the question *"How does PTx, when used as a persistence backend for Redis, compare to native Redis, to Pmem-Redis (a version of Redis* *designed for use with NVRAM), and to a simple custom kvs servers that relies on a standard C++ hashtable, backed by PTx for persistence?"* Our previous evaluation showed the performance of persisting state for various data structures used within a single process. We next evaluate the performance of a key value server that receives operations from a client and periodically persists the data structure. We evaluated five different systems, stock Redis with no persistence, stock Redis with an append-only log written to an NVRAM file system, Redis-pmem, Redis-PTx, and a custom written key value server that uses C++ hash tables and PTx for persistence. Redis-pmem is a version of Redis modified to store large values in persistent memory. Redis-PTx is our modified version of Redis, which uses PTx as its persistence backend.

Table 3.2 shows the performance of each system under a uniform key distribution with 2500 byte values. 25% of operations modified the data structure. Each system was pinned to a single core and persisted state once per second. Redis-PTx used dirty-bit based automatic tracking. We also evaluated other configurations (different key sizes, access distributions, read-write mix), and these results are representative.

Stock Redis using NVRAM as the file system was the best performing variant. Under this configuration, Redis writes each operation to an append-only disk and periodically calls `fsync`. Pmem-Redis also writes an append-only log to disk, but it writes larger values to NVRAM. This reduces memory pressure by removing large values from DRAM, but absent memory pressure, similar performance can be achieved by just storing the Redis append-only log on the NVRAM file system and writing to it through the file system API.

System	QPS	lines
Redis (no persistence)	166,471	0
Redis (NVRAM disk)	108,485	0
Redis-pmem	106,074	4,382
Redis-PTx	106,808	291
custom-PTx	345,881	428 (full server SLOC)

Table 3.2: Queries/second achieved with 1 second persistence for Redis, persisting its write-ahead log to disk, Redis-pmem, Redis using PTx as a backend, and a custom key value server using PTx and the C++ hash table implementation. Lines is the source code lines changed relative to stock Redis.

In our experiments, Redis' overhead was dominated by request parsing. To fully expose the overhead of persisting state, we wrote a custom key value server with a compatible C client API and similar semantics. Our custom protocol allowed for higher throughput without persistence (649K RPS), which gave us more headroom to evaluate PTx in an integrated application. With one second commit intervals, PTx significantly outperform every other configuration. We performed additional experiments with more frequent `commit`s and found that Redis-PTX exceeds or meets Redis-pmem's performance even when committing every 10 milliseconds (100X as frequently as Redis-pmem) between key modifications. Our custom key-value server also enables easy instrumentation to measure write amplification. (Such a measurement is otherwise difficult, e.g., in pmem-redis, because of how pervasive the source code changes are, and the various different parts of the code where NVRAM is accessed.) For the configurations we evaluated, regardless of `commit` interval, PTx achieved an average write amplification of approximately 8.4%.

We also tested our custom key value server with values that were either ran-

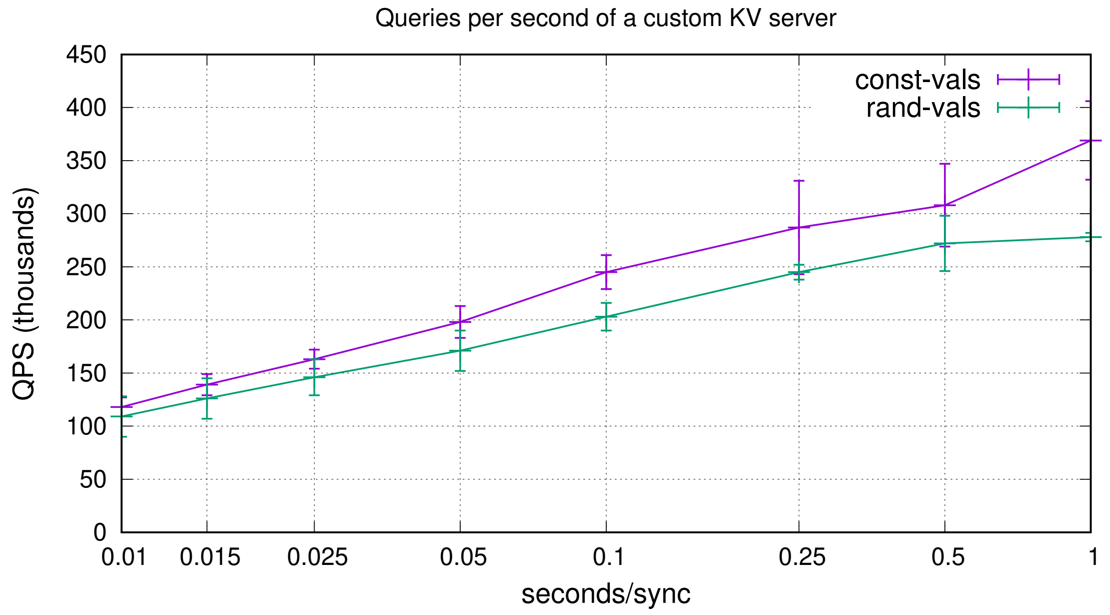

Figure 3.10: Queries per second (in thousands) of our custom key-value server as a function of `commit` frequency and value distribution

domized or constant. The results are shown in Figure 3.10. With constant values, areas of the heap that are reused are more likely to be written but unmodified. PTx will detect that the pages have not been modified and will not write the written pages to NVRAM. With fully randomized values, little to no NVRAM writes will be prevented by deduplication. As expected, constant values result in higher throughput, but this difference narrows as `commit` frequency increases. We expect that with constant values, the probability that the allocator will allocate memory that has not been modified between transactions increase as transaction size increases. In all cases, the throughput between random and constant values is within the margin of error.

PTx offers strong performance, when compared to DRAM and to existing mature systems, such as Redis and LMDB, is a remarkable result in context. Conventional wisdom, based on microbenchmarks, points to NVRAM as a storage medium that is faster than SSDs; in contrast, we show that NVRAM used with PTx promises an alternative that provides throughput comparable to DRAM, and often faster than existing customized protocols, while simultaneously providing consistency and persistence, *without* onerous programmer effort to ensure correctness. NVRAM with PTx can be thought of as executing at a configurable fraction of native DRAM speeds, with a commensurate "lag" in persistence. This observation enables a previously unavailable form of programming "durable" data structures that are at once performant, and relieving the development cycle from being burdened by the rigors of "annotating" changes, or optimizing data structures for different storage media.

3.4 Conclusion

PTx provides a powerful, high-level abstraction, but its implementation requires access to low-level abstractions that are not provided by POSIX. The primary high-level abstraction that PTx needs to circumvent is the buffer cache used for memory mapped files. The buffer cache loads file state into DRAM for reading or writing by the application before flushing to disk either when driven by some OS heuristic (e.g., memory pressure) or when explicitly flushed via a `msync` call. This abstraction is not suitable for PTx, both due to correctness and performance limitations. Data is flushed to persistent media (in this case, NVRAM) at unpredictable times, making it difficult to provide atomic semantics, and data corresponding to the same region of memory is potentially flushed multiple times within a transaction and at inappropriate granularities, hampering performance. PTx circumvents this by taking advantage of a new low-level abstraction introduced specifically for NVRAM: direct access to the data pages for a memory mapped file. This solution is safe, so long as the file system does not reallocate data blocks while the file is memory mapped. Maintaining this restriction requires some care for the file system, such as ensuring that all blocks are pre-allocated and fixed when a file is memory mapped, but this low-level access is sufficient to implement PTx efficiently. We discuss further optimizations that would be possible under the null-Kernel architecture in Section 5.4.

Chapter 4: Light-weight Contexts

In this chapter we discuss Light-Weight Contexts (*lwC*s), which is a new abstraction that decouples memory isolation, execution state, and privilege separation from within a process.

4.1 Introduction

Processes abstract the unit of isolation, privilege, and execution state in general-purpose operating systems. Computations that require memory isolation, privilege separation, or continuations at the OS level must be run in separate processes[1]. Unfortunately, switching and communicating between processes incurs the cost of invoking the kernel scheduler, resource accounting, context-switching, and IPC. The actual hardware-imposed cost of isolation and privilege separation, however, is much smaller: if the TLB is tagged with an address space identifier, then switching context requires as little as a system call and loading a CPU register.

Just as threads separate the unit of execution from a process, we assert that there is benefit to decoupling memory isolation, execution state, and privilege sepa-

[1]Language runtimes can provide these properties at the expense of additional overhead, language dependence, and an increased trusted computing base.

ration from processes. We show that it is possible to isolate memory and privileges, and maintain multiple execution states *within* a process with low overhead, thus streamlining common computation patterns and enabling more efficient and safe code.

We introduce a new first-class OS abstraction: the *light-weight context* (*lwC*). A process may contain multiple *lwC*s, each with their own virtual memory mappings, file descriptor bindings, and credentials. Optionally and selectively, *lwC*s may share virtual memory regions, file descriptors and credentials.

*lwC*s are not schedulable entities: they are completely orthogonal to threads that may execute within a process. Thus, a thread may start in *lwC a*, and then invoke a system call to *switch* to *lwC b*. Such a switch atomically changes the VM mappings, file table entries, permissions, instruction and stack pointers of the thread. Indeed multiple threads may execute simultaneously within the same *lwC*. *lwC*s maintain per-thread state to ensure a thread that enters a *lwC* resumes at the point where it was created or last switched out of the *lwC*.

*lwC*s enable a range of new in-process capabilities, including fast roll-back, protection rings (by credential restriction), session isolation, and protected compartments (using VM and resource mappings). These can be used to implement efficient in-process reference monitors to check security invariants, to isolate components of an app that deal with encryption keys or other private information, or to efficiently roll back the process state.

We have implemented *lwC*s within the FreeBSD 11.0 kernel. The prototype shows that it is possible to implement *lwC*s in a production OS efficiently. Our experience with implementing and retrofitting large applications such as Apache and nginx with *lwC*s has taught us that it is possible to introduce many new capabilities, such as rollback and secure data compartments, to existing production code with minimal overhead.

In this chapter we do the following:

- We introduce *lwC*s, a first-class OS abstraction that extends the POSIX API, and present common coding patterns demonstrating its different uses.

- We describe an implementation of *lwC*s within FreeBSD, and show how *lwC*s can be used to implement efficient session isolation in production web servers, both process-oriented (Apache, via roll-back) and event-driven (nginx, via memory isolation). We show how efficient snapshotting can provide session isolation while improving performance on web-based applications using a PHP-based MVC application on nginx. We show how cryptographic libraries such as OpenSSL can efficiently create isolated data compartments *within a process* to render sensitive data (such as private keys) immune to external attacks (e.g., buffer overruns a la Heartbleed [35]). Finally, we show how *lwC*s can efficiently implement in-process reference monitors, again for industrial-scale servers such as Apache and nginx, that can introspect on system calls and memory.

- We evaluate *lwC*s using a range of micro-benchmarks and application scenarios. Our results show that existing methods for session isolation are often slower than *lwC*s. Other common uses such as *lwC*-supported sensitive data compartments and reference monitoring have little to negligible overhead on production servers. Finally,

Function	Return Value		System Call
Create *lwC*	{*new*, *caller*, args}	←	`lwCreate`(resource-spec, options)
Switch to *lwC*	{*caller*, args}	←	`lwSwitch`(*target*, args)
Resource access	status	←	`lwRestrict`(*l*, resource-spec)
	status	←	`lwOverlay`(*l*, resource-spec)
	status	←	`lwSyscall`(*target*, mask, syscall, syscall-args)

Table 4.1: API for interacting with *lwC*s. Parameters in italics *new, caller, ...* are *lwC* descriptors. Arguments args are passed during *lwC* switches; resource-spec denotes resources (e.g. memory pages, file descriptors) that can be shared or narrowed.

we show that using the *lwC* snapshot capability to quickly launch an initialized PHP runtime can improve the performance of a production server.

The rest of this chapter is organized as follows: we discuss related work in Section 2.2 and describe the *lwC* abstraction, API, and design in Section 4.2. We present common *lwC* coding patterns in Section 4.3. We describe our FreeBSD implementation of *lwC*s in Section 3.3, and present an experimental evaluation in Section 4.4.

4.2 *lwC* design

*lwC*s are separate units of isolation, privilege, and execution state within a process. Each *lwC* has its own virtual address space, set of page mappings, file descriptor bindings, and credentials. Threads and *lwC*s are independent. Within a process, a thread executes within one *lwC* at a time and can switch between *lwC*s. *lwC*s are named using file descriptors. Each process starts with one *root lwC*, which has a well-known file descriptor number.

Table 4.1 shows the *lwC* API. A *lwC* may create a new (child) *lwC* using the `lwCreate` operation and receive the child's file descriptor. If a context *a* has a valid descriptor for *lwC c*, a thread executing inside *a* may switch to *c* using the `lwSwitch` operation. A *lwC c* is terminated (and its resources released) when the last *lwC* with a descriptor for *c* closes the descriptor. Common usage patterns of the *lwC* API will be shown in Section 4.3.

4.2.1 Creating *lwC*s

The `lwCreate` call creates a new (child) *lwC* in the current process. The operation's default semantics are similar to that of a POSIX *fork*, in that the child *lwC*'s initial state is an identical copy of the calling (parent) *lwC*'s state, except for its descriptor. Unlike with fork, however, child and parent *lwC* share the same process id, and no new thread is created. No execution takes place in the new *lwC* until an existing thread switches to it.

`lwCreate` returns the descriptor of the new child *lwC new* to the parent *lwC* with the *caller* descriptor set to -1. When a thread switches to the new *lwC* (*new*) for the first time, the `lwCreate` call returns with the caller's *lwC* descriptor in *caller* and the parent's *lwC* descriptor in *new*, along with any arguments from the caller in args.

By default, the new *lwC* gets a private copy of the calling *lwC*'s state at the time of the call, including per-thread register values, virtual memory, file descriptors, and credentials. Shared memory regions in the calling *lwC* are shared with the new *lwC*. The parent *lwC* may modify the visibility of its resources to the child *lwC* using the resource-spec argument, described in Section 4.2.3.

The implementation does not stop other threads executing in the parent *lwC* during an `lwCreate`. To ensure that the child *lwC* reflects a consistent snapshot of the parent's state, all threads that are active in the parent at the time of the `lwCreate` therefore should be in a consistent state. The application may achieve this, for instance, by barrier synchronizing such threads with the thread that calls `lwCreate`. A thread that does not exist in the parent *lwC* at the time of the `lwCreate` may not switch to the child in the future.

The `lwCreate` call takes several option flags. `LWC_SHAREDSIGNALS` controls signal handling in the child *lwC*, as described in Section 4.2.7. `LWC_SYSTRAP` indicates that any system calls for which the child does not hold the required OS capability should be redirected to its parent. This feature enables a parent to interpose and mediate its child's system call activity, as described in Section 4.2.6.

The fork semantics of `lwCreate` enable the convenient, language independent creation of *lwC*s based on the current state of the calling *lwC*. No additional APIs are required to initialize a new *lwC*. The new *lwC* can be viewed also as a snapshot of the state of the caller at the time of invoking `lwCreate`, enabling the caller to revert to this state in the future.

4.2.2 Switching between *lwC*s

The `lwSwitch` operation switches the calling thread to the *lwC* with descriptor *target*, passing args as parameters. `lwSwitch` retains the state of the calling thread in the present *lwC*. When this *lwC* is later switched back into by the same thread, the call returns with the switching *lwC* available as *caller* and arguments passed in args.

Note that returns from a `lwSwitch` and `lwCreate`, any signal handlers that were installed, and the instruction pointer locations of threads in a parent *lwC* at the time of a `lwCreate` define the only possible entry points into a *lwC*. (The root *lwC* has an additional one-time entry point when the process is launched.)

`lwSwitch` is semantically equivalent to a coroutine `yield`. In fact, as far as control transfer is concerned, *lwC*s can be viewed as isolated and privilege separated coroutines. Recall that a procedure is a special case of a coroutine. To achieve a (remote) procedure call among *lwC*s, the called procedure, when done, simply switches to its caller and then loops back to its beginning. This functionality can be provided easily as part of a library.

4.2.3 Static resource sharing

When a *lwC* is created using `lwCreate`, the child *lwC* receives a copy-on-write snapshot of all its parent's resources by default. The parent can modify this behavior using the *resource-spec* argument in the `lwCreate` operation. The *resource-spec* is an array of C unions: each array element specifies either a range of file descriptors, virtual memory addresses, or credentials. For each range, one of the following sharing options can be specified. `LWC_COW`: the child receives a logical copy of the range of resource (the default). `LWC_SHARED`: the range of resources is shared among parent and child. `LWC_UNMAP`: the range of resources is not mapped from the parent into the child. (The child may subsequently map different resources in the address range.)

When restricting the resources inherited by the child, care must be taken to minimally pass on the stacks, code, synchronization variables, and other dependencies of all threads in the parent *lwC*, to ensure predictable behavior if these threads switch to the child in the future.

4.2.4 Dynamic resource sharing

A *lwC* may dynamically map (overlay) resources from another *lwC* into its address space using the `lwOverlay` operation. The caller specifies which regions of a given resource type (file descriptor or memory) are to be overlayed, and whether the specified region should be copied or shared, in the *resource-spec* parameter. The `lwOverlay` call will only succeed if the caller *lwC* holds *access capabilities* (described below in Section 4.2.5) for the requested resources. A successful `lwOverlay` operation unmaps any existing resources at the affected addresses in the caller's address space.

4.2.5 Access capabilities

Access capabilities are associated with *lwC* file descriptors. Each *lwC* holds a descriptor with a universal access capability for itself. When a *lwC* is created, its parent receives a descriptor with a universal access capability for the child. A parent *lwC* may grant a child *lwC* access capabilities for the parent *lwC* selectively by marking resource ranges as `LWC_MAY_ACCESS` in the *resource-spec* argument passed to the `lwCreate` call.

Access capabilities may be restricted on a *lwC* descriptor with the `lwRestrict` call. The *resource-spec* parameter restricts the set of resources that may be overlayed or accessed by any context that holds the *lwC* descriptor *l*. The valid resource types are file descriptors, virtual memory addresses, and syscall numbers. Subsequent to the call, the descriptor will allow `lwOverlay` to succeed for any file descriptors and memory addresses, and `lwSyscall` for any syscalls, respectively, that are within the intersection of the resource-spec set and whatever capabilities *l* had previous to the call.

4.2.6 System call interposition/emulation

Consider an *lwC* C that was created with the `LWC_SYSTRAP` flag. If a thread in C invokes a system call for which C does not hold a capability according to the OS's sandboxing mechanism, the thread is switched to its parent *lwC* instead, if the thread exists in the parent (if the thread does not exist in the parent, the call fails with an error). When the thread is resumed in the parent *lwC* as a result of a faulting syscall by the child, the arguments in the switch contain the system call number attempted and the arguments passed to it. The parent can choose to decline the syscall and return an error to the child, or perform a syscall on behalf of the child, possibly with different arguments (see below). To signal the completion of the child's system call, the thread executing in the parent *lwC* switches back to the child with the return value and any error code as arguments to the switch call.

An authorized *lwC* may perform a syscall on behalf of another *lwC target* using the `lwSyscall` operation. The `lwSyscall` succeeds if the *lwC* calling the operation holds an access capability (see Section 4.2.5) for the *target* and syscall, and holds the OS credentials required to perform the requested syscall. The effects of a successful execution of `lwSyscall` are as if the *target* had executed the requested syscall, except that it returns to the calling context. The mask parameter allows the caller to modify this behavior by specifying aspects of its own context that are to be put in place for the duration of the system call. Specifically, the caller may specify that the target's file table, memory space, credentials, or any combination be replaced by the caller's equivalent for the duration of the call. This allows the efficient implementation of useful patterns, such as enabling a untrusted *lwC* to read (or append) a fixed number of bytes from (to) a protected file *without* having access to the file descriptor.

4.2.7 Signal handling

*lwC*s modify the standard POSIX signal handling semantics in the following way. We distinguish between *attributable* signals, which can be attributed to the execution of a particular instruction in a *lwC*, and *non-attributable* signals, which cannot. Attributable signals, such as `SIGSEGV` or `SIGFPE`, are delivered to the *lwC* that caused the signal immediately. Non-attributable signals, such as `SIGKILL` or `SIGUSR1`, are delivered to the root *lwC* and any *lwC*s in the process that were created with the LWC_SHARESIGNALS option by a parent *lwC* that is able to receive such signals. A non-attributable signal is delivered to a *lwC* upon the next switch to the *lwC*.

4.2.8 System call semantics

*lwC*s modify the behavior of some existing POSIX system calls. During a `fork`, all *lwC*s in the calling process are duplicated in the child process. Any memory regions that were `mmap`'ed as MAP_SHARED in some *lwC*s of the calling process are shared with the corresponding *lwC*s in the new child process, *within and across* the two processes. Any memory regions that are shared among *lwC*s in the parent process using the LWC_SHARED option in `lwCreate` are shared among the corresponding *lwC*s *within* the child process only. An *exit* system call in any *lwC* of a process terminates the entire process.

4.2.9 *lwC* isolation

Because *lwC*s do not have access to the state of each others' memory, file descriptors, and capabilities unless explicitly shared, they can provide strong isolation and privilege separation within a process. Since *lwC*s share executable threads, however, an application needs to make certain assumptions about the behavior of other *lwC*s in the same process, even if they don't share resources and don't have overlay capabilities for each other. Specifically, a *lwC* can block or execute a thread indefinitely or terminate the process prematurely by invoking `exit`.

We believe these assumptions are reasonable in practice because the *lwC*s of a process are part of the same application program. Denial-of-service within a process is self-defeating. On the other hand, *lwC*s can reliably prevent accidental leakage of private information across user sessions, isolate authentication credentials and other secrets, and ensure the integrity of a reference monitor.

A *lwC* can learn about certain activities of other *lwC*s by registering for non-attributable signals. An application that wishes to limit information flow across *lwC*s should create *lwC*s without the LWC_SHARESIGNALS option (the default).

4.2.10 *lwC* security

*lwC*s provide isolation and privilege separation within a process, but include powerful mechanisms for sharing and control among the *lwC*s of a process. Therefore, it is important to understand the threat model and the security properties provided by the *lwC* abstraction.

Threat model We assume that the kernel is trustworthy and uncompromised, and that the tool chain used to build, link, and load the application does not have exploitable vulnerabilities that can be used to hijack control before main() starts.

When a *lwC* is created, its parent has universal privileges on the *lwC*. Consequently, the security of a *lwC* assumes that its parent (and, by transitivity, all its ancestors) cannot be hijacked to abuse these privileges. In practice, the parent should drop all unnecessary privileges on the child immediately after the child is created, so this assumption is needed only with respect to the remaining privileges. When an application uses dynamic sharing, the same assumption must be extended to all *lwC*s that obtain privileges indirectly. The *lwC* API does not enable any inter-process communication or sharing beyond the standard POSIX API. Consequently, no new assumptions regarding *lwC*s in other processes are needed.

Security properties The properties of a *lwC* are constrained by the properties of the process in which it exists. A *lwC* cannot attain privileges that exceed those of its process, and the confidentiality and integrity properties of any *lwC* cannot be weaker than those of its process. The properties of the root *lwC* are those of the process. In applications that do not use dynamic sharing, the privileges of a non-root *lwC* are bounded by those of its parent and, transitively, by those of its ancestors; its integrity and confidentiality cannot be weaker than those of any of its ancestors. In applications that use dynamic sharing through the exchange of access capabilities via a common ancestor, the integrity (confidentiality) of a *lwC* depends on all siblings and descendants that have write (read) rights to it. For this reason, dynamic sharing should be used with caution.

In typical patterns of privilege separation, the root *lwC* should run a high-assurance component, i.e., one that is simple, heavily scrutinized, and exports a narrow interface. A component that protects sensitive state is at or near the root, to minimize its dependencies. More complex, less stable, network or user-facing components should be encapsulated in de-privileged *lwC*s at the leaves of a process's *lwC* tree and should execute with the least privileges required.

4.3 Common *lwC* usage patterns

In this section, we illustrate *lwC* use patterns for snapshots, isolation and protection rings. For some of the patterns, we use a web server as an illustrative setting. However, all the patterns are broadly applicable.

Snapshot and rollback A common *lwC* use pattern is snapshot and rollback, where a service process (such as a server worker process) initializes its state to the point where it is ready to serve requests (or sessions), snapshots this state, serves a request and rolls its state back to the snapshot before serving the next request. As compared to a setup where the process manually cleans up request-specific state after each request, the snapshot and rollback can improve performance by efficiently discarding the request-specific state with a single call, and also improves security by isolating *sequential* requests served by the same task from each other.

Algorithm 4 shows the pseudocode of a small library containing two functions—snapshot() and rollback()—and a main() server function illustrating their use. The server initializes its state and calls snapshot() on line 12 to create a snapshot. snapshot() duplicates the current *lwC* (copy-on-write) using `lwCreate` on line 2. The descriptor of the duplicated snapshot, called new, is returned at line 4 and stored in the variable snap. The program serves the request and then, to reset its state, calls rollback(). Control transfers to line 2 *in the snap* (the child) and then immediately to line 6 where the original *lwC* is closed (its resources are reclaimed). The snap recursively calls snapshot() (line 7). At line 2, it creates a duplicate of itself and returns that duplicate to main() at line 12. The cycle then repeats, with snap and its duplicate having taken the roles of the original *lwC* and the snap, respectively.

Algorithm 4 Snapshot and rollback

```
 1: function SNAPSHOT()
 2:     new,caller,arg = lwCreate(default_spec, ...)
 3:     if caller = -1 then                                    ▷ parent
 4:         return new
 5:     else
 6:         close(caller)
 7:         return snapshot()
 8: function ROLLBACK(snap)                                    ▷ never returns
 9:     lwSwitch(snap, 0)
10: function MAIN()
11:     ...                                                    ▷ initialize state
12:     snap = snapshot()
13:     ...                                                    ▷ serve request
14:     rollback(snap)
            ▷ kills current lwC, continues at line 12 in snap
```

In our evaluation, we use this pattern to roll back the state of pre-forked worker processes after each session in the Apache web server.

Isolating sessions in an event-driven server High throughput servers like nginx handle several sessions in single-threaded processes using event-driven multiplexing. However, they provide no isolation among sessions within a process. This shortcoming can be addressed using *lwC*s. Algorithm 5 illustrates the usage pattern.

The program defines a set of network socket descriptors to poll, one for each

Algorithm 5 Event-driven server with session isolation

```
 1: function SERVE_REQUEST(retlwc, client)
 2:     loop
 3:         if would_block(client) then
 4:             lwSwitch(retlwc, 0);
 5:         else if finished(client) then
 6:             lwSwitch(retlwc, 1);
 7:         else
 8:             serve(client)
 9: function MAIN
10:     descriptors = { accept_ descriptor }
11:     file2lwc_map = { accept_descriptor =¿ root }
12:     loop
13:         next = descriptors.ready()
14:         if next = accept_descriptor then
15:             fd = accept(next)
16:             descriptors.insert(fd)
17:             specs = { ... }                        ▷ Share fd descriptor only
18:             new,caller,arg = lwCreate(specs, ...)
19:             if caller = -1 then                    ▷ context created
20:                 file2lwc_map[fd] = new
21:             else
22:                 serve_request(root, fd)
23:         else
24:             lwc = file2lwc_map[next]
25:             from, done = lwSwitch(lwc, ...)
26:             if done = 1 then
27:                 close(next);close(from)
28:                 descriptors.remove(next)
29:                 file2lwc_map.unset(next)
```

client connection, on line 10 and sets a mapping of the listening socket descriptor to the current *lwC* on line 11.

Once a descriptor is ready the program moves past line 13 and either accepts and encapsulates a new descriptor in a worker *lwC* or resumes execution of a previous one that is now ready. In the former case, the worker's *lwC* is created on line 18 such that no descriptor other than `fd` is passed to it (line 17), the created *lwC* descriptor is mapped on line 20 and the loop resumes. In the latter case, the previously mapped worker *lwC* is retrieved on line 24. This *lwC* is now immediately switched into on the subsequent line. At this point execution resumes on line 18 *in the worker*. As a result, it enters the `serve_request` function on line 22.

When the worker is done executing it switches back into the root *lwC*. It uses the `lwSwitch` argument to indicate whether it is done with its work (arg = 1) or not (arg = 0). When it switches back to the root, control flow resumes at line 25. Depending on the argument passed in from the worker, the root *lwC* either closes the socket and the worker or leaves them intact for later service.

Since all worker *lwC*s obtain a private copy of the root's state, no worker sees session-specific state of other workers. This isolates the sessions from each other.

Sensitive data isolation A third common use pattern isolates sensitive data within a process by limiting access to a single *lwC* that exposes only a narrow interface. As an illustration, Algorithm 6 shows how to isolate a private signature key that is available to a signing function, but kept hidden from the rest of the (large and network-facing) program.

The main function initializes the program and loads the private signing key into the variable `privkey` (line 11). Next, it calls `lwCreate` to create a second *lwC* with the same initial state (line 13). The child *lwC*, which will become the isolated compartment with access to the `privkey`, is granted the privilege to overlay any part of the parent's virtual memory.

The parent *lwC* continues executing on line 16, where it deletes its copy of the private signing key and then revokes its privilege to overlay any part of the child

Algorithm 6 Sensitive Data Isolation

```
 1: function SIGN(key, data, out_buffer)
 2: function SIGN_SSTUB(caller,arg)
 3:     loop
 4:         lwOverlay(caller,{VM,arg,sizeof(arg),SHARE})
 5:         sign(privkey, arg.in, arg.out)
 6:         lwOverlay(caller,{VM,arg,sizeof(arg),UNMAP})
 7:         caller,arg = lwSwitch(caller, 0)
 8: function SIGN_CSTUB(buf)
 9:     caller,res = lwSwitch(child, buf)
10: function MAIN
11:     ...                                          ▷ initialization, load privkey
12:     child,caller,arg =
13:     lwCreate({VM,0,MAX,MAY_OVERLAY}, 0)
14:     if caller != -1 then
15:         sign_sstub(caller,arg)
16:     privkey = 0                                  ▷ erase key
17:     lwRestrict(child, {VM,0,MAX,NO_ACCESS})
18:     loop
19:         ...
20:         sign_cstub(buf)
21:         ...
```

lwC's memory. Any code executed in the parent after this point (line 17) has no way to access the private key. When this code wishes to sign data, it calls `SIGN_CSTUB` passing as argument a structure that contains the data to sign and a large enough buffer to hold the returned signature.

The `SIGN_CSTUB` function performs a `lwSwitch` to the child *lwC*, passing a pointer to the buffer as the argument. The first time the child is switched to, it returns from `lwCreate` with caller != -1 and calls `SIGN_SSTUB` (line 15), from which it does not return.

`SIGN_SSTUB` now uses `lwOverlay` to map the buffer from the parent *lwC* as a shared region into its own address space (line 4), calls the `SIGN` function with the private key, and then unmaps the buffer from its address space. Finally, the function calls `lwSwitch` to return control to the parent *lwC*, which resumes by returning from the `lwSwitch` in line 9. Upon future invocations of `SIGN_CSTUB`, the child *lwC* returns from the `lwSwitch` in line 7 and loops back.

In our evaluation with web servers, we use this pattern to isolate parts of the OpenSSL library that handle long-term private keys, thus protecting the keys from vulnerabilities like the widespread Heartbleed bug [35]. (Heartbleed remains a threat even after global key revocations and reissues [86, 87].)

Protected reference monitor Next, we describe a pattern that allows a parent *lwC* to intercept any subset of system calls made by its child and monitor those calls. In our evaluation, we use this pattern to implement a reference monitor for system calls made by the web server.

Algorithm 7 Reference Monitor

```
1:  function MONITOR(child)
2:      _,call = lwSwitch(child, NULL)
3:      loop
4:          if is_allowed(call) then
5:              spec = { type = CRED, SANDBOX }
6:              rv = lwSyscall(child, spec,
                               call.num, call.params)
7:              out.err,out.rv = errno, rv;
8:          else
9:              out.err,out.rv = EPERM, -1;
10:         _,call = lwSwitch(child, out)
11: function MAIN
12:     specs = { ... } ▷ Share (COW) all but private data
13:     child,c,_ = lwCreate(specs, LWC_SYSTRAP)
14:     if c = -1 then                      ▷ parent becomes refmon
15:         monitor(child)                  ▷ Never returns
16:     privdrop() && run()                 ▷ Child starts here
```

Algorithm 7 shows the pseudocode of the pattern for the case where the monitoring parent is the root *lwC*. On line 13, the root creates a child *lwC* but reserves a private region, which may contain secrets (e.g., encryption keys) of which the child is not allowed to get a copy. The child is created with the flag `LWC_SYSTRAP`, so any system calls that the child lacks the capability for trap to the root *lwC*. Once the child *lwC* is created, the root *lwC* enters the monitoring function, which never returns.

Within the monitoring function, the root, now acting as the reference monitor, yields to the child immediately (line 2). The reference monitor regains control when the child makes a system call that it does not have the capabilities for. The reference monitor checks whether the call should be allowed (line 4) and, if so, makes the call *in the context of the child* (line 6). It yields to the child with the system call's result and error code. If the system call should be disallowed, the reference monitor yields to the child with error code EPERM. The reference monitor loops to handle the next system call.

The child starts execution on line 16 where it immediately drops privileges for all system calls that should be monitored. This causes all these system calls to trap to the reference monitor, which handles them as described above.

For simplicity, our example reference monitor merely filters system calls, a capability already provided by many operating systems. A more interesting monitor could inspect the system call arguments or other parts of the child's state by overlaying in the appropriate regions, or perform arbitrary actions and system calls on behalf of the child.

4.4 Evaluation of *lwC*s

In this chapter, we evaluate *lwC*s using micro-benchmarks, and when applying the usage patterns discussed in Section 4.3 in the context of the Apache and nginx web servers. Our experiments were performed on Dell R410 servers, each with 2x Intel Xeon X5650 2.66 GHz 6 core CPUs with both hyperthreading and SpeedStep disabled, 48GB main memory, running FreeBSD 11.0 (AMD64) and OpenSSL 1.0.2. The servers were connected via Cisco Nexus 7018 switches with 1Gbit Ethernet links. Each server has a 1TB Seagate ST31000424SS disk formatted under UFS.

4.4.1 *lwC* switch

Table 4.2 compares the time to execute a `lwSwitch` call compared to context switching between processes (using a semaphore), between kernel threads (using a semaphore, which we found to be faster than a mutex), and user threads. The user threads use the `getcontext` and `setcontext` calls specified by POSIX.1-2001. A *lwC* switch takes less than half the time of a process or kernel thread switch. The reason is that a *lwC* switch avoids the synchronization and scheduling required for a process or thread context switch, instead requiring only a switch of the VM mapping. Somewhat surprisingly, a kernel thread switch is on par with a process context switch when both use the same form of synchronization. The reason is that the kernel code executed during a switch between two kernel threads in the same process or in different processes is largely the same.

User threads are only moderately faster than *lwC* switches, because in FreeBSD

11, the user context switch is implemented by a system call. In Linux glibc, it is instead implemented in userspace assembly. In an experiment with Linux 3.11.10 on the same hardware, user thread switches run in 6% of the time required by semaphore-based kernel thread switches.

lwC	process	k-thread	u-thread
2.01 (0.03)	4.25 (0.86)	4.12 (0.98)	1.71 (0.06)

Table 4.2: Median switch time (in microseconds) and standard deviation over ten trials.

4.4.2 *lwC* creation

Next, we measured the total cost of creating, switching to, and destroying *lwC*s with default arguments (all resources shared COW with the parent) within a single process. When no pages are written in either the parent or child *lwC* during the lifetime of the child, the system is able to create, switch into once, and destroy an *lwC* in 87.7 microseconds on average, with standard deviation below 1%. This result is independent of the amount of memory allocated to the process. Each page written in either parent or child, however, causes a COW fault, which requires a page frame allocation and copy. When 100, 1000, 10000, and 100000 pages are written in the child during the experiment described above, the average total time taken per *lwC* increases to 397, 3054, 35563, and 34182 microseconds, respectively. Standard deviation was below 7% in all cases. The cost of maintaining a separate *lwC* is approximately linearly dependent on the number of unique pages it creates, and is lowest when *lwC*s in a process share most of their pages.

The results of our microbenchmarks can be used to estimate the cost of using *lwC*s in an application, given an estimate of the rate of *lwC* creations and switches, and the number of unique pages in each *lwC*. Later in this section, we evaluate the overhead of *lwC*s in the context of specific applications: Apache and nginx.

4.4.3 Reference monitoring

Following the pattern described in Section 4.3, we have implemented an in-process reference monitor using *lwC*s. When a process starts, the reference monitor gains control first and creates a child *lwC*, which executes the server application. The child *lwC* is sandboxed using FreeBSD Capsicum and disallowed from using certain system calls, which are instead redirected to the parent *lwC* using the LWC_SYSTRAP option. Our reference monitor restricts access to the filesystem, though other policies that restrict any system call or inspect memory (using `lwOverlay`) can readily be implemented within our basic schema. We compare the *lwC* reference monitor (**lwc-mon**) to two other techniques:

Inline Monitoring (inline) This is a baseline scheme where the reference monitor checks are inlined with the application code. The monitored process is `LD_PRELOAD`ed with a library that intercepts each system call and checks arguments. Inlining provides a lower bound on overhead, but does not provide security since the monitored process can overwrite the checks or otherwise bypass the interception library.

Process Separation (procsep) This method provides a secure reference monitor in a separate process. The monitored process runs in a sandbox based on FreeBSD Capsicum [88]: the sandbox ensures that the monitored process is unable to issue prohibited system calls (e.g. **open**). At initialization, but prior to entering the sandbox, the monitored process connects to the reference monitor process over a Unix domain socket, which it can subsequently use to communicate with the reference monitor, even while sandboxed. All **open** calls (which the sandbox restricts) must be vectored through this socket, which allows the reference monitor to inspect and restrict the access as necessary. If the access is to be granted to the sandboxed application, the reference monitor shares a file descriptor over the socket.

Figure 4.1 shows the overhead of monitoring open, read and write system calls, while an application is accessing a file stored in an in-memory file system. The application calls each system call 10,000 times and we report the average of 5 runs. Faster system calls have higher relative overhead since the fixed cost of redirecting the system call has to be paid. **lwc-mon** does not require data copying or IPC and hence outperforms **procsep** by a factor of two or more.

4.4.4 Apache

Modern web servers are designed to efficiently map user sessions to available processing cores. For instance, the popular Apache HTTP server provides multi-threading using kernel threads (**threads**) in one configuration and pre-forked processes that map to different cores (**prefork**) in another. Higher performance servers,

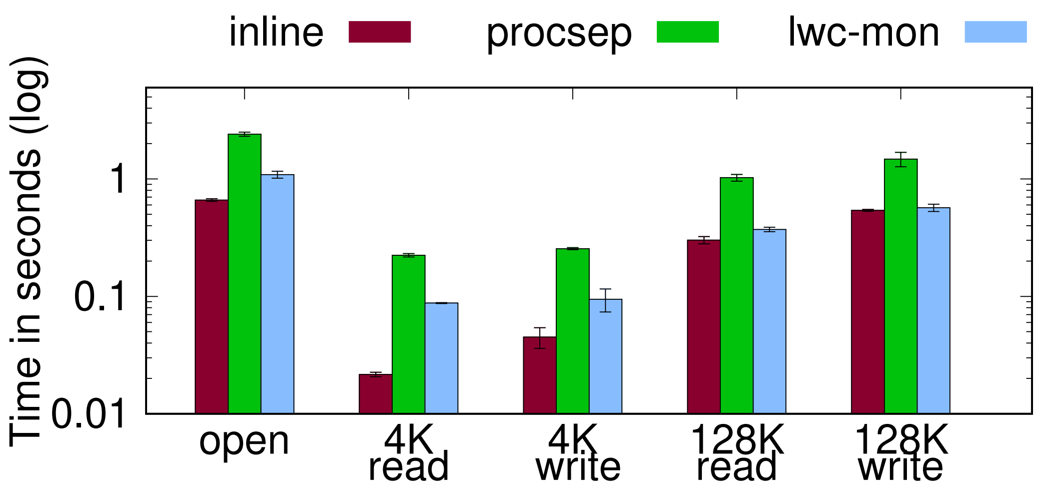

Figure 4.1: Cost of 10,000 monitored system calls in seconds (log scale). Error bars show standard deviation.

such as nginx, use an event loop (based on kqueue or epoll) within a process, and have the option of spawning multiple processes that map to cores, each with their own event loop.

Consider the problem of isolating individual user sessions to separate the privileges of different user sessions or to implement per-user information flow control. None of the above mentioned server configurations provide such isolation: multi-threaded and event-driven configurations serve different sessions concurrently in the same process; pre-forked processes sequentially share among different sessions. Apache can be configured to fork a new process for each user session (**fork**), which provides memory isolation and privilege separation. As our results demonstrate, however, this configuration has low performance for small session lengths, due to the overhead of forking processes[2].

[2]In fact, we had to patch Apache (in server/mpm_common.c) to continuously check the status of child processes (rather than at 1s intervals) to get this configuration to perform at all at small to modest session lengths.

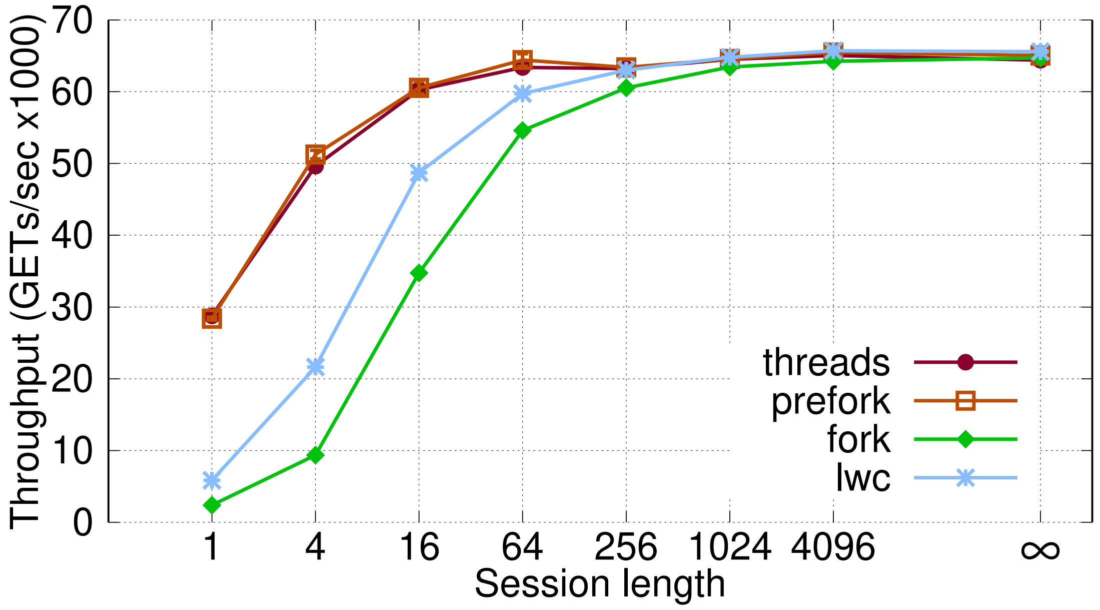

Figure 4.2: Apache throughput in (GETs/sec) of 128 concurrent clients, 45 byte docs, over HTTP. Error bars show standard deviation, which was below 3.7%.

*lwC*s can provide memory isolation, privilege separation, and high performance. We have augmented the pre-fork mode in Apache (version 2.4.18) to provide session isolation using the snapshot and rollback pattern from Section 4.3. Within each Apache process, we create a *lwC* that serves a user session; when the session ends, the *lwC* switches (reverts) to its initial (untainted) state before serving the next user session, thereby ensuring the isolation property.

In the following set of experiments, we use ApacheBench (`ab`) to issue HTTP and HTTPS requests to our Apache server. We modified `ab` to support varying client session lengths by using HTTP Keepalive and terminating a session after a certain number of requests. We launch a single ApacheBench instance which repeatedly makes up to 128 concurrent requests for a small 45 byte document. We chose small document requests to make sure the results are not I/O-bound.

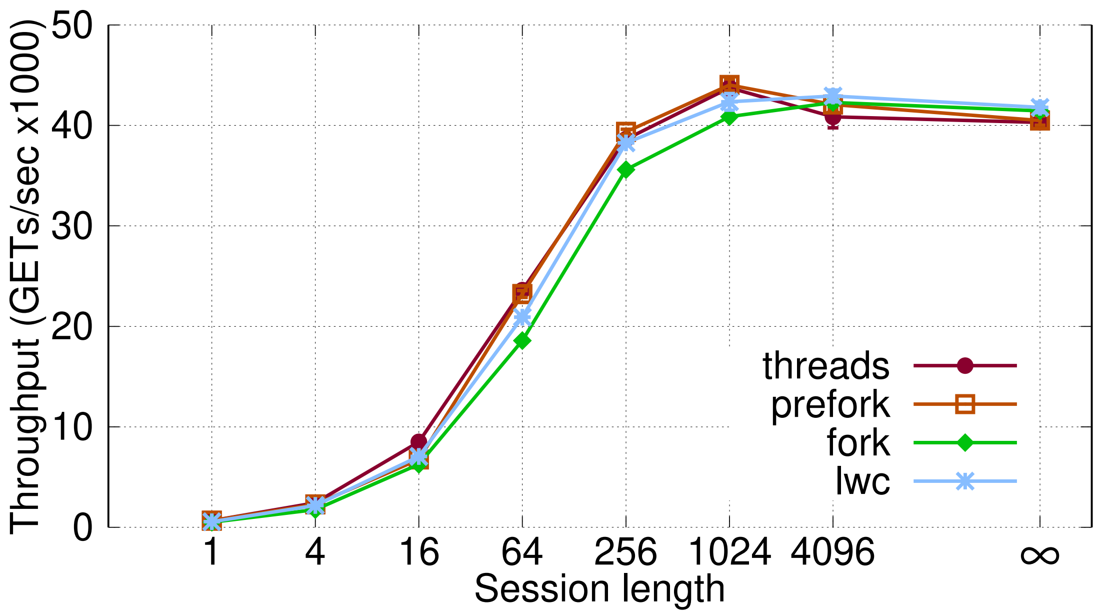

Figure 4.3: Apache throughput in (GETs/sec) of 128 concurrent clients, 45 byte docs, over HTTPS. Error bars show standard deviation, which was below 3.7%.

Figure 4.2 and Figure 4.3 show the number of GET requests served per second by the different Apache configurations at different session lengths, and for HTTP and HTTPS respectively. For HTTPS, the server uses TLSv1.2, ECDHE-RSA-AES256-GCM-SHA384 with 4096 bit keys. The results were averaged over five runs of 60 seconds each.

At session length ∞, each client maintains a session for the duration of the experiment. The **threads** and **prefork** configurations, which provide no isolation, perform comparably for all session lengths and protocols. **fork** and **lwc** configurations provide isolation: **lwc** has better throughput in all cases, and has a significant advantage for short sessions (256 and below), particularly for HTTP. (In HTTPS, the high CPU overhead for session establishment dominates overall cost; however, emerging hardware support for crypto will diminish these costs, exposing once again the costs of isolation.) Moreover, **lwc** achieves performance comparable to the best configuration *without isolation* for sessions lengths of 256 and larger.

We also repeated the experiment with GET requests for 900 byte documents. These documents are 20x larger but still small enough not to saturate the network link. The trends and relative throughput between the different configuration were very close to those in Figure 4.2 and Figure 4.3, with the absolute peak throughput within 10%.

We have integrated reference monitoring within Apache (and nginx). Figure 4.4 shows the throughput of Apache **prefork** in different reference monitor configurations when used to serve short (45 byte) documents. The results were averaged over five runs of 20 seconds each. In this experiment, the **open** and **stat** system calls are monitored and checked against a whitelist of allowed directories. These results show that a reference monitor implementation based on in-process *lwC* incurs lower overhead than an implementation based on process separation even for large applications where the monitored system calls constitute only part of what the applications do. The overhead of reference monitoring increases with session length due to the increase in relative number of reference monitored system calls (open and stat) compared to other system calls (accept, read, send, close).

4.4.5 Nginx

To enable session isolation in nginx (version 1.9.15), we allocate a *lwC* for each new connection: each event for a single connection is isolated within the *lwC*,

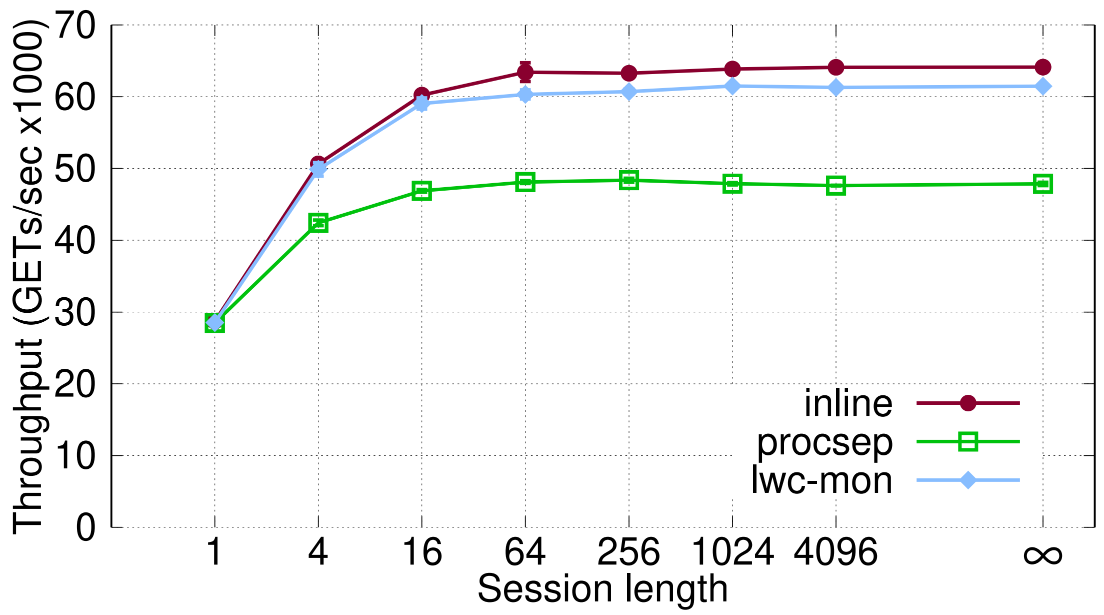

Figure 4.4: Throughput of different Apache reference monitoring configurations in (GETs/sec) of 128 concurrent clients, 45 byte docs. Error bars show standard deviation, which was below 2%.

following the session isolation pattern from Section 4.3. Note that in the nginx case, each process may serve many different connections simultaneously, and our implementation creates a *lwC* per active connection within the process. We have also integrated a reference monitor with nginx.

We experiment with different nginx configurations: the stock **nginx**, **lwc-event** augments nginx's event loop to create a new *lwC* per connection, and **lwc-event-mon** combines a reference monitor with the per-connection *lwC*. In each case we configured nginx to use 10 worker processes, as we found that this had the best performance. We launch four ApacheBench instances, each of which repeatedly makes up to 75 concurrent requests for a small 45 byte document.

Figures 4.5 and 4.6 shows the average number of queries served by each of the configurations over five runs of 60 seconds each for HTTP and HTTPS respectively.

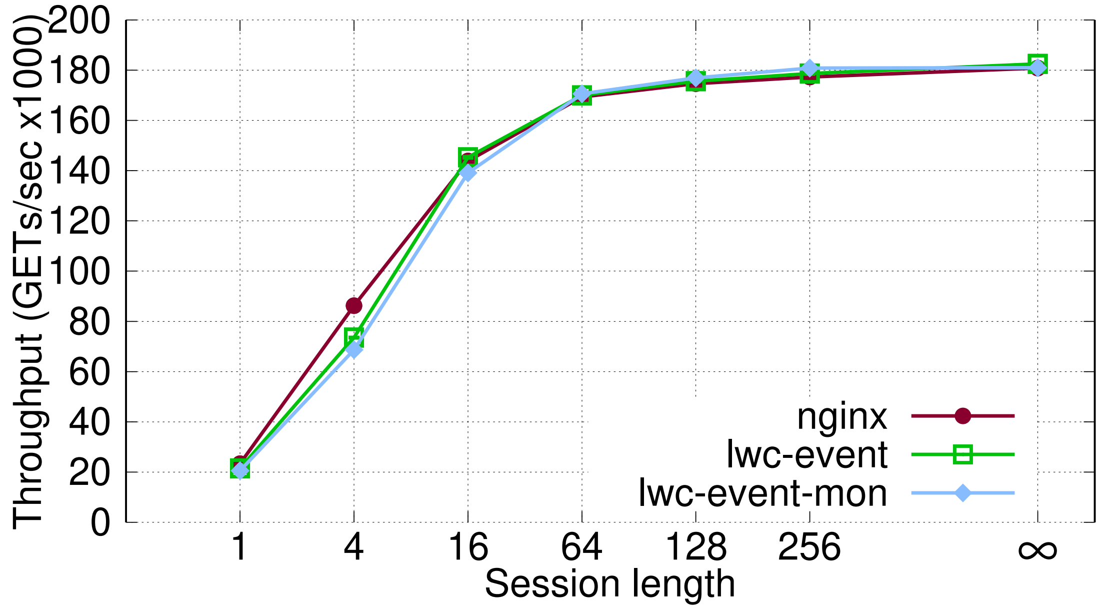

Figure 4.5: Nginx throughput in GETs/sec for HTTP requests with 10 workers, 45B documents, 300 concurrent requests. Error bars show standard deviation, which was below 0.9%.

The standard deviation did not exceed 0.9%.

nginx is considered the state of the art high-performance server. It uses a highly optimized event loop and is about 2.88x quicker than Apache. Introducing *lwC*s in this base configuration (named **lwc-event** in the results) has no significant impact on the throughput of this high-performance configuration. Similarly, reference monitoring adds only minimal overhead. For both HTTP and HTTPS, with isolation and reference monitoring, *lwC*-augmented nginx performs comparably to native nginx.

Large scale servers may need to maintain tens of thousands of concurrent user sessions. Using *lwC*s for session isolation increases the amount of per-session state. Therefore, our next experiment explores how using *lwC*s for session isolation affects nginx's performance under a large number of concurrent client connections.

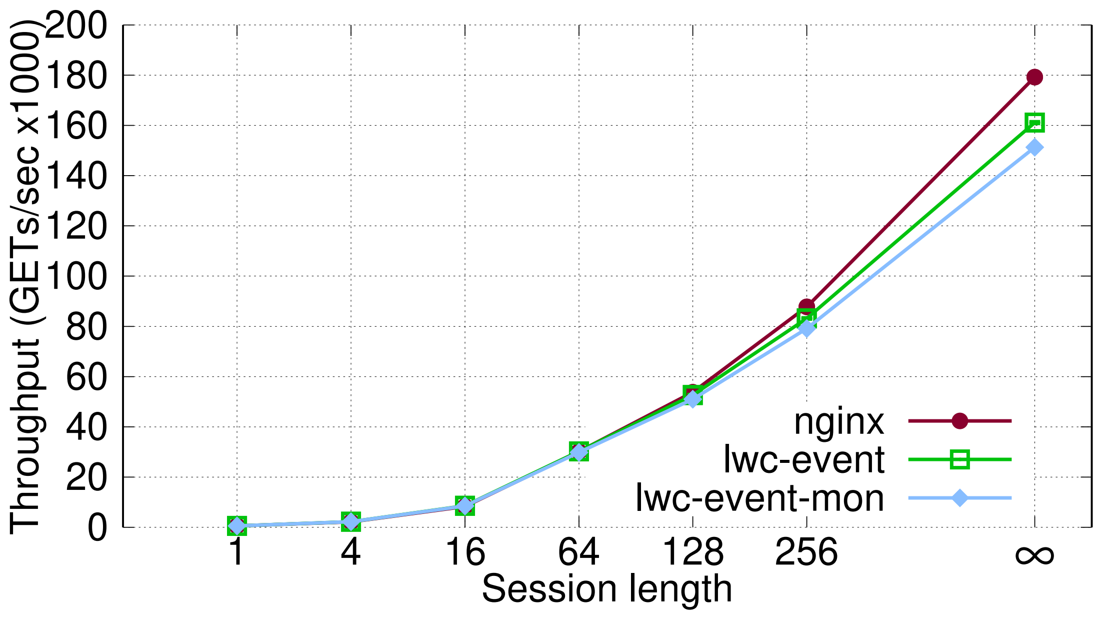

Figure 4.6: Nginx throughput in GETs/sec for HTTPS requests with 10 workers, 45B documents, 300 concurrent requests. Error bars show standard deviation, which was below 0.9%.

We experimented with two configurations: in the first, we use between 6 and 76 ApacheBench instances, and each instance issues 250 concurrent requests for a 45 byte document. The session length was 256 and we used 10 nginx workers. The second configuration is identical except the ApacheBench instances request 900 byte documents.

Figure 4.7 shows the average number of requests served, over 5 runs of the experiment, as a function of the number of client sessions for stock nginx and **lwc-event** for both file sizes.

For small documents, **lwc-event** matches the performance of native nginx up to 6500 clients. Beyond, the performance of both configurations declines following the same trend, but the absolute throughput of **lwc-event** falls below that of nginx by up to 19% at 19,500 concurrent clients. In investigating this result further,

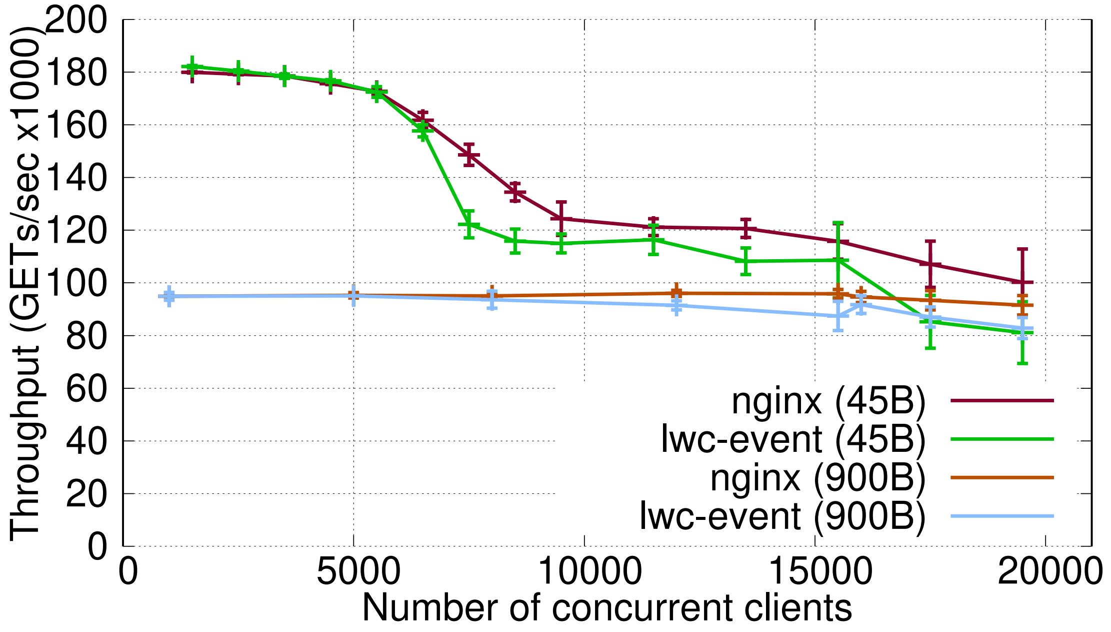

Figure 4.7: Nginx cumulative throughput in GETs/sec with 10 workers, session length 256, 45B and 900B documents, increasing number of concurrent clients. Error bars show standard deviation.

we find that FreeBSD kernel threads, in particular, the interrupt handler thread, gets CPU bound after 6500 clients, and the CPU consumption of the nginx worker threads *reduces* with higher numbers of clients as the nginx worker threads block waiting for the kernel to demultiplex packets. The **lwc-event** configuration further pays an extra cost of *lwC* switches, which reduces performance compared to stock nginx. However, given that **lwc-event** provides session isolation, this is a still a strong result.

For 900 byte documents, the performance of stock nginx and **lwc-event** remain similar until ~12000 simultaneous clients. Performance of stock nginx is not affected by increasing numbers of clients: this is because the rate of incoming requests is lower, which means the kernel threads do not saturate the CPU. With increasing numbers of clients, eventually the cost of *lwC* switches, which were amortized over serving a larger document, become a measurable factor.

Overall, our results show that using *lwC*s, it is possible to implement features such as session isolation and reference monitoring at low cost for both HTTPS and HTTP sessions, and even in a high-performance server under a challenging workload.

4.4.6 Isolating OpenSSL keys

*lwC*s provide a particularly effective way to isolate sensitive data from network-based attacks such as buffer overflows or over-reads. The sensitive data is stored in a *lwC*, within the process, such that the network-facing code has no visibility into pages that store the sensitive data. In this way, unless the kernel is compromised, the data is guaranteed safe, but access to functions that require the data can be rapid, using a safe *lwC*-crossing interface.

As an example, we have isolated parts of the OpenSSL library that manipulate secret information within Apache and nginx. In our case, the web server certificate private keys are isolated; note that such a scheme would have rendered attacks such as Heartbleed completely ineffective since the buffer over-read that Heartbleed relied on would not have visibility into the memory storing the private keys. We evaluate this scheme using the following configurations:

In-process LwC Sensitive data is stored in a *lwC* within the process, following the pattern from Algorithm 6 in Section 4.3. The network-facing code within the process has no visibility into the sensitive data; access is through a narrow interface exported via *lwC* switch entry points. The isolated *lwC* has a copy of the original process at the time of creation and may call whatever functions are available within its address space. Our encapsulated OpenSSL library takes advantage of this fact because the isolated *lwC* hosts a COW copy of the OpenSSL code and global state and need not be aware that it is running in a restricted environment. None of the changes in the sensitive *lwC* are visible to the network facing code.

We evaluate the cost of providing this isolation by performing SSL handshakes (TLSv1.2,ECDHE-RSA-AES256-GCM-SHA384 with 4096 bit keys) with the nginx web server. The server was configured to spawn four worker processes. We used ApacheBench with concurrency level 24 and a session length of 1. In our experiments, native nginx required 99.7 seconds to complete ten thousand SSL handshakes, whereas the configuration with a *lwC* isolated SSL library required 100.4 seconds. With *lwC*s, isolating SSL private keys is essentially free.

Our prototype isolates only the server certificate private key, but not session keys or other sensitive information. More fine-grained isolation of the OpenSSL state, such as that described in [36], can be implemented readily using *lwC*s.

4.4.7 FCGI fast launch

We demonstrate the utility of *lwC* snapshotting by adding a "fast launch" capability to a PHP application. When a PHP request is served, a PHP script is read from disk, compiled by the interpreter, and then executed. During execution, other PHP files may be included and executed. We modified the PHP 7.0.11 programming language to add a `pagecache` call that allows the script to "fast-forward" using previous snapshots. Our implementation augments PHP-FPM [89], which functions as a FCGI server for nginx. Our test application is based on the MVC skeleton application that is included with the Zend PHP framework [90], which provides the core functionality for creating database-backed web-based applications such as blogs.

Before a PHP script performs any computation that depends on request-specific parameters (e.g., cookie information), the script may invoke the `pagecache` call, which implements the snapshot pattern (Algorithm 4). The first time a `pagecache` is invoked, we take a snapshot and then revert to it on subsequent requests to the same URL, effectively jumping execution forward in time. We use a shared memory segment to store data that must survive a snapshot rollback, including request-specific data and network connection information.

Our experiments run PHP-FPM with 11 workers. PHP itself includes an opcode cache (which caches the compilation of each script in memory) and our results include configurations where the PHP opcode cache is enabled and not. When combining the opcode cache and the *lwC* snapshot, we warm up the opcode cache before taking the snapshot. The results in Table 4.3 are an average of five runs and overall standard deviation was less than 2%.

stock php no cache	*lwC* php no cache	stock php cache	*lwC* php cache
226.1	615.8	1287.5	1701.4

Table 4.3: Average requests per second over 60 seconds with 24 concurrent requests.

With or without the opcode cache, the *lwC* snapshot is able to skip over much of the initialization of the runtime and whatever PHP execution would otherwise occur before the `pagecache` call. This result is remarkable in that it shows *lwC*s can provide significant performance benefit to highly optimized end-to-end applications such as web frameworks, *while adding isolation between user requests.*

4.5 Conclusion

The abstraction provided by *lwC*s enables higher security when used for isolation or reference monitoring and better performance when used for memoization with OS supported snapshots, yet *lwC*s could not be efficiently implemented with the abstractions provided by FreeBSD. Instead, an efficient implementation of *lwC*s requires access to new, low-level abstractions that decouple isolation, execution state, and privilege separation from the process abstraction. These low-level abstractions can then be composed in novel ways to create new high-level abstractions, such as *lwC*s. To maintain compatibility with traditional process isolation, the only requirement is that the low-level abstractions do not expose information that would not otherwise be available to a calling process. In Section 5.5, we show how *lwC*s could be implemented in the null-Kernel architecture, which allows both low and high-level abstractions to be safely exposed simultaneously by codifying this restriction.

Chapter 5: The null-Kernel

In this chapter, we present a null-Kernel, a new OS architecture that supports applications and abstractions with efficiency, usability, and security goals that may at times be in conflict. The null-Kernel achieves this by providing a safe way to simultaneously expose both low and high-level OS abstractions simultaneously.

5.1 Introduction

Abstractions imply choice, one that OS designers must confront when designing a programming interface to expose. Traditional monolithic kernel designers choose a high-level portable interface that necessarily hides many hardware details. On the other extreme, designs such as the Exokernel optimize for performance and low-level hardware access. Abstraction choices have further implications in how easily and quickly new hardware and application models can be supported. In this chapter, we describe the null-Kernel, a new model for structuring system software that attempts to relieve OS designers of this choice and enable access to and composition of OS interfaces at different levels of abstraction.

The null-Kernel is designed to address the growing need to easily provide high-level programming interfaces to new hardware, such as GPUs, crypto/AI accelerators, smart NICs/storage devices, NVRAM etc., and to efficiently support new application requirements for functionality such as transactional memory, fast snapshots, fine-grained isolation, etc., that demand new OS abstractions that can exploit existing hardware in novel and more efficient ways.

At its core, the null-Kernel derives its novelty from being able to support and compose across components, called Abstract Machines (AMs) that provide programming interfaces at different levels of abstraction. The null-Kernel uses an extensible capability mechanism to control resource allocation and use at runtime. The ability of the null-Kernel to support interfaces at different levels of abstraction accrues several benefits: New hardware can be rapidly deployed using relatively low-level interfaces; high-level interfaces can easily be built using low-level ones; and applications can benefit from being able to use interfaces, and compose abstractions using more than one if necessary, as appropriate.

The null-Kernel capability system can also be used to partition hardware between different components (AMs) that provide different abstractions. For instance, the null-Kernel can be used to simultaneously support a traditional OS that provides a system call interface, and an exokernel that provides low-level access to new hardware. Such a system could easily enable optimizations not possible now: for instance, the BSD Socket interface necessarily implies copying incoming data from kernel to user space memory, and typically an additional copy is incurred when the data is initially copied from the NIC. Zero copy stacks exist, but do not eliminate the initial copy. A null-Kernel that combines a BSD-like AM and an exokernel interface for Smart NICs can be programmed to assemble incoming TCP segments and copy them directly into process memory, bypassing the BSD AM entirely. Applications can use the BSD interface for traditional high-level services while simultaneously benefiting from a very high performance networking stack.

We describe design principles for AMs that go beyond strict partitioning, and enable cooperating AMs to provide additional functionality. These design principles can be used to retrofit existing kernels, allowing applications to simultaneously use the high-level interface they provide while benefiting from additional access to low-level hardware features that were previously hidden by the traditional OS. For instance, cooperating high- and low-level AMs can simultaneously provide virtual memory (high-level) and access to page-access bits (low-level, currently unavailable). Such a facility could be used to implement new primitives (e.g., software transactional memory) or optimize existing (e.g., garbage collection).

In the next section, we describe the null-Kernel structure, its capability system, and how interfaces provided by AMs can be composed using the null-Kernel. In Section 5.3 we discuss how an existing kernel can be retrofitted to interface with an exokernel, and provide examples of how new types of application primitives such a hybrid system can support. We discuss related work in Section 2.3.

5.2 The null-Kernel

Figure 5.1 shows a high-level schematic of the null-Kernel. The null-Kernel architecture decomposes the system into three components: abstract machines (AMs),

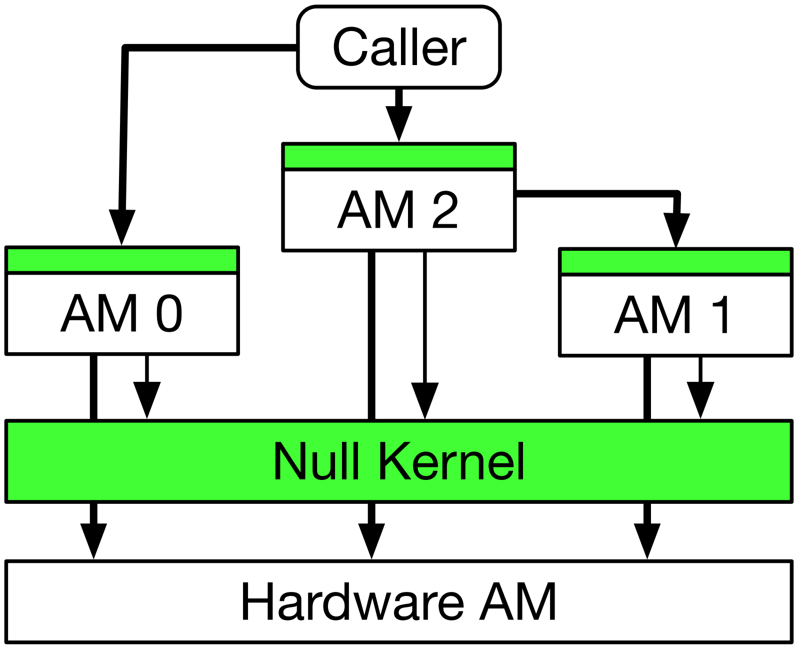

Figure 5.1: An overview of the null-Kernel showing system components: the null-Kernel, abstract machines, and callers.

callers, and the null-Kernel itself.

Abstract machines are software layers that provide specific functionality, and expose a set of operations that *callers* may invoke. In a traditional OS, the kernel is the AM, and this set of operations is the system call interface. Callers are processes or threads, as recognized by the kernel. In a null-Kernel architecture, there may be other layers of software (i.e., different AMs) that provide different interfaces, which would also be available to eligible callers, which may include other AMs.

The null-Kernel, shown in green in the figure, controls access to AM operations by only allowing invocations when the caller presents capabilities with sufficient access rights. The capability structure supported by the null-Kernel is extensible: AMs define new capabilities and specify which access rights are required for any given AM operation. Since the capability system is extensible, the null-Kernel can recognize new operations (and indeed complete AMs) at runtime.

null-Kernel Structure Figure 5.1 shows how OS software in the null-Kernel model is structured. The hardware presents a programming interface, which we term the Hardware-AM[1]. The other AMs in the figure export different sets of operations that ultimately make use of the Hardware-AM. AMs can be layered, e.g., AM-2 is partially built using AM-1's operations. In a null-Kernel, callers, with proper capabilities, may invoke operations exported by a "high-level" AM such as AM-2, or by "low-level" AMs such as AM-0, or any combination simultaneously. This is the key insight behind the null-Kernel: as long as a caller has proper capabilities, they may invoke operations at any level of abstraction, and thus the OS architecture is not confined to one model. More importantly, if the underlying resources are disjoint, or if the AMs cooperate (as described next), these calls compose, and can safely be executed in parallel or in any combination.

AMs structured with the null-Kernel capabilities permit many patterns of resource access and optimizations that are either cumbersome or impossible otherwise. These include "bypassing" layers by delegating capabilities and controlled sharing of resources between "peer" AMs. Next we describe the capability subsystem in more detail followed by examples demonstrating these access and optimization patterns.

[1]Obviously, the Hardware-AM is not "abstract" but we (ab)use the term for uniformity.

5.2.1 null-Kernel Capabilities

In this section, we describe the null-Kernel capability system in more detail. AMs, including the Hardware-AM, define “objects” and “access rights” on objects. The null-Kernel capability system is extensible in that it operates over (dynamically defined) AM objects and rights. Capabilities are unforgeable references to a pair consisting of an AM object and a set of access rights on that object. Operations defined by AMs refer to one or more pairs of objects and access rights. For example, the Hardware-AM may define a memory page as an object, and `read` and `write` as access rights. A DMA operation that copies data onto a page would require the `write` access right on that page object. This requirement is reflected to the null-Kernel as described below; a caller may invoke an operation (DMA-write) only if they have the capabilities associated with the operation (in our example, the caller must have a capability that grants the `write` right to that memory page).

The null-Kernel capability system derives directly from prior work in capabilities [72, 91]. Like existing systems, in the null-Kernel, capabilities can be associated with object, rights pairs, delegated to others, derived to produce weaker capabilities (by reducing the rights set), and revoked. In the null-Kernel, when a capability is revoked, all derived capabilities are also revoked. Much like other capability systems, the basic security of the null-Kernel requires that a principal (a caller or AM) can only get access to a capability by either being granted the capability explicitly or deriving it from a stronger capability. In particular, colluders cannot grow their collective set of capabilities beyond what is explicitly granted to them.

Extensibility The novelty of null-Kernel capabilities derives from the fact that null-Kernel itself does not associate capabilities to the operations they guard. This association is made by AMs and is, therefore, extensible. Specifically, AMs can define new objects at their level of abstraction (e.g., the Hardware-AM can define memory pages as objects, whereas a higher-level VM-AM that provides virtual memory can define address-spaces and memory regions as objects). AMs also define custom rights on objects, and this set too is extensible at runtime. Again, as an example, both the Hardware-AM and the VM-AM can define `read` and `write` as rights on their respective objects (physical pages for the Hardware-AM, memory regions and address spaces for the VM-AM). The operations supported by the low-level Hardware-AM mirror those of the access rights (the read/write operation succeeds only if the caller has `read/write` access to a memory page). The VM-AM can associate much richer semantics with operations: for example, it may define a **mapReadable** function that takes an address space and a memory region as input, and the caller may only map a memory region into an address space if they have capabilities that provide `write` on the address space and `read` on the memory region. The null-Kernel provides an API that allows AMs to express capability requirements for each call. As long as the capabilities are delegated correctly any caller at any 'layer' of the system may use operations exported by an AM. The invoked AM maintains its correctness as long as the capabilities are checked prior to the operation being executed.

Capability hierarchies and delegation The null-Kernel naturally allows AMs to build and, in turn, export interfaces based on capabilities received from lower layers. These exported interfaces (and their associated capabilities) implicitly form a capability hierarchy. Hierarchical capabilities are different from simple delegation in which an AM directly grants received capabilities to others. (Delegation is useful for the layer bypassing model we discuss later.) For both hierarchical and delegated capabilities, AMs should follow two basic principles to ensure correctness for higher-layers:

- Logical separation: An AM should give potentially conflicting capabilities (e.g., write capabilities to the same object) to mutually trusting principals only (principals who understand each other's invariants).
- Essential capability hiding: A higher-level AM should not give out any capability it has on a lower-level AM, if the capability can be used to violate the higher-level AM's own invariants.

Capability Checks It is crucial to note that the null-Kernel, as described, is a schematic for how OSs should be structured. This schematic does not specify *how* capability checks are implemented, only that operations across AMs should be guarded by checks. This lack of specificity of implementation is on purpose since it provides unconstrained latitude in how (and when) the checks are implemented. For example, capability checks could be implemented in hardware (using the ISA [47,92], MMU, processor protection rings [3,72]), with programming language techniques (a safe compiler only generates code for capabilities it is provided [69]), using virtualization (guest OSs implement AMs constrained using the hypervisor interface [93]), and so on. Similarly, capability unforgeability can also be implemented using different mechanisms: EROS [72] protects capabilities using protection rings, whereas Amoeba [94] uses random placement of capabilities in a sparse address space. Other systems [95] [96] protect capabilities with cryptography primitives. The null-Kernel could employ any or all of these methods.

5.2.2 null-Kernel Structures

We conclude this section with two examples of how null-Kernel capabilities can be used to create interesting optimizations and sharing structures between AMs.

Layer Bypassing Consider a system (Figure 5.2) that exposes both a high-level filesystem AM (fs-AM) that operates on the level of files and directories, as well as a low-level disk AM (d-AM) that operates at the level of blocks. The fs-AM is implemented on top of the d-AM using raw block read/writes exported by the d-AM.

Most callers may prefer to use the file system through the fs-AM. However, applications, such as high performance databases, that want low level control over how data is arranged, may use the d-AM directly. With a null-Kernel, both these cases can be supported simultaneously by exposing both AMs to callers, subject to constraints of hierarchical and delegated capabilities. In particular, following the principle of "logical separation", the d-AM should give the direct callers and

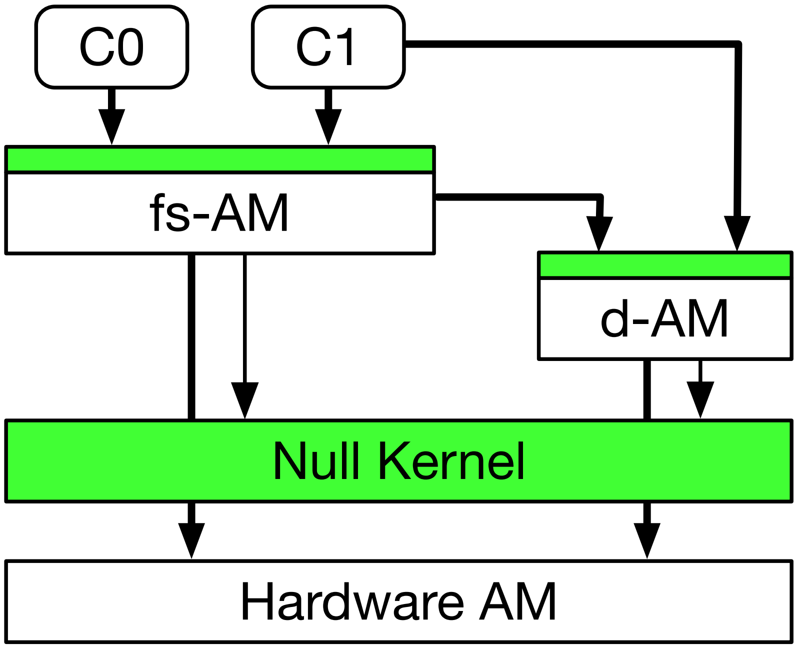

Figure 5.2: A representation of a file system AM built on top of and exposing capabilities for a disk AM.

the fs-AM capabilities to disjoint disk blocks to ensure that they do not overwrite the other's data. Indeed, in existing systems, raw disks or partitions are often provided exclusively to high performance applications for exactly this reason (and with exactly this constraint).

A more interesting use-case is that the fs-AM itself can delegate block capabilities it receives from the d-AM to its callers. This would enable applications to write to file-system managed data blocks directly without going through the fs-AM. The null-Kernel enables such *layer bypassing* since it allows any caller with the appropriate capabilities to call any AM (the d-AM in this case). In this case, the fs-AM must adhere to the principle of "essential capability hiding" by never delegating write capabilities that pertain to file system metadata blocks to guarantee file system integrity.

AM peering The null-Kernel also supports non-hierarchical, peering structures between AMs. We illustrate this using VM paging as an example. Consider the virtual memory AM (VM-AM). Upon memory pressure, the VM-AM writes pages to disk. To accomplish this, we assume that the VM-AM has been delegated write capabilities to a set of disk blocks by the disk AM (d-AM). The VM-AM uses these capabilities to write pages to disk as needed. To page these items back in, the VM-AM invokes an operation in the d-AM that requires a block capability with read access and a page capability with write access. The d-AM may then asynchronously write into the page from the block and notify the VM-AM when the operation has completed. This peer-to-peer interaction between cooperating AMs is natively supported by the null-Kernel.

5.3 null-Kernel in Practice

In the previous section, we outlined the basic structure of the null-Kernel and described use cases where the relevant subsystems were written to conform to our model. Many null-Kernel ideas, however, are applicable to current OSs as well; in this section, we describe how salient parts of the null-Kernel can be integrated into production kernels and the types of optimizations this can enable.

Figure 5.3 depicts a standard OS, such as FreeBSD, extended to recognize the null-Kernel as we describe next. The system also includes a new AM, the EXO AM, which exports a low-level interface to hardware, similar to that provided by exokernels. FreeBSD is not structured as a null-Kernel AM and there are several

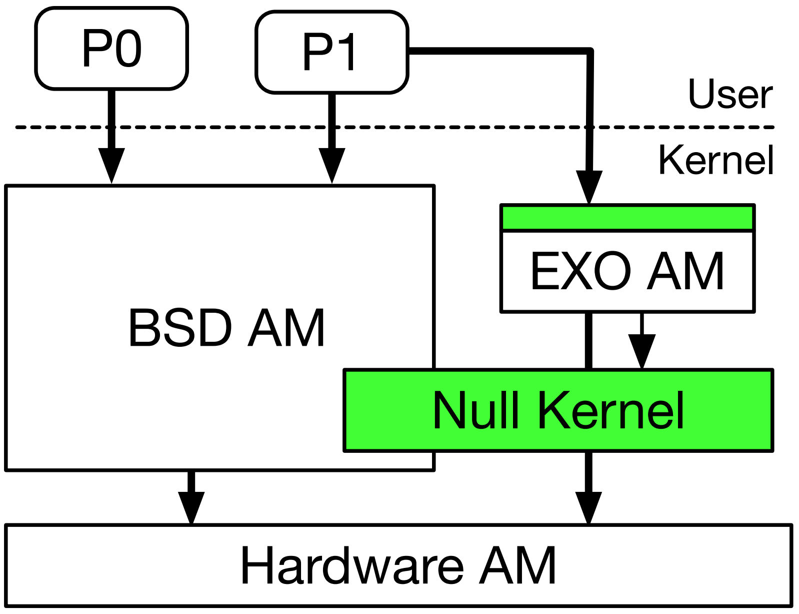

Figure 5.3: Architecture for retrofitting the null-Kernel into a BSD system to expose include safe exokernel like AM.

options as to how an EXO AM could co-habit with FreeBSD. One option to give the EXO AM access to the hardware is to let it run in supervisor mode, alongside the BSD AM.

Callers in this hybrid system are BSD processes, augmented with capabilities which can be used to access the EXO AM. Processes (which run in processor ring-3) calling into the EXO AM must incur a processor ring switch, and hence the "user-kernel boundary" separates processes from the EXO AM as well.

In this structure, the BSD AM and the EXO AM cooperate, and must share the hardware capabilities without conflict. For instance, the BSD AM could choose to not use its hardware capabilities for certain devices. The EXO AM can safely export its minimal interface and be used as a base for higher level abstractions on these devices. With more cooperation, the EXO AM could also provide read-only access to hardware primitives that are used by the BSD AM (e.g., by exporting processor status and memory reference bits). Such hybrid access to high- and low-level interfaces enables new use patterns that are not possible with either interface in isolation.

Access to new hardware The EXO AM can provide low-level access to new hardware such as GPUs, FPGAs, or smart-NICs for which the BSD kernel does not have support. New devices added to the machine would add additional hardware AMs to the system. Hardware vendors or kernel developers would then write a thin abstraction of the hardware AM and expose it via the EXO AM. At this point, the new hardware could be directly used by processes (with proper capabilities).

New Abstractions The ability to layer AMs would give us the opportunity to build higher level AMs in terms of the EXO AM. These higher level AMs would offer different abstractions that might be suitable for the hardware, and each application that wanted to use the new hardware could choose the AM that best meets its needs.

New abstractions need not be limited to new hardware. For instance, in co-operation with the BSD AM, the EXO AM could expose hardware features such as page reference bits in page tables which are usually hidden by the BSD AM. These tracking bits could be used by applications to augment the BSD VM subsystem and implement novel features such as efficient software transactional memory (TM) or fast garbage collection. Currently, a generic software implementation of TM requires compiler augmentation of every single memory access [97]. This overhead can be entirely avoided if page reference bit were made available through the EXO AM.

Simultaneous high- and low-level access The hybrid BSD/EXO AM system can be used to implement layer bypassing as discussed earlier. The BSD AM could provide capabilities for disk blocks to processes, which could then use the EXO AM to implement their own optimizations *within* the blocks allocated by the BSD filesystem.

High-level AM over different low-level AMs The hybrid system would allow different AM's functionality to enable new use cases. For example, suppose new hardware in the form of NVRAM storage devices is available, and the EXO AM exports a low-level block interface to these devices. A higher-layer AM could provide a memory-mapped file interface to the NVRAM storage, and process logic could use this facility to implement efficient crash recovery. Here the null-Kernel allows programmers to use a high-level, well-understood paradigm (memory-mapped files) to program their application logic, and integrate it with low-level access to new hardware to implement new functionality (efficient crash recovery).

AM composition The examples above assume that either AMs partition resources, or are able to expose safe "enough" interfaces such that composite services, that use operations from multiple AMs do not cause deadlock or fault the system in some other manner. A sufficient condition to ensure both safety and progress is for each exported AM call to run to completion upon invocation, and for the AM to maintain all of its safety and progress invariants (e.g., release held locks) prior to call return (including for calls that it services in parallel). The OS system call interface maintains such an invariant, but internal kernel interfaces, that assume specific locking sequences and at times undocumented pre-conditions, do *not*, thereby making kernel modifications fraught with danger. To support composability, AMs could simply implement the sufficient condition we have described. Articulation of more precise and efficient criteria is likely feasible and remains part of our future work.

5.4 NVRAM and the null-Kernel

In this section, we will discuss how NVRAM can be supported in a null-Kernel in a way that makes PTx possible with lower overhead and more flexibility than offered in existing systems.

PTx, discussed in Chapter 3, acts as an abstraction for efficiently persisting in-core data structures over NVRAM hardware. To persist efficiently, PTx relied on newly added Linux support for direct hardware access, which is a form of layer bypass in terms of the null-Kernel architecture. PTx uses direct access to persist to NVRAM without the intervention or overhead of other software layers.

Existing POSIX abstractions do not support direct access, so stock Linux was modified to support direct access via a modified `mmap` system call. In addition to the mmap system call, maintainers modified the block device, file system, and virtual memory system. In this section, we discuss how PTx could have been implemented in a null-Kernel architecture with lower overhead and less programmer intervention.

Note that kernel subsystems already export interfaces that, in effect, act as AMs. Interoperability between subsystems depend on programmer discipline: so long as the interface is used as expected, the kernel can be extended, but unspoken contracts and requirements make doing so difficult. We consider those interfaces our starting point as the AM interface, but note that in a null-Kernel, these interfaces would be exposed to users and safety and compatibility constraints would be enforced by capabilities, rather than the use of convention and programmer discipline.

There are multiple ways to interact with these new AMs, but performance likely requires that abstractions built in terms of subsystem AMs be executed in supervisor mode. Towards that end, we posit that users have the ability to inject small programs that are written in a memory safe language and have progress guarantees that are then injected into the kernel's address space. These programs could invoke internal AMs and can expose new abstractions to user space by adding entries to the system call AM. The enforcement of capabilities and progress guarantees will be ensured by the invoked AMs and programming language restrictions guarantee memory safety.

To support direct access, the virtual memory AM must be modified to support capabilities for NVRAM. This call may be as simple as `allocate-nvram-region(size)`, which allocates a region of NVRAM pages that may then be mapped into an address space with a `map-region(region,aspace, offset)`, that maps either NVRAM or DRAM pages at a given address space and offset. So long as the virtual memory system does not allocate the same NVRAM regions to multiple callers, this call is safe and is the only modification necessary in a null-Kernel. To gain direct access to hardware, a caller requests an NVRAM capability from the virtual memory system and then requests that this capability be mapped in an address space. This form of direct access significantly lowers the bar in supporting new hardware.

However, the form of direct access differs from that offered in Linux, which makes NVRAM accessible through the filesystem. In this form, the access control mechanism and namespace for NVRAM is specified by the filesystem abstraction. With the null-Kernel, we have the flexibility to define a new abstraction that provides a namespace and access control that may better fit a specific use case, but if the file system abstraction is useful the null-Kernel architecture can support it through the `mmap` call.

To support direct access `mmap`, we need to consider how file systems would likely handle the buffer cache in a deconstructed Linux null-Kernel. The buffer cache is a cache of file contents in system memory that can be mapped or copied into a process's address space in response to a read fault over a memory mapped region. A reasonable null-Kernel file system could either 1., request pages from the memory AM and copy file contents into those pages whenever requested by the memory AM in response to a read fault within the memory mapped region or 2., fill in a buffer that will be copied into user space by the memory AM in response to a read fault within the memory mapped region. In the former case, shown in Figure 5.4, the file system AM can support direct access by providing AM backed pages to the memory AM in response to page in requests by the memory AM. In the latter case, both the memory and file system AM would need to be modified to support direct mapping.

Note that existing in-kernel abstractions were nearly capable enough to support layer bypass for NVRAM, but users had to wait until kernel developers were

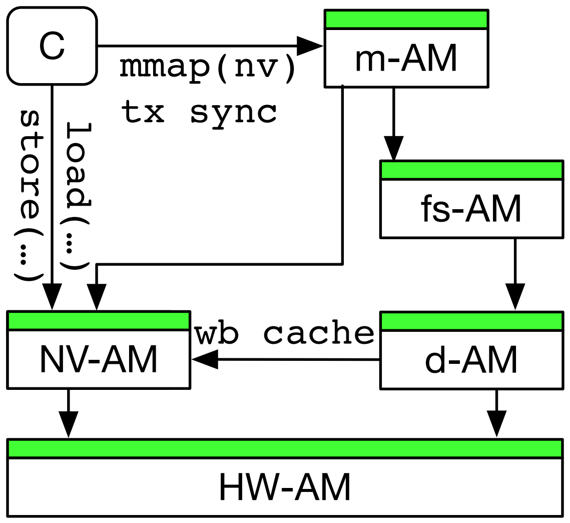

Figure 5.4: Achieving direct access / layer bypass with a deconstructed linux null-Kernel

able to make more significant modifications. Furthermore, the abstraction offered was one size fits all and introduces additional overhead. For instance, the file system must provide a mapping between file offsets and "blocks," yet PTx also provides a mapping between virtual address and NVRAM offsets. A PTx mapping between virtual addresses and NVRAM physical page offsets would eliminate one layer of mapping.

It is also important to note that a null-Kernel architecture supports both a minimal abstraction, where NVRAM pages are directly mapped without intervention of a block device and file system AM, as well as a higher level file system representation. So long as the memory system adheres to logical separation and gives out NVRAM capabilities exclusively, both lower and high level abstractions are available simultaneously.

5.5 Isolation Abstractions and the null-Kernel

In this section, we will discuss how higher level isolation abstractions, such as process isolation, can be built in a null-Kernel architecture by defining higher level constructs that contain and limit access to sets of lower-level capabilities.

The process, a high level abstraction, encapsulates several lower level abstractions for isolation (address spaces), privilege (user credentials), and execution state (threads). Processes are a successful abstraction and have worked well for decades, but increased security demands are leading to new proposals for intraprocess isolation, such as Wedge [36], ERIM [57], and *lwC*s. Each of these proposals could be built on top of the high level process abstraction, but performance requirements necessitate unsafe kernel modifications to access lower level primitives.

Each of the newly proposed intraprocess isolation abstractions make different choices that impose alternate security, usability, and design sensibilities on their users, but this is an unnecessary imposition. Each of these systems could have been simultaneously implemented in a null-Kernel that provided safe access to the same lower abstractions. This implies that merely changing the composition of low level abstractions affords significant flexibility in offering different visions for how to address similar application requirements. We focus on how *lwC*s would be implemented in a null-Kernel, but note that Wedge could be built with the same lower level AMs discussed below, whereas ERIM, which relies on new hardware, would require an additional modification to the virtual memory AM to provide access to new isolation instructions.

In a null-Kernel, abstractions should be decomposed into the lowest level set of primitives that can be exposed while maintaining compatibility with high level abstractions. Under this model, the decomposition of the process abstraction would consist of AMs that provide capabilities and corresponding instructions for manipulating execution state, memory isolation, and privilege.

Execution state and memory isolation are represented with capabilities defined by the scheduling and virtual memory AM, respectively, whereas privilege is defined by the set of capabilities that a caller can reach. The scheduling AM represents execution context with an OS context capability. A scheduling AM exposes instructions through which a OS context may be have its registers modified or scheduled to run on a core. Later we will see that these registers may be used to define the address space in which the OS context executes. Address space and memory region capabilities are granted by the virtual memory AM and provide memory isolation.

To generalize capabilities, we suggest capabilities be represented by uniformly distributed pseudo-random integers from a large namespace that cannot be guessed. Capabilities can thus be stored or passed through registers, address spaces, files, or sockets. By convention, capabilities may be effectively passed between AMs by storing capabilities at predetermined locations that have meaning in certain contexts (e.g., the address space a scheduled OS context executes within is stored in the CR3 register).

In terms of a null-Kernel, a POSIX process is a set of threads (i.e., a OS context whose CR3 register is constant) that share all capabilities except for the OS context capability itself. The set of open files and sockets (i.e., the file table) and the credentials of the process (e.g., the user id, group id, quotas, etc) are represented as a collection of capabilities granting access to system resources and are passed as a set whenever system calls are invoked via the high level system call AM. For simplicity we will assume the set of capabilities that a OS context may access are only accessible through a pointer that is stored in a special register we will call the *capability register*. Under this assumption, the high level system call AM is always passed the capability register by the caller.

The *lwC* AM introduces the *lwC* abstraction and a set of calls for manipulating or switching contexts, the primary purpose being to partition a process's capabilities into potentially disjoint and mutually untrusted actors. Under the stipulation that capabilities are only accessible by a OS context through the capability register, switching *lwC*s is equivalent to atomically modifying the registers for the OS context so that switches manifest coroutine semantics and switch the capability register pointer to another capability set to provide isolation. Critically, the *lwC* capability is a derived capability that encloses a capability set that is inaccessible to a OS context and can only be accessed through the *lwC* AM. The only way to switch the capability register to the *lwC* capability set is through the *lwC* AM, which is necessary to provide the higher level security invariants promised by the abstraction. We will now describe how the *lwC* AM is implemented in terms of the null-Kernel.

`lwCreate` The *lwC* AM exports a new capability type: the *lwC*. This call takes a resource specifier and the capability set of the caller. The newly created *lwC* contains a capability set that is derived by filtering the passed in capability set with the constraints specified by the resource specifier. The newly created context also contains register values set such that it can resume execution at the point of its creation when `lwSwitch` is first called.

`lwSwitch` Switching an *lwC* is an operation that replaces the context executing on an OS context by performing three tasks atomically. First, it saves the state of the current execution (i.e., saves the register values for the OS context). Second, it switches the capability register to the capability set enclosed by the target *lwC*. And lastly, it changes the OS context registers such that execution resumes where the target *lwC* left off.

`lwOverlay` Capabilities are passed between *lwC*s during creation and with the `lwOverlay` call. With this call, one *lwC* requests access to capabilities housed in the capability set of another target *lwC*. So long as the target *lwC* has not restricted the caller, the specified capabilities of the target will be imported into the caller's capability set.

`lwRestrict` The `lwRestrict` call allows the caller to downgrade a *lwC* capability such that it may not be used to overlay a set of specified resources. The set of restricted resources is associated with the *lwC* capability internally and consulted whenever a `lwOverlay` is attempted.

`lwSyscall` The `lwSyscall` allows a more privileged *lwC* to perform a system call on behalf of a lesser privileged *lwC*. `lwSyscall` works in concert with Capscicum, a sandboxing mechanism present in FreeBSD that blocks system calls for sandboxed processes, which is a form of privilege dropping that we extended for *lwC*s. The ability to prohibit system calls is equivalent to requiring system call capabilities to successfully invoke any calls on the system call AM. Under this equivalence, `lwSyscall` may be implemented by creating a new capability set that is the union of the necessary system call capability and the capability set of the target *lwC* and then subsequently using this newly created capability set to invoke the system call.

The *lwC* AM is a fairly simple composition of lower level AMs that would likely be exposed in any null-Kernel implementation. This is driven be hardware: fundamentally, any performant form of isolation will rely on the same underlying primitives that are available on the hardware (i.e., the lowest level AM). While we did not show it above, other security proposals such as Wedge are similarly trivial to implement with the same AMs used above. Where the null-Kernel shines relative to an exokernel, which would also expose low level primitives, is the use of capabilities to provide compatibility between abstractions. With a null-Kernel architecture, it is possible to simultaneously expose *lwC*s, the Wedge abstraction, process sandboxing (Seccomp or Capscicum), and the MPK abstraction within the same OS instance.

5.6 Conclusion

The null-Kernel is a new structure for system software that enables abstractions for efficient access to new hardware and admits new optimizations for existing hardware. The key enabling feature is its ability to codify the safe co-existence of potentially competing abstractions for the same underlying hardware resources. The null-Kernel achieves this with an extensible capability mechanism and a set of principles that allow each abstraction to define their own invariants such that both a high-level abstraction as well as the low-level abstractions that it may encapsulate may be exposed to users simultaneously.

Chapter 6: Conclusion and Future Work

In this dissertation, we described two systems abstractions, PTx and *lwC*s, as well as the null-Kernel architecture, a new OS architecture that simplifies the development of and compatibility of competing abstractions. Our contributions include the design and implementation of PTx and *lwC*s, the design of the null-Kernel architecture, and a discussion of how PTx and *lwC*s, which both require access to low and high-level abstractions for performance and functionality, could be more easily constructed within a null-Kernel architecture. These contributions support the following thesis: *Supporting novel hardware such as NVRAM and new abstractions like fine-grained isolation while maintaining efficiency, usability, and security goals, requires simultaneous access to both high-level OS abstractions and compatible access to their low-level decompositions.*

PTx is a high-level abstraction that enables the persistence of standard, unmodified data structures without the use of a specific programming language or manual annotation. The performance of PTx is dependent on lower level abstractions: it relies on direct-access to persistent media (NVRAM), bypassing traditional memory-mapped file abstractions, and a lower-level virtual memory interface that provides access to hardware-set page-modified bits. PTx operates on data structures within DRAM and optionally tracks modifications through the page-modified bits before directly and atomically persisting volatile state to an NVRAM-optimized data structure resident on NVRAM. Our results showed strong performance, PTx often outperforms Redis and LMDB, which are production systems, as well as manual-annotation systems proposed by researchers. Further, our results show that coarse-grained persistence on the order of a second approaches the execution performance of native DRAM.

*lwC*s are a new OS abstraction that provides units of isolation, privilege, and execution state independent of processes and threads. The *lwC* abstraction is built on top of lower-level abstractions than are traditionally available and provide new intra-process isolation functionality while maintaining compatibility with existing abstractions. In addition to intra-process isolation, *lwC*s provide fast OS-level snapshots and co-routine control transfer between contexts. Our results show that fast roll-back, compartmentalization of secrets, isolation, and monitoring of user sessions can be used within production Apache and nginx web servers to improve security while nearly maintaining or even exceeding the performance of unmodified Apache and nginx.

The null-Kernel is a new structure for system software that enables abstractions for efficient access to new hardware and admits new optimizations for existing hardware. The null-Kernel posits an extensible capability mechanism that distributes system resources across software that provides programming interfaces at different layers of abstraction. Equipped with proper capabilities, callers, such as user processes, can simultaneously program to any or all of these abstractions as appropriate. We describe requirements of the null-Kernel's basic capability mechanism and show how the null-Kernel can be used to implement new abstractions and optimizations.

6.1 Future Work

PTx and *lwC*s both address specific problems relating to persisting unmodified in-core data structures and providing intra-process isolation and snapshots respectively. The null-Kernel, on contrast, suggests a new way to structure a system to hasten the development of new abstractions. Our work on these systems has raised some additional research questions about potential extensions to PTx and *lwC*s, as well as how the null-Kernel may be implemented in practice.

6.1.1 Extensions to PTx

The basic PTx design can be optimized and extended relatively easily, which discuss next.

Versioning PTx only stores a single valid snapshot, but can be extended to store/restore an arbitrary number. Supporting multiple active snapshots would require PTx to associate a sequence number with each mapped address in the data header and only free blocks when no retained snapshot refers to the associated sequence number. (This design would only accommodate a linear history, i.e. it would not support forks).

Paging To support colored regions that exceed the size of DRAM, PTx would have to support paging. A rudimentary form of paging is implicitly supported already, in that the OS should page out DRAM pages from the colored region as necessary when under memory pressure, but this is not ideal. With OS support, the kernel could unmap unmodified colored pages under memory pressure and fault them back in from NVRAM when accessed. Similarly, if data blocks in NVRAM were the size of a page, NVRAM blocks could be mapped as read-only when under memory pressure.

Optimistic write-ahead When we write from the modified set to NVRAM, we do so non-destructively. An implication of this is that we could write this data either in parallel during the `commit` call, or even between `commit` calls (using a free CPU core). The former case is a potential optimization that would be part of a more advanced implementation. The latter would benefit from hints from the programmer that modified data in the colored region is unlikely to be written again, and thus, could be written to NVRAM. Critically, hints should be conservative: incorrectly hinting that an object will not be rewritten during a transaction does not affect correctness, but it would lead to unnecessary writes to NVRAM and be a counter-productive optimization.

Small transactions Transactions whose write set fits into a single blockset can be committed without using the undo log. Instead, the write of the blockset's header serves an an atomic commit point. This optimization works for transactions of up to $41 * 256 = 10,496$ bytes, increases performance, and further reduces write amplification.

6.1.2 Persistent *lwC*s

Applications may use PTx to persist data structures, but when restarted, the application itself must re-initialize all other application state and invoke the PTx `restore` operation for each colored region. It would reduce startup time and simplify use if the application could be restarted from the last successful `commit` call and resume execution where it left off. This could be achieved if we combined two abstractions: *lwC*s and PTx.

*lwC*s are not currently persistent, but may function as a persistent snapshot mechanism when combined with PTx and modified slightly. The virtual memory of the context is easily persisted: whenever `lwSwitch` is invoked, the memory and registers of the yielding *lwC* is optionally persisted to NVRAM via `commit` and the target *lwC* is restored via the `restore` operation. Similarly, `lwCreate` would create a persistent snapshot of the memory of the currently executing *lwC*. The best way to handle other *lwC* state, such as open sockets and file descriptors could be handled similar to existing checkpointing systems, such as CRIU [98][1], but we suggest that instead whenever an operation on a restored file is first invoked, a callback is invoked that may set up and restore the descriptor or socket as appropriate. Sufficient low-level access to the process file table that allows alternate file operations to be

[1]CRIU itself functions by the selective exposure of low-level abstractions for querying and setting process state, such as socket state.

associated with any given descriptor makes this possible. This flexibility would allow file descriptor restoration to be done in an application specific way, whether it attempt to restore network connections, replay the results of a previous invocation (e.g., in testing or debugging), or invalidate the descriptor. In some cases, such as preserving application histories, active connections may never need to be restored, whereas in others connection end-points may have moved and a new connection may need to be established.

lwC snapshots can be used to checkpoint and restore state, but they can also be used to build an application's history, which would be useful for auditing, debugging, and other contexts where history may want to resumed at arbitrary points, such as fuzzing [99]. The history of an application (or context) would be collected by periodically invoking `lwCreate` and proceeding with the execution. Due to the deduplication built into PTx, each snapshot would be relatively small.

6.1.3 Implementing a null-Kernel

Monolithic kernels already have well defined interfaces that are available for extending the kernel with a kernel module. Modules, which execute with full privilege, are intended to use these interfaces but they may call any kernel symbol and read or write arbitrary regions in the kernel's address space. Kernel modules work because they are constrained by convention to only call symbols in predetermined ways and not read or write memory for which they should not have access. However, if the constraints that safe modules generally follow can be codified with capabilities, the interface exposed to kernel modules is a natural starting point for exposing lower-level AMs in a manner consistent with a null-Kernel.

Wrapping these interfaces with capability checks and exposing them via the system call interface would provide the functionality we desire, but performance would likely be unacceptable. High-level AMs built in terms of these low-level AMs may require many calls to the lower-level AMs to represent a single higher level abstraction. If each call to the lower-level AMs requires a switch into and out of supervisor mode, high performance higher-level abstractions could not be built that are compositions of many calls to the lower-level AMs without inducing significant overhead. Instead, we propose giving users the ability to inject into the kernel new abstractions that are written in a restricted language that provides memory safety and guarantees that capability checks are performed.

We have begun extending FreeBSD to act more like a null-Kernel and expose lower and higher-level abstractions simultaneously. We have provided lower-level AMs for virtual memory and OS tasks and allow users to access this via a safe subset of the Rust programming language that executes within supervisor mode. Significant work remains to fulfill the full null-Kernel vision, though even a minimal set of lower-level abstractions enables new higher-level such as *lwC*s and PTx, as discussed in Section 5.5 and Section 5.4 respectively. Working from a monolithic kernel, new hardware and existing abstractions can be safely decomposed and made available to users with the aide of proper capability checks on an incremental basis.

6.2 Concluding Thoughts

Changing application requirements and the accelerating pace of hardware advances increase the needed rate of iteration for OS abstractions. Increasing security requirements have driven the need for abstractions that promote greater security and isolation. A raft of new security proposals, including *lwC*s, Capsicum [100], Pledge [101], ERIM [57], and many others have been proposed in recent years. Some of these systems have been adopted, but often when adopted, they are adopted haphazardly. Owing to the fact that we still lack a unified mechanism to enable the adoption of new abstractions, many compelling proposals are not adopted or are only adopted haphazardly and with heroic effort. Critically, many of these systems rely on the same lower-level abstractions, even though they make different tradeoffs and design decisions that may suit different kinds of applications. Systems for managing NVRAM, such as PTx, tell a similar story. We have a raft of new proposals, none of which are the best for all circumstances, but all of which rely on the same fundamental low-level abstractions. It would be better to expose to users the low-level abstractions alongside the high-level abstractions that depend on them. When new hardware is introduced, the lowest-level abstraction possible should be exposed. Whenever a higher-level abstraction is offered, the lower-level abstractions upon which it is built should be left accessible, with safety provided by following the principles and structure of the null-Kernel. This vision for future systems development will speed the pace of innovation by providing a low-level substrate upon which new abstractions may always be built, without eliminating the portability that time-tested higher-level abstractions provide.

Bibliography

[1] 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 *OSDI*, pages 49–64, 2016.

[2] James Litton, Deepak Garg, Peter Druschel, and Bobby Bhattacharjee. Composing abstractions using the null-kernel. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 1–6, 2019.

[3] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[4] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, November 1984.

[5] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150. ACM, 2013.

[6] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[7] Stephen R Walli. The POSIX family of standards. *StandardView*, 3(1):11–17, 1995.

[8] Redis Labs. Redis. https://redis.io, 2020.

[9] Joseph Izraelevitz, Jian Yang, Lu Zyang, 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, 2019.

[10] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 478–496, 2017.

[11] 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*, pages 1–15, 2014.

[12] 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 ACM SIGOPS 22nd symposium on Operating systems principles*, pages 133–146, 2009.

[13] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, Roy H Campbell, et al. Consistent and durable data structures for non-volatile byte-addressable memory. In *FAST*, volume 11, pages 61–75, 2011.

[14] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-tree: Reducing consistency cost for NVM-based single level systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 167–181, 2015.

[15] Shimin Chen and Qin Jin. Persistent B+-trees in non-volatile main memory. *Proc. VLDB Endow.*, 8(7):786–797, February 2015.

[16] Se Kwon Lee, K Hyun Lim, Hyunsub Song, Beomseok Nam, and Sam H Noh. WORT: Write optimal radix tree for persistent memory storage systems. In *15th USENIX Conference on File and Storage Technologies*, pages 257–270, 2017.

[17] Fei Xia, Dejun Jiang, Jin Xiong, and Ninghui Sun. Hikv: A hybrid index key-value store for dram-nvm memory systems. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 349–362, Santa Clara, CA, July 2017. USENIX Association.

[18] Kornilios Kourtis, Nikolas Ioannou, and Ioannis Koltsidas. Reaping the performance of fast NVM storage with uDepot. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 1–15, 2019.

[19] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey III, Dhruva R. Chakrabarti, and Michael L. Scott. Dalí: A Periodically Persistent Hash Map. In Andréa W. Richa, editor, *31st International Symposium on Distributed Computing (DISC 2017)*, volume 91 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 37:1–37:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[20] Pengfei Zuo, Yu Hua, and Jie Wu. Write-optimized and high-performance hashing index scheme for persistent memory. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 461–476, 2018.

[21] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. NVWAL: Exploiting NVRAM in write-ahead logging. *ACM SIGOPS Operating Systems Review*, 50(2):385–398, 2016.

[22] Seunghee Shin, Satish Kumar Tirukkovalluri, James Tuck, and Yan Solihin. Proteus: A flexible and fast software supported hardware logging approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 178–190, 2017.

[23] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building durable transactions with decoupling for persistent memory. In *ACM SIGARCH Computer Architecture News*, volume 45, pages 329–343. ACM, 2017.

[24] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven Swanson. Atomic in-place updates for non-volatile main memories with kamino-tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, page 499–512, New York, NY, USA, 2017. Association for Computing Machinery.

[25] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M Chen, and Thomas F Wenisch. High-performance transactions for persistent memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 399–411, 2016.

[26] Intel Corporation. Persistent memory programming. `https://pmem.io/pmdk`, 2020. Accessed 25-May-2020.

[27] 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. *ACM SIGARCH Computer Architecture News*, 39(1):105–118, 2011.

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

[29] Dhruva R Chakrabarti, Hans-J Boehm, and Kumud Bhandari. Atlas: Leveraging locks for non-volatile memory consistency. *ACM SIGPLAN Notices*, 49(10):433–452, 2014.

[30] Qingrui Liu, Joseph Izraelevitz, Se Kwon Lee, Michael L Scott, Sam H Noh, and Changhee Jung. iDO: Compiler-directed failure atomicity for nonvolatile

memory. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 258–270. IEEE, 2018.

[31] Haris Volos, Andres Jaan Tack, and Michael M Swift. Mnemosyne: Lightweight persistent memory. *ACM SIGARCH Computer Architecture News*, 39(1):91–104, 2011.

[32] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and fast persistence for volatile data structures. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 789–806, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-structured non-volatile main memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 703–717, 2017.

[34] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery.

[35] CERT Vulnerability Note VU#720951: OpenSSL TLS heartbeat extension read overflow discloses sensitive information. `http://www.kb.cert.org/vuls/id/720951`.

[36] 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.

[37] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. *2016 IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 23-25, 2015*, pages 20–37, 2016.

[38] 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.

[39] Andrea Mambretti, Kaan Onarlioglu, Collin Mulliner, William Robertson, Engin Kirda, Federico Maggi, and Stefano Zanero. Trellis: Privilege separation for multi-user applications made easy. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 437–456. Springer, 2016.

[40] 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, Berkeley, CA, USA, 1994. USENIX Association.

[41] 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, Washington, DC, USA, 1995. IEEE Computer Society.

[42] Gabriel Palmer. The case for thread migration: Predictable IPC in a customizable and reliable OS. In *Proceedings of the Workshop on Operating Systems Platforms for Embedded Real-Time applications (OSPERT '10)*, 2010.

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

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

[45] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS X, pages 304–316, New York, NY, USA, 2002. ACM.

[46] Emmett Witchel, Junghwan Rhee, and Krste Asanovic. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Proceedings of the 20th Symposium on Operating Systems Principles (SOSP '05)*, Brighton, UK, October 2005.

[47] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 457–468, Piscataway, NJ, USA, 2014. IEEE Press.

[48] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav H. Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy, S&P 2015, San Jose, CA, USA, May 17-21, 2015*, pages 20–37, 2015.

[49] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, OSDI '99, pages 45–58, Berkeley, CA, USA, 1999. USENIX Association.

[50] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An operating system for many cores. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[51] Mark Miller. *Robust composition: Towards a unified approach to access control and concurrency control.* PhD thesis, Johns Hopkins University, 2006.

[52] Adrian Mettler, David Wagner, and Tyler Close. Joe-e: A security-oriented subset of java. In *NDSS*, volume 10, pages 357–374, 2010.

[53] Google Caja Team. Google-Caja: A source-to-source translator for securing javascript-based web. `http://code.google.com/p/google-caja`.

[54] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, pages 340–353, 2005.

[55] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 147–163, 2014.

[56] Marco Patrignani, Pieter Agten, Raoul Strackx, Bart Jacobs, Dave Clarke, and Frank Piessens. Secure compilation to protected module architectures. *ACM Transactions on Programming Languages and Systems*, 37(2), April 2015.

[57] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (mpk). In *Proceedings of the 28th USENIX Conference on Security Symposium*, SEC'19, page 1221–1238, USA, 2019. USENIX Association.

[58] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, December 1993.

[59] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullager. Native Client: A sandbox for portable, untrusted x86 native code. *2009 IEEE Symposium on Security and Privacy, SP 2016, Berkeley, CA, USA, May 17-20, 2009*, pages 79–93, 2016.

[60] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin—Madison CS Department, April 1997.

[61] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference*, pages 213–223, January 1995.

[62] William R. Dieter and James E. Lumpp, Jr. User-level checkpointing for LinuxThreads programs. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 81–92, Berkeley, CA, USA, 2001. USENIX Association.

[63] Hua Zhong and Jason Nieh. CRAK: Linux checkpoint/restart as a kernel module. Technical Report CUCS-014-01, Columbia University CS Department, November 2001.

[64] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 193–206, Berkeley, CA, USA, 2010. USENIX Association.

[65] Intel Corp. *Intel 64 and IA-32 Architectures Software Developer's Manual: Vol. 3D*, June 2016.

[66] Udo Steinberg and Bernhard Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, 2010.

[67] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 335–348, Hollywood, CA, 2012. USENIX.

[68] Christopher A Small and Margo I Seltzer. Vino: An integrated platform for operating system and database research. 1994.

[69] 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. *ACM SIGOPS Operating Systems Review*, 29(5):267–283, 1995.

[70] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel design for isolation and assurance of physical memory. In *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*, IIES '08, pages 35–40, 2008.

[71] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*, volume 27, 2008.

[72] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. Eros: A fast capability system. *SIGOPS Oper. Syst. Rev.*, 33(5):170–185, December 1999.

[73] Norman Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev.*, 19(4):8–25, October 1985.

[74] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 263–278, Berkeley, CA, USA, 2006. USENIX Association.

[75] Butler W Lampson and Howard E Sturgis. Reflections on an operating system design. *Communications of the ACM*, 19(5):251–265, 1976.

[76] R. M. Needham and R. D.H. Walker. The cambridge cap computer and its protection system. In *Proceedings of the Sixth ACM Symposium on Operating Systems Principles*, SOSP '77, pages 1–10, New York, NY, USA, 1977. ACM.

[77] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernels meet recursive virtual machines. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, OSDI '96, pages 137–151, New York, NY, USA, 1996. ACM.

[78] T. Harris, A. Cristal, O. S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, and M. Valero. Transactional memory: An overview. *IEEE Micro*, 27(3):8–29, 2007.

[79] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[80] P.J. Plauger, Meng Lee, David Musser, and Alexander A. Stepanov. *C++ Standard Template Library.* Prentice Hall PTR, USA, 1st edition, 2000.

[81] Torvald Riegel. Transactional Memory in GCC. Available at `https://gcc.gnu.org/wiki/TransactionalMemory`, 2012.

[82] Doug Lea. A memory allocator. `http://gee.cs.oswego.edu/dl/html/malloc.html`, 2020. Accessed 21-April-2020.

[83] Howard Chuh. http://www.lmdb.tech/doc/, 2020. Accessed 27-May-2020.

[84] Intel Corporation. Pmem redis. https://github.com/pmem/pmem-redis, 2020.

[85] memcachd. `https://github.com/memcached/mc-crusher/blob/bba6b5cb46603e4c0f04f4aa4ea43ffaa3f7d6c0/test-suites/test-nvdimm`, 2020. Accessed 27-May-2020.

[86] Liang Zhang, Dave Choffnes, Tudor Dumitraş, Dave Levin, Alan Mislove, Aaron Schulman, and Christo Wilson. Analysis of SSL Certificate Reissues and Revocations in the Wake of Heartbleed. In *ACM Internet Measurement Conference (IMC)*, 2014.

[87] Zakir Durumeric, James Kasten, Frank Li, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, J. Alex Halderman, Vern Paxson, and Michael Bailey. The matter of Heartbleed. In *ACM Internet Measurement Conference (IMC)*, 2014.

[88] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. A taste of Capsicum: Practical capabilities for unix. *Commununications of the ACM*, 55(3), March 2012.

[89] The PHP Group. FastCGI Process Manager (FPM). `http://php.net/manual/en/install.fpm.php`, 2016.

[90] Zend. MVC Skeleton Application. `https://framework.zend.com/downloads/skeleton-app`, 2016.

[91] Adam Lackorzynski and Alexander Warg. Taming subsystems: Capabilities as universal resource access control in l4. In *Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, IIES '09, pages 25–30, 2009.

[92] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware support for fast capability-based addressing. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, pages 319–327, New York, NY, USA, 1994. ACM.

[93] Anil Madhavapeddy and David J Scott. Unikernels: the rise of the virtual library operating system. *Communications of the ACM*, 57(1):61–69, 2014.

[94] Andrew S Tanenbaum, Sape J Mullender, and R van Renesse. Using sparse capabilities in a distributed operating system. 1986.

[95] Mark Anderson, RD Pose, and Chris S. Wallace. A password-capability system. *The Computer Journal*, 29(1):1–8, 1986.

[96] Jerry Vochteloo, Stephen Russell, and Gernot Heiser. Capability-based protection in the mungi operating system. In *Object Orientation in Operating Systems, 1993., Proceedings of the Third International Workshop on*, pages 108–115. IEEE, 1993.

[97] Gokcen Kestor, Luke Dalessandro, Adrián Cristal, Michael L Scott, and Osman Unsal. Interchangeable back ends for stm compilers. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, 2011.

[98] CRIU Project. CRIU: Checkpoint and restore in userspace. `https://criu.org/Main_Page`, 2020. Accessed 19-November-2020.

[99] Wen Xu, Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. Designing new operating primitives to improve fuzzing performance. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2313–2328, 2017.

[100] Robert N. M. Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: practical capabilities for unix. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[101] OpenBSD Foundation. Pledge(2) - OpenBSD manual pages. `https://man.openbsd.org/pledge.2`, 2020. Accessed 22-November-2020.