

Operating Systems are a Service

by

Kenneth Ryan Hancock

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Kenneth Ryan Hancock 2025

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner: Ashvin Goel
Professor, Dept. Electrical and Computer Engineering
and Computer Science, University of Toronto

Supervisor(s): Ali José Mashtizadeh
Professor, Dept. of Computer Science, University of Waterloo

Internal Member: Samer Al-Kiswany
Professor, Dept. of Computer Science,
University of Waterloo

Internal Member: Mina Tahmasbi Arashloo
Professor, Dept. of Computer Science, University of Waterloo

Internal-External Member: Hiren Patel
Professor, Dept. of Electrical and Computer Engineering
University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contribution

The majority of this thesis encapsulates the storage system built for the Aurora Single Level Store [131, 132] and MemSnap [133]. For Aurora, I was the second author and MemSnap was an equal contribution coauthor. The storage system, ObjSnap, and its use within Aurora and MemSnap, and its design and evaluation is my main contribution.

For OrcBench [71], I am the first author and designed the methods used in modeling the function invocation rates.

For SlimSys, I am the first author and designed, built, and evaluated the main systems outlined in the paper. The analysis was designed and built by my coauthor, Tavian Barnes.

Abstract

OS containers have set the standard for the deployment of applications in modern systems. OS containers are combined sandboxes/manifests of applications that isolate the running applications and its dependencies from other applications running on top of the same kernel. Containers make it easy to provide multi-tenancy and control over the application, making it ideal for use within cloud architectures such as serverless.

This thesis explores and develops novel systems to address three problems faced by containers and the services that use them. First, OS containers currently lack a fast checkpoint-restore mechanism. Second, container security is still inadequate due to its underlying security mechanisms, which provide coarse-grained policies that are abused. Third, the lack of a benchmark for serverless clouds, one of the largest consumers of containers, and specifically checkpoint-restore.

This thesis outlines solutions to these problems. First, ObjSnap, a storage system designed and built for two modern single-level store systems, Aurora and MemSnap, which enable checkpoint restore for container systems. ObjSnap is a transactional copy-on-write object store that can outperform other storage systems by up to $4\times$. Second, we introduce SlimSys, a framework that tackles security issues found within containers by binding a policy to kernel resources. Lastly, we introduce Orcbench, the first benchmark used to evaluate serverless orchestrators.

Acknowledgements

I would like to acknowledge the people who have supported me. My collaborator and friend Emil Tsalapatis, whose guidance and constant help I will always cherish. My mother, Barbara, who has been a constant source of support. Trevor Brown for his guidance and mentorship. Finally, thank you to all my friends and family who have helped me throughout this journey.

Dedication

This thesis is dedicated to two people. First, my wife Jacqueline, whose unwavering support and love made this thesis even possible. Second, Nolan, who gave me the little nudge out of the door I needed, and has been one of the most positive influences in my life.

Table of Contents

Examining Committee	ii
Author’s Declaration	iii
Statement of Contribution	iv
Abstract	v
Acknowledgements	vi
Dedication	vii
List of Figures	xi
List of Tables	xiv
1 Introduction	1
1.1 Towards Container Checkpoint/Restore	1
1.2 ObjSnap: An Object Store for SLSES	4
1.3 SlimSys: Extending Container Security	4
1.4 OrcBench: Enabling Container-based Serverless Clouds	5
1.5 Outline of Contributions	6

2	ObjSnap—A Single Level Store Object Store	7
2.1	Background	9
2.1.1	File Systems	11
2.1.2	Our Approach	12
2.2	Design Overview	13
2.2.1	ObjSnap	15
2.2.2	ObjSnap API for Single Level Stores	15
2.3	Design	16
2.3.1	Global Checkpoint Worker	19
2.3.2	The Generational Allocator	20
2.3.3	Recovery	22
2.3.4	Implementation	23
2.4	Evaluation	23
2.4.1	Checkpoint Coalescing Breakdown	24
2.4.2	Generational Moving Allocator	25
2.4.3	Versus File Systems	27
2.4.4	Supporting An SLS–MemSnap	28
2.5	Related Work	30
2.6	Conclusion	31
3	SlimSys: Fine-grained System Call Debloating using Resource Binding	32
3.1	Background and Motivation	34
3.1.1	System Call Disabling	35
3.1.2	Capabilities	35
3.1.3	Limitations of Software Debloating	36
3.1.4	Our Approach: Syscall-Resource Binding	37
3.2	Threat Model	38
3.3	Design Overview	38

3.3.1	SlimSys	39
3.4	Syscall-Resource Binding	41
3.4.1	Creating Policy	41
3.5	Enforcing Policy	45
3.5.1	Dynamic Revocation	46
3.6	Implementation	47
3.7	Evaluation	47
3.7.1	Attack Surface Reduction	49
3.7.2	Vulnerability Comparison	50
3.7.3	Performance	51
4	OrcBench: A Representative Serverless Benchmark	58
4.1	Overview of the Microsoft Azure Traces	59
4.2	Methodology	60
4.2.1	Modeling Functions	61
4.3	Evaluation	65
4.3.1	Models	65
4.4	Related Work	67
5	Conclusion	68
	References	71

List of Figures

2.1	Performance and scalability degradation for various production grade file systems under a multithreaded workload doing small synchronous IO to a single file.	10
2.2	The main components of the ObjSnap design. ObjSnap’s checkpoint coalescing mechanism consists of the Flat Combining region, which selects combiner agents to combine other threads, and commits all thread dirty sets as a micro checkpointer. The global checkpointer worker aggregates checkpoints in our journal region on disk and commits them. The generational, moving allocator, will provide data to move for combiners and maintain performance consistency over the lifetime of the SSD.	17
2.3	ObjSnap’s Generational Allocator. During an allocation, the allocator will allocate from a segment sequentially until full. Once full, the segment is placed onto the tail of the hot list. The allocator will eventually start moving data that is still resident in the head segment of the host list. These pages are moved by a combiner threads by appending a move set to the current transaction. The combiner thread will perform IO to move pages to a cold segment	21
2.4	ObjSnap with a 30 GiB and 24 threads, each writing a 1 page transaction in a Zipfian distribution. The closer to zero the alpha parameter is the more uniform it is, the closer to 1, the more concentrated the hot set of the distribution becomes. We see that as we become more concentrated the more operations per second achieves as less moving is required to keep a steady stream of segments available for allocation.	25

2.5	A comparison of ObjSnap versus various forms of traditional file systems, ZFS, FFS (with soft updates), ext4 (with journaling), and XFS. The benchmark shown is a single page durable write, in which up to 24 threads randomly dirty and write to a page, followed by an <code>fsync</code> , or <code>objsnap_checkpoint</code> call. Shown is the performance in terms of Operations per second, latency, tail latency, and throughput efficiency. Throughput efficiency is a measure of on disk throughput divided by application goodput.	26
2.6	Multi page write performance of ObjSnap versus another copy-on-write object store, ZFS. The workload is pinned at 24 threads, and run the same workload with larger dirty sets of each thread. We further show a multi-object checkpoint with ObjSnap, in which the dirty set is randomly spread between five objects to show that the write path is independent of the number of objects.	28
2.7	Memcached performance with ObjSnap. We use ObjSnap with the MemSnap SLS to transparently persist writes to a Memcached instance, and measure the throughput compared to a non-persistent Memcached instance. ObjSnap keeps the cost of persistence stable for write ratios of 10–90% and for up to 24 concurrent threads. By comparison, MemSnap’s original object store falls behind in performance after 9 threads, and crashes for more than 12 threads.	30
3.1	A diagram showing the SlimSys architecture and the process of data forwarding. data forwarding starts by capturing calls within the C library through our interposition layer. This translates the call to our service to actually perform the call on the application’s behalf. The policy is updated, and then the values returned to the application. Any systems related to the C library call are disallowed, to force an application to use the interposition layer. . .	40
3.2	Creating a SlimSys policy.	54
3.3	An example revocation policy that can be presented to SlimSys to be combined at runtime in the kernel. In this example it can be used to protect sensitive files from being exfiltrated from an application by disconnecting previously allowed connections.	55
3.4	A breakdown for each workload showing the relative cost of the capability check to the entire system call and the cost of the capability check overall to the entire system for a given workload.	56
3.5	Performance overhead of SlimSys’s syscall-resource binding.	57

4.1	Invocation counts of functions broken up by execution time in milliseconds over the course of a day	61
4.2	An example of sampling from an ECDF of a model with a hyper-parameter of $N = 3$, and how we spread invocations within a sampled interarrival time. Each vertical line on the timeline from 0s to 0.5s represents a separate function.	64
4.3	Example CPU and memory histograms which back each model for generating function stubs.	65
4.4	Error Rates within the OrcBench Models	66
4.5	Example traces of our models which have been scaled to the number of functions to the functions they originally represent. We compared these modeled functions to the sum of the traces of the original functions.	67

List of Tables

2.1	The API provided to the single level store checkpointing subsystem.	16
2.2	The effects of wait times and our throughput efficiency. We run a single page write workload at 24 threads and track checkpoints that finished as a waiter, or a combiner agent and their respect latencies and amounts. We then show the average size of a checkpoint given that wait time (in pages). With checkpointing, we see a non-linear relation with throughput efficiency due to a trade off that occurs between μ Checkpoint throughput and breakpoints of the average checkpoint size, showing that a 2 us wait is optimal.	24
2.3	In this table we show the scalability performance of the original MemSnap and ObjSnap. We perform a similar single page durable write and measure the latency. Due to bottlenecks innate in the design of MemSnap, we see as thread count rises, latency increases faster than ObjSnap.	29
3.1	Shows the limitations of system call based debloating and its ineffectiveness of reducing the attack surface of the kernel for lighttpd. Perfect system call debloating reduces the kernel system call API by 91%, but reduces reachable code by 29%, and Temporal Specialization only achieves an additional 4% reduction.	36
3.2	Shows the effectiveness of Seccomp debloating techniques. System call debloating assumes perfect debloating of a program, lighttpd, which uses 24 system calls. Temporal Specialization removes 9 system calls, resulting in 15 total calls.	47
3.3	Shows five applications and how much kernel code is covered by a variety of common system calls executed during the lifetime of the application. syscall-resource binding, results in an 81% average reduction across these calls. This reduction does not include the reduction of code resulting from excluding the code paths from the other 277 system calls.	48

3.4	A list of examples attacks and which system protects against it. Attacks that Temporal debloating and Confine are unable to defend against are ones which manipulate kernel state through arguments to system calls.	50
-----	--	----

Chapter 1

Introduction

Containers, such as Docker [32], have become the standard for application deployment. Containers are an operating system abstraction, in which the kernel uses various namespaces (e.g., PID namespaces, file namespaces, network namespaces) to ensure application isolation. This design has made containers a lightweight deployment mechanism compared to standard virtual machines.

Containers have a wide variety of applications and uses through various fields, in particular in the cloud, and have enabled new techniques to develop and deploy software, such as Serverless. In serverless environments, users provide small pieces of code to be executed when triggered (e.g., through an HTTP request) and offload the responsibility of deployment and scaling to the cloud provider. Serverless platforms such as those offered by Azure and Amazon heavily use containers as a way of packaging user code for future deployment and then deploy these containers into running virtual machines. However, the potential of these techniques are not fully realized as containers have fundamental limitations. This thesis illustrates and evaluates three systems that solve problems within three areas of containers: checkpoint-restore, security, and benchmarking.

1.1 Towards Container Checkpoint/Restore

The first large gap in containers is the lack of a fast and efficient checkpoint-restore mechanism (CR). Having this mechanism would have substantial benefits for the cloud and specifically serverless and help solve its cold start problem. The cold start problem is a serverless performance bottleneck in which the construction and initialization of a container and associated virtual machine dominates the execution time of the function itself [126].

Checkpoint-restore is being explored as a solution to mitigate the cold start problems and follows the insight that the cost of reading the already initialized state from disk is far faster than creating this state from scratch. Using checkpoint restore, the cloud provider checkpoints an already initialized virtual machine and an application container. Upon a trigger, the cloud restores the checkpointed state, resulting in reduced cold-start latencies.

Directly checkpointing a running container versus a virtual machine offers substantial performance benefits for restore times, while also reducing the usage of the checkpointed image on disk and in memory. However, fast container checkpoint-restore is more difficult, and current systems such as CRIU can take multiple seconds to restore a container [2]. The difficulty of restoring a container comes from rebuilding each individual object that makes up an application, as opposed to black-box restoring the virtual memory of a VM [51, 56, 101].

Systems such as ZAP [86, 104] were some of the first systems to introduce transparent checkpoint restore of processes and groups of processes on a commodity system. However, these systems only captured the memory state of these applications and, further, required larger stop times of 10–100ms. OpenVz [36, 100] also used container checkpointing as a way to migrate, but was too slow for use within a serverless environment.

The Single Level Store A promising solution to container checkpoint-restore is in the form of a new operating system paradigm—the single-level store. Single-level stores are reemerging as an alternative to traditional operating systems. Within an SLS, the OS subsumes the responsibility of persistence by unifying virtual memory and storage, allowing developers to write applications as if they never crash, greatly simplifying persistence. With an SLS, the OS takes the responsibility of ensuring that any application object’s (e.g., file descriptor, memory mapped region) in memory version is identical to its persisted one, making it an ideal candidate as a mechanism for container checkpoint-restore. Primitives such as single address spaces [50] provide part of the semantics of an SLS, but through an orthogonal approach. SAS-like semantics are essential for building an SLS, but alone do not provide the persistence guarantees and performance required by applications on top of an SLS.

SLS systems like EROS [127] and KeyKOS [72] were explored in the 1960s but fell out of favor due to the increasing gap between CPU and storage. In an SLS, entire applications are persisted through a kernel checkpoint mechanism, which checkpoints all execution state (e.g., CPU registers, file descriptors, memory). SLSes require that the application be halted for a period of time to consistently serialize it. However, as application’s were becoming faster and dirtying more memory at higher rates, the storage devices of the time could not keep up, requiring increasingly larger application pauses.

Modern SLSes such as Aurora and MemSnap offer an alternative to traditional file-based persistence, whose API has its set of nontrivial issues. The implementation of each file system varies, leading to various bugs and performance discrepancies between systems [26]. In addition, users must manage two forms of their data: their in-memory data and the on-disk versions. This becomes challenging, particularly with large data structures that have intricate internal relationships, such as the processes that make up a container application. Ensuring atomic updates to these data structures requires careful ordering of writes to storage and reliance on complex, high-overhead abstractions such as a userspace write-ahead logs (WAL).

The advent of new high-bandwidth, low-latency SSD devices, and increased PCIe slots on host has closed the gap between CPU and IO, prompting the development of two modern SLSes: Aurora [131, 132] and MemSnap [133]. These systems continue the vision of the SLS as a persistence mechanism; however, they required a low-latency, high-throughput storage system to meet their fast checkpoint restoration goals.

Aurora was the first SLS enabled and designed for POSIX that uses commodity SSD's. Aurora introduced a fast, submillisecond checkpointing mechanism that was completely transparent to a running application and that required no code modifications. Aurora's design introduced its POSIX object model, which underpinned how the operating system persisted kernel representations of application state. Aurora was designed to serialize the fine-grained state and relationships of an application as seen in the kernel.

Aurora's ability to checkpoint at high frequency is further attributed to its ability to serialize the majority of data in the time between two checkpoints. The majority of application state is backed by memory mapped region's, which when marked as copy-on-write can safely allow for the background write of them without corrupting the serialized state. However, Aurora's transparent checkpoint mechanism was insufficient for applications like databases that require small, low-latency, durable writes, requiring a re-evaluation of its design.

MemSnap was introduced as an extension to Aurora and introduced it's slim SLS checkpoint API. In this API, a user marks the start and end of a transaction. During execution between a start and end call, MemSnap will transparently capture the individual write set's of each thread and atomically checkpoint them to disk, subsuming the need for file-based logic and systems such as userspace WAL's.

Metropolis [130] is a recent system that uses Aurora as a base for its fast container restore. In addition, Metropolis extended the capabilities of an SLS even further through its object-sharing model. Metropolis leveraged the unique perspective that SLSes have over the objects that make up an application, allowing for objects such as shared libraries to be

fully shared between containers and for containers on the same host to warm each other up by paging in each other’s libraries.

1.2 ObjSnap: An Object Store for SLSes

The first system we present in this thesis is the design of an SLS storage system. Through this system, we enable the SLS used for systems like Metropolis which enable efficient checkpoint-restore of containers. ObjSnap is designed to handle the unique workload of SLSes, which is high-throughput, write mostly, synchronous, random, page-size IOs. SLSes, like MemSnap, have the additional requirement that checkpoints be low-latency and support multiobject checkpoints and versioning, resulting in the need for a custom storage design. ObjSnap enables these SLSes to efficiently act as a foundation to solve problems in areas such as container checkpoint-restore.

ObjSnap, is the evolution of Aurora and MemSnap, and was designed to maintain correctness and performance under highly concurrent, write-heavy workloads as seen by single-level stores. ObjSnap uses a *checkpoint coalescing* mechanism to achieve the goals of high throughput and low latency. Checkpoint coalescing amortizes copy-on-write-induced metadata costs across thousands of checkpoints, while also reducing contention on systems like the kernel buffer cache. Minimizing metadata overhead in I/O operations is essential to achieving our goal of low latency and maximizing the effectiveness of available storage bandwidth, ensuring it can keep up with memory modification rates of a running application.

We evaluated ObjSnap against state-of-the-art production file systems and show that ObjSnap is capable of outperforming existing traditional file systems by 4×. In addition, ObjSnap is designed for efficient use of available storage IO, producing 2–3 times less metadata on average, ensuring that the available IO good output of the overarching SLS can keep up with memory dirtying rates.

1.3 SlimSys: Extending Container Security

The second problem we explore in this thesis is to make containers a better isolation mechanism with respect to security. Although containers are optimal for dependency isolation and deployability, they are still inadequate as a multi-tenancy mechanism. One such issue with containers stems from the fact that they are isolated using security mechanisms that provide coarsely grained policies [1, 21, 22, 44], which are prone to abuse.

However, these mechanisms restrict an application by reducing its allowed set of system calls and cannot prevent the misuse of common system calls used in almost all applications (e.g., `mmap`, `connect`, `sendmsg`).

Deriving the set of allowed system calls is done through debloating systems, which run various forms of run-time and static analysis to determine the policy. However, these systems [38, 58, 61, 66–69, 85, 124] are used to create Seccomp or Pledge policies, which means that they remain coarse-grained. For example, in our evaluation of these systems we found on applications like `lighttpd`, a perfect system call debloating technique reduces the number of available system calls by 91% but only reduces the kernel code attack surface by 29%. This lack of concrete containers isolation is the main reason the cloud continues to deploy containers within microVM’s [37] to ensure tenant isolation.

This thesis introduces SlimSys, an enforcement system that de-bloats the underlying kernel code as opposed to minimizing the system call API using our novel resource-centric approach. SlimSys introduces a novel model, *System call-resource binding*, that views an application as a set of well-defined system resources (e.g., files, sockets, event queues), rather than a global set of system calls. SlimSys defines resources by the system call and the arguments that create it. Each resource has its own fine-grained policy, which specifies the set of system calls and their associated arguments that are allowed.

We implement SlimSys’s Syscall resource binding with a userspace runtime, a kernel module, and a new program analysis tool. In our evaluation, we and show that SlimSys’s syscall-resource binding model reduces the kernel attack surface in common system calls by an additional 81% over Seccomp-based debloating techniques. A ceiling for a hypothetical perfect system call debloating system for an application such as `lighttpd` web server only reduces the kernel attack surface by 29%. In comparison, SlimSys reduces the kernel attack surface by 84%. Syscall resource binding is an efficient mechanism that imposes an average overhead of 1% as opposed to Seccomp’s 3%.

1.4 OrcBench: Enabling Container-based Serverless Clouds

Our work on containers and SLS systems such as Metropolis brought to light the lack of available performance benchmarks in the field of serverless. Serverless as a field lacks core benchmarks with respect to the invocation rate of functions, reducing our ability to evaluate systems such as Metropolis. Serverless benchmarks traditionally evaluate the

types of functions seen within the cloud [62, 63] and how those flagship functions perform. However, do not evaluate how often these functions are invoked.

Serverless clouds are difficult to model, as they behave differently from existing clouds and rarely make data surrounding their services publically available. A representative model of the serverless cloud benefits the evaluation of future container orchestrators.

The last system introduced in this thesis is OrcBench [71], the first benchmark that evaluates the invocation rate of serverless functions and thus is able to evaluate orchestrator designs. The functions invoked are modeled from real-world Microsoft Azure traces [126] and model the usages of the resources and the execution times of these traced functions.

1.5 Outline of Contributions

This thesis introduces and evaluates three novel systems built during my research, which encapsulate storage systems, container security, and serverless benchmarking.

1. ObjSnap (Chapter 2), a storage system built to support two modern single level stores: Aurora [131, 132], published at HotOS 2021 and SOSP 2021, in which I was the second author, and MemSnap [133], published at ASPLOS 2024 in which I was the coauthor. I was also coauthor on Metropolis, a serverless invoker, which is to be submitted (citation to be added).
2. SlimSys (Chapter 3), a security system used within containers that replaces mechanisms like seccomp [21], and allows for finer-grained policies which bind the policy to kernel resources and the call that the system uses them.
3. OrcBench [71](CLOUD 2022) (Chapter 4), the first benchmark that models the invocation pattern of serverless functions in the cloud.

Chapter 2

ObjSnap—A Single Level Store Object Store

Single-level stores (SLS) offer a simple to traditional file IO based APIs. Almost 20 years after the last wave of single-level stores such as EROS [127], there has been a new wave of single-level stores that provide persistence for modern low-latency and high-throughput applications [132, 133, 139]. These systems revisit the unified virtual memory-storage paradigm motivated by the advent of large address spaces, low-latency storage devices, and PCIe bandwidth that rivals memory bandwidth.

Aurora, TreeSLS, and MemSnap are three such modern state-of-the-art single level stores. Aurora [132] revisits the idea for traditional UNIX operating systems on conventional PCIe attached flash, while achieving 10–100 checkpoints per second. TreeSLS [139] reduces latency through a custom OS with the use of NV-DIMMs. MemSnap [133] introduces a low latency and high throughput persistence API that subsumes mechanisms such as write-forward logs (WAL) in databases. Each system has successively pushed towards more frequent smaller checkpoints of application state.

Unfortunately, none of the systems have gone in depth on the challenges and optimizations of their single level store’s storage subsystem. Existing SLSes gain most of their performance by optimizing application checkpointing, atomic checkpointing of memory state [131, 132, 139], and custom APIs to provide fine-grained control over checkpointing [133].

As these systems persist at ever finer granularities, they place much higher demands on the storage back-end in terms of throughput and latency. The key idea of all single-level stores is to unify virtual memory and storage in the operating system to make persistence simpler and more efficient. In contrast to the file API, SLSes provide a simpler interface for

atomically persisting changes across multiple objects on disk, without requiring tracking at the application level. Ultimately, this interface gives the single-level store more flexibility for its persistence layer, with the potential for better performance.

SLses have a unique workload pattern of high-throughput, write-mostly, synchronous, random, page-size IOs to disk. In systems like Aurora, these transactions are created by the checkpointing at some set frequency, while systems like MemSnap provide microsecond persistence for individual application transactions. These requirements of high throughput and low latency have prompted the developers of these systems to create custom storage subsystems for their needs.

Existing SLS systems gain most of their existing performance gains from optimizing checkpoint creation, but still suffer from bottlenecks in their storage system that limit their usefulness. For example, MemSnap is an SLS that exposes a fine-grained SLS API for latency-sensitive applications such as databases. Through this API MemSnap creates checkpoints with minimal overhead, and its storage system has design limitations that limit its performance for many workloads. First, MemSnap’s storage system contains optimizations to reduce metadata overhead, which lead to cases in which data integrity was at risk. Second, MemSnap scales poorly with multiple threads due to bottlenecks in the on-disk tree structures that underpin the entire storage system. Third, MemSnap’s data store ages poorly as performance degrades over time due to fragmentation.

To truly unlock the potential of the SLS architecture, we require novel storage subsystems that satisfy two performance goals: high throughput and low latency. Goodput is a bottleneck for applications, which is a challenge because small transactions have large metadata overheads that limit the available storage bandwidth. Latency is critical for performance because low-latency SLses almost exclusively perform synchronous writes, and storage latency is the dominant factor in end-to-end latency. High tail latency thus limits the practicality of SLses like MemSnap as a replacement for userspace mechanisms such as write-ahead logging.

Existing conventional file systems fall short of these performance requirements for IO patterns typical of SLses. For example, in modern file systems a 4 KiB write and `fsync` causes approximately 24–28 KiB of additional metadata to be written to disk. The high metadata overhead limits the overall goodput available to the application by 6–7×. File systems also typically sacrifice latency to gain additional throughput. Minimizing persistence latency requires excessive use of `fsync` common in databases and key value stores. However, `fsync` targets whole-file persistence and is inappropriate for SLses where the dominant IO pattern is small, random, and synchronous writes.

This paper introduces ObjSnap, a storage system designed for single-level stores running

on commodity SSDs. We present the first fine-grained design analysis and evaluation of an SLS object store along with their unique challenges. ObjSnap extends the μ CheckpointAPI into the storage layer and maintains correctness and performance under highly concurrent loads. Specifically, ObjSnap includes a mechanism *checkpoint coalescing* that achieves high throughput and low latency.

ObjSnap is a copy-on-write object store which uses its *checkpoint coalescing* mechanism to amortize the cost of metadata across thousands of μ Checkpoint’s while simultaneously reducing contention in the system. Checkpoint coalescing occurs in two phases, in which threads first flat-combine unrelated transactions into a single IO. Second, a checkpoint worker amortizes the cost of metadata updates across thousands of μ Checkpoint’s.

We evaluate ObjSnap against state-of-the-art, production file systems and show that ObjSnap is able to outperform existing traditional file systems by 4x, while producing 2–3x less metadata on average. We further replace the object store of a state-of-the-art SLS, MemSnap, with ObjSnap and show it improves performance, scalability, and consistency, while reducing metadata bloat, thus improving the lifespan of underlying storage devices.

2.1 Background

Single-level stores provide an alternative storage interface for applications that eschew the file IO interface in favor of integrated virtual memory and storage abstraction. This simplifies the development and correctness reasoning of persistent applications. Several research and commercial systems have implemented SLSES [72, 127, 132, 133, 139], but none of these systems have focused on the problems of building the storage backend, rather they focus on the challenges of checkpointing application, OS and memory state efficiently and consistently. The storage layer has largely held all these systems back. ObjSnap is the first system to solely focus on providing an object store for single-level stores.

Single Level Stores Single-level stores provide an alternative to traditional file system APIs via a unified storage and virtual memory hierarchy. These systems integrate application checkpointing, the virtual memory subsystem, and an object store to provide a simplified abstraction for persistence. Single-level stores use *incremental checkpointing*, which tracks changes in memory (application and operating system state) that have occurred since the last checkpoint and serializes them into an on-disk checkpoint.

Single-level stores such as EROS [127] and KeyKOS [72] provide persistence at a coarse granularity of once every 30s. Aurora [132], maintains the vision of fully transparent

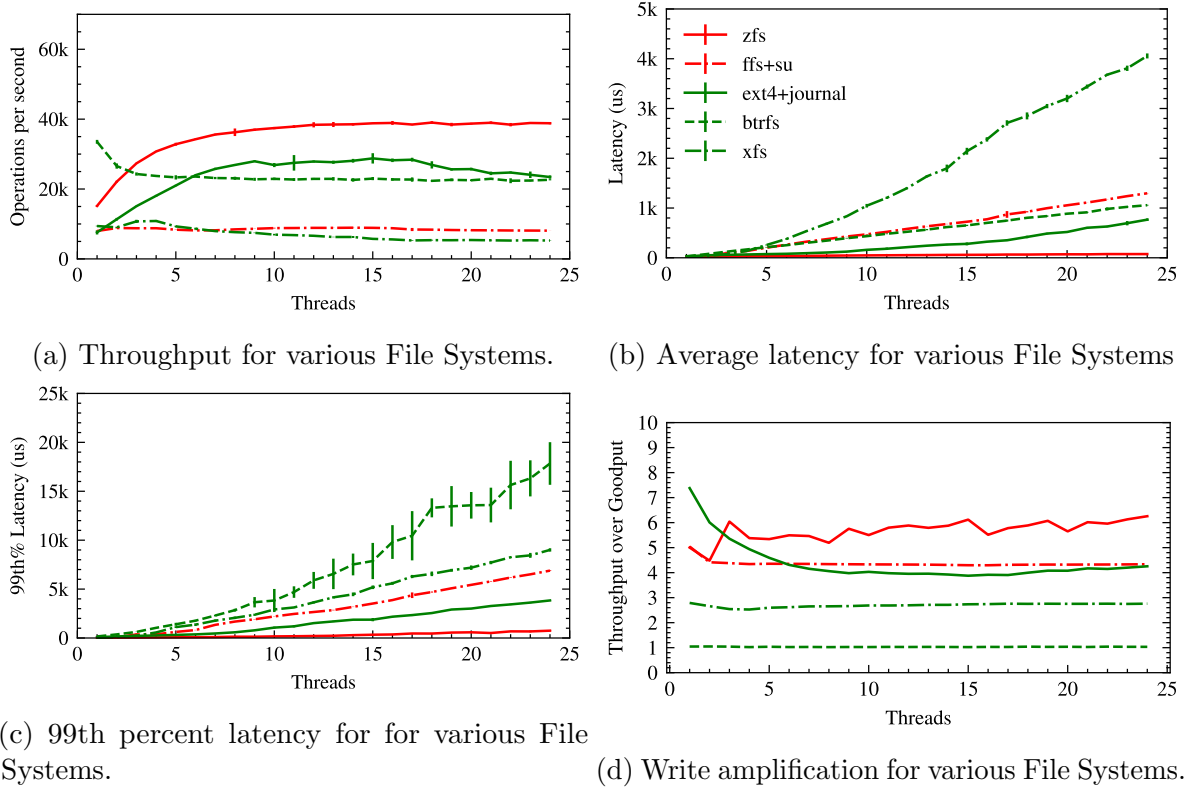


Figure 2.1: Performance and scalability degradation for various production grade file systems under a multithreaded workload doing small synchronous IO to a single file.

application persistence every 10 ms, however, the object store cannot checkpoint fast enough for applications that would benefit the most from this persistence model (i.e., databases [11, 25, 30, 35]). Furthermore, global checkpointing requires stopping all threads to consistently capture the program state, leading to large reductions in overall application performance. TreeSLS [139] is designed specifically for expensive persistent memory class storage devices, which have largely lost popularity and are currently dropped by its main manufacturer [3].

MemSnap [133] overcomes the bottlenecks in the application checkpointing provided by Aurora. MemSnap provides a finer-grained, thread-level persistence checkpointing mechanism, allowing it to be used in performance-critical applications such as databases. However, MemSnap’s object store suffers in three main ways.

First, the object store’s performance is limited by on-disk write amplification induced

by its copy-on-write data structures as is common to most COW based storage systems. MemSnap attempts to alleviate these problems by deferring the flushing of certain tree metadata. However, delaying flushing of this metadata with each transaction results in correctness issues compared to a completely correct implementation.

Second, MemSnap’s disk throughput cannot scale above 8 threads. Scalability bottlenecks exist within highly contended data structures in the object store, such as MemSnap’s global metadata inode, and within the kernel buffer cache and virtual file system [81].

Third, MemSnap’s object store ages poorly because many small allocations result in the storage becoming fragmented. Fragmentation reduces the overall performance of the storage stack and devices.

2.1.1 File Systems

File systems are the most prevalent persistence mechanism. File systems provide a strongly consistent global view of persisted objects for all applications on a host. File systems efficiently achieve consistency and persistence using one or more techniques, including copy-on-write [46, 84, 120], log-structured [82, 122, 140], journaling [34, 114, 129, 129], or soft-updates [64, 96]. However, no matter the design, the POSIX APIs these systems expose lack the qualities required by fine-grained transactional systems, e.g., multi-file atomic writes.

While file systems are an important starting point for any storage system, single-level stores require atomic multi-object updates on the disk to enable applications such as databases. Traditionally, file-based databases depend on complex userspace storage systems that write data two or more times through write-ahead logs or similar mechanisms to achieve these semantics.

Copy-on-Write File Systems and Object Stores Copy-on-write file systems have a strong consistency guarantee due to its non-destructive writes and the ability to maintain multiple versions of its objects. These systems typically expose light-weight checkpoints/snapshots for backup and recovery. However, these are designed such that these checkpoints occur at the microsecond granularity as opposed to the submillisecond granularity. Systems like ZFS have complex checkpoint pipelines which help with duplication, check-summing, and compression. However, COW systems suffer from large amounts of metadata that are induced by small, persistent write workloads within databases. Large amounts of file system metadata reduce the available throughput left for the application while also reducing the storage devices’ lifetime.

Motivation for a New Object Store To illustrate the problem with existing file systems, we measure the efficiency of ZFS [46], Btrfs [120], XFS [129], ext4 [93], and FFS [95]. Figure 2.1 shows the results of using the Fio [33] benchmark, where multiple threads synchronously write to a random page in a 10 GiB file. We plot each file system’s performance relative to the number of threads concurrently writing. We ensure that each file system is using a 4 KiB block size.

Figure 2.1 shows four graphs: Throughput, measured in operations per second, average latency, 99th-percentile latency, and the ratio of disk throughput to application throughput (i.e. goodput). In the first three graphs, we see that no system scales well. ZFS has the best scaling due to underlying mechanisms like its file range locks, but suffers from contention on disk bandwidth caused by metadata-induced write amplification. 85% of ZFS’ throughput is used by metadata updates, reducing goodput such that 8 threads can saturate the device.

The fourth graph shows the overall efficiency of each file system’s storage throughput. This metric determines an upper bound on the operations per second an application can perform given a file system and throughput of a storage device. On average, ZFS writes 5 to 6 pages for each operation (4 to 5 pages for metadata, 1 for user data). For a high end PCIe 3.0 SSD with a maximum throughput of 2 GiB/s, ZFS saturates the SSD throughput with only 35k to 41k user operations per second. Btrfs maintains a low overall write amplification at the cost of the highest average and tail latency.

ZFS even at one thread has 25–50% more metadata than other systems due to its usage of an intent log and periodic COW based snapshots. The intent log records operations to ensure that ZFS complies with POSIX semantics. The variability in the ZFS metadata is because the benchmark may be completed before or after a checkpoint was completed. ZFS can delay checkpoints up to 30 seconds, leading to large bursts of metadata to the disk.

This experiment shows that conventional file systems do not provide the performance required for SLSes because they suffer from non-obvious bottlenecks that only manifest for SLS-specific IO patterns. Accommodating these patterns requires us to rethink not just file system design but also parts of the storage stack like the buffer cache, to adapt for small, random, write-mostly IO.

2.1.2 Our Approach

ObjSnap is a copy-and-write object store designed to reduce metadata overhead through *checkpoint coalescing*. ObjSnap is designed for single-level stores, such as MemSnap, that generate a write-heavy random IO workload. ObjSnap extends the concept of *μCheckpoints*,

which are atomically persisted per-thread transactions, into the storage stack to solve the problems in existing SLS storage systems.

ObjSnap takes advantage of SLS semantics for its checkpoint coalescing mechanism, which amortizes the metadata overhead under highly concurrent workloads. ObjSnap uses *flat combining* to journal updates, and a background *global checkpointer* to update object metadata. The combination of these two mechanisms amortizes metadata between threads, and coalesces μ Checkpoint’s even further into a single global checkpoint.

Lastly, ObjSnap uses a generational allocator to achieve the goal of constant performance as the system ages. The generational allocator uses μ Checkpoint’s as a way to offload work to checkpointing threads to move old data to cold regions on disk, thus continually defragmenting the store.

2.2 Design Overview

Single level stores transform application memory reads and writes to storage reads and writes to persist application state. The page granularity of application memory writes appears as a mostly random workload. In turn, this means that from the perspective of the object store, the checkpointing mechanism is a random, write-mostly, transactional workload. In the limiting case, SLS systems will create single-page checkpoints [133].

To provide an atomic snapshot of application state, single-level stores must temporarily pause processes within an application, to serialize various kernel states (e.g., file descriptors and thread state), and ensure virtual memory pages cannot be modified while its memory regions are being checkpointed.

Checkpoint throughput and latency determine the application overhead. Faster throughput allows the single-level store to achieve finer-grained checkpointing. Lower latency improves application performance, since applications must explicitly wait for checkpoints to flush to disk or rely on external synchrony [102] for transparent durability that adds checkpoint latency to perceived application latency [132, 133, 139].

The object store must aggregate random writes into fewer sequential IOs and reduce metadata overhead as much as possible to maximize performance. Applications typically dirty memory pages more than files and further write to memory more often than write to disk. As discussed previously, file system metadata overheads scale poorly when using direct IO or frequent `fsyncs` with small 4 KiB writes, reducing the overall throughput available by 3–5 $\times$. To ensure that our latency objectives are met, a fundamental goal is to

minimize metadata overhead while ensuring that random writes are sequentially written on disk to maximize disk throughput.

The dual goal of a high throughput and low latency object store is difficult to achieve with a general storage system such as a traditional file system. Unlike traditional file systems, ObjSnap benefits from the end-to-end principle and our ability to tune how the single-level store’s checkpointing system presents checkpoint data to the object store. This enables simplifying assumptions and the use of a custom interface that will help us achieve our performance goals.

Assumptions The ObjSnap design makes the following assumptions about the data presented to it. First, page-level locking is handled by the checkpointing mechanism or by the running application. That is, the write set of two μ Checkpoint’s occurring simultaneously will be disjoint, meaning any two threads concurrently persisting a checkpoint must not share a page between their write sets. Second, the staging of user data into traditional caching systems, such as the buffer cache, is not necessary, and transactions should take the shortest path to disk once called. In an SLS, user data are cached by the virtual memory subsystem. Third, we assume that all writes are submitted through a checkpoint; this could be a larger checkpoint in the case of transparent checkpoints or smaller checkpoints such as those submitted through an SLS API.

ObjSnap assumes that there are few on-disk objects. Objects within ObjSnap are either large objects or aggregations of smaller kernel objects. Larger objects represent applications memory regions. For example, a SQL-style application would allocate a large region of memory for each of its tables. Aggregated objects are groups of similar kernel data structures that have been grouped by the SLS and passed to ObjSnap for checkpointing, for example, file descriptors and thread state.

Challenges To achieve the goal of a low tail latency and high-throughput object store, ObjSnap must overcome several design challenges. First, the bandwidth reduction induced by the use of copy-on-write object designs. COW greatly simplifies many of the semantics necessary for ObjSnap; however, COW systems create large amounts of metadata overhead for its write operations.

Second, overcoming system contention at all levels of the storage stack. Global on-disk or in-memory data structures are synchronization points (e.g., file system trees, or super blocks) whose contention can reduce overall throughput at high thread counts. For example, the usage of mechanisms like the buffer cache is particularly challenging. The buffer cache

is a useful and required mechanism for the metadata of on-disk data structures and provides the basis for memory over commit. However, this mechanism is a bulky system which interacts with every subsystem of the kernel, leading to larger amounts of contention that is directly in the write path of a checkpoint.

2.2.1 ObjSnap

The ObjSnap design is based on the idea of *checkpoints coalescing*, which amortizes the cost of checking in two phases. First, ObjSnap uses the assumption of disjoint write sets between threads to flat-combine them into a single μ Checkpoint. This reduces the write amplification induced by journaling, while having the additional benefit of aggregating user data into a single sequential IO.

Second, ObjSnap uses a global checkpoint worker to apply on disk μ Checkpoint's to the object trees. This worker reduces overall buffer cache and object contention as it is the sole writer of the underlying object B-trees. Furthermore, batching updates to the trees reduces the write amplification induced by copy-on-write.

ObjSnap performance is critically dependent on *IO short cutting* that ensures that no read operations occur in the write path. This is critical as dispatching read IO's in the main transaction path pushes that read latency to all threads currently being combined in a checkpoint. We use this in two points of the design. First, our moving allocator, required to ensure sequential IO, ensures that data blocks that need to be moved are warm in the OS buffer cache. Second, ObjSnap's journal entry metadata is kept in memory for the global checkpoint worker to efficiently update the trees with the metadata (i.e., key-value pair) in each journal entry.

ObjSnap's main mechanisms were designed to support high-throughput, low-latency workloads required by modern SLSes. However, all its main mechanisms could be applied to other storage systems. To reduce contention, object stores could make use of flat combining discrete write sets, or to ensure consistent performance, could use the generational moving allocator as part of their copy-on-write design to maintain sequential writes.

2.2.2 ObjSnap API for Single Level Stores

Table 2.1 shows the ObjSnap API. The API is a slim, page-oriented, checkpoint interface that persists checkpoints generated by an SLS. Objects are opened or created through `objsnap_open()` / `objsnap_create()` API calls. The single level store can read or fault in

Function	Description
<code>object_id = objsnap_create()</code>	Create a new object.
<code>object_id = objsnap_open(name)</code>	Open an object.
<code>objsnap_read(object_id, offset, size, page)</code>	Read a page.
<code>objsnap_checkpoint(page_vector)</code>	Create a checkpoint.

Table 2.1: The API provided to the single level store checkpointing subsystem.

pages through the `objsnap_read()` call. Writes to disk are only done through a checkpoint call, `objsnap_checkpoint()`, which persists a set of pages with a single operation.

2.3 Design

Figure 2.3 shows the main components of ObjSnap. ObjSnap uses a flat combining [75] mechanism, which coalesces concurrent thread transactions into a single IO and journaling operation. By using flat combining, ObjSnap creates larger, more efficient IO's, and reduces overall contention in the system. Journal entries are coalesced with our global checkpointing worker, which commits their contents to the trees. Our disk storage layout consists of a superblock that contains the global transaction number, a region of blocks that hold journal entries, a region of inode blocks, and finally user data. The generational moving allocator manages the user data region to reduce fragmentation.

To better understand the design of ObjSnap and how it achieves its goal of a high throughput, low latency checkpointing object store, we will follow a single page checkpoint through the system. The SLS creates or opens an ObjSnap object to correspond to any persisted memory region of an application. To create a checkpoint, the SLS calls `objsnap_checkpoint` and passes to it the dirty page as an argument. The dirty page is the thread's write set for the checkpoint.

During a checkpoint, the first set of metadata reductions occurs through a flat-combining process. ObjSnap uses an array to represent the current checkpointing state of each thread (i.e., `READY`, `CHECKPOINTING`, `DONE`). The thread first indicates that it is ready to checkpoint by atomically setting its state to `READY`. The checkpointing thread will wait for a few microseconds. This gives any thread currently combining the opportunity to claim the checkpointing thread's write set by setting its state from `READY`, to `CHECKPOINTING`. After waiting, the thread checks whether its state has changed. If the state changes during this wait, the thread sleeps until its state is set to `DONE` and then exits.

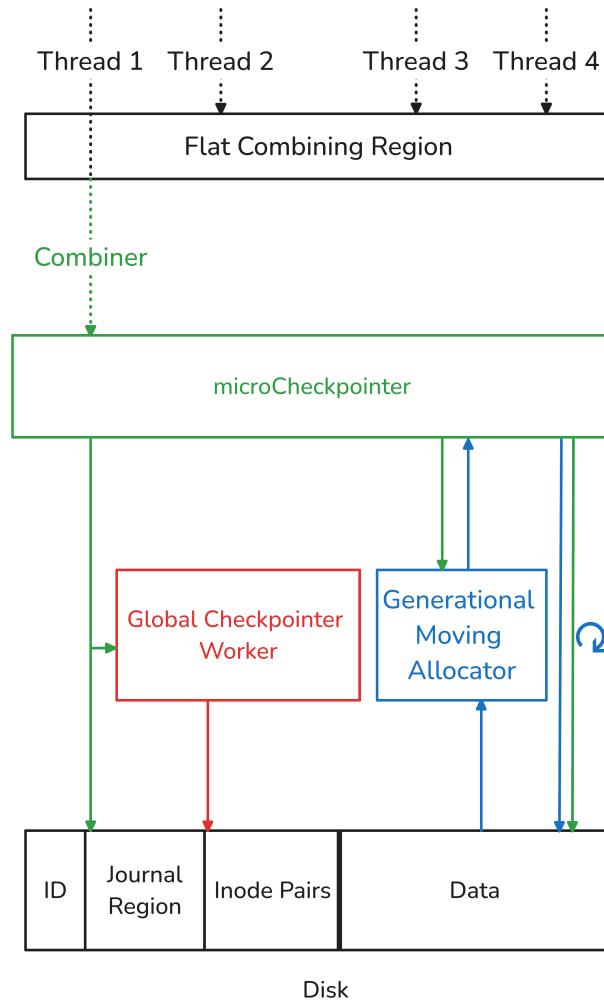


Figure 2.2: The main components of the ObjSnap design. ObjSnap’s checkpoint coalescing mechanism consists of the Flat Combining region, which selects combiner agents to combine other threads, and commits all thread dirty sets as a micro checkpointer. The global checkpointer worker aggregates checkpoints in our journal region on disk and commits them. The generational, moving allocator, will provide data to move for combiners and maintain performance consistency over the lifetime of the SSD.

If a thread completes its wait without another thread claiming its write set (i.e., changing its state), it will become a combiner. Once a combiner, it first attempts to claim its own write set, and then iterates over each thread currently in the **READY** state and attempts to claim its write set as part of its own. If the thread fails to claim its own write set, it still continues as a combiner. From here, the combiner will allocate the blocks needed and append the pages marked for moving by the allocator (see Section 2.3.2).

The combiner allocates a journal entry for the checkpoint, which is a list of tuples, each of which corresponds to the data pages being written. Each tuple contains the ID of the object to which the page belongs and the logical index within that object. Journal entries are 4 KiB blocks that reside in a contiguous 100 MiB region at the start of the disk. ObjSnap uses this region as a circular queue, where the head pointer is bumped during journal allocation. During checkpoint coalescing, the background worker will remove entries from the region by bumping the tail pointer, indicating that those entries are no longer needed.

After writing the tuples to the entry, we associate it with a newly allocated transaction ID, write the data, and then write the journal entry. Once these writes complete, the combiner sets the state of each thread it was combining to indicate their completion, and finally checks its own to ensure its done, this is required as the combining agent may not have successfully added its own write set to the checkpoint.

This process forms our first reduction of metadata and aggregates smaller checkpoints into larger, sequential IOs. In this example a single page, without write combining, would incur overhead from the journal entry write. Combining even with a few other dirty pages amortizes this cost and reduces contention on the buffer cache.

Avoiding Journal Contention ObjSnap ensures that operations do not interfere with the journal and the flat combining by guaranteeing that they can be handled by the flat combiner or that they do not need to interact with the journal at all. ObjSnap data writes are handled through flat combining regardless of their function. This includes both user transactions and data moves by the allocator (see Section 2.3.2). Metadata writes are performed exclusively by the checkpoint worker and progress independently of data writes. Read operations also do not require reading the journal, as they are handled by a combination of the virtual memory pager and volatile data structures to resolve reads to a physical disk address. Read operations are infrequent during steady state in modern SLSes because they do not target overcommit scenarios, and are prevalent during system startup and application restore.

Versus write-ahead logging Write-ahead logging in user space is used to overcome the lack of multi file atomicity within traditional file systems. Databases use WALs to aggregate transactional data into one file, before flushing the data to the underlying data structures that may be spread across many files. However, this comes at the cost of double writes, where data must be moved from that file to the correct place, resulting in write amplification. File systems like ext4, use journaling to protect against metadata inconsistencies incurred by traditional file system operations (e.g., `mkdir`).

ObjSnap’s journal is designed to support multi-object atomicity , and reduce metadata writes (i.e., tree blocks) in the main checkpoint path, reducing latency. ObjSnap’s combiner thread will write data to the data region of the disk, with the global checkpointing worker eventually updating trees to point to already written data, resulting in no rewritten data, at no cost of crash consistency.

2.3.1 Global Checkpoint Worker

The global checkpoint worker is responsible for periodically draining the journal area, updating the object trees with new blocks, and reclaiming any overwritten data pages back into the allocator. The worker amortizes the cost of updating the COW object trees by batching hundreds to thousands of updates into one operation. This batching avoids the need for COW operations between successive updates, and so reduces the write amplification normally caused by COW on metadata blocks. The worker also combines updates to the tree with block reclamation, freeing any overwritten blocks to back to the allocator.

The worker is a background thread that periodically wakes up and measures the amount of data written to the journal. If the amount exceeds a threshold, the worker checkpoints the journal by removing the oldest entries from the journal region and using them to update the trees. For each (`object`, `offset`, `data`) tuple found in the journal, the worker updates the tree corresponding to `object` with the key-value pair (`offset`, `data`). During these writes, the worker also creates new tree nodes as required for COW.

ObjSnap amortizes the cost of these copy-update operations using the checkpoint worker to batch tree updates. Each of these key-value updates would normally require a set of copy-update operations across the path from the leaf to the root node, but with batching, multiple keys may have partially or completely overlapping paths. This overlap reduces the total tree metadata generated for these transactions.

The worker finishes the checkpoint operation by updating the object metadata structure to point to the new root. Each object has two root pointers, one for each of the last two updates to the tree. The worker updates the object with the new root node by overwriting

the older of the two pointers with the newly created tree root for the object. ObjSnap uses transaction numbers to determine which of the two trees is newer: the root node of the tree is assigned the latest transaction number in the update.

ObjSnap minimizes the latency of the checkpoint worker by keeping all tree nodes in memory to avoid reads during tree updates. We use B-trees, which are short and wide, and so keeping the nodes in memory has minimal overhead, while it allows the worker to quickly consume journal entries by updating the trees with memory operations.

The worker also performs memory reclamation during tree update operations. The worker retrieves any blocks in the trees whose offsets are being written to and, since these blocks are now inaccessible, adds them to a dead list. Upon successful completion of a checkpoint, the worker iterates on the dead list and frees all blocks back to the allocator.

2.3.2 The Generational Allocator

The generational allocator optimizes the data layout on disk to maximize sequential writes throughout the object store’s lifetime. The allocator allocates 4 KiB pages from 8 MiB segments. The segments are classified as hot or cold, giving us two generations to minimize garbage collection churn. The allocation strategy is SSD friendly and prioritizes writing and garbage collection at the same granularity as the disk flash translation layer (FTL) to improve performance and increase device lifetime, inspired by observations inspired by previous work [87]. Combiners will move data on the allocator’s behalf by adding moving data to its write set.

The allocator groups the blocks into segments which are physically contiguous groups of pages on the disk. Segments start empty and are allocated sequentially until full. Each segment also contains a backmap that includes the starting offset of the segment and an array of object ID (16 bits) and object page offset (32 bits) pairs for each 4 KiB page it manages. The backmap enables segment cleaning by making it easy to identify which objects are still using the segment. The segment metadata consumes 0.15% of disk space in memory for its metadata.

Once a segment is completely full, it is placed in an in-memory linked list of hot segments. When the allocator requires more free segments, it looks at the head of the hot list (i.e., the oldest hot segment). The allocator then creates a move transaction using the backmap. During an allocation, a combiner passes its checkpoint structure to the allocator. The allocator will ensure that the write sets of its own checkpoint is disjoint with the checkpoint of the combiner. Any collisions will be resolved by removing the overlapping pages from the allocator checkpoint.

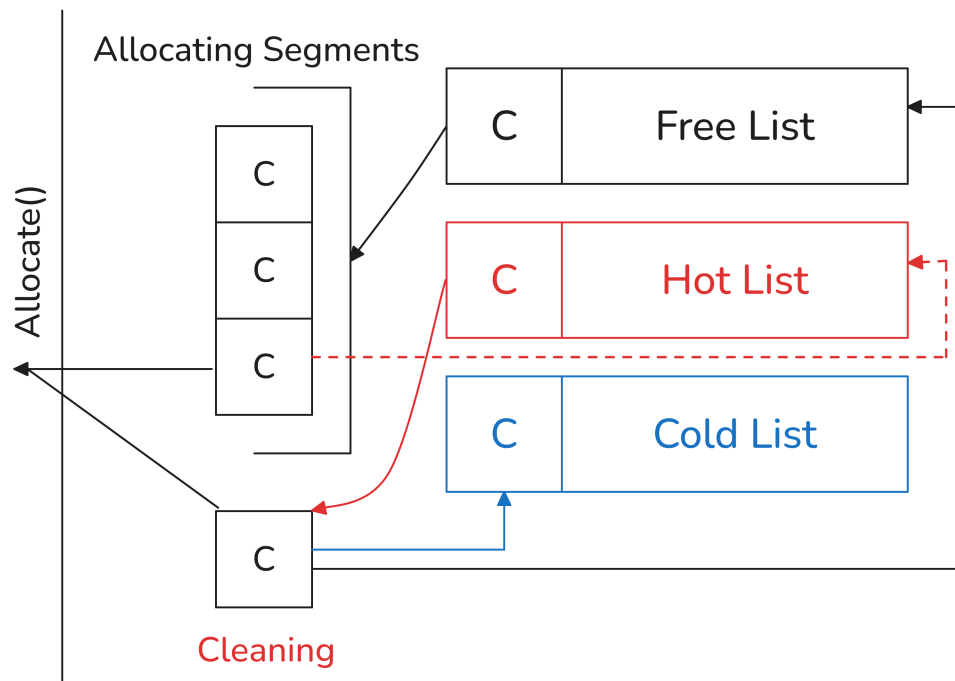


Figure 2.3: ObjSnap's Generational Allocator. During an allocation, the allocator will allocate from a segment sequentially until full. Once full, the segment is placed onto the tail of the hot list. The allocator will eventually start moving data that is still resident in the head segment of the host list. These pages are moved by a combiner threads by appending a move set to the current transaction. The combiner thread will perform IO to move pages to a cold segment

Blocks of data that are continually written to are considered hot because frequent writes ensure that the latest version of the block will be in a recently allocated segment close to the tail of the hot list. Previous versions of the block will be freed by the background checkpointer, without ever reaching the cold list.

Pages of data written through an allocator move is considered cold. The move will cause the original block to be freed by copy-on-write, while the moved block is now stored in a cold segment. The allocator cleans the data by embedding move operations into existing in-flight transactions. This avoids any contention between the allocator and threads doing data writes. The allocator prefetches the to-be-cleaned data before passing them to the combiner to prevent costly read IOs.

The background worker will continually re-evaluate candidate cold segments to clean, based on how empty the segment is. During allocation, the allocator will choose to clean the head of the hot list or the best candidate out of the cold list. The cold list scanning occurs every 100 global checkpoints.

The generational allocator is designed to work well with the applications used within SLSeS. We observe that memory regions within applications are either very hot and highly written to (e.g., database buffer caches) or very cold (e.g., memory-mapped libraries). The allocator naturally partitions the data into hot and cold sets and will converge to just the hot data that remain resident in a set of hot segments. This partitioning results in fewer required moves to clean a segment, as the life cycle of pages within a cold segment and hot segment is disjoint.

2.3.3 Recovery

Recovery after a crash or during remount involves rebuilding the in-memory state of ObjSnap's various data structures. We use the assumption that there are few objects, so we can read inodes and their metadata quickly. Each inode requires 8 KiB of disk space (4 KiB for each version of the inode), meaning 20k objects requires 150 MiB of sequential reading. We first read our global transaction number located at the head of the disk, and we scan our inodes to check that each inode transaction number is older than the global transaction number. If any transaction number is larger, we know that this inode is a part of an incomplete checkpoint, and we use its previous version. No data is lost from an incomplete global checkpoint, as the data is accessible through the journal entries.

We populate our in-memory segment structures for the generational segment allocator by reading the on-disk data structures. We read the state of each inode object tree to determine the data on disk is allocated. We never need to read the actual data, but rather

just the B-Tree nodes. We then read our journal region by reading in all its entries (1 GiB region) to further find allocated data on disk. We aggregate the chunks that are free (have no data within it) and place all other chunks on the hot list, in order of most empty to least.

ObjSnap rebuilds the state of our journal. We determine the head and tail of the journal region by checking our global transaction ID with each journal entry. Any journal entry that has a transaction ID lower than the global has been checkpointed, and any higher has not. We always write user data first before writing the journal entry. The journal also places the transaction number in the final 512 bytes of the journal block, to ensure the metadata are written first. If we read a transaction entry larger than our own on the block, we know the user data it points to, and journal metadata has been written.

Recovery is bound to the bandwidth of our underlying storage. Suppose that we have a terabyte of data, with 2 GiB of read bandwidth. Suppose that our storage contains ten 64 GiB objects, and 10 k 512 KiB objects. We would need to read a total of around 200 MiB of tree data, 80 MiB of inodes, and 1 GiB of the journal region at its fullest. As each object is independent, each object is read in concurrently with each thread reconstructing the required segment-free lists as data is found. There is little connection on each segment's free list due to the size of the disk and number of segments. For this example, our recovery takes less than a second.

2.3.4 Implementation

ObjSnap is a single kernel module written in 4 KSLOC written in C for the FreeBSD 12.3 kernel. For all our benchmarks, we use dual Intel Xeon Silver 4116 CPUs (Skylake-SP) running at 2.1 GHz with 96 GiB of memory. For storage, we used a single Intel 900P PCIe SSD. Our worker host used to generate the load for our Memcached evaluation is an identical machine.

2.4 Evaluation

This section focuses on evaluating the write performance of ObjSnap in the constraint of smaller than memory workloads, as this has been the target workload for current single level stores. We first evaluate the performance of our checkpoint coalescing mechanism and generational moving allocator. We then evaluate ObjSnap against five state-of-the-art, production grade file systems. Finally, we evaluate ObjSnap as a storage system for

Wait(us)	Waiter (us)	Combiner (us)	Ckpt (Pages)	Write Amp.	
				Ckpt Off	Ckpt On
0	119	122	1.01	2.00	2.26
2	130	135	2.64	1.38	2.13
5	127	129	3.42	1.30	1.96
10	128	128	4.57	1.21	1.79
30	151	144	7.40	1.14	1.58

Table 2.2: The effects of wait times and our throughput efficiency. We run a single page write workload at 24 threads and track checkpoints that finished as a waiter, or a combiner agent and their respect latencies and amounts. We then show the average size of a checkpoint given that wait time (in pages). With checkpointing, we see a non-linear relation with throughput efficiency due to a trade off that occurs between μ Checkpoint throughput and breakpoints of the average checkpoint size, showing that a 2 us wait is optimal.

MemSnap, a state-of-the-art SLS, and compare ObjSnap with MemSnap’s original storage system. We perform a microbenchmark to show the effects of scaling within the two systems and then evaluate both systems’ performance when persisting the Memcached application.

2.4.1 Checkpoint Coalescing Breakdown

To illustrate the interactions between our flat combiner and the global checkpoint worker, we run a single page μ Checkpoint benchmark. Through this benchmark, we show that as the static waiting period performed by all threads before entering the main increases, the write amplification of the system decreases. However, this is at the cost latency, but due to the interactions between the global checkpointing worker, we see there is an optimal wait period in regards to the latency, metadata trade off.

We run a single page μ Checkpoint benchmark at 24 threads. The benchmark creates a 10 GiB on disk object, and dispatches a set of worker threads, which randomly select a page uniformly to write to within that object. We have each thread perform the same number of μ Checkpoint (200 k each), to ensure that there is an exact amount of work done. We then capture the total amount of data written to disk, to determine the total write amplification that had occurred during the benchmark. The results are shown in Table 2.2.

To show the write amplification cost of checkpointing more clearly, we have two runs at each wait period, one with checkpointing on and the other with it off. Intuitively,

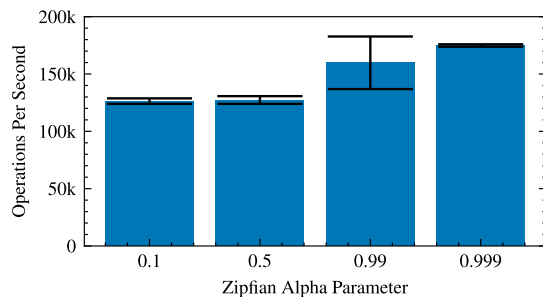


Figure 2.4: ObjSnap with a 30 GiB and 24 threads, each writing a 1 page transaction in a Zipfian distribution. The closer to zero the alpha parameter is the more uniform it is, the closer to 1, the more concentrated the hot set of the distribution becomes. We see that as we become more concentrated the more operations per second achieves as less moving is required to keep a steady stream of segments available for allocation.

as threads wait for longer, more μ Checkpoint’s are aggregated, and we tend towards a throughput-to-good-put ratio of 1. However, we see that the best trade-off for latency to write amplification is at a wait period of 5 us. This increases latency by 7% over 2 us, but reduces write amplification by 10%.

Waiting can increase latency; however, the dip in latency we see is because the waiter’s amortize the cost of writing to the journal with other combining threads. Eventually, the cost of waiting outweighs the cost saved, and latency increases. Write amplification steadily decreases, as journal entries are more compact, and the global checkpoint worker becomes gradually more efficient.

2.4.2 Generational Moving Allocator

To illustrate how the generational moving allocator can benefit from workloads that demonstrate a hot and cold working set of data, we run the single page μ Checkpoint workload as before, with a few changes. First, we use a 30 GiB object to ensure that there can be a larger cold set and run the benchmark for 5 minutes, so the moving allocator is properly stressed. Lastly, we use a zipf distribution for our workers over a uniform one. The zipf distribution takes a single hyperparameter to skew the distribution. When the hyperparameter approaches 0, the distribution acts uniformly, and as it approaches 1, the hot set becomes hotter and less large. The results are seen in Figure 2.4.

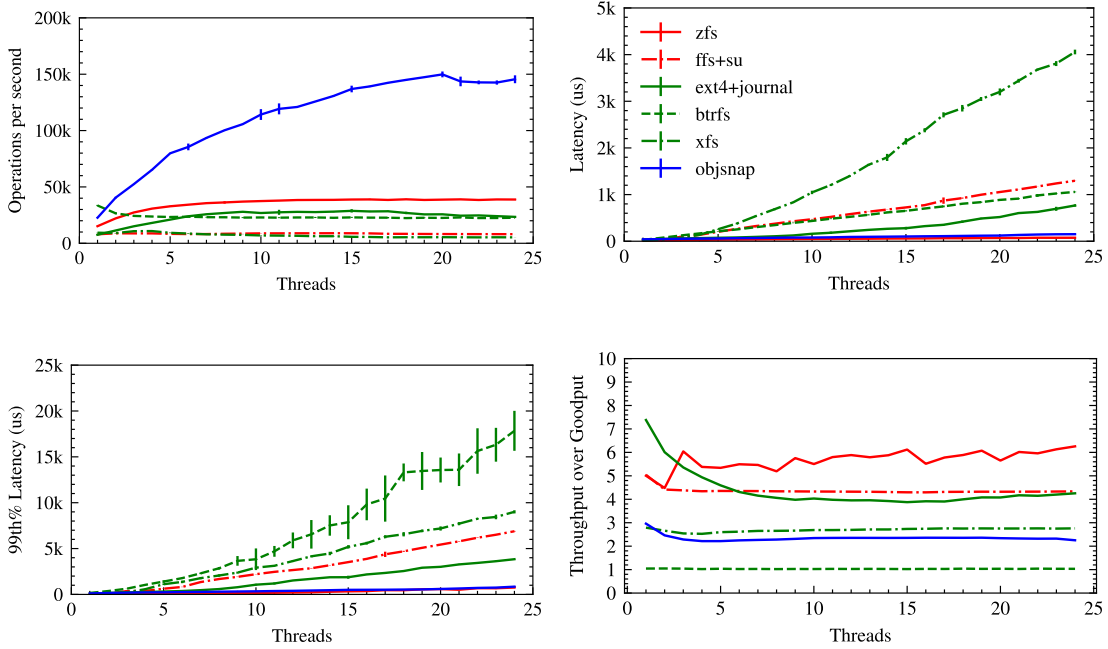


Figure 2.5: A comparison of ObjSnap versus various forms of traditional file systems, ZFS, FFS (with soft updates), ext4 (with journaling), and XFS. The benchmark shown is a single page durable write, in which up to 24 threads randomly dirty and write to a page, followed by an `fsync`, or `objsnap_checkpoint` call. Shown is the performance in terms of Operations per second, latency, tail latency, and throughput efficiency. Throughput efficiency is a measure of on disk throughput divided by application goodput.

As seen in Figure 2.4, as our hot set becomes more concentrated (i.e., approaches 1), our throughput increases because threads typically never need to move data for the allocator. As these hot pages are written to, copy-on-write causes the page to be moved to a new segment, and thus is farther down the hot list. By the time a segment reaches the head of the hot list, it is typically empty, meaning it can be moved back to the free list on an allocation. In this way copy-on-write acts as a passive mechanism for hot data placement, resulting in reduced work to move pages within a segment and increasing performance.

2.4.3 Versus File Systems

Single Page Write Performance. We evaluate ObjSnap’s small durable write performance against five different production grade file systems. We use a single page write workload, as this stresses the consistency mechanisms designed by each object store the most. We run our uniform single page durable write benchmark in each of the file systems, and compare it to ObjSnap. The benchmark is run for 5 minutes on each run to ensure that the moving allocator is properly engaged. We further constrain the size of our hot list (50 GiB) of our allocator so that moving allocations occur earlier to ensure that ObjSnap starts moving even at low thread counts. Within each file system, write and serialization occur through a `write` call, followed by an `fsync`, while in ObjSnap, it occurs through the `objsnap_ckpt` call.

To maintain fairness, we only use a single object as this is the closest in consistency semantics that can be offered by traditional file systems. Multi-object transactions, although supported by ObjSnap, are not supported by these systems without the use of a user space write-ahead log. Data journaling is enabled on ext4 to ensure protection against write tearing to ensure fairness of durability semantics compared to systems like ObjSnap, btrfs, and ZFS. FFS is running the soft-updates journaling mechanism.

Figure 2.5 shows the results of our experiment. As explained in Section 3.1, we see the amount of write amplification of the various file systems. In this graph, a throughput-to-good-put ratio of one is perfect use of the available disk bandwidth, meaning that every byte written to results in a byte written to disk. Among the traditional file systems ZFS, sees the best scaling in terms of latency and throughput, however, at the cost of almost 3x ObjSnap’s write amplification. Btrfs, surprisingly, shows little write amplification; however, its design shows a noisy and large latency, with writes on average taking 4 ms to complete at 24 threads.

Each traditional file system is designed to uphold POSIX style semantics and read heavy workloads, making them unsuitable for write heavy systems that require fine durability or multi-object atomics. Systems like Btrfs trade latency for write amplification, while systems like ext4 see better scaling in terms of latency and write amplification, however, cannot come close to saturating the available disk throughput. ObjSnap assumes a transactional workload, and takes advantage that memory caching is handled by the above layer (i.e., the SLS virtual memory subsystem). With this assumption, the ObjSnap reduced write amplification design allows for a higher efficiency use of the available disk throughput and thus is able to achieve greater performance.

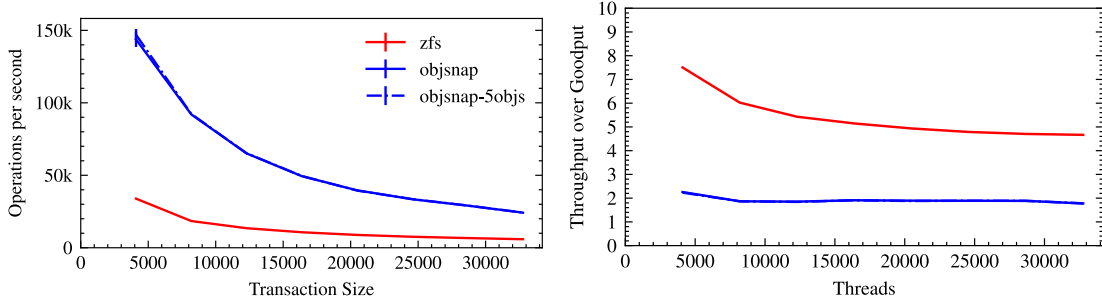


Figure 2.6: Multi page write performance of ObjSnap versus another copy-on-write object store, ZFS. The workload is pinned at 24 threads, and run the same workload with larger dirty sets of each thread. We further show a multi-object checkpoint with ObjSnap, in which the dirty set is randomly spread between five objects to show that the write path is independent of the number of objects.

Multi Page Write Performance. We next evaluate ObjSnap’s multipage write performance and write amplification. In this benchmark, we show three configurations: ZFS, ObjSnap using a single dirty object, and ObjSnap with five dirty objects. These two ObjSnap configurations are used to show that ObjSnap’s write path is agnostic to the number of objects. Furthermore, ObjSnap has the additional benefit that the multi-object write is atomic, unlike in a traditional file system. Figure 2.6 shows the results.

We see that ObjSnap outperforms ZFS in throughput and efficiency. ZFS gains efficiency as the dirty set size increases; this is due to better amortization of its various COW related overheads, but starts plateauing to a final ratio of throughput to good-put of almost 5. ObjSnap’s throughput declines steadily, as the latency for writes increases due to larger μ Checkpoint’s, until at 3 page μ Checkpoint’s our disk throughput (2 GiB/s) becomes saturated.

2.4.4 Supporting An SLS–MemSnap

We will now evaluate the original MemSnap’s scaling in a microbenchmark and compare it to MemSnap supported with ObjSnap. We run a single page write transaction workload as before and show the latency of both MemSnap and MemSnap supported ObjSnap at higher thread counts. We see, due to innate bottlenecks found within the original MemSnap object store, that durable writes scale greatly relative to ObjSnap. Large coarse grained locks alongside inefficient usage of throughput leads to 250 us latency at 24 cores within the

Threads	MemSnap(us)	ObjSnap(us)
1	35 us	41 us
5	57 us	58 us
10	101 us	81 us
15	146 us	105 us
20	184 us	118 us
24	238us	147 us

Table 2.3: In this table we show the scalability performance of the original MemSnap and ObjSnap. We perform a similar single page durable write and measure the latency. Due to bottlenecks innate in the design of MemSnap, we see as thread count rises, latency increases faster than ObjSnap.

original MemSnap object store. The jump we at 21 cores in latency for ObjSnap-supported MemSnap occurs due to NUMA imbalances, and as the workload fully saturates, the available hardware threads of the system NUMA effects become less pronounced.

Memcached We chose Memcached as our workload as it has two important qualities. First, the system allows for it’s data region to be within a single memory mapped region. Originally, this was used to support persistent memory hardware and can be easily modified to fit our use case. Second, it is an application which has a concurrent writers. Many key-value stores, which offer persistence, like RocksDB are unable to properly load ObjSnap as it is designed with the drawbacks of a file system in mind, and so have very few writers to the disk.

We use ObjSnap as the supporting object store for MemSnap which tracks pages that dirtied during write transaction. This required only a few lines of code as high-level transactional locks are already in place.

Memcached is set with 4096 byte value sizes, as this is required for key-value isolation needed by MemSnap. We use a 30 GiB memory mapped region within Memcached, and run it with 24 threads. We run `memtier_benchmark`, on an identical machine, running 12 threads, and each thread supporting 64 connections. We then vary the read, write ratios going from 90% read, 10% write to 10% read, 90% write. We run three configurations, the baseline Memcached, the original MemSnap, and then the MemSnap supported with ObjSnap. The results are shown in Figure 2.7.

From Figure 2.7, we see that ObjSnap moves toward its single page write performance as

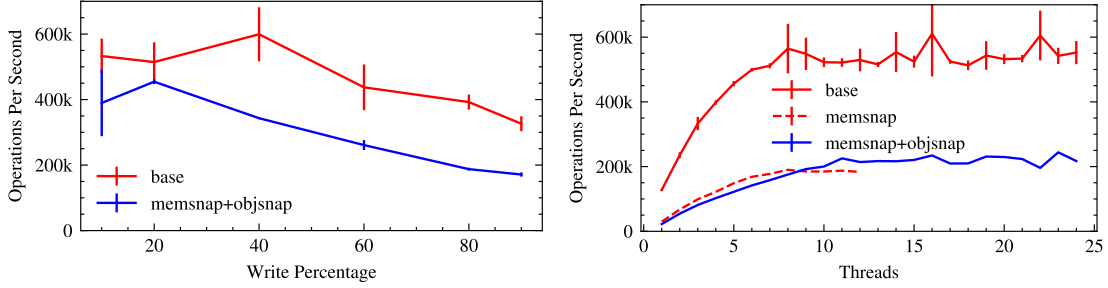


Figure 2.7: Memcached performance with ObjSnap. We use ObjSnap with the MemSnap SLS to transparently persist writes to a Memcached instance, and measure the throughput compared to a non-persistent Memcached instance. ObjSnap keeps the cost of persistence stable for write ratios of 10–90% and for up to 24 concurrent threads. By comparison, MemSnap’s original object store falls behind in performance after 9 threads, and crashes for more than 12 threads.

the write percentage of the workload increases to 90%. At even a 90% write rate, ObjSnap is able to persist all key-value pairs at only a 40% reduction in throughput to a pure in-memory Memcached. ObjSnap’s close gap to a fully volatile Memcached is due to its ability to use the available cores. Threads waiting for IO’s result in user requests spreading out between the various workers, while also reducing contention.

To illustrate how Memcached scales, we performed an additional experiment. We pin our read-write ratio to 50% reads and 50% writes, and run the same benchmark, but scaling the number of threads from 1 to 24 threads. We see ObjSnap continues to scale, while Memcached plateaus at 6–8 threads. Memcached persisted with ObjSnap is able to further close the gap even further with these available cores. In this experiment, the original MemSnap storage system was unreliable and was seeing consistent crashes due to not enough space on the disk for 12 threads and beyond, so it is omitted past this point. Even with this omission we see at 12 threads that the ObjSnap supported MemSnap surpasses the performance of the original MemSnap by 10%.

2.5 Related Work

Persistent Data Structures Write optimized object stores hinge on the use of highly concurrent modified B trees known as a B^ϵ -Tree [47, 52, 77]. B^ϵ -Trees are write optimized and are modified such that each internal node of the tree has an operation log attached to

it. Log-structured merge trees (LSM trees) are a data structure specifically designed to reclaim spacial locality of the disk for write heavy workloads [103], and whose primary user is RocksDB [11]. LSM’s use an append-only approach in which writes are aggregated into a single log file.

Non-Volatile and Kernel Bypass File Systems Various file systems focus on using non-volatile byte addressable storage devices [78, 118] to achieve higher throughput. Although other systems present a better API to directly bypass the kernel, reducing the cost of using storage devices from userspace [141]. Other work has looked at how to better optimize how the operating system interacts with its storage devices to improve overall system performance [143].

Memory Mapped IO Memory-mapped IO poses significant performance and correctness challenges. Current OS storage stacks often pose a performance bottleneck for memory-mapped IO, and existing work has attempted to optimize or bypass it [106, 107].

Persistence APIs for Traditional File Systems The limitations of existing file system persistence APIs have been well documented. Existing work has expanded conventional file system APIs with fast userspace journaling [128], atomic `msync` operations [108], multifile sync operations [135] or general transaction-like operations for faster persistence [110]. Other work also provides stronger persistence abstractions to conventional file systems using device support for atomic writes [79, 99, 138].

2.6 Conclusion

This chapter illustrated the design, implementation, and evaluation of ObjSnap, the fundamental storage system that enables SLSES. Through ObjSnap we enable the ability for SLS systems to efficiently checkpoint and restore, individual objects, or bundle these objects into large images for the purposes of checkpointing and restoring containers, as is done in both Aurora and Metropolis. In the next chapter, we will break away from storage and look at the next problem pertaining to containers: security.

Chapter 3

SlimSys: Fine-grained System Call Debloating using Resource Binding

Over the years, the system call interface has expanded to accommodate increasingly more complex applications. This expansion has resulted in the addition of many system calls to commodity operating systems. System calls expose a large attack surface for malicious actors to break out of OS protection mechanisms [73, 142] and gain full control of a host. However, most applications use only about 2–5% of all available system calls [112].

Software debloating is a well-studied technique that enforces the principle of least privilege. System call debloating systems, in particular, reduce the attack surface of the kernel API by determining the minimal set of system calls necessary for an application [58, 61, 68, 145]. Application debloating involves the removal of dead code paths or library code that was not used during the life of the applications [38, 40, 67, 69, 144]. These systems mitigate the available code that a malicious actor can use.

System call debloating systems transform the minimal set of system calls into a policy. These policies are enforced using systems such as Seccomp or Pledge, which ensures that only the system calls outlined in the policy are allowed [1, 44].

However, a large number of system calls share various subroutines in the kernel, often the most used, sharing the most code. These kernel restriction systems (i.e. Seccomp and Pledge) cannot account for this large amount of sharing. Further, current state-of-the-art debloating mechanism quantify their restrictions as the percentage of systems calls disabled versus actual code paths restricted. For example, on large applications like `lighttpd`, a perfect system call debloating technique reduces the number of available system calls by 91% but only reduces the kernel code attack surface by 29% (Section 3.1).

However, work has been done to create finer approaches [68], however, focusing on increasing the number of disabled system calls. As such, the most general system calls, which are used in every application (e.g., `mmap`, `read`, `write`), continually act as large attack surfaces, which expose vulnerabilities within the kernel.

We present SlimSys, an enforcement system that debloats the underlying kernel code as opposed to minimizing the system call API using our novel resource-centric approach. SlimSys introduces a novel model, *System call-resource binding*, which views an application as a set of well-defined system resources (e.g., files, sockets, event queues), rather than a global set of system calls. SlimSys defines resources by the system call and the arguments that create it. Each resource has its own fine-grained policy, which specifies the set of system calls and their associated arguments that are allowed.

SlimSys solves the unique challenge of working with dynamic arguments only known at run-time by introducing a mechanism known as *data forwarding*. For example, the IP argument for a `connect` system call is unknown until the program resolves the domain name at runtime. Data forwarding safely specializes the policies of resources based on dynamic state.

We introduce a novel analysis, which we use to generate our SlimSys policy. Our analysis combines a live trace of the application with a novel hybrid static-dynamic program analysis tool that creates the policy. The trace supplies vital context on dynamic state that is difficult to extract in larger applications, while the static analysis uses the trace and application binary to determine missed system calls and assist the developer in creating more robust policies.

SlimSys’s syscall-resource binding allows for resources to be restricted even further dynamically at runtime. For example, developers can enforce exactly-once semantics for system calls. This is useful for server applications that use `bind` or `listen` only once during execution [68].

Lastly, we evaluate SlimSys using a novel technique to determine the amount of kernel code reduced over the amount of system calls disabled. This technique involves using kernel fuzzing to determine the total amount of code accessible by a system call and kernel tracing to determine the amount of code actually run by a system call during application runtime.

We implement SlimSys’s syscall-resource binding with a userspace runtime, a kernel module, and a new program analysis tool. In our evaluation, we and show that SlimSys’s syscall-resource binding model reduces the kernel attack surface in common system calls by an additional 81% over Seccomp-based debloating techniques. A hypothetical perfect system call debloating for the `lighttpd` web server reduces the kernel attack surface by only 29%. In comparison, dynamic capabilities reduce the kernel attack surface by 84%.

Syscall-resource binding is an efficient mechanism that imposes an average overhead of 1% as opposed to Seccomp’s 3%.

3.1 Background and Motivation

Attackers use techniques, such as buffer overflows, to compromise vulnerable programs and run arbitrary code. The attacker’s goal is to escalate their privileges and bypass protection mechanisms, such as containerization and process-level security policies [12, 22, 90]. Breaking operating system protection mechanisms requires executing system calls to trigger and exploit vulnerabilities within the kernel. Compromising the kernel invalidates all OS-level isolation mechanisms.

Removing unused code is important to mitigating vulnerabilities within the kernel. Studies have shown that the number of bugs is proportional to the number of lines of code [125], with an average of 23 bugs per thousand lines of code [39]. Furthermore, kernel bugs are often latent in the code, requiring years to uncover [105].

For example, `mmap` can be used as an entry point for attackers to access vulnerabilities in multiple OS subsystems and drivers. `mmap` can be used to directly modify files, create private file mappings, and dynamically allocate anonymous memory. CVE-2016-1583 [18] attacks the implementation of `proc` file systems of `mmap` using sequences of `/proc` files as arguments to corrupt the kernel memory. Previous exploits [19] have used `mmap` on a device to execute vulnerable device driver code. Each new file type or device increases the attack surface of `mmap`, with device drivers being particularly vulnerable as they are often untested.

To mitigate these attacks, researchers have proposed several techniques for system call debloating [38, 58, 61, 66–69, 85, 124]. System call debloating limits the system calls available to an application to minimize the kernel’s attack surface, because most applications use a small fraction of all available calls.

System call debloating uses binary or source level analysis [58, 66, 67, 69] to determine a static set of system calls required by an application. Temporal [68] and SysXCHG [61] create policies that further constrain the application after known phases, such as initialization. Application debloating systems remove unused functions from libraries and applications [38, 85], or limit applications to only the most popular system calls [124].

3.1.1 System Call Disabling

System call debloating techniques currently use system call disabling mechanisms, such as Seccomp-BPF [1, 94] or BSD’s Pledge [44], provided by the OS to limit the system call interface. Users generate a policy, which is an allowlist/blocklist for system calls, for each application and pass it to the kernel after launching the application.

Systems like Seccomp-BPF suffer from two major limitations: first, they cannot make filtering decisions based off of system call arguments, as it cannot safely dereference user space pointers. Second, these policies are mostly static and cannot be changed or changed exactly once. This means that applications that require a system call are given access to it for its execution. Constraining a Seccomp policy, such as in the work of Temporal Specialization, can only ever occur once, as the system calls that modify policy (i.e., `seccomp` and `prctl`) must be removed on the transition, or attackers can use these system calls to modify their own policy.

3.1.2 Capabilities

Seccomp alone is not a complete sandbox and requires additional support for isolating applications. One such mechanism is Capabilities, which restrict an application’s functionality at process or file descriptor granularity. Capabilities that are attached to the process (e.g., Linux Capabilities) have a similar role to Seccomp. For example, `CAP_NET_ADMIN` captures all calls from network-related systems such as `bind` and `listen`. Process-centric capabilities are global, and an application that is given a capability retains it for its entire lifetime.

Finer-grained capabilities like file descriptor capabilities (e.g., Linux Landlock [4], or FreeBSD Capsicum [136]) help further refine the allowed functionality to a single file descriptor. However, these systems have three pitfalls: first, file descriptor capabilities function on parent-child relationships, meaning that acquiring specific capabilities requires providing a parent file descriptor that can grant the capability. For example, the `open` call is disallowed, and callers must use the `openat` call with the “at” file descriptor having the correct capabilities to open a file. This means that systems must be aware that they are running a file descriptor capability system.

Second, these capability systems lack the ability to revoke previously granted capabilities in response to some event. A malicious actor can aggregate many file descriptors with many different capabilities and use whatever file descriptor they require to achieve their goal. For example, an attacker can abuse a socket file descriptor with the connect capability to

Name	Syscalls	Blocks	Reduction
Kernel	277	19668	0%
Perfect Debloating	24	14033	29%
Temporal [68]	15	13205	33%

Table 3.1: Shows the limitations of system call based debloating and its ineffectiveness of reducing the attack surface of the kernel for lighttpd. Perfect system call debloating reduces the kernel system call API by 91%, but reduces reachable code by 29%, and Temporal Specialization only achieves an additional 4% reduction.

connect and communicate with any address. Lastly, these systems are not dynamic and have no ability to handle unknown user data.

3.1.3 Limitations of Software Debloating

Quantifying the Kernel Attack Surface Current system call debloating techniques restrict the system call API, which does not effectively reduce the overall amount of kernel code accessible to the attacker. To better illustrate this, we use a kernel fuzzer, Syzkaller [31], to determine the amount of kernel code that can be reached by an example application, lighttpd [5], an HTTP server. Fuzzing has been shown to be useful in understanding the reachability of code within a security setting [76].

Fuzzing is a useful technique to help better acquire a quantitative number of the kernel attack surface. By capturing the source lines of code traversed during execution of a specific system call during fuzzing, we can acquire a quantitative number of reachable code. This approach is general and has the advantage of being able to reach difficult to statically analyze code paths such as paths reached through function pointers. To determine an application’s normal kernel surface, we capture multiple sample runs of an application (i.e., through benchmarks and test cases) to understand how the system call interface is actually used during execution.

For many applications, this method acts as a lower bound of what is possible, as longer fuzzing increases the amount of found reachable code paths, increasing the kernel attack surface. For example, in server-style applications, in which the system calls and their arguments are more discrete, while also being some of the most vulnerable to external code manipulation attacks.

We calculate reachable code paths for lighttpd’s 24 system calls, using a modified

Syzkaller (Section 3.7). To determine these system calls, we ran `lighttpd` with an associated workload [14] and captured the used system calls. We then removed 9 system calls through the Temporal Specialization [68] technique and re-ran the kernel fuzzer. Fuzzing the 24 system calls measures the reachable code within the kernel with perfect coarse-grained system call debloating, as this is the exact set of system calls required by the application. To determine all accessible code paths for the entire kernel, we fuzzed all 277 system calls for 120 hours.

Table 3.1 shows the results of this kernel fuzzing, with the numbers within representing the *basic blocks*. A *basic block* is defined as a contiguous sequence of instructions terminating in a branch or jump and approximates lines of code.

A hypothetical perfect coarse-grained debloating removes 91% of the system call interface but only removes 29% of reachable code. Temporal debloating, a state-of-the-art technique, removes 9 additional system calls, but only debloats the kernel by an additional 4%. In this sample cast, we show that the removal of system calls does not effectively reduce the attack surface in the kernel as these system calls use large shared kernel subroutines.

3.1.4 Our Approach: Syscall-Resource Binding

SlimSys introduces a resource-centric model and enforcement mechanism called *syscall-resource binding* that uses a per-resource policy and replaces the Seccomp enforcement mechanism. Per resource policies bind system call argument pairs to a defined resource. For example, a TCP socket that can only connect `bind` or `connect` to specific addresses and whose `accept` call can only create sockets with specified policies.

The SlimSys policy provides crucial context to the enforcement agent about all allowed system calls during execution. The policy outlines a set of resources, defined by the system call and arguments that create it, with each resource having its own policy specifying the allowed system calls that can execute on it. In this way, no matter what file descriptor a resource uses, the policy remains the same as it was attached to the kernel resource during creation.

Defining resources as the system call arguments that create it means that we naturally separate allowed code paths and mitigate how attackers may use resources maliciously. For example, the `accept` system call is both an operation on a resource and a system call that creates one. Defining resources in this way naturally separates the policy for listening and connected sockets.

SlimSys’s syscall-resource binding can restrict applications further at run-time. The additional context given by a resource-centric policy can be used to define how they respond

to other system calls. We can specify a secondary policy to dictate the revocation of capabilities on specific resources. This secondary policy can be used to define the system call that causes an application to acquire a taint label and describe how resources lose capabilities upon gaining that label. For example, we can restrict a web server to use the `bind` or `listen` system calls only once. We can describe a policy in which, upon execution of either call, we remove all `listen` or `bind` capabilities.

3.2 Threat Model

We make the following two assumptions. First, we assume that an attacker has gained complete control over an application by exploiting an underlying vulnerability that allows arbitrary code execution. Second, the attacker carefully manipulates the injected code to exploit kernel vulnerabilities to escalate their privilege.

SlimSys minimizes the kernel attack surface available to the attacker by limiting the reachable code. Thus, attackers cannot exploit vulnerabilities that occur in unintended kernel code paths.

Attacks that use side channels to violate isolation are beyond the scope of this work. Time-of-check to time-of-use (TOCTTOU) attacks, which occur between system call argument validation and system call execution can not occur, are in scope of this work and are defened by our system as arguments used within SlimSys are fully copied before enacting policy.

3.3 Design Overview

SlimSys is a novel kernel level enforcement system that uses our system call resource binding model to mitigate the exploitation of kernel vulnerabilities. Applications run unmodified and with no knowledge of the dynamic capability mechanism underneath. Dynamic capabilities are lightweight and can be updated at runtime, allowing for features such as our data forwarding mechanism and revocation policy. Our work introduces both a new enforcement mechanism to replace systems like Seccomp and an analysis tool that creates a policy for our system-call resource binding model.

3.3.1 SlimSys

Figure 3.1 shows SlimSys’s three main components: the SlimSys service, userspace interposition shim, and kernel module. The figure further shows how our data forward binding updates kernel policy in response to dynamic data.

- The **SlimSys Kernel Module** enforces the SlimSys policy during system call execution.
- The **Interposition Shim** directly intervenes in specific C library calls to facilitate data forwarding. It helps in addressing the challenge of dynamic arguments that change between executions.
- The **SlimSys Service** is responsible for initiating the application and shim and installing resource policies into the kernel module. The service daemon communicates with the shim to perform C library functions on its behalf and safely update the policy.

SlimSys’s design follows the principle of least privilege, the core idea behind system call debloating. The advantages of argument-level debloating stem from two insights.

First, system call code paths are typically deterministic and depend on the unique tuple of arguments passed to the system call. This determinism is what attackers exploit to access known vulnerabilities in the kernel by crafting the arguments needed to reach vulnerable code. By enforcing allowed tuples of arguments, we ensure that a potential attacker is restricted to well-defined behavior, enforcing the execution within a known path.

Second, the arguments describe the intent of a system call on a resource. For example, an application may intend to use the `sendmsg` system call with a TCP socket. A malicious actor can manipulate the arguments to instead run the `sendmsg` system call on a UDP socket, resulting in exploitation [23]. By forcing system call and argument tuples to be bound to a resource, we ensure that system calls can pass only to the intended resources.

Data Forwarding

SlimSys syscall-resource binding uses data forwarding to update kernel policy in response to dynamic arguments only known at runtime. Figure 3.1 shows an example of this process in which an application attempts to make a DNS call. The application transparently calls the `gethostbyname` shim function, which serializes the arguments to the SlimSys service.

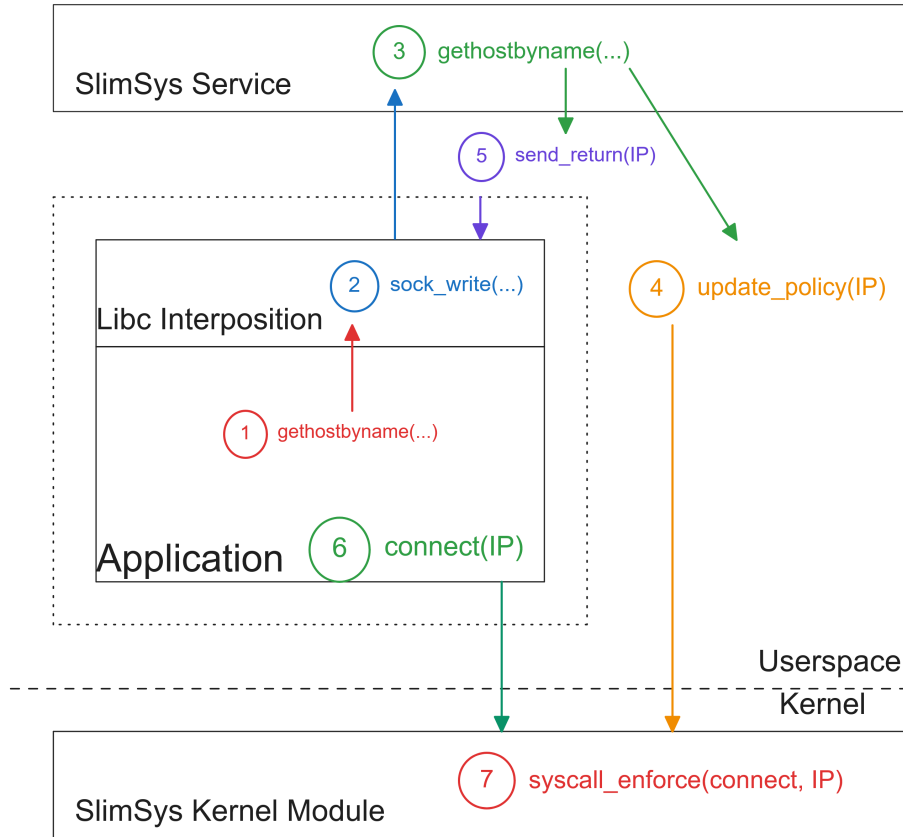


Figure 3.1: A diagram showing the SlimSys architecture and the process of data forwarding. data forwarding starts by capturing calls within the C library through our interposition layer. This translates the call to our service to actually perform the call on the application's behalf. The policy is updated, and then the values returned to the application. Any systems related to the C library call are disallowed, to force an application to use the interposition layer.

The service then makes the call, updates the policy, and returns the results of the call to the interposition shim. The shim returns to the calling application, which is now allowed to make the future call to `connect` as the resource policy has been updated to allow the IP address returned by `gethostbyname` to `connect`. Dynamic data can come from a variety of other sources, for example:

- An IP address resolved from a DNS service request.
- A file name returned by the `mktemp` system call.
- Reading a symbolic link through the `readlink` system call.

Attackers cannot abuse data forwarding to escalate privileges, as policy is static, and data forwarding only attaches static known policy to kernel objects created through systems calls.

3.4 Syscall-Resource Binding

Syscall-resource binding is the model of binding system call policy to each kernel resource, with each resource being defined as the system call and the arguments that create it. Each resource policy outlines the set of operations that are allowed on the resource.

In this section, we describe the process of creating a SlimSys' policy. The creation of policies is broken down into three steps. First, an online trace of the application running a test or benchmark. Second, convert the trace into a candidate policy and determine any forward data binding that may have occurred during runtime. Third, we analyze the binary using a hybrid static-dynamic approach that combines the generated trace with static analysis. This assists in reasoning about states that would be difficult to infer with static analysis alone. For example, inferring missed system calls within not executed error paths. The analysis helps to refine the policy beyond what is possible with the trace.

Furthermore, we illustrate that system resource binding can also be used as a revocation mechanism. Our mechanism, which uses the additional context provided by an argument-centric policy, further restricts system calls in response to other system calls on a resource.

3.4.1 Creating Policy

To illustrate our policy and its creation, we use an example program shown in Figure 3.2. In this example, a connection is made to a host provided in the programs' argument list.

Tracing

Our first step is to run the application using our runtime library interposition layer that interposes on system calls and C library functions that are applicable to forward data binding (e.g., `gethostbyname`, `getaddrinfo`). We run the application using a provided test case or benchmark and capture all system calls, arguments, return values, the calling process ID, and the corresponding call stack at each interposed function.

This trace solves problems that are difficult to determine by static analysis alone. First, it allows us to capture all dynamic states that are indeterminate through static analysis. For example, configuration details such as what dynamic libraries are loaded (e.g., optional plugins/modules for a server). Second, the trace captures system calls that occur within code paths accessed from indirect jumps, such as function pointers and dispatch table pointers. Static analysis cannot reasonably analyze these code paths offline.

In our example, we see the trace created from running our application with a provided test case. The trace shows all system calls related to DNS such as `open(resolv.conf)`, and the C library function, `gethostbyname`.

Policy Creation

We analyze the trace and generate a candidate policy. The policy creation mechanism reads the trace and identifies calls that create resources (i.e., file descriptors) and the subsequent calls that operate on that resource. We determine the operations allowed on the resource by following its file descriptor as used within the program trace. Operations that use the file descriptor are added to its resource policy. Systems calls that are not bound to a file descriptor, such as anonymous `mmap` calls, are added to the application's global policy.

Our tool maintains a process-specific mapping of file descriptors to its current resource policy, and we ensure that this mapping is consistent through edge cases. This is done because two processes with the same file descriptor integer may not point to the same resource. We clone our mapping for new processes created through `fork`. Lastly, we capture new file descriptor values that occur through system calls such as `dup/dup2`.

The trace also handles functions related to dynamic data, such as `gethostbyname`. The trace contains the return values of these calls, and our policy generator binds these return values to a name known as a forward data binding variable. Our generation tool contains rules for specific system calls, like `connect`, which checks if argument data passed to the system call came from data bound to a forward data binding variable. When a match

occurs between the data within an argument and the data captured in a variable name, we replace the data with that of the forward data-binding variable name.

In our example, we see two specific forward data binding variables, `FD` and `HOST1`. `FD` always refers to the file descriptor created by the per-resource policy create call, and is used as a placeholder in the operation list. `HOST1` refers to the future return value of `gethostbyname(www.google.ca)`. The policy creation tool inferred the use of the data from `gethostbyname` in the `connect` call.

Hybrid Analysis

Our hybrid analysis uses dataflow analysis and our runtime trace to refine the initial policy. We use the trace in several ways. First, we determine static arguments for system calls. Second, we determine forward data binding collisions in which a return value from a captured forward data binding variable was the same as a static value used in a system call. This occurs because the creation tool incorrectly inferred forward data binding. Third, we determine all system calls that do not appear directly in the trace. Some system calls are operations that act on an existing resource; in this case, the analysis tool finds and attaches these calls to the correct per-resource policy. Others are system calls that create new system resources (e.g., `open`), whose call and discovered static arguments are appended as a separate per-resource policy.

Our dataflow analysis lets us associate system calls that operate on a file descriptor with the source code location that created it. The analysis operates in two domains: a concrete domain that performs integer constant propagation, and a symbolic domain that performs origin tracking for file descriptors. Using the concrete domain, we can identify specific static values that are passed as arguments to system calls, helping us to identify relevant flags for the policy. The symbolic domain associates the return value of each resource-creating system call with a unique symbol, which is propagated along the data flow as it is passed to future system calls.

The dataflow analysis tool extracts a control flow graph from the application binary to determine paths that lead to system calls. The analysis works by breaking down the binary into a control flow graph, using the trace, and marking the nodes in the CFG that have been visited.

The tool performs an interprocedural data flow analysis to track resources as they pass through the program control flow. We iterate over the basic blocks that remain unmarked and use dataflow analysis to capture any static arguments to those calls. When an unmarked

basic block contains a system call that creates a file descriptor, a new resource policy object is created and appended to the current policy, with all discovered static arguments.

If a system call operates on a file descriptor (e.g., `setsockopt`, `connect`), we scan the trace's call stack to determine which per-resource policy to attach the operation to. This model assumes that creation occurs along the higher levels of the CFG. In the case that the analysis is unable to determine the per-resource policy, we output a warning and exclude the operation from the list.

The cases where a separate thread created the file descriptor from the thread performing the operation on it are not handled by the analysis. This case would require capturing it within the trace itself.

In our example, we see how the analysis captures missed system calls. The initial trace missed an open and subsequent write to a log file. This is because it is within an error path related to the connect call (not triggered while running the application on the test case). The hybrid analysis is able to determine these missed system calls as well as its static arguments and append them to the policy.

The last step is to scan the policy to determine if a single resource can potentially have two possible per-resource policies; this is known as a resource collision. Due to developers being able to use more dynamic representations of arguments, two per-resource policies can collide, for example, `open("/eg/hello.txt")` and `open(DIR "/eg/")`. In this example, the kernel module would not be able to determine which policy to use. We assume that a resource is strictly defined by one, per-resource policy. When a resource collision occurs, we output an error for the developer to correct.

The policy creation process can be incrementally expanded and refined through new test cases and reapplication of our analysis tool. Outputted policy files can be merged at the per-resource level. Once merged, allowed operations within a given per-resource policy are deduplicated and scanned for conflicts.

Dynamic Identifiers

To handle more dynamic programs, the policy supports resource identifiers using restricted regular expressions. In our example, we handle one connection to a single host string; however, this would only allow our program to ever connect to the host provided by the developer test case. The forward data binding for `gethostbyname` can be relaxed so that any domain can be resolved and passed to the `connect` call. Although this is less restrictive, the connect call and its arguments are still bound to the specific socket resource. For

example, an attacker would be unable to connect a UDP socket to any of the IP addresses allowed, even if the creation of such a resource was allowed [23].

3.5 Enforcing Policy

Capability enforcement is handled by the SlimSys kernel module. The SlimSys shim enforces calls from the C library which use forward data binding, while the kernel module enforces filtering and applying the resource policy to calls from the system.

Initializing Kernel Filters

After the daemon has created the application but before it is executed, the daemon iterates over the policy and serializes it into a series of registrations and system call operation addition calls. The SlimSys daemon makes note of forward data binding variables (e.g., *HOST1* in Figure 3.2) and creates an index of the policies that are affected by a change in this variable.

We register a system call and its arguments to create a policy for a specific system call that produces a file descriptor (e.g., `open`, `socket`). We use a list of structures to specify the construction arguments used to create the resource. We wrap our arguments in this structure to provide type information to the kernel and provide whether an argument is static, a regular expression, or a forward data-bound variable.

To expand the per-resource policy with allowed operations, the daemon adds each call to the registered resource policy. A system call number is used to identify the system call operation, followed by a list of argument structures.

Enforcement

During execution, the system calls that create file descriptors are checked for the policy attached to the process. When found, a pointer to the per-resource policy structure is attached to the internal file descriptor structure for future use. When an operation on a file descriptor is performed, the file descriptor object is pinned to stop its modification or removal, and the policies operation list is used to determine if the operation should be allowed.

Any system call, whether it creates a resource or not, has its arguments fully copied within the kernel first. This prevents TOCTTOU attacks that attempt to provide an

allowed set of arguments during the filtering phase and update those arguments after the kernel validates them. System calls and their associated arguments that are absent in the policy are not allowed.

Frequent System Call Optimization

The per-resource system call bit field is an optimization to reduce the overhead of syscall-resource binding incurred by frequently occurring system calls such as `kevent` or `write`. During enforcement, SlimSys first checks for a bit to be set, determined by the system call number. If the bit is set, SlimSys skips traversing their resource policy structure and comparing various arguments, reducing the overhead of the capability check to almost zero. The effects of this optimization are evaluated in Section 3.7.

Updating

Supporting data forwarding requires the ability to update a resource policy. SlimSys uses `LD_PRELOAD` to inject into the application a userspace shim, which interposes on the C library services such as DNS services (e.g., `gethostname`). The shim interposes on the calls from the C library used in forward data binding and translates them into an interprocess communication to the trusted SlimSys daemon. The daemon then executes the call on behalf of the application, updates the policy, and then returns the data to the calling application. Services like DNS are composed of deterministic system calls such as the opening of the `"/etc/resolv.conf"` file, or the connection to the resolving gateway. By disabling these system calls in the policy, we ensure that the application must use the interposed function as opposed to creating DNS requests from scratch within the application code itself.

3.5.1 Dynamic Revocation

The additional context provided by a resource-centric model and argument-level filtering can be used to provide revocation semantics. SlimSys can be used to restrict server applications to only once semantics for system calls like `listen` or `bind` once. More generally, it can be used to revoke per-resource policy system calls upon other system calls occurring within the application.

Name	Syscalls	Blocks	Reduction
Kernel	277	19668	0%
Perfect Debloating	24	14033	29%
Temporal Specialization	15	13205	33%
Dynamic Capabilities	24	3469	82%

Table 3.2: Shows the effectiveness of Seccomp debloating techniques. System call debloating assumes perfect debloating of a program, `lighttpd`, which uses 24 system calls. Temporal Specialization removes 9 system calls, resulting in 15 total calls.

3.6 Implementation

SlimSys is written for FreeBSD 13.1. Its userspace component is written in C++ and is approximately 1k lines of code. This number includes the userspace service and shim layer and the client library. The SlimSys kernel module is written in 2.2 k lines of C code and includes the logic for fine-grained filtering and dynamic revocation. Our tracing mechanism is a set of macros which wraps system calls to include logging, is built in 300 lines of C code and uses LLVM’s implementation of `libunwind`. Our hybrid analysis tool is built on Ghidra and is 4 k lines of Java.

3.7 Evaluation

Our evaluation consists of three parts. First, a breakdown of the effectiveness of SlimSys as an attack surface reduction technique. The breakdown includes a comparison to coarse-grained system call debloating mechanisms, as well as a refinement technique, Temporal Specialization. The second part is a qualitative analysis of the vulnerabilities found within kernels and which systems defend them. The last part of our evaluations shows the overall cost of syscall-resource binding in terms of performance.

All experiments were run on a 3.7GHz Intel Xeon with 8 cores, a Samsung SSD 960 EVO, and 16 GiB of memory. SlimSys was evaluated on FreeBSD 13.1, and all experiments related to Seccomp were evaluated on Ubuntu 22.04.

System Call	lighttpd	nginx	redis	memcached	sqlite	Total	Reduce (%)
read	836	-	294	529	365	1634	69.0
write	1452	1115	700	240	901	1578	44.1
open	1131	1143	480	279	1067	1581	48.1
sendmsg	-	243	-	747	-	4208	88.2
accept	-	-	198	-	-	4461	95.6
fcntl	84	106	82	81	203	2962	96.2
socket	217	205	171	171	-	1517	87.4
bind	142	157	155	155	-	1296	88.3
setsockopt	95	115	100	99	-	1289	92.1
listen	109	110	109	109	-	2481	95.6
getsockopt	81	-	-	-	-	1782	95.5
mmap	320	323	270	326	521	2190	83.9
accept4	226	268	-	166	-	1863	88.2
kevent	517	646	676	395	-	1498	62.7
Average Reduction							81.1%

Table 3.3: Shows five applications and how much kernel code is covered by a variety of common system calls executed during the lifetime of the application. syscall-resource binding, results in an 81% average reduction across these calls. This reduction does not include the reduction of code resulting from excluding the code paths from the other 277 system calls.

3.7.1 Attack Surface Reduction

To measure the effectiveness of state-of-the-art debloating techniques, we use the kernel coverage framework that can be compiled into commodity OSes. When a kernel is compiled with kernel coverage, every edge of a basic block has a special function that writes the current PC address of the basic block into a shared memory mapping accessible in userspace. The coverage excludes code paths that execute due to hard interrupts (e.g., disk driver IO); however, for the purposes of our experiments, this is not required.

To show the effectiveness of syscall-resource binding in reducing the overall kernel code reachability, we used an example application, `lighttpd`. We first determined the total reachable code paths in the kernel using Syzkaller [31], a tool used for kernel fuzzing, which uses the kernel coverage mechanism. We modified Syzkaller so that we could enable or disable a system call or set of system calls as needed.

The `Lighttpd` workload uses 24 system calls, and so we enabled the coverage of only those system calls within Syzkaller, and fuzzed for 120 hours. This data point represents what is possible with perfect coarse-grained debloating and captures all code paths for each system call. We then removed system calls through the Temporal Specialization technique and ran the kernel fuzzing. We capture the reachability of system calls with syscall-resource binding by enabling kernel coverage and running the application and workload.

Table 3.2 shows the final reachable blocks within `lighttpd`. We see that using perfect coarse-grained debloating results in a 91% reduction in accessible system calls, but only a 29% reduction in accessible code. Furthermore, refinement techniques, like Temporal Specialization, restrict the use of 9 additional system calls, but only reduce what is reachable by an additional 4%. In comparison, syscall-resource binding reduces the reachable code in the kernel by 82%.

We next show the effectiveness of syscall-resource binding in debloating common system calls used by large applications. Table 3.3 shows a breakdown of what is accessible through system calls for larger applications: `Lighttpd` [5], `nginx` [9], `Redis` [10], `memcached` [?], and `SQLite` [13].

We measure the number of kernel basic blocks accessible by an application for a given system call versus the number of all basic blocks accessible by that system call. We acquired the total accessible code of a system call by fuzzing the kernel and enabling only the coverage of a single system call at a time. This is done to ensure that we do not pollute the coverage data with unrelated code paths. To acquire application-specific numbers, we run the application alongside its benchmark once for each system call. We show that

CVE	Description	SlimSys	Temporal	Confine
2016-1583	Syscall exploit	✓		
2017-5123	Container breakout	✓		
2018-15664	Syscall exploit	✓	✓	✓
2019-14271	Code injection	✓		
2019-10144	Capability abuse	✓	✓	✓
2019-13272	ptrace exploit	✓	✓	
2022-0492	Abuse cgroups	✓	✓	✓
2022-0185	File system API mismatch	✓	✓	✓

Table 3.4: A list of examples attacks and which system protects against it. Attacks that Temporal debloating and Confine are unable to defend against are ones which manipulate kernel state through arguments to system calls.

syscall-resource binding has an average reduction of 81% between common system calls (Figure 3.3) used within our five applications.

This large reduction can be attributed to the syscall-resource binding model of binding system calls to their arguments. Applications are bound to the code paths outlined by its policy for each system call, which greatly reduces the kernel attack surface.

Limitations of Policy Creation

Our policy generation technique required that we manually insert the `DIR` identifier for the directory used as the root folder for `nginx` and `lighttpd`, as this cannot be captured with our analysis. `Redis` and `memcached` both required no additional developer input for the created syscall-resource binding policy.

3.7.2 Vulnerability Comparison

We compare SlimSys with two software debloating techniques, Temporal Specialization [68] and Confine [67] and present the techniques to mitigate example vulnerabilities. These CVEs are triggered from common system calls and can be used by an attacker to escalate privileges. We look at the guarantees which SlimSys provides as opposed to two such systems, Confine [67] and Temporal Syscall Specialization [68], and test them to a list of CVEs whose exploitation leads to privilege escalation. These systems use different techniques to create filters, but both use Seccomp filters to represent them.

Both Confine and Temporal System Call Debloating use allow/deny filters that do not stop malicious use of fine-grained arguments. Temporal system call filtering traces the system calls used by the application in two distinct phases, and removes Seccomp filters to their application after initialization to eliminate unnecessary system calls like `ptrace`. Temporal uses a combination of Andersen’s algorithm [42] and pruning techniques to determine the appropriate set of system calls allowed in each phase. Confine uses a combination of static analysis techniques throughout the code, and its shared libraries including `libc` to find all system call sites accessible from the binary. This technique is an upper bound of what system calls are available, but does not prevent CVEs in the code paths of commonly used system calls.

Table 3.4 lists a selection of CVEs that lead to privilege escalation and are triggered by executing common system calls. CVE-2016-1583 is exposed by `mmap` when it calls to paths in the `/proc` file system to trigger recursive page fault handling, Confine and temporal specialization cannot prevent calls to `mmap` without preventing most applications from executing. SlimSys prevents `mmap` from being called with files in the `proc` file system, as this capability is not required by applications. CVE-2017-5123 which is triggered by `waitid` is similarly unmitigated by Confine and Temporal for most multiprocess applications.

Coarse-grained debloating works for system calls with less general functionality, like `ptrace`. CVEs which can be triggered with `ptrace` (CVE-2019-13272), file system mounting (CVE-2022-0185), symlink creation (CVE-2018-15664) and cgroup management (CVE-2022-0492) are properly mitigated because their functionality relates to system management. These calls are not necessary for normal applications.

3.7.3 Performance

Figure 3.4 shows the breakdown of the cost of our syscall-resource binding checks on each system call within the main request path for our applications. We enabled kernel tracing after the server has initialized but before the benchmark has run to capture overhead for system calls related to the request of each application. These breakdowns help provide candidates for systems calls that should disable fine-grained argument inspection and use our system call bitfield optimization. For our optimized versions, we disabled fine-grained argument inspection in the two most expensive system calls of each application.

Every workload was associated with a benchmark. Nginx and Lighttpd are web-based, so we used the `wrk` benchmark [14]. Redis uses the supplied `redis-benchmark` [116], and Memcached uses `libmemcached`’s Memaslap [89]. Each benchmark was run with 8 threads, over 5 minutes, and re-run 15 times for each configuration. For overhead percentages, we

compared them to unaltered versions for each relative operating system, meaning that Seccomp overheads are calculated overtop of the same application and benchmark on Linux.

Figure 3.5 shows our final evaluation of four server workloads, Redis, Lighttpd, Nginx, and Memcached. In the figure, each workload was run with three separate configurations. The first configuration is under SlimSys with a syscall-resource binding’ policy, the second is an optimized version that disables fine-grained argument inspection for the top two systems calls by overhead. The final configuration is a comparison against Seccomp.

We also calculate, on average, how many system calls are performed during a request to better illustrate how dependent each application in the critical is reliant on system calls. Both Memcached and Redis make relatively few system calls per request, leading to a lower overall overhead. These workloads are in memory key value stores, so all calls are related to communicating with the client, with no calls related to fetching data.

Between both Redis and Memcached, we see that both the un-optimized and optimized versions are lighter weight than Seccomp. Seccomp uses Berkley packet filters as its underlying kernel mechanism. The cost of Seccomp scales to the size of the application policy. The Seccomp BPF program is a switch statement, which translates the system call number to an accept/reject of the system call.

Furthermore, BPF programs are not executed in the main context of the kernel, they are executed in their own language-level VM, similar to Java, adding additional overheads. Using BPF, every system call must traverse the program on every request. To ensure the Seccomp policy is as low overhead as possible, each application uses its own Seccomp policy that only captures the systems calls that occurred during the benchmark.

SlimSys has the benefit of being a kernel module that hooks directly into the process system call vector. Further, at a given system call, our overheads come from either traversing and comparing the system’s call/argument pairs that represent per-resource policies or the traversing of the per-resource policies themselves. SlimSys scales better because per-resource policies are much smaller than a global Seccomp BPF program. For the applications we evaluated, the resources that represent connections had very slim resource policies (around 3-4 operations).

Within Lighttpd, the `write` call is the majority of the overhead. This is because for every `kevent` call, there are numerous `write` calls. `write` is a much lighter call than `kevent`, and because of this, the capability check, which adds 21% overhead, is more pronounced. Nginx has much higher throughput; however, it makes fewer system calls in the main request path, with the main overhead being the tens of thousands of `kevent` calls that occur per second. This makes `kevent` the most pronounced overhead for the Nginx application, even though the capability check only accounts for a fraction of a percent of the overhead.

An observation is that Nginx and Lighttpd have very similar overheads due to the relationship of system calls per request and overall throughput of each of the applications. Nginx has $4\times$ the throughput, but makes 30% less calls than Lighttpd. Which accounts for why Lighttpd has slightly more overhead than Nginx.

Between Figure 3.5 and Table 3.3 we show that we still gain all the benefits of fine-grained debloating for most system calls, while disabling fine-granularity for a select few. Some of these system calls are also quite limited in their individual reductions (See Table 3.3 for each reduction of `write`, `kevent`). Furthermore, we still are able to enforce per-resource policy on these calls, and the optimization is essentially equivalent to allowing an operation, with all arguments allowed. In this way, the operation is allowed only on a specific kernel resource.

In our optimized versions of Lighttpd, and Nginx, we disabled fine-grained argument inspection in top two contributors of overhead in the application. In Lighttpd, we see an improvement in performance from 4.6% overhead to 0.5% overhead. In Nginx, we see an improvement from 4.4% overhead to 2.6% overhead. This improvement is not as pronounced, as we can see in Figure 3.4 that the overhead is spread more evenly between various system calls.

```

int main(...) {
    int sockfd = socket(AF_INET,
        SOCK_STREAM, 0);

    struct hostent *server =
        gethostbyname(argv[1]);

    struct sockaddr_in addr =
        create_addr(server);

    int error = connect(sockfd, &addr,
        sizeof(addr));

    if (error) {
        int log = open("log.out", ...);
        write(log, "Problem!");
    }

    make_request(sockfd);
}

```

(a) An example program that takes a domain name as an argument (e.g., `www.google.com`) and connects to it to make a request.

```

HOST1 = gethostbyname(www.google.ca)

// Acquired using Hybrid Analysis
open("/tmp/log.txt", ...) {
    write($FD, ...),
}

socket(AF_INET, ...) {
    connect(
        $FD, sockaddr {family : AF_INET,
            port: 80, addr: $HOST1}
    )
}

```

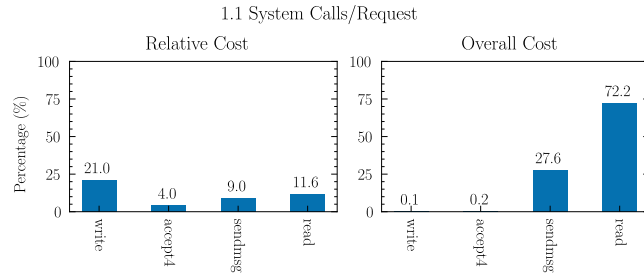
(b) A policy created through our policy creation. The `open` call, and the associated `write` is discovered through our hybrid analysis.

Figure 3.2: Creating a SlimSys policy.

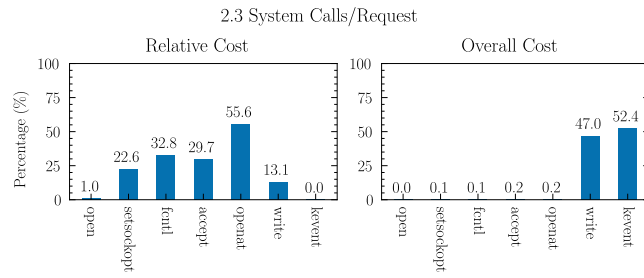
```
open("/usr/home/passwords.txt", ...) {  
    read {$FD, ...} MY_SENSITIVE_LBL  
}  
  
socket(AF_INET, ...) {  
    REMOVE_ON(MY_SENSITIVE_LBL) connect(  
        $FD, sockaddr {family : AF_INET,  
                        port: 80, addr: $HOST1}  
    )  
    REMOVE_ON(MY_SENSITIVE_LBL) read(..),  
    REMOVE_ON(MY_SENSITIVE_LBL) write(..)  
}
```

(a) SlimSys Policy.

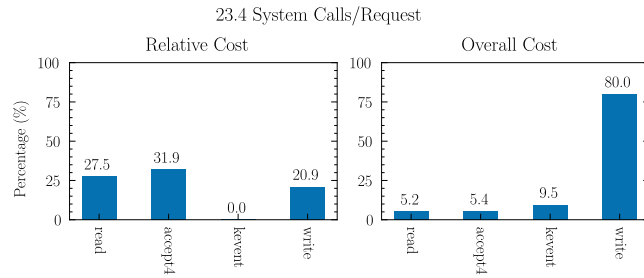
Figure 3.3: An example revocation policy that can be presented to SlimSys to be combined at runtime in the kernel. In this example it can be used to protect sensitive files from being exfiltrated from an application by disconnecting previously allowed connections.



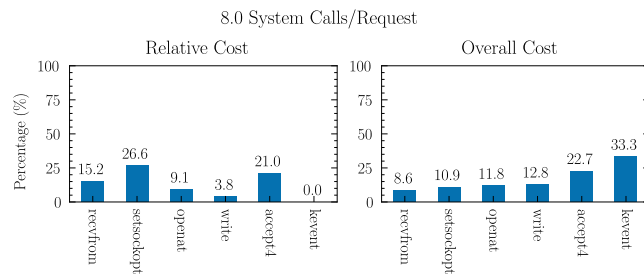
(a) Memcached



(b) Redis



(c) Lighttpd



(d) Nginx

Figure 3.4: A breakdown for each workload showing the relative cost of the capability check to the entire system call and the cost of the capability check overall to the entire system for a given workload.

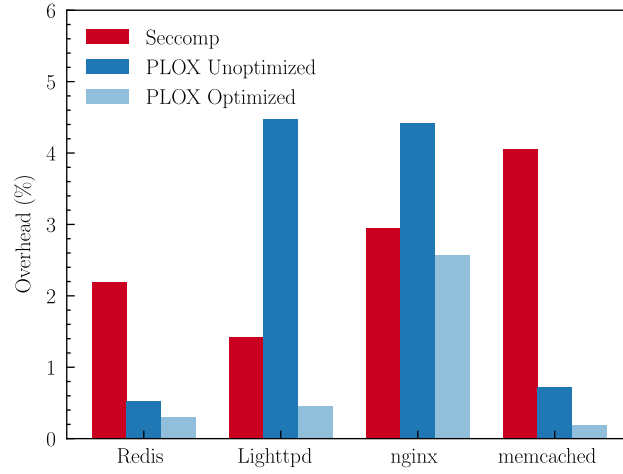


Figure 3.5: Performance overhead of SlimSys's syscall-resource binding.

Chapter 4

OrcBench: A Representative Serverless Benchmark

The serverless paradigm has seen a growth in popularity over the last few years [49, 109] with AWS seeing a 209% increase in functions executed in 2020 [117]. Serverless simplifies service deployment by shifting the responsibility of infrastructure from developers to the cloud provider. Developers focus on the functionality of their applications while the cloud takes care of most of the deployment challenges.

Building a cost-efficient serverless platform is challenging because developers are only charged for the resources used during the execution of a function. Offloading the application deployment and management from developers to providers introduces complex research challenges including resource management, auto-scaling, workload consolidation, storage systems, billing, and orchestration. A benchmarking tool that simulates the characteristics of real-world workloads is essential to explore these frontiers.

Currently, there are no benchmarking tools that capture the characteristics of real serverless workloads. Previous efforts resorted to building tools that were limited by the lack of publicly available data from serverless providers. These tools provide performance and cost estimates for running a serverless application on a given cloud platform [53, 92], or provide a synthetic set of functions that represent applications within the serverless paradigm [80].

We present OrcBench, a serverless workload generator that generates data center workloads that model the recently released Microsoft Azure dataset [126]. OrcBench generates workloads which mimic the interarrival time, resource usage, and execution times of the real world Azure traces. OrcBench supports varying cluster sizes and workload

intensity, and can be used to generate complete serverless applications that chain multiple functions.

The modeling of Azure traces is challenging for two reasons. First, the traces do not include the exact invocation time, instead batching all invocations at minute-level granularity. Second, the data set contains 52 thousand functions and a total of 8.8 billion function invocations over a two-week period. Third, there are many applications with different interarrival patterns and resource usage.

To accurately model these functions, we group them based on their interarrival times using EP-means clustering [74]. EP-means clusters the empirical cumulative distribution functions (ECDF) of each function’s interarrival times. Clustering in this way allows us to group similar-behaving functions together. We use the centroid of each cluster as a representative function for its group.

These representative functions are used in place of the hundreds to thousands of functions in a cluster. Each group also has a probability distribution for both CPU and memory constraints. When a user requests a simulated function this underlying distribution is sampled to give each function how much memory and CPU to consume.

Clustering treats each function independently without considering the relationships between functions. An application may consist of multiple functions that form a chain that is executed in series. We use time-series analysis to discover the relationships between functions within each application and allow us to model these chains.

OrcBench grouped 8622 functions into 17 distinct groups, and the models produced from these groups were used to generate an equivalent number of synthetic functions and traces. These 8622 simulated functions were invoked 5.6 million times in 30 minutes with an average error rate of 15%. A major source of error comes from extrapolating sub-minute behavior from the one-minute resolution timestamps in the Azure data set. Our two highest invoked models (representing 113 functions) accounted for 2.8 million invocations while having only an average error rate of 4.7%. The remaining functions were rarely invoked or timer-based functions that were excluded from our modeling.

4.1 Overview of the Microsoft Azure Traces

The Microsoft Azure dataset [97] is a collection of 52 thousand functions which were invoked 8.8 billion times over a 14 day period. The three main objects in the data are functions, applications, and owners that are identified through anonymous hash IDs. Owners can

own multiple applications, and applications can be made up of many functions that can be invoked by each other to form function chains. This captured data is broken into three major parts: a time series of invocations, execution time, and memory usage.

Invocations The invocation time series contains the number of invocations of a function at each minute of the 14-day trace.

Execution Time The dataset contains the average execution time and a fixed set of percentiles for each function. The execution time percentiles for 0^{th} , 1^{st} , 25^{th} , 50^{th} , 75^{th} , 99^{th} and 100^{th} are included. The execution times do not include the cold start of the function run-time.

Memory Usage The data set includes the average memory usage for each *application* and is also broken down into a fixed set of percentiles. Unlike the other data recorded per function, memory usage is recorded for the entire application because Azure packages resource allocation limits for functions belonging to the same application together for pricing [98].

4.2 Methodology

Modeling the Microsoft Azure dataset requires overcoming three challenges: First, the data set is temporal (see Figure 4.1). Each function fluctuates throughout the day, often seeing a steady rise over working hours (i.e., 9am to 5pm) and declining when the workday is over.

Second, anonymization of the data reduces the assumptions we can make based on additional information such as function naming. This particularly affects our ability to determine where functions exist within a chain.

Lastly, the traces of each function have low-resolution timestamps of minute granularity. The execution times of most functions is far smaller (i.e., 90% of functions have an execution time of less than 10 seconds [126]) than the one-minute resolution requiring us to extrapolate a model for execution behavior for timescales less than one minute.

It is important to translate the dataset from an invocation time series to interarrival time as this decouples time from our model. However, the low resolution of the data makes this difficult. For example, functions commonly have large contiguous sections of non-zero

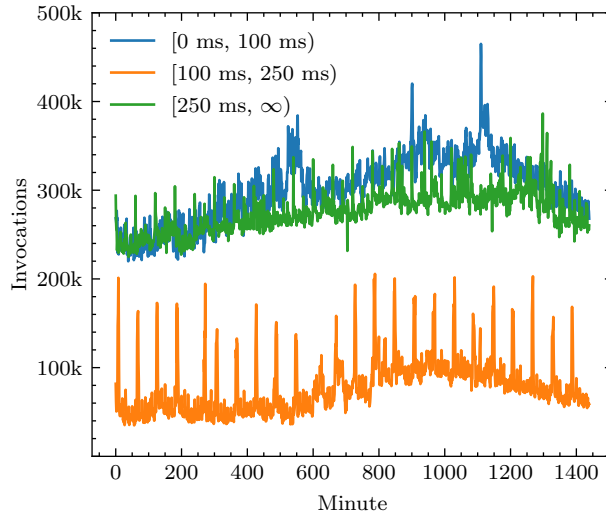


Figure 4.1: Invocation counts of functions broken up by execution time in milliseconds over the course of a day

entries in their traces, which when naively averaged to calculate the interarrival time leads to a single highly frequent data point. In a real-world application, we would expect a far more continuous set of data points in regard to interarrival time. Averaging would also completely flatten our invocation rate, causing us to lose important patterns that can occur during the trace.

4.2.1 Modeling Functions

The modeling of the function invocation rates can be broken down into three stages: trimming, data expansion, and grouping.

Trimming To reduce temporalness (Figure 4.1), we first isolate the invocation data in a single 30 minute window of time between 12:00 PM and 12:30 PM. By sampling a smaller window of time, we reduce the temporal behavior within our model at the cost of accuracy. An incredibly short window would be unaffected by temporal behavior, but the model could miss interesting patterns that may be expressed in the data. As this window grows, the model captures more interesting patterns.

However, if it grows too large, these patterns can be difficult to detect because of temporal behavior. An outcome of choosing this time window is that our models are a

representation of the workload at that specific time. For example, models created around peak hours invoke functions more frequently than models created from an earlier or later time window.

We then trimmed the data to contain only functions that were invoked at least 10 times during the selected 30 minute time period and also excluded functions triggered by timers. Timer functions can be trivially modeled, and rarely invoked functions lack data for proper modeling.

After trimming, we divide the data into the top 1% of highly invoked functions and the remaining functions. We model these two groups using the same techniques but separately. Clustering highly invoked functions with lesser ones masks the interesting behavior of these infrequently invoked functions.

The data were further divided between weeks, with the first week being used for modeling, while the second was used for evaluation. We excluded weekends as outliers because functions exhibited very different behavior.

Data Expansion The next step is to translate our invocation traces to inter-arrival time. Translating from invocations per minute to inter-arrival time makes our models independent of each minute. This translates our model from a function that tells us the number of invocations to expect at a given minute to how long we should expect until the next invocation.

As previously stated, if we naively average our traces to calculate our interarrival times, we are left with very few data points for modeling. This also has the disadvantage of flattening each function that causes the loss of interesting patterns in the data. We can extract more data points if we assume that, for any given function, the number of invocations at any given minute is independent of any other minute within the same function.

The assumption of minute-to-minute independence allows us to instead view each minute in the invocation data as a Poisson process. We can then generate as many data points as invocations by using each minute as a hyper parameter to the Poisson distribution. The sampling of this distribution allows us to transform a discrete dataset into a continuous one, which better represents a realistic invocation pattern.

We believe this assumption to be sound, since this stage of the modeling focuses only on each individual function rather than relationships between functions. We could not make this assumption between two minutes of two separate functions, as they may belong to the same application and one function may trigger another during its execution.

Grouping We then create empirical cumulative distribution functions (ECDF) for each function with these newly expanded interarrival data. The ECDF is a model of a function which is then grouped to form clusters with other similarly shaped functions. A cluster centroid is chosen, which is the model that replaces all functions of the cluster.

We cluster the functions using EP-Means that clusters similar ECDFs together. However, a challenge occurs when clustering the interarrival times. Specifically, the distances between the interarrival times are not representative of the behavior. For example, two functions that have constant interarrival times of 0.01 s and 0.1 s seem close, but have a $10\times$ difference in invocation frequency.

To remedy this, we use EP-Means clustering on the inverse of the interarrival times. We create ECDFs of this frequency data and use EP-Means to retrieve each grouping. From here we translate back to interarrival time and re-calculate the centroids of each group.

Sampling Once models have been created, we now face the challenge of properly sampling a function that represents many functions. If we were to sample our created model once for each of its represented functions, then at any given minute we would likely just see the average of the model. To overcome this, OrcBench allows for a hyper-parameter (N) which sets how many functions a model is used for, which we call the sampling group.

We then randomly spread invocations for each function being represented within the sampling group throughout the range of the interarrival time sampled (see Figure 4.2). Although we are modifying the interarrival time for that set of invocations, if we sample the interarrival time and randomly place these invocations, then the average interarrival time would trend towards the initial sampled time.

Application Workload Generation We now have a group of applications and their root functions. The user can choose one of these applications to use for their workload. If more than one candidate root is attached to the application, the user must select one. We associate these root functions with their model from §4.2.1 to generate function invocations. On each invocation, OrcBench simulates the execution of the call graph of the application and records the trace.

Users can also get OrcBench to generate function stubs for their application. OrcBench does this by sampling the underlying histogram of CPU and memory data from the root model (see Figure 4.3 as an example). We found no strong correlation between CPU time and memory when looking at each of our models, so we can sample these histograms to help generate our synthetic functions. These samples are then input into a template function which, when invoked, will allocate memory and execute for the specified amounts.

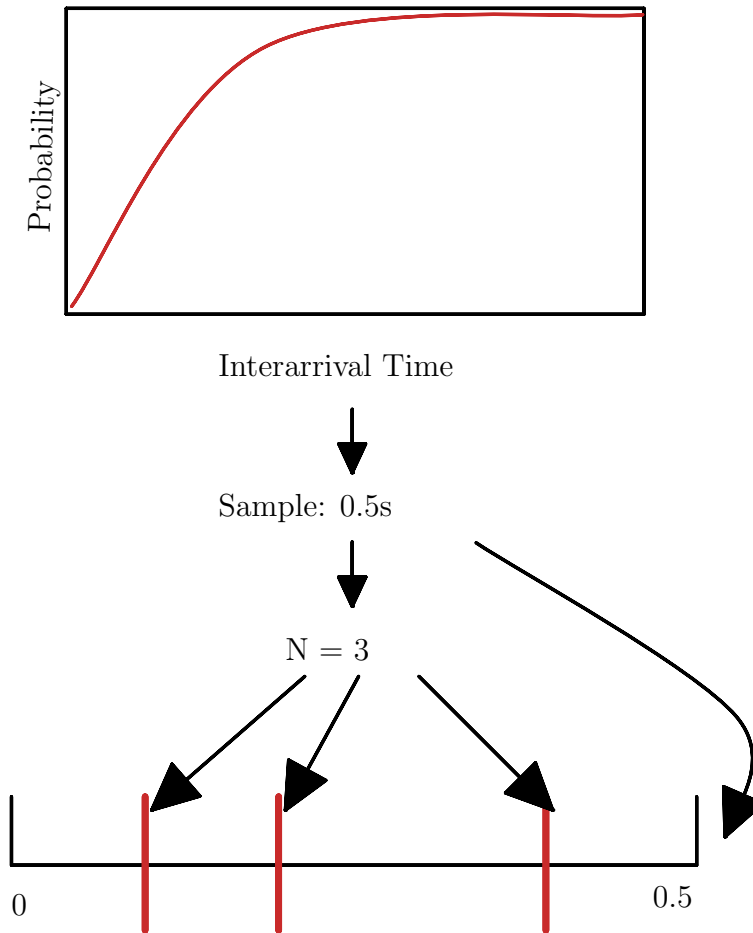


Figure 4.2: An example of sampling from an ECDF of a model with a hyper-parameter of $N = 3$, and how we spread invocations within a sampled interarrival time. Each vertical line on the timeline from 0s to 0.5s represents a separate function.

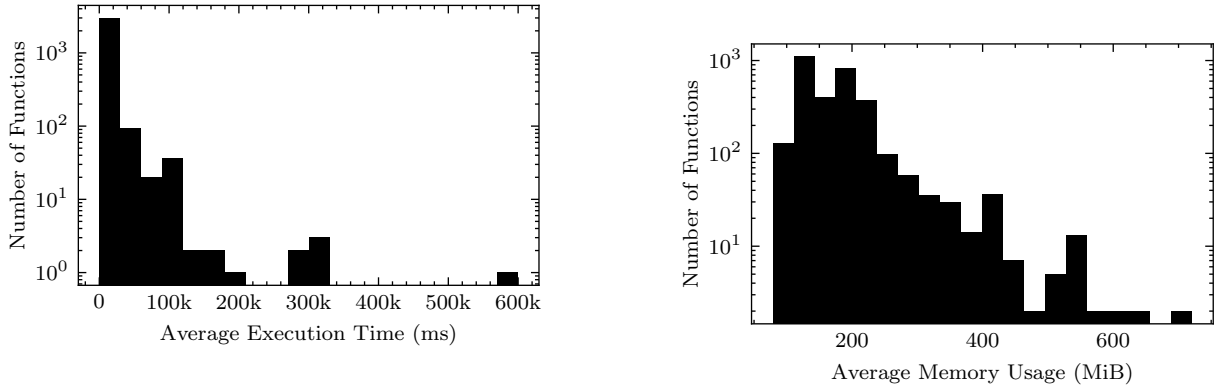


Figure 4.3: Example CPU and memory histograms which back each model for generating function stubs.

4.3 Evaluation

Our evaluation examines the following. How accurate are the individual function traces created by OrcBench?

In §4.3.1 we evaluate how close our models are to each of the functions we are modeling and how much each model contributes to the total error. Lastly, in §?? we evaluate an inferred application chain.

4.3.1 Models

Generated Model Traces Figure 4.5 shows example OrcBench traces that compare invocation data to that of its model. We compare the model with the second week of the data set. As previously stated, we use a hyperparameter for the number of functions each model will represent during the trace. All evaluations were performed with $N = 10$.

Model Error Rate Our clustering found 17 distinct models that represent a total of 8622 functions, with a workload scaled to one that matches the Azure dataset executing 5.6 million invocations over a 30-minute period.

We examined our results to determine the following.

- How much error does each model have relative to the group of original functions that the model is representing?

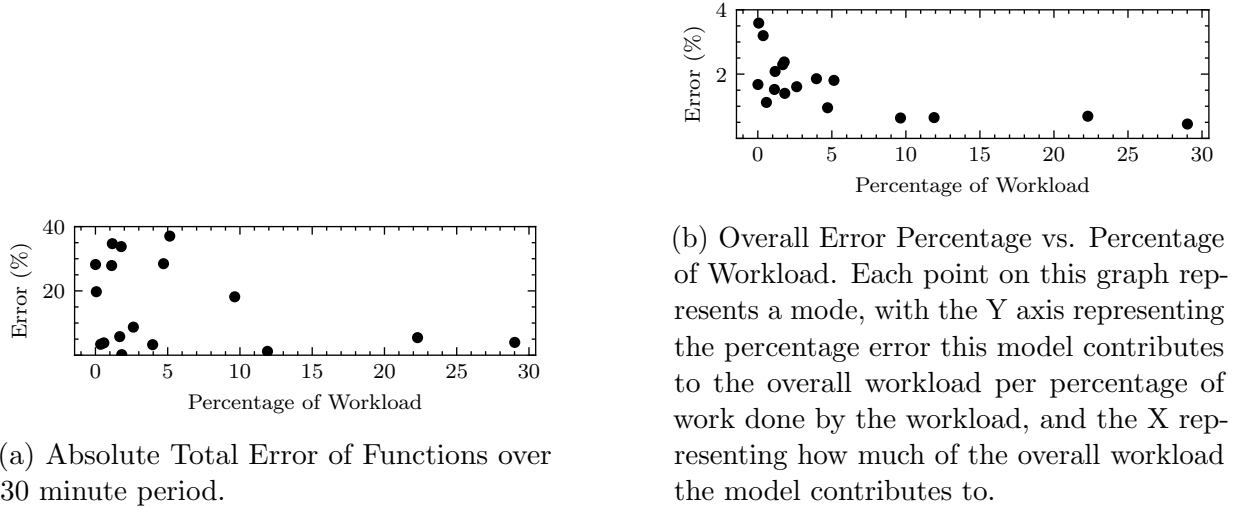


Figure 4.4: Error Rates within the OrcBench Models

- How much error do models have relative to each other?

We simulated a trace using each model and calculated the mean squared error (MSE) between the simulated trace and each original function trace within its group. We added this error to a sum and then normalized it with all other models.

We then normalized the total invocations the model contributes to the error. This is the ratio of normalized error to normalized model invocations, that is, how much error a model contributes per percentage of the total workload. Figure 4.4b shows this value for each model. Note that many higher error models represent less than 1% of the total workload. Figure 4.4a shows how much total error accumulated during the course of a 30 minute run. We aggregate the total MSE between the model trace and each function trace.

We see a trend with the error that shows that models that represent a larger weight of the overall workload have a lower error. This occurs because these models have far more data points to create a better fitting function leading to a more refined centroid being created during clustering. Furthermore, as more function invocations occur within an individual minute, the extrapolated Poisson model used for each minute becomes more accurate. The average absolute error for all of our models is 15%.

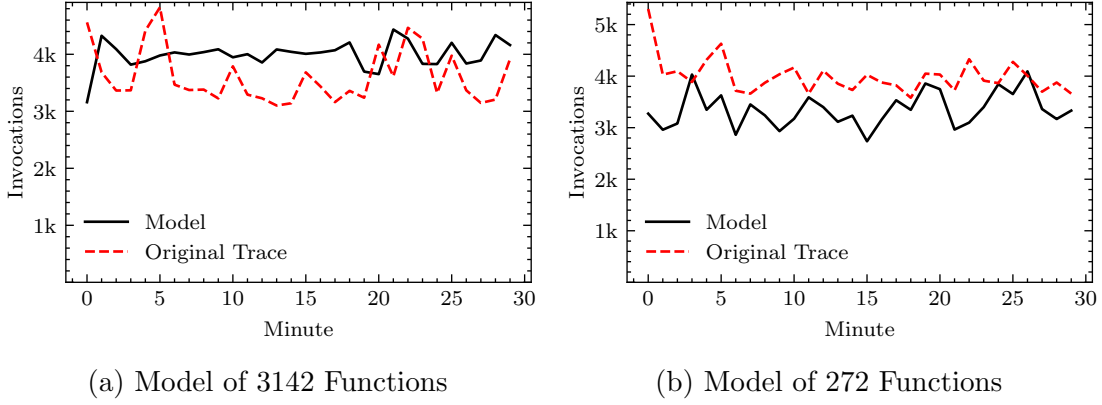


Figure 4.5: Example traces of our models which have been scaled to the number of functions to the functions they originally represent. We compared these modeled functions to the sum of the traces of the original functions.

4.4 Related Work

Work within serverless computing that focuses on orchestration infrastructure [83, 88, 121, 123] often uses open source applications to evaluate and compare their designs. The solutions focus on individual applications and their latency/throughput. Due to the lack of any global workload benchmark, readers are left to assume how these design decisions affect the infrastructure as a whole.

Benchmarks like FaaSdom [92] and SeBs [53] focus on benchmarking cloud providers themselves to give insights into each cloud provider’s expected function runtime and cost. These benchmarks provide a strong basis for micro benchmarks over a larger workload-based one like OrcBench.

Chapter 5

Conclusion

This thesis illustrated the motivation, design, and evaluation of three systems created during my Ph.D. This section summarizes these works and outlines possible follow-up directions and challenges faced by each system.

ObjSnap and SLSes ObjSnap (Chapter 2), a write-optimized, high-throughput, low-metadata transactional object store that outperforms traditional storage systems by $4\times$ and with $3\times$ less metadata. ObjSnap acts as a foundation for SLS checkpoint restore and is the storage layer for Aurora and MemSnap. ObjSnap shows the advantages of using a block-level WAL in conjunction with copy-on-write semantics. The use of copy-on-write as a mechanism for block movement such as required in fields such as storage tiering should be further explored. Furthermore, additional extensions to ObjSnap should be explored to support small key-value style databases such as RocksDB, which are designed to reduce write amplification. LSM databases, such as RocksDB [11] are specialized in reclaiming spatial locality and reducing total disk usage; however, compactions make these systems highly complex and difficult to reason about in terms of performance. ObjSnap’s generational allocator and data movement mechanisms amortize data movement across the workload to consistent IO over a databases lifetime, resulting in reduced CPU and latency caused by processes such as compactions. ObjSnap has all the tools necessary as a general storage system for applications that require the semantics and SSD longevity provided by LSM data structures.

The Case for Swap ObjSnap and MemSnap explored a cohesive virtual memory/persistence abstraction, breaking traditional persistence and presenting a simpler mechanism for

transactional semantics. However, ObjSnap and MemSnap left the problem of overcommitment out of scope, which is a fundamental problem for database workloads. Further work could be done in reapplying ideas found in MemSnap towards traditional swap systems, which have been left behind since their use on slow, hard disks. The swap system is a natural place to merge the gap between memory and storage, while maintaining the transactional persistent memory semantics provided by MemSnap.

By presenting better swap systems and API's we could translate the ideas of SLSeS into traditional POSIX OSes, MemSnap as a system could further be used as the swap system itself, allowing for transactional swapping of data of large address spaces, this would fundamentally change and simplify how we build databases, which must build large complex systems to ensure the swap system is never engaged.

Concurrency and Storage A challenge we faced while building ObjSnap is that many of the underlying mechanisms in the kernel performed poorly under high concurrency (greater than 12 threads). For example, in the buffer cache, the use of compare-and-swap (CAS) style locks scale poorly past 12 threads, due to contention at the CPU cache coherency level. The key to solving this issue was to look at various work within the concurrency field, which showed an innate bottleneck of CAS-style locks. These locks, even on a failed CAS would still acquire a cache line in exclusive mode, resulting in pulling cache lines away from cores that would have acquired the lock, creating large amounts of thrashing resulting in dramatically reduced throughput of IO (up to 2–5x in some cases). By backing off on failed CASes, we were able to solve this issue. The storage field and concurrency field can not be separated, it is pivotal for experts in storage to maintain an understanding of the work done within concurrency, as problems solved there will frequently be more apparent as storage devices become faster.

Better Container Systems My research branched into the creation of SlimSys (Chapter 3), a container security system that links the policy to individual resources, and further introduces a way to quantitatively evaluate the kernel attack surface through system-call fuzzing. A large challenge and potential future work for SlimSys is isolating better techniques for analysis. We needed to supplement our analysis with static traces to simplify this process; however, this comes at the cost of usability. Users are required to create more robust tests and benchmarks to create robust traces, which when done incorrectly can lead to operational toil for the user.

Orcbench Lastly, I introduced OrcBench, the first serverless invocation-based benchmark. This benchmark allows for the creation of scalable synthetic traces which can be used to evaluate new orchestrators for serverless. Due to the extent of data anonymization, the error rates of our models are larger than necessary. Furthermore, complex relationships between functions are difficult to verify unless de-anonymized data are present. However, the methodology we used is applicable to any future data that may be released and would benefit greatly from any additional context regarding invoked functions.

Additional work is needed to make OrcBench functions more substantial. Although OrcBench is capable of creating function definitions that mimic memory usage and CPU, they are shallow functions and could be improved to better imitate real applications. Combining OrcBench with a suite of functions such as FunctionBench [63] would increase its value for evaluating systems like Metropolis.

References

- [1] BPF(2) - Linux Manual Page. <https://man7.org/linux/man-pages/man2/bpf.2.html>.
- [2] Criu. https://criu.org/Main_Page.
- [3] End-of-Life (EOL) Announcement. <https://www.intel.com/content/www/us/en/support/articles/000057951/memory-and-storage/intel-optane-memory.html>.
- [4] landlock(7). <https://www.man7.org/linux/man-pages/man7/landlock.7.html>.
- [5] Lighttpd. <https://www.lighttpd.net/>.
- [6] Memcached. <https://www.memcached.org/>.
- [7] memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark.
- [8] Namespaces. <https://man7.org/linux/man-pages/man7/namespaces.7.html>.
- [9] nginx. <https://nginx.org/en/>.
- [10] Redis. <https://redis.io/>.
- [11] RocksDB — A persistent key-value store. <https://www.rocksdb.org>.
- [12] Selinux. <https://github.com/SELinuxProject>.
- [13] Sqlite. <https://www.sqlite.org/index.html>.
- [14] wrk. <https://github.com/wg/wrk>.
- [15] *zones(5) Standards, Environments, and Macros*, SunOS 5.10 edition, April 2004.

- [16] *CHROOT(2) BSD System Calls Manual*, 12.1-RELEASE edition, January 2012.
- [17] SFS: Random write considered harmful in solid state drives. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, San Jose, CA, February 2012. USENIX Association.
- [18] Cve-2016-1583. <https://www.cvedetails.com/cve/CVE-2016-1583/>, 2016.
- [19] Cve-2018-8781. <https://nvd.nist.gov/vuln/detail/CVE-2018-8781>, 2018.
- [20] PostgreSQL's fsync() surprise. <https://lwn.net/Articles/752063/>, April 2018.
- [21] *SECCOMP(2) Linux Programmer's Manual*, linux man-pages 5.06 edition, November 2019.
- [22] *CGROUPS(7) Linux Programmer's Manual*, linux man-pages 5.06 edition, April 2020.
- [23] Cve-2020-14386. <https://www.cvedetails.com/cve/CVE-2020-14386/>, 2020.
- [24] What's LXC? <https://linuxcontainers.org/lxc/introduction/>, March 2020.
- [25] PostgreSQL: The world's most advanced open source database. <https://www.postgresql.org/>, April 2021.
- [26] PostgreSQL's fsync() surprise. <https://lwn.net/Articles/752063>, January 2021.
- [27] bcache: The COW filesystem for Linux that won't eat your data. <https://www.sqlite.org/wal.html>, August 2023.
- [28] Ghidra. <https://ghidra-sre.org/>, 2023.
- [29] HC-tree. <https://sqlite.org/hctree/doc/hctree/doc/hctree/index.html>, April 2023.
- [30] MySQL Website. <https://www.mysql.com/>, April 2023.
- [31] syzkaller - kernel fuzzer. <https://github.com/google/syzkaller>, 2023.
- [32] Docker. <https://www.docker.com/>, 2024.
- [33] fio - Flexible I/O tester. [fio-FlexibleI/Otester](https://github.com/axboe/fio), September 2024.

- [34] JFS Layout: How the Journaled File System Handles On Disk Layout. <https://trac.netlabs.org/lucide/raw-attachment/ticket/356/jfslayout-pg1.pdf>,, September 2024.
- [35] MongoDB: The most popular Database for Modern Apps . <https://www.mongodb.com/>, September 2024.
- [36] OpenVz. <https://openvz.org/>, November 2024.
- [37] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *NSDI 2020*, 2020.
- [38] Ioannis Agadakis, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: Debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference, ACSAC '19*, page 70–83, New York, NY, USA, 2019. Association for Computing Machinery.
- [39] Fumio Akiyama. An example of software system debugging. In *IFIP Congress*, 1971.
- [40] Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. Lightweight, multi-stage, compiler-assisted application specialization. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroSP)*, pages 251–269, 2022.
- [41] Chloe Alverti, Vasileios Karakostas, Nikhita Kunati, Georgios Goumas, and Michael Swift. Daxvm: Stressing the limits of memory as a file interface. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 369–387, 2022. <https://doi.org/10.1109/MICRO56248.2022.00037>.
- [42] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, Citeseer, 1994.
- [43] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 53–64, New York, NY, USA, 2012. Association for Computing Machinery. <https://doi.org/10.1145/2254756.2254766>.
- [44] Bob Beck. Pledge, and Unveil, in OpenBSD. In *BSDCan 2018*, June 2018.

- [45] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. Hacking blind. In *2014 IEEE Symposium on Security and Privacy*, pages 227–242, 2014.
- [46] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. 215, 2003.
- [47] Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '03*, page 546–554, USA, 2003. Society for Industrial and Applied Mathematics.
- [48] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of High-Coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, December 2008. USENIX Association.
- [49] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. The rise of serverless computing. *Commun. ACM*, 62(12):44–54, nov 2019.
- [50] 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.
- [51] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286, 2005.
- [52] Alex Conway, Abhishek Gupta, Vijay Chidambaran, Martin Farach-Colton, Rick Spillane, Amy Tai, and Rob Johnson. Splinterdb: Closing the bandwidth gap for nvme key-value stores. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC’20, USA, 2020. USENIX Association.
- [53] Marcin Copik, Grzegorz Kwasniewski, Maciej Besta, Michal Podstawski, and Torsten Hoefler. Sebs: A serverless benchmark suite for function-as-a-service computing, 2020.
- [54] Microsoft Corporation. Microsoft vulnerability cve-2019-1372. <https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2019-1372>, March 2020.

- [55] Andrew Crotty, Viktor Leis, and Andrew Pavlo. Are you sure you want to use mmap in your database management system. In *CIDR 2022, Conference on Innovative Data Systems Research*, 2022.
- [56] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX symposium on networked systems design and implementation*, pages 161–174. San Francisco, 2008.
- [57] Pubali Datta, Prabuddha Kumar, Tristan Morris, Michael Grace, Amir Rahmati, and Adam Bates. *Valve: Securing Function Workflows on Serverless Computing Platforms*, page 939–950. Association for Computing Machinery, New York, NY, USA, 2020.
- [58] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 459–474, San Sebastian, October 2020. USENIX Association.
- [59] Docker. Empowering app development for developers. <https://www.docker.com/>, March 2020.
- [60] Jake Edge. Deep argument inspection for seccomp. <https://lwn.net/Articles/799557/>, March 2019.
- [61] Alexander J. Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. Sysxchg: Refining privilege with adaptive system call filters. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS '23*, page 1964–1978, New York, NY, USA, 2023. Association for Computing Machinery.
- [62] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinisky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '19*, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

- [63] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinisky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.
- [64] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: A solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2):127–153, may 2000. <https://doi.org/10.1145/350853.350863>.
- [65] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS*, volume 3, pages 163–176, 2003.
- [66] Masoud Ghaffarinia and Kevin W. Hamlen. Binary control-flow trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1009–1022, New York, NY, USA, 2019. Association for Computing Machinery.
- [67] Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 443–458, San Sebastian, October 2020. USENIX Association.
- [68] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1749–1766. USENIX Association, August 2020.
- [69] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. C2c: Fine-grained configuration-driven system call filtering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1243–1257, New York, NY, USA, 2022. Association for Computing Machinery.

- [70] XYHR Haas and X Hu. The fundamental limit of flash random write performance: Understanding, analysis and performance modelling. *IBM Research Report, 2010/3/31, Tech. Rep*, 2010.
- [71] Ryan Hancock, Sreeharsha Udayashankar, Ali José Mashtizadeh, and Samer Al-Kiswany. Orcbench: A representative serverless benchmark. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*, pages 103–108, 2022.
- [72] Norman Hardy. KeyKOS Architecture. *SIGOPS Operating System Review*, 19(4):8–25, October 1985.
- [73] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. Cross container attacks: The bewildered eBPF on clouds. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5971–5988, Anaheim, CA, August 2023. USENIX Association.
- [74] Keith Henderson, Brian Gallagher, and Tina Eliassi-Rad. Ep-means: An efficient nonparametric clustering of empirical probability distributions. New York, NY, USA, 2015. Association for Computing Machinery.
- [75] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA ’10, page 355–364, New York, NY, USA, 2010. Association for Computing Machinery.
- [76] Zhenghao Hu and Brendan Dolan-Gavitt. Irqdeblob: Reducing driver attack surface in embedded devices. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1608–1622, 2022.
- [77] William Jannen, Jun Yuan, Yang Zhan, Amogh Akshintala, John Esmet, Yizheng Jiao, Ankur Mittal, Prashant Pandey, Phaneendra Reddy, Leif Walsh, Michael A. Bender, Martin Farach-Colton, Rob Johnson, Bradley C. Kuszmaul, and Donald E. Porter. Betrfs: Write-optimization in a kernel file system. *ACM Trans. Storage*, 11(4), nov 2015. <https://doi.org/10.1145/2798729>.
- [78] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP ’19*, page 494–508, New York, NY, USA, 2019. Association for Computing Machinery.

- [79] Dong Hyun Kang, Changwoo Min, Sang-Won Lee, and Young Ik Eom. Making application-level crash consistency practical on flash storage. *IEEE Transactions on Parallel and Distributed Systems*, 31(5):1009–1020, 2020. <https://doi.org/10.1109/TPDS.2019.2959305>.
- [80] Jeongchul Kim and Kyungyong Lee. Functionbench: A suite of workloads for serverless cloud function service. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 502–504. IEEE, 2019.
- [81] Steve R Kleiman et al. Vnodes: An Architecture for Multiple File System Types in Sun UNIX. In *USENIX Summer*, volume 86, pages 238–247, 1986.
- [82] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.
- [83] Swaroop Kotni, Ajay Nayak, Vinod Ganapathy, and Arkaprava Basu. Faastlane: Accelerating Function-as-a-Service workflows. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 805–820. USENIX Association, July 2021.
- [84] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. High performance metadata integrity protection in the WAFL Copy-on-Write file system. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 197–212, Santa Clara, CA, February 2017. USENIX Association.
- [85] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ziegler, Wolfgang Schröder-Preikschat, Daniel Lohmann, and Rüdiger Kapitza. Attack surface metrics and automated compile-time os kernel tailoring. In *NDSS*, 2013.
- [86] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *USENIX Annual Technical Conference*, pages 323–336, 2007.
- [87] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. {F2FS}: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.
- [88] Zijun Li, Yushi Liu, Linsong Guo, Quan Chen, Jiagan Cheng, Wenli Zheng, and Minyi Guo. Faasflow: Enable efficient workflow execution for function-as-a-service. In *Proceedings of the 27th ACM International Conference on Architectural Support*

for *Programming Languages and Operating Systems*, ASPLOS 2022, page 782–796, New York, NY, USA, 2022. Association for Computing Machinery.

- [89] Libmemcached. Libmemcached. <https://libmemcached.org/libMemcached.html>.
- [90] Linux. Apparmor. <https://apparmor.net/>.
- [91] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS’11, page 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.
- [92] Pascal Maissen, Pascal Felber, Peter Kropf, and Valerio Schiavoni. Faasdom: A benchmark suite for serverless computing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, DEBS ’20, page 73–84, New York, NY, USA, 2020. Association for Computing Machinery.
- [93] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.
- [94] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX’93, page 2, USA, 1993. USENIX Association.
- [95] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, aug 1984. <https://doi.org/10.1145/989.990>.
- [96] Marshall Kirk McKusick and Jeffery Roberson. Journaled soft-updates. *Proceedings of EuroBSDCon*, 2010.
- [97] Microsoft. Azure functions dataset 2019. <https://github.com/Azure/AzurePublicDataset/blob/master/AzureFunctionsDataset2019.md>, 2019.
- [98] Microsoft. Azure functions developer guide. <https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference?tabs=blob#function-app>, 2022.
- [99] Changwoo Min, Woon-Hak Kang, Taesoo Kim, Sang-Won Lee, and Young Ik Eom. Lightweight application-level crash consistency on transactional flash storage. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*,

- USENIX ATC '15, page 221–234, USA, 2015. USENIX Association. <https://dl.acm.org/doi/10.5555/2813767.2813784>.
- [100] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers checkpointing and live migration. In *Proceedings of the Linux Symposium*, volume 2, pages 85–90, 2008.
 - [101] Michael Nelson, Beng-Hong Lim, Greg Hutchins, et al. Fast transparent migration for virtual machines. In *USENIX Annual technical conference, general track*, pages 391–394, 2005.
 - [102] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Trans. Comput. Syst.*, 26(3), sep 2008.
 - [103] Patrick O’Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O’Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, jun 1996.
 - [104] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, December 2003.
 - [105] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *15th USENIX Security Symposium (USENIX Security 06)*, Vancouver, B.C. Canada, July 2006. USENIX Association.
 - [106] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped i/o on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 277–293, 2021.
 - [107] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped i/o for fast storage devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC’20, USA, 2020. USENIX Association. <https://dl.acm.org/doi/abs/10.5555/3489146.3489202>.
 - [108] Stan Park, Terence Kelly, and Kai Shen. Failure-atomic msync(): A simple and efficient mechanism for preserving the integrity of durable data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys ’13, page 225–238, New York, NY, USA, 2013. Association for Computing Machinery. <https://doi.org/10.1145/2465351.2465374>.

- [109] Andrea Passwater. 2018 serverless community survey: huge growth in serverless usage. <https://www.serverless.com/blog/2018-serverless-community-survey-huge-growth-usage>, 2018.
- [110] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Application crash consistency and performance with ccfs. *ACM Trans. Storage*, 13(3), sep 2017. <https://doi.org/10.1145/3119897>.
- [111] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: On the complexity of crafting Crash-Consistent applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.
- [112] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. *SIGARCH Comput. Archit. News*, 39(1):291–304, mar 2011.
- [113] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, page 161–176, New York, NY, USA, 2009. Association for Computing Machinery. <https://doi.org/10.1145/1629575.1629591>.
- [114] Vijayan Prabhakaran, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis and evolution of journaling file systems. In *USENIX Annual Technical Conference, General Track*, volume 194, pages 196–215, 2005.
- [115] Anthony Rebello, Yuvraj Patel, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Can applications recover from fsync failures? *ACM Trans. Storage*, 17(2), jun 2021. <https://doi.org/10.1145/3450338>.
- [116] redis. redis-benchmark. <https://redis.io/docs/management/optimization/benchmarks/>.
- [117] New Relic. Serverless benchmark report. <https://newrelic.com/resources/ebooks/serverless-benchmark-report-aws-lambda-2020>, 2020.

- [118] Yujie Ren, Changwoo Min, and Sudarsun Kannan. CrossFS: A cross-layered Direct-Access file system. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 137–154. USENIX Association, November 2020.
- [119] Ohad Rodeh. B-trees, shadowing, and clones. *ACM Trans. Storage*, 3(4), feb 2008.
- [120] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.
- [121] Francisco Romero, Gohar Irfan Chaudhry, Íñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. FaaS: A transparent auto-scaling cache for serverless applications. arXiv, 2021.
- [122] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.
- [123] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. Icebreaker: Warming serverless functions better with heterogeneity. New York, NY, USA, 2022. Association for Computing Machinery.
- [124] O.S. Saydjari, J.M. Beckman, and J.R. Leaman. LOCK trek: navigating uncharted space. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 167–175, 1989.
- [125] Syed Muhammad Ali Shah, Maurizio Morisio, and Marco Torchiano. An overview of software defect density: A scoping study. In *2012 19th Asia-Pacific Software Engineering Conference*, volume 1, pages 406–415, 2012.
- [126] Mohammad Shahradd, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider, 2020.
- [127] Jonathan S. Shapiro and Jonathan Adams. Design evolution of the EROS Single-Level store. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA, June 2002. USENIX Association.
- [128] Kai Shen, Stan Park, and Meng Zhu. Journaling of journal is (almost) free. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies, FAST’14*, page 287–293, USA, 2014. USENIX Association. <https://dl.acm.org/doi/10.5555/2591305.2591333>.

- [129] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.
- [130] Emil Tsalapatis. Application persistence, performance, and deployment in a unix-compatible single level store operating system, 2025.
- [131] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora operating system: Revisiting the single level store. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 136–143, New York, NY, USA, 2021. Association for Computing Machinery.
- [132] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. The aurora single level store operating system. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 788–803, New York, NY, USA, 2021. Association for Computing Machinery. <https://doi.org/10.1145/3477132.3483563>.
- [133] Emil Tsalapatis, Ryan Hancock, Rakeeb Hossain, and Ali José Mashtizadeh. Memsnap μ checkpoints: A data single level store for fearless persistence. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 622–638, 2024.
- [134] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 559–572, New York, NY, USA, 2021. Association for Computing Machinery.
- [135] Rajat Verma, Anton Ajay Mendez, Stan Park, Sandya Mannarswamy, Terence Kelly, and Charles B. Morrey. Failure-atomic updates of application data in a linux file system. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, page 203–211, USA, 2015. USENIX Association. <https://dl.acm.org/doi/10.5555/2750482.2750498>.
- [136] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *USENIX Security Symposium*, volume 46, page 2, 2010.

- [137] Jinpeng Wei and Calton Pu. TOCTTOU vulnerabilities in UNIX-Style file systems: An anatomical study. In *4th USENIX Conference on File and Storage Technologies (FAST 05)*, San Francisco, CA, December 2005. USENIX Association.
- [138] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled io stack for flash storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies, FAST’18*, page 211–226, USA, 2018. USENIX Association. <https://dl.acm.org/doi/10.5555/3189759.3189779>.
- [139] Fangnuo Wu, Mingkai Dong, Gequan Mo, and Haibo Chen. Treesls: A whole-system persistent microkernel with tree-structured state checkpoint on nvm. In *29th ACM Symposium on Operating Systems Principles (SOSP 23)*. USENIX Association, 2023.
- [140] Jian Xu and Steven Swanson. NOVA: A log-structured file system for hybrid Volatile/Non-volatile main memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, February 2016. USENIX Association.
- [141] Sujay Yadalam, Chloe Alverti, Vasileios Karakostas, Jayneel Gandhi, and Michael Swift. Bypassd: Enabling fast userspace access to shared ssds. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS ’24*, page 35–51, New York, NY, USA, 2024. Association for Computing Machinery.
- [142] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, and Kui Ren. Demons in the shared kernel: Abstract resource attacks against os-level virtualization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, CCS ’21*, page 764–778, New York, NY, USA, 2021. Association for Computing Machinery.
- [143] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. Optimizing the block i/o subsystem for fast storage devices. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–48, 2014.
- [144] Haotian Zhang, Mengfei Ren, Yu Lei, and Jiang Ming. One size does not fit all: Security hardening of mips embedded systems via static binary debloating for shared libraries. In *Proceedings of the 27th ACM International Conference on Architectural*

Support for Programming Languages and Operating Systems, ASPLOS '22, page 255–270, New York, NY, USA, 2022. Association for Computing Machinery.

- [145] Quan Zhang, Chijin Zhou, Yiwen Xu, Zijing Yin, Mingzhe Wang, Zhuo Su, Chengnian Sun, Yu Jiang, and Jianguang Sun. Building dynamic system call sandbox with partial order analysis. *Proc. ACM Program. Lang.*, 7(OOPSLA2), oct 2023.