# Application Persistence, Performance, and Deployment in a UNIX-Compatible Single Level Store Operating System

by

Aimilios Tsalapatis

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

© Aimilios Tsalapatis 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: | Andrew Warfield<br>VP & Distinguished Engineer<br>Amazon |
| Supervisor(s): | Ali José Mashtizadeh<br>Associate Professor<br>Dept. of Computer Science<br>University of Waterloo |
| Internal Member: | Samer Al-Kiswany<br>Associate Professor<br>Dept. of Computer Science<br>University of Waterloo |
| Internal-External Member: | Seyed Majid Zahedi<br>Assistance Professor<br>Dept. of Electrical and Computer Engineering<br>University of Waterloo |
| Other Member(s): | Bernard Wong<br>Associate Professor<br>Dept. of Computer Science<br>University of Waterloo |

## Author's Declaration

I hereby declare that I am the sole author of this 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.

**Abstract**

This thesis presents the Aurora single level store, an OS design that uses continuous checkpointing for application persistence and deployment. Aurora provides submillisecond application checkpoint and restore operations to efficiently turn applications into on-disk images and back. Fast checkpointing/restore as an OS service also serves as a foundation for further research into open problems like efficient persistence APIs for memory-mapped data and serverless computing. Aurora's single level store-based persistence has recently become practical because of advances in hardware and file system technology. Modern SSD storage devices have low latency at 10µs, allowing us to persist application checkpoints to the disk with minimal latency overhead. Modern CPUs also have IO throughput that rivals that of their memory bandwidth, making it possible to continuously checkpoint and forward in-memory application state to the disk. This thesis describes three systems that demonstrate the efficiency and flexibility of the single level store paradigm. We first present Aurora (SOSP 2021), an OS design capable of continuous application checkpointing at a fast enough granularity to provide transparent persistence. We follow up with MemSnap (ASPLOS 2024), an OS single level store API and associated virtual memory mechanism. MemSnap persists application data, e.g., database data, more efficiently than the file API. Finally, we present Metropolis, a serverless invoker that uses the single level store paradigm to create serverless function instances at submillisecond latency.

**Acknowledgements**

I would like to thank my advisor, Prof. Ali Mashtizadeh, for his steadfast guidance throughout my PhD. I would also like to thank Prof. Samer Al-Kiswany, with whom I had the privilege to work with on the Metropolis project.

I would also like to thank all my colleagues at the RCS Lab: My coauthors Ryan Hancock, Tavian Barnes, and Rakeeb Hossain who worked on the conference versions of the papers that form the basis of this thesis; and Haoyu Gu, Oscar Zhao, Stefanie Dukovac, Micheal Friesen, Fatemeh Hassani, and Zahra Rezapour for their helpful discussions and feedback.

Lastly, I would like to thank my family for their support throughout my PhD.

# Table of Contents

**Examining Committee** ii

**Author's Declaration** iii

**Abstract** iv

**Acknowledgements** v

**List of Figures** x

**List of Tables** xiii

**1 Introduction** 1

1.1 The Aurora Single Level Store . . . . . 2

1.2 MemSnap: SLS µCheckpoints: for Low-Latency Persistence . . . . . 3

1.3 Metropolis: An SLS-Based Serverless Invoker for Fast Cold Starts . . . . . 4

**2 Background** 5

2.1 Single Level Stores . . . . . 5

2.2 Existing SLSes . . . . . 6

2.3 Checkpoint-Restore (CR) as a Technique . . . . . 7

2.4 The Evolution of Checkpoint Restore . . . . . 9

2.5 Implementing Checkpoint-Restore . . . . . 11
2.5.1 The Checkpoint Operation . . . . . 12
2.5.2 The Restore Operation . . . . . 14
2.6 Checkpoint-Restore as an OS Interface . . . . . 15
2.7 Applications of Checkpoint-Restore . . . . . 16
2.8 Persistence and Reliability . . . . . 16
2.9 Other Related Work . . . . . 19

**3 The Aurora Single Level Store 22**
3.1 Introduction . . . . . 22
3.2 Background . . . . . 24
3.3 Using Aurora . . . . . 28
3.4 The Aurora Operating System . . . . . 30
3.4.1 Aurora's Architecture . . . . . 31
3.5 Making POSIX Persistent . . . . . 33
3.5.1 Basic Persistence . . . . . 33
3.5.2 Persistence Challenges . . . . . 35
3.5.3 Completing Persistence . . . . . 36
3.6 Making Persistence Fast . . . . . 37
3.7 Aurora Object Store . . . . . 41
3.8 Implementation . . . . . 42
3.9 Evaluation . . . . . 42
3.9.1 The Aurora File System . . . . . 43
3.9.2 Persisting POSIX Objects . . . . . 46
3.9.3 Checkpointing Memory Objects . . . . . 47
3.9.4 Application Checkpointing . . . . . 49
3.9.5 Transparent Persistence for Memcached . . . . . 51
3.9.6 Customized RocksDB . . . . . 53

3.9.7 Comparison with Linux CRIU . . . 55
3.10 Related Work . . . 56
3.11 Limitations and Future Work . . . 57
3.12 Conclusion . . . 59

**4 MemSnap: SLS µCheckpoints for Low-Latency Persistence 61**
4.1 Introduction . . . 61
4.2 Motivation . . . 63
4.3 Design . . . 67
4.4 MemSnap-based Crash Consistency . . . 71
4.5 Scope and Limitations . . . 74
4.6 Evaluation . . . 74
4.7 Case Studies . . . 76
4.7.1 Case Study: SQLite . . . 77
4.7.2 Case Study: RocksDB . . . 82
4.7.3 Case Study: PostgreSQL . . . 86
4.8 Related Work . . . 90
4.9 Conclusion . . . 91

**5 Metropolis: An SLS-Based Serverless Invoker for Fast Cold Starts 92**
5.1 Introduction . . . 92
5.2 Background . . . 94
5.3 Design . . . 98
5.3.1 Overview . . . 98
5.3.2 Triggering Snapshotting/Invocations . . . 101
5.3.3 Fast Address Space Creation . . . 102
5.3.4 Collaborative Warmups . . . 103
5.3.5 Using Introspection . . . 104

5.3.6 Security Performance Considerations . . . . . 106
5.4 Implementation . . . . . 107
5.5 Evaluation . . . . . 107
5.5.1 In-memory Invocation Times . . . . . 108
5.5.2 Paging Strategies . . . . . 110
5.5.3 Disk Density . . . . . 113
5.5.4 Sustained Invocation Throughput . . . . . 113
5.5.5 Comparison to Other Systems . . . . . 114
5.6 Related Work . . . . . 117
5.7 Conclusion . . . . . 118

**6 Future Work** **119**

**7 Conclusion** **122**

**References** **124**

# List of Figures


3.1 Aurora system diagram. . . . . 31

3.2 VM Memory Diagram of FreeBSD. Address spaces include page tables and a list of mapped regions. Each region is backed by mappable groups of physical pages called VM objects. VM objects represent anonymous or file memory and back each other to enforce COW semantics. . . . . 39

3.3 FileBench microbenchmarks that evaluate the Aurora file system to both ZFS and FFS. The Aurora file system and its checkpoint-based consistency scheme is competitive with existing file systems. . . . . 44

3.4 Memcached at max throughput over varying checkpoint periods. . . . . 51

3.5 Memcached with throughput pegged at 120 k ops/s over varying checkpoint periods. . . . . 51

3.6 A comparison between multiple RocksDB configurations running the Prefix_dist Facebook workload. Configurations that do not provide write persistence are labeled "No Sync" while those that do are labeled "Sync". . . . . 53

4.1 Comparison of the three techniques for marking pages as read only. Traversing the mapping's page tables is expensive even for small dirty sets, and traversing the page table for each page adds overhead for large dirty sets. MemSnap's trace buffer is both fast and scalable. . . . . 69

4.2 The three conditions required for MemSnap-based persistence. ① MemSnap persists only dirty pages that are in a MemSnap region. ② Two threads cannot simultaneously dirty different parts of the same page. ③ A page dirtied by a thread cannot be dirtied by another until it is flushed out. . . 72

4.3 Comparison of MemSnap with Aurora's region and application checkpointing. We measure the latency of a synchronous persistence operation for randomly distributed dirty sets of different sizes. MemSnap outperforms Aurora's region checkpointing by 7× for small IOs because it processes individual pages. 77

4.4 Performance of MemSnap vs the WAL+Checkpoint baseline for transaction sizes from 4 KiB to 1 MiB. The actual IO done is larger than the transaction size because of write amplification. The latency for both configurations scales linearly with transaction size, but MemSnap is faster both for average and tail latency across all transaction sizes. MemSnap has a fixed per-4 KiB page latency across all transaction sizes, while the baseline's latency rises for smaller sizes. . . . . 79

4.5 TATP benchmark. The baseline slightly outperforms MemSnap for small database sizes by because of its WAL index, but heavily degrades in performance for larger databases. . . . . 81

4.6 Performance of four variations of PostgreSQL running a TPC-C benchmark: Baseline FFS, FFS with `mmap`'d files, FFS with `mmap` and also directly writing data instead of using the buffer cache (i.e., bufdirect, labeled "bd"), and MemSnap. . . . . 87

5.1 Overview of the Metropolis platform. The invoker on the local machine registers new functions by setting up the container instance, which automatically triggers a checkpoint after it is setup. On invocations it retrieves the image from the cloud if needed, restores it, and hands off a socket to it to directly receive arguments from the orchestrator. Metropolis applies uses introspection and sharing (§ 5.3) during the restore process to accelerate sharing. . . . . 99

5.2 The Metropolis sharing mechanism. . . . . 100

5.3 Performance gains for the socket handoff optimization for simple functions echoing a 6MB argument. Removing the invoker from the path between the orchestrator and the function reduces communication overhead by 45% for compiled languages (C, Go) and 25% for the Python interpreter and Node.JS JIT. . . . . 105

5.4 Metropolis and CRIU invocation times from images on a ramdisk and physical disk. . . . . 110

5.5 Paging Strategies vs Execution times. Sharing is performant compared to tracing as it avoids IO for the working set. . . . . 111

5.6 Invocations per second on a single machine with two PCIe SSDs. The disk restores achieve $\tilde{8}0\%$ of the SSD throughput, because of some latency bound operations in our storage code. The memory case seems to be limited but requires 12 cores just processing helloworld to saturate. . . . . 114

5.7 The size of a function's address space in the image. Metropolis' passive sharing technique measurably reduces the total checkpoint size. Delta checkpointing is effective for trivial functions (`helloworld`, `pyaes`), but it cannot reduce the size of large shared libraries in larger workloads like `video_processing`. 115

5.8 Breakdown of Metropolis(m) and vHive(v) cold start latency for five FunctionBench workloads. Metropolis has better performance both at invocation and execution time, except for `json_serdes` for which Metropolis is using an outdated version of the package. . . . . 116

5.9 Cold starts for Metropolis snapshot restore modes. Metropolis consistently outperforms Catalyzer by 2–3× due to its optimized deserialization path. . 116

# List of Tables


3.1 A breakdown of CRIU's checkpointing overheads for a 500 MB Redis process. 27

3.2 A subset of the command line interface. . . . . 29

3.3 A subset of the Aurora application API. . . . . 30

3.4 Checkpoint and restore times for POSIX objects. . . . . 47

3.5 Checkpoint times for userspace data objects using differing modes of Aurora's API. . . . . 48

3.6 Checkpoint stop times and restore times for several popular applications. . 49

3.7 Comparing Aurora's full checkpoint performance to CRIU with a 500 MiB Redis instance. Aurora is two orders of magnitude faster in terms of stop time both for metadata and data copies. Aurora writes out the checkpoint 3 faster, even though CRIU does not flush to disk. . . . . 55

4.1 Breakdown of RocksDB's total execution time between userspace and the kernel. Only 18.3% of CPU time is spent on the in-memory transaction - the rest is spent on persistence across userspace and the kernel. . . . . 64

4.2 Latency breakdown for synchronous Aurora region checkpointing during RocksDB's `dbbench` benchmark. Most of the overhead comes from the shadowing mechanism that applies to entire mappings, even though the amount of dirty data is 64 KiB. . . . . 66

4.3 Comparison between different persistence mechanisms. `fsync` and `msync` flush an entire file or region's dirty set without atomicity. Atomic `msync` requires contiguity and does not track the per-thread dirty set. Aurora is not microsecond-scale and does not track the dirty set per thread. MemSnap combines strong guarantees, performance and per-thread set tracking. . . . 66

4.4 The MemSnap API. Users either explicitly register page-aligned regions of memory into the snapshot or implicitly track them throughout execution. MemSnap atomically persists these pages. Threads that should block until the data persists do so using `msnap_persist` with the `MS_SYNC` flag or `msnap_wait`. . . . . 67

4.5 Breakdown of an `msnap_persist` call for 64 KiB worth of dirty pages. The call protects the pages to reset tracking in 5.1 µs, and spends 39.7 µs waiting for the IO to complete. Minimizing COW application time quickly frees up the modified pages to be accessed and modified by other threads and thus avoids contention. . . . . 71

4.6 The latency of persistence for `fsync`, `msnap_persist`, and Aurora. We use `fsync` after writing out a random or sequential series of 4 KiB pages and compare its latency with that of `msnap_persist` for a random IO pattern of the same size. MemSnap outperforms disk writes for large write sizes because disk writes always have one outstanding IO. . . . . 76

4.7 The number and cost of persistence-related system calls for both MemSnap and the baseline. MemSnap only requires the `msnap_persist` call that is both less frequent and cheaper than `fsync`. The baseline in contrast uses WAL and checkpoint that generates a significant amount of file IO. . . . . 80

4.8 CPU usage and total `dbbench` execution time for MemSnap and the baseline. MemSnap runs 2× to 5× faster than the baseline while doing a similar amount of work in userspace. . . . . 80

4.9 RocksDB latency comparison between MemSnap RocksDB, a WAL-based baseline, and Aurora's region checkpointing. MemSnap provides 3× lower latency than Aurora as it persists data with a single call. MemSnap's costs scale down with the dirty set, unlike Aurora's region checkpoints. . . . . 85

4.10 Breakdown of MemSnap vs Aurora's persistence cost. Aurora's region COW tracking causes 80% of its total latency. MemSnap's page tracking avoids these overheads. MemSnap also avoids contention between concurrent persistence operations compared to Aurora. . . . . 86

5.1 Breakdown of the Disk Usage for a Firecracker image and a HelloWorld application. The application is a less than 1 KiB of Python. . . . . 96

5.2 Shows the breakdown of total response time into restore time and execution time for helloworld across three language runtimes. . . . . 108

5.3 Shows the pairwise benefit of collaborative warmups across any two benchmarks in FunctionBench. The biggest benefit is for workloads warming up other instances of themselves, which share all libraries and most memory. . 112

5.4 Image sizes and density per TB of storage for FunctionBench functions, using full images versus Metropolis images. The sizes include the function snapshot and all dependencies. Metropolis' on-disk sharing scheme reduces the size of function images by up to 98.9%. . . . . 113

# Chapter 1

# Introduction

Single level storage (SLS) systems provide persistence of applications as an operating system service. Their advantage lies in removing the semantic gap between the in-memory representation and the serialized on-disk representation that uses file IO APIs. This gap often leads to increased code complexity and software bugs [193, 44]. Instead, applications only use memory semantics to access data, and the operating system persists this state to disk. Developers design programs as if they never crash and thus do not write code for persistence and recovery. After a crash, the SLS restores the application, including all state (i.e., CPU registers, OS state, and memory), which continues executing oblivious to the interruption.

SLSes have been impractical to build for decades for performance reasons, but this has changed with the advent of new storage technologies. Past systems suffered from a large performance gap between memory and disks in terms of bandwidth and latency. This was compounded by write-amplification due to the tracking of memory modifications at page granularity, and the overhead of CPU and OS state. Modern flash, coupled with fast PCIe Gen 4–5, has largely closed the performance gap with memory.

We introduce the Aurora Operating System, a novel non-traditional single level storage system that enables persistence and manipulation of execution state. Aurora is based on the FreeBSD kernel and is the first SLS that can run unmodified POSIX applications. Aurora provides persistence at the granularity of process trees or containers, and supports multi-process applications with nearly all POSIX primitives. This allows for the persistence of complex applications like Firefox, a popular web browser.

Aurora differs from previous systems in several ways. EROS [209] requires application cooperation to achieve performance and its main contributions are optimizing checkpointing

and swap for spinning disks. IBM's AS/400 uses runtime and compile time hooks along with application hints to achieve good performance [217].

Aurora provides a system level service for manipulating arbitrary application state. It goes much further than traditional SLSes by blurring the line between applications and data. Users can operate on running applications to persist, copy, revert, or transfer them the same way they would a file. Aurora makes state manipulation an explicit operation, which programs often need to do in an ad hoc manner by themselves. Aurora creates application checkpoints that encapsulate all information required to recreate the application, even across reboots and machines.

This thesis argues that application persistence and manipulation of execution state naturally belong in the operating system, and enable novel applications in modern systems. Aurora allows us to solve a wide range of complex systems problems, from reducing application startup times and increasing density of serverless computing, to simplifying database design.

This thesis includes portions based on previously published papers. This introductory chapter is partly based on a HotOS 2021 workshop paper published with Ryan Hancock, Tavian Barnes, and Ali Mashtizadeh [225]. Chapter 2 is based on an SOSP 2021 conference paper published with Ryan Hancock, Tavian Barnes, and Ali Mashtizadeh [226]. Chapter 3 is based on an ASPLOS 2024 paper with Ryan Hancock, Rakeeb Hossain, and Ali Mashtizadeh [227], while Chapter 4 is based on a paper under submission with Ryan Hancock, Ali Mashtizadeh, and Samer Al-Kiswany.

## 1.1 The Aurora Single Level Store

We first present the Aurora single level store (SLS), an OS that transparently provides persistence to unmodified applications. Aurora uses high-frequency checkpoint/restore in the kernel to persist entire applications and not just their data, allowing applications to resume executing after a crash oblivious of the interruption. Unlike previous SLSes like EROS, Aurora persists applications at a high frequency using modern low-latency and high-throughput storage devices.

Persistence is a difficult property to ensure, and even mature applications have had significant data loss bugs over the years [8, 10, 193, 44]. The complexity of efficiently persisting data is caused by the use of the file API for persistence. The file API requires developers to write userspace code that explicitly flushes data throughout the storage stack, inlcuding the virtual memory subsystem, the buffer cache, and the file system layer. The complexity of modern storage stacks makes writing such code complex and error-prone.

Aurora correctly and efficiently persist entire multiprocess applications using two mechanisms: The POSIX system model, and system shadowing. Aurora checkpoints the application as it is represented in the kernel so that it accurately and efficiently captures sharing relationships between application processes like common mappings and files. This approach is faster and simpler than currently userspace mechanisms that extract application data using the `proc` file system and deduplicate it at checkpoint time. Aurora also uses a virtual memory mechanism called system shadowing to parallelize application execution with the flushing of checkpoints, using COW to track and protect the pages in each application checkpoint.

## 1.2 MemSnap: SLS µCheckpoints: for Low-Latency Persistence

MemSnap is an extension to Aurora that provides efficient persistence for high-performance applications like databases. Databases currently use the file API for persistence, but the API has significant limitations like lack of atomic flush operations, the inability to flush non-contiguous memory regions with a single call. These limitations force developers to use mechanisms like writeahead logs (WALs) to properly persist data, but these mechanisms are complex to write and difficult to combine with memory-mapped IO. Aurora's transparent checkpoints remove the complexity of writing persistence code, but persist state at a millisecond granularity and cannot be used as a direct replacement for the low-latency commit operations normally done by databases.

MemSnap introduces µCheckpoints, a mechanism designed to atomically persist to disk arbitrary sets of application data pages. Applications that use µCheckpoints operate on their data as if the system never crashes, so developers need only implement in-memory transactions without worrying about persistence. MemSnap integrates with the existing commit protocols of the database, turning in-memory commits into on-disk commits without developer effort.

Applications do not explicitly move data between memory and the disk with the file API, but instead modify them as if they are completely in memory then mark changes as persistent during a commit operation with a single call. MemSnap transparently tracks, flushes, and protects data while in flight, simplifying the development of persistent memory-mapped applications. MemSnap implements these mechanisms using a novel virtual memory-based mechanism for page tracking and protection at a thread granularity. This mechanism implements page tracking using page table-based techniques akin to those used by copy on write.

## 1.3 Metropolis: An SLS-Based Serverless Invoker for Fast Cold Starts

Metropolis is an extension of Aurora that adapts its restore path for serverless computing. In serverless computing, the unit of execution is the *function instance*, an on-demand sandbox with the function's code and runtime. Creating sandboxes for the function instances and initializing their runtimes is done on demand during invocations. However, the short runtime of most functions means that the function initialization operation adds significant end-to-end latency to the function's invocation.

Metropolis creates new function instances by restoring functions from prepared images that hold the state of preinitialized instances (function snapshotting). Metropolis improves over existing state-of-the-art snapshotting systems in two regards: First, it does not require checkpointing entire sandboxes but only the applications running in them. Functions are unique, but are run in identical sandboxes, so sandboxes can be prepared ahead of time instead of being created at invocation time. We present a design that uses Metropolis' application restores with prepared microVM [66] sandboxes to combine the speed of application restores with the safety of virtualization-based sandboxing.

Second, Metropolis improves function invocation latency by sharing common function image state between similar functions. Most functions share large amounts of state between them, like their language runtime and shared libraries. Metropolis takes advantage of this to safely share common state between function images, and by extension function instances. Safe sharing allows Metropolis to store function images in memory instead of on cloud storage, speeding up invocations compared to other snapshotting systems.

# Chapter 2

# Background

This thesis presents a single level store architecture that brings efficient checkpoint restore to Unix-like OSes. Checkpoint-restore is a powerful primitive that solves many otherwise difficult systems problems like application persistence, deployment, and migration. Existing checkpoint-restore systems are limited by fundamental performance bottlenecks that force them to make tradeoffs in their design. Such tradeoffs make these CR systems competitive for one application domain, e.g., application migration, but prevent them from being used in other domains like transparent persistence. In contrast, this work presents a CR system based on single level stores that satisfies all use cases with a unified design.

This chapter describes the motivation behind revisiting single level stores (SLSes), why these systems are becoming increasingly relevant for current- and next generation systems, and why past approaches fail to satisfy the demands of modern software. We provide an introduction to SLS operating systems (OSes), and define them in comparison to conventional OSes. We explain that SLSes are fundamentally related to efficient checkpointing, but existing SLSes do not address this link. Finally, we explain why existing SLS and SLS-adjacent designs fail to solve the CR problem for modern applications, and demonstrate the need for a novel SLS architecture.

## 2.1 Single Level Stores

This work focuses on providing single level store (SLS) semantics to conventional UNIX applications, with focus on persistence. Single level stores are operating systems that provide a uniform interface for accessing data, whether that is volatile, persistent, or OS metadata. The goal of this uniformity is twofold: First, it makes the programming model

simpler for application developers. Developers use the same code patterns to access data regardless of its type and location in the memory hierarchy. The second goal is to hide data and even application movement in the system from the user. That is, the system can move data and application state from memory to the disk or to a remote machine without requiring developer code. Such data movement is done for persistence, fault tolerance, or load balancing.

Existing single level stores have all been incompatible with UNIX applications and require OS designs built around the SLS model (2.2). Academic SLSes have been mostly designed to explore the SLS model for operating systems, and do not support existing POSIX applications. SLSes developed in industry similarly have their own specific APIs and do not consider the problems of integrating with existing POSIX applications.

This thesis explores providing an SLS design for POSIX applications. To be exact, we provide transparent application and data movement and an optional uniform data interface for UNIX systems. The challenges we address are completely different from those of existing SLSes, which could be designed from scratch and therefore expose system interfaces that made SLS semantics easy to implement. In contrast, we deal with providing transparent data movement for data organized as anonymous memory regions, files, or even POSIX pseudofiles like pipes, without changing the underlying OS primitives.

To address this challenge we use kernel-level application checkpointing. Application checkpointing allows us to handle all application data in a uniform representation, that of the application checkpoint image (2.3). This in turn allows us to transparently move application state throughout the storage hierarchy and between machines as expected by the SLS paradigm. This has the additional advantage of providing SLS semantics on top of existing OSes, simplifying application development.

We provide SLS semantics for UNIX applications because with them we can solve a wide array of difficult systems problems. Treating application state as data allows us to inspect it, modify it, and move it throughout the storage hierarchy and the network. Provided these operations are efficient, we can provide services like persistence, fault tolerance, and application introspection directly from the OS without developer effort(2.6).

## 2.2 Existing SLSes

One of the first single level stores was the Multics [97] operating system. Multics was a single level store in that it allowed for transparent memory overcommit by hiding the data movement between memory and disk from the user similar to modern paging subsystems.

The AS/400 [217] system from IBM provided SLS semantics, including simpler persistence semantics than conventional systems. The AS/400 used an object-based storage API for which it enforced persistence using custom microcode, OS code, and a compiler toolchain.

The Locus [236] operating system provides reliability in case of machine failures by being distributed between multiple computers. Locus used a distributed directory structure to index and access data across a network, and mainly concerned itself with making task execution reliable, e.g, by transparently replicating data between nodes. It provides reliability just for data and does not persist or move application state, but its overall design was influential for subsequent single level store-based designs.

The Apollo Domain/OS [177] uses a single level store design as a basis for a distributed operating system. Domain/OS exposes all data as objects, mappable virtual memory regions that can represent RAM, disk space, or even OS primitives like IPC mailboxes. This SLS-based design allows Domain/OS to provide a uniform access interface to applications for all data in the system. However, Domain/OS mainly uses SLSes to make OS interfaces simple in a distributed environment, and does not provide application checkpointing or persistence.

EROS [209] is a single level store that focuses on durability, and uses checkpoint-restore for that purpose. EROS is a capability-based system that expands the single level store design to provide transparent persistence. EROS provides periodic system global checkpoints wherein it persists the entirety of the system to the disk, both metadata and data. However, the EROS paper states that the system provides persistence guarantees 'at frequencies comparable to those of conventional buffer cache flushes' [209], meaning that it can only periodically checkpoint applications. EROS is thus not appropriate for applications like databases that require high-frequency persistence.

TreeSLS [245] is a recent single level store that builds on the work published in this thesis and adapts it for systems that exclusively use persistent memory. TreeSLS mainly concerns itself with optimizing checkpointing for persistent memory-based machines, and is not applicable to conventional systems.

## 2.3 Checkpoint-Restore (CR) as a Technique

Checkpoint-restore systems gather the state of a running application into an executable image. These images look just like data and users can move them between machines, store them, copy them, modify them and inspect them. Users can also restore the image, recreating the running application exactly as it was at checkpoint time. Restored applications execute normally and are indistinguishable from applications created with regular `fork`/`exec`.

Application images can be restored on a different system from the one the checkpoint was created in. Checkpoint-restore includes all information related to the application, even if that information normally resides in the kernel, e.g., the layout of the virtual address space of a process. For this reason, restores work even if the machine was rebooted after the checkpoint, or even if the checkpoint was created on a different machine and moved to this one.

Checkpoint restore turns an application into an image by gathering all of its state, serializing each component, and storing it into some data format. This is possible because from the point of view of the OS, applications are simply collections of interrelated system resources like address mappings, file descriptors, and threads. These resources may refer to each other, but are ultimately independent. Each of these resources are also serializable into a data record, and are also deserialized from that record.

For example, a user may checkpoint a virtual address mapping that may represent data like the process heap by reading it out into a file. This involves reading the mapping metadata, e.g., its virtual address, permissions, and `madvise` hints using `/proc`, then reading the data pages of the mapping and storing them with the serialized metadata. Deserializing the mapping just requires using the `mmap`, `mprotect`, and `madvise` system calls to recreate the mapping metadata, then copy the data in the mapping with `memcpy`.

Checkpoint-restore is possible because of these two properties: Applications are collections of self-contained resources, and each resource is serializable/deserializable by itself. Checkpointing an application only requires checkpointing each primitive separately, then collecting all serialized data into a checkpointing image. Restoring the running application at a later time only requires deserializing each primitive, then collecting the restored instances into an application. The application itself is unaware of the operation, and the restored instance is indistinguishable from the original.

Checkpoint-restore has found widespread use in some form or another as a way to solve systems problems related to monitoring, preserving, and inspecting application state. Its main advantage over other techniques is its transparent nature: Applications do not require modification to work with checkpoint-restore, in fact, we can use checkpoint-restore to provide functionality that normally requires bespoke application code.

We now go through a brief history of checkpoint-restore as a technique to explain its use cases and highlight that there are potential application domains that have not been explored. Checkpointing has been used for coarse-grained persistence mechanisms, application migration, and fault tolerance. However, other applications like including transparent persistence and application deployment are notably absent. We expand on all potential use cases of checkpoint-restore later in this chapter (2.6.

## 2.4 The Evolution of Checkpoint Restore

Checkpoint-restore has been a longstanding technique that has evolved over time as the intended application domains have changed. Early forms of application checkpointing were popular in distributed operating systems like Demos/MP [197], LOCUS [236], V [90], Amoeba [220, 219], and Sprite [107]. These systems used checkpoint-restore to migrate processes between different OSes. These systems migrate processes by checkpointing an application in one node, then moving it to a remote node for load balancing.

Checkpoint-restore mechanisms in these systems assume a distributed OS and make simplifications not possible in single-node OSes. For example, both Demos/MP and Sprite use some variation of a 'home node' to avoid fully checkpointing a process: When migrating a process from a source to a destination machine, these systems leave a stub process in the source into which they redirect certain system calls that read application state from the kernel. This optimization avoids having to read that state into the checkpoint image at checkpoint time, but is not applicable to single-node operating systems.

Early application checkpointing systems for persistence focused on the cost of gathering the memory state of the application during checkpointing [159]. This work builds on earlier systems that used checkpointing for reliability [154] and uses then-new virtual memory hardware to optimize checkpoint times. Checkpointing for reliability initially used a stop-the-world approach wherein the checkpointed application was stopped while the checkpoint was created and flushed to the disk, to ensure the application did not modify the data being checkpointed. The advent of virtual memory made it possible to checkpoint concurrently with application execution, and the cited work uses COW at the system level to prevent application writes to memory that has not yet been checkpointed. However, this system does not consider either multiprocess applications or non-memory state: In fact, the system is not allowed to have open file handles at checkpoint time, or be making remote procedure calls.

This initial style of checkpointing has also been used by systems that provided coarse-grained fault tolerance [112]. The cited system uses single-process COW-based checkpointing every 2 minutes to checkpoint distributed applications, but still limits itself to single processes. The work's main contribution is demonstrating that checkpointing can be expanded to distributed applications, but did not include any new optimizations for single-node checkpointing.

Condor [222] was one of the first systems to provide checkpointing for Unix processes, but only for process migration. Condor handles single Unix processes and considers it impossible to support all Unix abstractions, instead aiming for a subset of features they

consider practical. Limitations include programs that use fork() or exec(), which is very frequent in modern applications.

Subsequent work brought more complete checkpointing to Unix: `libckpt`[195] was the first publicly available full checkpointing system. `libckpt` is simple and supports only single process applications. The system minimizes checkpoint overhead using `fork` to implement COW to minimize the stop time of the checkpoint. `libckpt` was designed to provide coarse-grained asynchrony, once every 10 minutes, and so does not consider the complications of continuous incremental persistence, or of multiprocess checkpointing.

CRAK [251] is the first modern attempt to add checkpointing to widely used OSes, and the authors note in the paper that adding checkpointing is a difficult task. It expands on a previous library called `epckpt` [194]. It uses checkpointing for the purpose of task migration. A limitation that CRAK has, similar to previous systems, is that it checkpoints each process separately. Unlike previous systems, CRAK also does not use COW to reduce stop time. As a result, it costs 200 ms to checkpoint even small utilities like find and grep. The main appeal of CRAK over previous kernel attempts is that it is self-contained as a kernel module.

Zap [184] is followup work to CRAK and introduces the concept of pods, a precursor to modern containers. Zap uses kernel-side namespaces to prevent collisions when restoring an application between resources of a newly created app and any existing ones. While Zap handles multiprocess applications, the work initially did not describe any optimizations for continuous checkpointing of multiprocess applications, and is left as followup work.

The aforementioned followup [150] adds performance optimizations to Zap to improve checkpoint and restore times. On the restore side, Zap uses successive `fork` operations to reconstruct applications. This requires careful ordering of the creation operations and introduces non-fundamental dependencies between process restore operations that increase restore time. As a result, Zap requires restore times of above 10 ms for all workloads. This is acceptable latency for application migration, which is Zap's main use case, but does not work for use cases like application deployment (Section 2.6).

Zap is also designed for one-off checkpoints because it focuses on the application migration, so its design is not appropriate for use cases like transparent persistence. For example, Zap uses COW to make checkpointing concurrent with execution, but does not do differential checkpointing as required for transparent persistence (Section 2.6). This means that every application checkpoint gathers all of the application's memory state, even if it has not been changed between checkpoints. Even for single checkpoints, Zap requires 10s of milliseconds in application downtime to gather application state.

The DragonflyBSD [35] system is the only current UNIX system that implements

checkpointing as a system call. However, this call only checkpoints a single process and does not handle multiprocess applications. The system call also does not handle applications that use pseudofiles like pipes. This is due to the complexity of checkpointing application metadata.

The only widely available system for checkpoint/restart in Linux for the last 15 years is CRIU [17]. CRIU is a checkpoint-restore system for Linux built mostly in userspace, but also supported by Linux kernel checkpoint-restore APIs [34]. CRIU repurposes conventional system interfaces like the system call interface and pseudo file systems, using them to extract application state out of the kernel and into a checkpoint file. CRIU implements restore operations using sequences of `fork` and `exec` calls to recreate the application process tree, then restoring the original application on top of it one process at a time.

CRIU is used mostly for application migration, and has checkpoint stop times in the 100s of milliseconds (Chapter 3). On the restore side, CRIU requires 10s of milliseconds to restore even small applications, making it inappropriate for use cases like application deployment (Chapter 5). Optimizations for like VAS-CRIU [231] optimize checkpoint and restore for in-memory checkpoints by modifying Linux's virtual memory system with a COW mechanism for entire address spaces, but do not address CRIU's limitations for domains like application persistence as it is designed to be completely in memory.

## 2.5 Implementing Checkpoint-Restore

We now go into more detail on how the design of a typical checkpoint-restore system to provide better understanding of the performance considerations. Checkpoint-restore systems, whether in the kernel, like Zap, or in userspace, like CRIU, make the same tradeoffs in their design depending on their intended application domain. We first present the performance considerations of these systems, then present the application domains and explain how checkpoint-restore adapts to them.

These systems use established system APIs to read the state of an application out of the system. These interfaces include system calls that expose application state to the user, for example:

- `capget()` for task capabilities
- `getpid()` for getting the PID of a task
- `getsockopt()` for querying the state of a socket file descriptor
- `sched_getattr()` for a task's scheduler attributes

The other type of interface for reading application state is pseudo file systems like `/proc`, `/sysfs`, and `/debugfs`. These file systems expose a directory tree that mirrors the process tree and provides readable files, each of which corresponds to a system resource used by an application. For example, /proc represents each process in the system as a directory named after the PID of the process. The directory then has files that represent the threads, file descriptors, and mappings in use by the process. Reading each such file returns a serialized representation of the resource. Going back to the mapping example, each directory includes a file called `maps` that holds all the permissions and location of all mappings in the process, while a file called `mem` holds the actual memory contents.

```
#ls -1 /proc/`pidof bash`
<snip>
ksm_merging_pages
ksm_stat
maps
mem

#cat /proc/`pidof bash`/maps
:!cat /proc/5350/maps
55c06d200000-55c06d2f2000 r-xp 00000000 fd:00 4199950 /usr/bin/bash
55c06d355000-55c06d360000 rw-p 00000000 00:00 0
55c08b5d7000-55c08b5f8000 rw-p 00000000 00:00 0 [heap]
7fbbfe400000-7fbc0c197000 r--p 00000000 fd:00 4194393 /usr/lib/locale/locale-archive
7fbc0c2b7000-7fbc0c2ba000 rw-p 00000000 00:00 0

7fbc0c2ba000-7fbc0c42a000 r-xp 00000000 fd:00 4228194 /usr/lib64/libc.so.6
7fbc0c4a6000-7fbc0c4ae000 rw-p 00000000 00:00 0
7fbc0c4f4000-7fbc0c4f6000 rw-p 00000000 00:00 0

7fbc0c4f6000-7fbc0c4fa000 r--p 00000000 00:00 0 [vvar]
7fbc0c4fa000-7fbc0c4fc000 r-xp 00000000 00:00 0 [vdso]
7fbc0c4fc000-7fbc0c526000 r-xp 00000000 fd:00 4228191 /usr/lib64/ld-linux-x86-64.so.2
7ffdf880d000-7ffdf882e000 rw-p 00000000 00:00 0 [stack]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

Listing 2.1: Contents of the `proc` pseudo file system for a simple bash process. Non-memory related files omitted. The `maps` file in the directory holds all the metadata for the virtual address mappings of the process.

### 2.5.1 The Checkpoint Operation

Checkpointing an application requires stopping it, extracting all of its state, then resuming it to let it continue executing. Stopping the application aims at quiescing it, so that

the image is consistent: Checkpoints must represent the application as it was at a single point in time, and so we cannot allow changes to its state while checkpointing. The next step is actually extracting the state, which is done using one of the APIs outlined above. Finally, the application resumes running. The application is not aware of the checkpointing operation having taken place, because none of its state has been modified, and because the stop and resume operation looks like a regular scheduling event.

This stop-extract-resume cycle is the main source of overhead for checkpointing operations. Reading application state must be as fast as possible, because it requires that the application is stopped for the duration. For this reason, checkpointing is essentially a blocking operation for the application and its latency is visible to the latter.

Creating a checkpoint image requires gathering two types of application resources: small and structured, or large and unstructured. By structured here we mean that the resources' image is a data structure with well-defined fields that correspond to some attribute of the resource. For example, the image for a thread structure is a C structure with a field for the thread ID, fields for the register values at checkpoint time, fields that refer to the process the thread belongs to, and so on. In contrast, other application resources are mostly unstructured: For example, the checkpoint image for an address mapping includes the pages of the mapping, which cannot be interpreted outside of the application.

Checkpointing structured and unstructured resources each has its own set of performance challenges. For structured resources the main difficulty is in quickly extracting and serializing the data structures out of the system while also representing any references between system resources in the corresponding image. An example is checkpointing a file handle shared between multiple processes in an application. This involves reading the data for the file handle, e.g., through the `/proc` file system, and also linking it to the checkpointed image of the underlying file it refers to, e.g., a real file in the file system, or a pseudofile like a UNIX socket. Both reading out the data and correctly processing it so that the resulting checkpoint properly represents the application can be time-consuming.

For large unstructured data the main challenge is in quickly reading the data out. In most cases, this unstructured data is the anonymous memory pages used by the application. Naively reading out the entire memory contents of an application with multiple GiB of data takes orders of magnitude longer than the rest of the checkpointing process, so it is by default the main overhead in the checkpointing process.

Minimizing the overhead of writing out large unstructured data is possible when checkpointing continuously during the runtime of the process by doing differential checkpointing. With differential checkpointing, on each checkpointing iteration the checkpoint gathers only the application state that has been modified since the last. This technique is mainly used

for data pages, to minimize the amount of data that must be read out of the application on each checkpoint.

### 2.5.2 The Restore Operation

Restoring a process requires deserializing each serialized system resource into a running instance, then fixing any references between the restored instances. Similar to checkpointing, current systems implement restore on top of already existing interfaces - in this case mostly the system call interface. For example, restoring a pipe pair pseudofile in a process requires calling `pipe`, to create an empty pipe, then calling `write` to restore its data. The restore operation then moves the pipe's file descriptors to the proper offset in the file descriptor table using `dup2`, and possibly sends the file descriptors over to other restored processes in the application using `sendmsg` over a UNIX socket.

Restoring an application is portable across machines and reboots: We can deserialize records on different machines than the ones we created them in. For example, we can checkpoint a pipe pair including their data in one machine, and restore it in another. Extending this to application images, we can store application checkpoints to persistent storage or to remote machines, and restore the application on top of a different kernel, e.g. after we reboot the machine, or on a different machine altogether.

The most important issues in restore performance are how quickly we can restore each resource, and how quickly we can connect them together. Restoring a resource may take multiple steps, each of which requires a system call. In the example of the pipe, we need to first create an empty instance before filling it in with data, then moving the file handles to the right descriptor number. Most resources require multiple such steps, therefore adding latency.

Another consideration is connecting the restored resources together, for example for sharing. Complex sharing relations are not easy to replicate through the file system interface and may require specific sequences of operations at restore time. For example, sharing anonymous memory between two restored processes is only possible by first creating the anonymous region with `mmap` in a restored process, then calling `fork` to create the other process and possibly `mremap` to move the mapping to the right virtual address. These sequences of operations prevent parallelizing parts of the restore process and add extra latency.

Restoring from differential checkpoints has the additional consideration of combining differential checkpoint images at restore time. This is necessary because each differential checkpoints only hold the changes between checkpointing iterations, so restoring the

application as it was on the last checkpoint requires combining all the individual differential checkpoints.

This work presents a checkpoint-restore system that provides high performance both in the checkpoint and restore path, and supports both differential checkpoints and restores. This systems checkpoints and restore checkpoints directly in the kernel, and avoids costly processing in two ways. First, it uses an application image format that represents the application as it is in the kernel, not as it is seen from userspace. Second, it directly recreates application state in the kernel at restore time, instead of using system calls to reiterate parts of the execution. This way, our system does both microsecond stop time incremental checkpoints, and do microsecond latency incremental restores. This is only possible because it system avoids most of the processing required by current systems during these operations.

## 2.6 Checkpoint-Restore as an OS Interface

The generality of the main checkpoint-restore operation makes it very useful in solving problems that normally require application introspection. We can broadly divide this class of problems in two. Some problems require data movement in and out of the application, like persistence, fault tolerance, and application migration. Other problems require fine-grained introspection into the application's OS state, like debugging, application deployment, secure rollbacks, and speculative execution/software transactional memory.

Checkpoint-restoring transparently is also important because it lets us apply the aforementioned mechanisms to unmodified applications. Mechanisms like Persistence, fault tolerance, and scalable deployment normally require significant developer effort to implement and require application-specific logic that cannot be reused. Checkpoint-restore lifts this burden from the developer while promising performance close to or sometimes even better than application-level code.

This generality and versatility of checkpoint-restore make it ideal to implement it as an OS-level mechanisms. OS mechanisms must provide simple and intuitive primitives without limiting developers with arbitrary policy, thus following the principle of least astonishment. OS mechanisms must also combine with other mechanisms and support existing applications, no matter their complexity. For these reasons we believe checkpoint-restore should be implemented directly as an OS interface and be tightly integrated with the rest of the system.

For checkpoint-restore mechanism to be usable and general it must support all applications domains with a single interface, however, existing systems do not satisfy this

requirement. Existing checkpoint-restore systems each target one of the application domains of checkpointing, but none of them support all use cases and are thus not general enough to act as an OS interface 2.6.

Most systems are CRIU-based, and their dependence on conventional systems interfaces requires specializing their design for a specific application domain to be efficient. These systems are tailored for a single use case, which also leads to them being opinionated in their design, making performance tradeoffs that heavily penalize certain use cases for CR in favor of others. Such systems have limited functionality when integrated into a generic OS. Other systems like Zap do not consider application many of the application domains we list below, of checkpointing, e.g., persistence and deployment, and as such do not address their specific challenges. More specifically, Zap does not integrate with the storage hierarchy in any way, and does not provide consistency guarantees.

Transparent persistence is especially important in this work because we are using checkpointing to explore SLS design. Existing SLSes do not provide transparent fine-grained persistence to applications. Instead, they either focus on exposing a unified data access interface, or provide coarse-grained persistence with no guarantees. For SLSes to truly hide the difference between memory and disk, and still support applications that rely on persistence, they must provide transparent persistence for application data whether it is in memory or disk. This work builds a checkpointing system that can support this requirement.

## 2.7 Applications of Checkpoint-Restore

2.6

Below we list the different application domains of checkpoint-restore systems.

## 2.8 Persistence and Reliability

Checkpointing for persistence sends checkpoints to persistent storage to allow an application survive system crashes and power failures. The OS normally provides persistence only for data, and for that it requires the application to write the state it wants to persist to files.

With checkpointing, application developers do not write persistence code and can design the application as if it completely in memory. Writing correct persistence code is notoriously error-prone and requires significant developer effort. The reason is that application developers must create and persist consistent images of the application's state. Checkpointing removes the need for such persistence code by gathering the entirety of the application's state transparently to the application, including executable system state.

A critical challenge in application checkpointing for persistence is providing fine granularity. Any state modifications done after the last persisted checkpoint are lost, so checkpoints must be as frequent as possible to minimize data loss. This requires techniques like differential checkpointing to be efficient. However, existing systems do not persist at a fine granularity because the overheads in the checkpointing process prevent it from scaling down to these time scales.

Fine-grained transparent checkpointing combines well with techniques like record-replay [166] and external synchrony [181] to reduce its costs. Transparent persistence requires ensuring that an application only communicates with external entities like clients after persisting its entire state, e.g., to avoid confirming a database store operation that is then lost because of a crash fault. This in turn requires a full application checkpoint after every operation, which significantly impacts performance. Record-replay and external synchrony both help amortize the cost of checkpointing by respectively using a simple event log to persist individual operations, and batching communication to outside entities so that a single checkpoint ensures consistency for multiple clients.

Another problem that checkpoint-restore solves is that of application migration. This is implemented similarly to checkpoint-based persistence, but by flushing the checkpoint to a remote machine instead of sending it to the disk. For migration we can checkpoint the application on one machine, kill it, send the checkpoint over to another machine, then restore it. The result is that the application is now running on a different machine [107].

Application migration requires fast checkpointing and may employ techniques like precopy and postcopy migration [92] along with checkpointing to minimize downtime and reduce migration time. Precopy migration is related to differential checkpointing and reads application pages into an image before the checkpoint, so that during the checkpoint it can minimize application downtime by skipping any pages that have been read out and have not been modified since. However, precopy migration has increased migration times due to having to read pages multiple times. CRIU [17] is an example of a checkpoint system built for this purpose.

Checkpoint-based application fault tolerance is a subset of application migration and requires the same tight performance guarantees as in the persistence use case. Application fault tolerance requires differential checkpointing of the application to minimize downtime, restoring an image on the remote machine in case the original goes down. This approach has been extensively studied for virtual machines [99], where it has also been combined with external synchrony to minimize checkpointing overhead.

Again similarly to application migration we can use checkpoint/restore to deploy applications quickly. Applications normally include runtimes that take significant time to

start up, e.g., the Python runtime, or even require warming up in the case of runtimes for JIT languages, e.g., JavaScript and Java [212, 237]. However, the initialization of these runtimes is deterministic as it happens before the applications receive any input. In this case, checkpointing an initialized application and creating new instances by restoring from an image is faster than creating a new instance with `exec()`. This technique depends on fast restore times and is used in domains like serverless computing [108].

Checkpointing, and differential checkpointing in particular, is helpful in debugging applications by capturing their state as it continues executing. This provides developers with the ability to do *time-travel debugging* [172, 144] where they can and re-execute the application from any checkpoint, effectively rolling it back in time to reiterate the execution. This feature depends on fine checkpointing granularity and compact checkpoints, so that the developer can debug with high fidelity.

Checkpoint-restore also combines with record-replay systems [122, 166]. These systems capture all nondeterministic events in an application's execution into a log, allowing its replication by replaying the events of the log on top of a new instance of the application. These events include user input and benign nondeterminism like races to take a lock between threads. These systems generate significant amounts of data (GiB per second), so the log size and thus total runtime of the application is bound by the available space, but checkpointing can replace parts of the log with checkpoints to trade off visibility into parts of the execution for the ability to run for longer or even indefinitely.

Finally, checkpoint restore provides the ability to implement secure rollbacks and speculative execution as system operations. These operations allow applications to mark a state in their execution for reentry, and go back to it at a later time. This is similar to `setjmp/longjmp`, but for entire applications instead of just CPU state. Secure rollbacks use this ability to wipe away state from an application, e.g., sensitive data provided by a user, so that the application can be reused without risking leaking data [70]. Speculative execution is similar to software transactional memory: It allows applications to take a checkpoint, then continue executing while making an assumption in the code that may be incorrect. If the assumption is incorrect, and the results of the execution are invalid, the application can roll back to before it made the assumption and retry execution, now with the right assumptions in place. This requires both fast in-memory checkpoints, and fast in-memory restores.

Otherworld [103] focuses on persistence through kernel failures at the expense of ignoring other data loss events like power loss. It achieves this by keeping two kernels in memory at a given time, one executing and one designated the *crash* kernel. In case the original kernel crashes, the crash kernel takes over and extracts all application-related data from

the original before letting applications execute.

Similarly, KUP [141] uses checkpoint-restore to implement an OS update mechanism that does not require application restart. KUP uses CRIU to checkpoint and persist applications before replacing the running kernel with an updated one, then restoring the applications on top of it and letting them run. KUP avoids the overheads normally associated with checkpointing by retaining the physical memory and virtual-physical mappings used by applications through the kernel update to avoid checkpointing memory.

## 2.9 Other Related Work

This thesis is on single level stores and checkpoint-restore, but uses a variety of systems concepts and techniques. We list those below to provide better context into this work.

### Single Address Space OSes

Single address space OSes SASOSes are OSes that explore the idea of storing all information in the system in a single address space. They are similar, but not identical, to single level stores in that they explore removing the concept of a process from an OS altogether, and concern themselves with taking advantage of 64-bit address spaces as a way to organize and address data. In contrast, single level stores do not focus as much on sharing virtual memory between processes, but mostly on blurring the distinction between volatile memory and persistent disk. The problems that single level stores are traditionally associated with are related to unifying memory and storage for overcommit and to some extent persistence.

The advent of large 64-bit virtual address spaces brought with it research into using it as a way to organize data and build system that exclusively use memory semantics for data processing. SASOSes aimed at simplifying programming for the user by keeping a single representation of data across memory and the disk. These systems focused on novel programming models and did not keep compatibility with widely used Unix semantics.

Grasshopper [102] is an operating system that explores persistence as a first class OS primitive for a SASOS. Grasshopper organizes all data in the system into objects, and attempts to solve the problem of crash consistency by essentially only updating on-disk data in a manner reminiscent of transactions. To this end, it uses a programming model wherein all data is in memory regions called containers, and are acted on by agents (reminiscent of conventional system processes) called loci while their access is gated off by software capabilities. This programming model is more intuitive in terms of its persistence guarantees, but is incompatible with widely used Unix-like systems and requires a ground-up redesign of the OS.

The Sombrero SASOS explores OS-hardware co-design to achieve "protection without isolation" [216]. Sombrero introduces abstractions called "protection domains" that allow processes to safely and in an organized fashion share data without traditional virtual address spaces. To this end, Sombrero builds a system reminiscent of capabilities that must be enforced by specialized hardware to be efficient.

Mungi [125] explores kernel design for distributed SASOSes. Mungi organizes data into non-expandable memory objects, and uses password capabilities to control access to them from multiple users. Mungi is also distributed and supports sharing data between multiple nodes on a network. However, while Mungi explores the organization of data, including indexing and mediating access, and provides SLS semantics through uniform access to volatile and persistent data, it does not actually provide persistence, or address any of the challenges of checkpoint restore.

The Angel operating system [240, 239] is an evolution of an earlier microkernel OS that uses a SASOS design to avoid the overheads of message passing. The goal of Angel OS is being distributed and supporting distributed shared memory while still which it still retaining SASOS semantics. Its main goal is evaluating message-passing vs DSM for microkernel designs.

## Virtual Machines

Virtual machine checkpointing is a very well explored technique that is similar in principle to application checkpointing but has different challenges and advantages. In virtual machine checkpointing, workloads run in a virtual machine that apart from the application includes an entire userspace, kernel, and even virtual hardware [144, 229, 82, 119]. All these components' state are in a process in the virtual machine's host and represented as data. The process in the host itself is simple, mostly uses anonymous memory and does not have complex OS state. This makes checkpointing virtual machines simpler, as a lot of the complexity of extracting application state out of the system do not apply - in this case we are treating the entire system as an indivisible unit.

The downside is that by gathering all state in the VM and not just that of the application we must increasing the total amount of state we have to gather. This is a significant amount, e.g., the guest kernel's memory, and makes virtual machines checkpointing inherently more heavyweight than application checkpointing (Chapter 5). Restoring a virtual machine is also fundamentally slower than restoring an application, because we need to create new virtual hardware and retrieve a large image.

Remus [99] is a transparent virtual machine replication system for high availability. Remus places the workload into a virtual machine and combines virtual machine checkpoint-

ing with *external synchrony* to minimize the stop time for the application, using external synchrony to parallelize virtual machine execution with checkpointing. This allows it to trade off latency towards the client for lower checkpointing overheads. RemusDB [171] expands on this work by relaxing the external synchrony limitation through integration of the client to the asynchronously replicated server. In RemusDB the client distinguishes between restartable and non-restartable operations. Restartable operations do not require external synchrony from the server,

# Chapter 3

# The Aurora Single Level Store

## 3.1 Introduction

Single level storage (SLS) systems provide application persistence as an operating system service. Their advantage comes from eliminating the semantic gap between the in-memory and the serialized on-disk representations to decrease application code complexity and reduce software bugs [193, 44]. Applications written for an SLS hold data solely in memory, and the operating system persists the entire application. Developers write programs as if they never crash and do not write code for persistence and recovery. After a crash, the SLS restores the application from disk, including all execution state (i.e., CPU registers, OS state, and memory), and resumes execution oblivious to the interruption.

At a high level, a single level store accurately captures all application state and stores it on disk with low overhead. The capturing frequency determines the maximum amount of application work lost on a failure. Furthermore, the system's storage bandwidth limits the frequency.

SLSes were impractical for decades because of performance reasons, but this has changed with the advent of new hardware. Past systems suffered because memory bandwidth and latency were orders of magnitude faster than disks. Write amplification from the page granularity memory tracking of SLS systems amplifies the storage bandwidth required. Modern flash coupled with fast PCIe Gen 4–5 are closing the performance gap with memory.

Another problem with existing and prior single level stores is their incompatibility with the ubiquitous POSIX API. IBM's i Series mainframes, the successor to the AS/400, provide a custom operating system, runtime and compiler that work in concert to provide

persistence [217]. The IBM compilers semi-automatically insert API calls into the application for persistence. Research systems like EROS also worked by providing a custom API [209].

We introduce the Aurora Operating System, a novel single level store that enables the persistence and manipulation of execution state. Aurora is based on the FreeBSD kernel and is the first SLS to run POSIX applications. Aurora supports all POSIX abstractions and complex multi-process applications like the Firefox web browser.

Aurora has three main challenges: First, POSIX state is inherently difficult to capture as state spans both userspace and the kernel, and is not always associated with a process. Second, saving application state frequently requires incremental tracking to reduce system overhead. Third, the bulk of the state is memory that is tracked using the memory management unit (MMU), adding runtime overhead.

Aurora solves these problems using two techniques. First, it treats POSIX objects (e.g. UNIX domain sockets, System V shared memory, and open files) as first class objects. This allows Aurora to handle applications that share memory or files between processes, without duplicating work or leaving edge cases unhandled. Second, we introduce system shadowing to efficiently track system-wide updates to memory.

Users can persist, copy, revert, or transfer running applications the same way they would a file. Aurora creates application checkpoints by default every 10 ms that encapsulate all information required to recreate the application, even across reboots and machines. The 10 ms target is selected to provide modest overhead for transparently persisted applications. Applications using the Aurora API achieve faster checkpoint frequencies with low overhead.

The persistence and manipulation of execution state enables a broad range of applications. For example, Aurora replaces database storage engines, which reduces application complexity and overhead. The persistence API enables databases to use the hardware MMU to track changes over typical software tracking techniques. Capturing applications post initialization accelerates warm starts for serverless computing, just-in-time compilation, and other applications with costly initializations. Keeping execution history enables time travel debugging and record/replay. Transferring applications between machines enables transparent migration and fault tolerance.

Aurora makes the following contributions:

- We develop an architecture that supports both unmodified and customized POSIX applications. Unmodified applications gain transparent persistence at regular intervals (by default 10 ms). Customized applications can achieve microsecond-level persistence.

- We expand the concept of a single level store with new primitives for the manipulation of execution state to enable new mechanisms. The ability to provide fast incremental checkpoints and restores has applications to systems from debugging to serverless computing.
- We introduce two key ideas: The *POSIX object model* that captures OS state with minimal effort and performance overhead, and the *system shadowing* technique that enables low overhead tracking of memory.
- Finally, we observe that new flash devices, an abundance of fast PCIe lanes, and large virtual address spaces make single level stores practical. We offer the Aurora implementation and evaluation as proof of this.

Our evaluation shows that transparent persistence is practical. Aurora transparently adds persistence to Memcached, a popular key-value store, with overheads of 9%–82% depending on the persistence granularity compared to a baseline with no persistence. We achieve much lower overeads using the Aurora API. Modifying RocksDB to use the Aurora API replaced 81k lines of code with 109 lines, while improving throughput by 75%.

## 3.2 Background

Single level stores abstract away the distinction between persistent and volatile storage to provide transparent persistence, simplifying application programming by removing the need for explicitly serializing and moving data between devices. There have been several approaches to single level stores with different designs and trade-offs.

IBM's i Series mainframes, the successor to the AS/400, provide a custom operating system, runtime and compiler that work in concert to provide persistence [217]. The IBM compilers semi-automatically insert API calls into the application for persistence. This approach does not provide persistence to arbitrary POSIX applications and only supports some programming languages.

EROS [209] and KeyKOS [152] provide persistence through system-wide checkpointing. These systems periodically checkpoint all system state and store it in persistent storage. After a crash the operating system restores the applications to resume as if no failure had occurred. EROS provides system checkpoints as frequent as a buffer cache flush in traditional OSes, which typically occur every 30 seconds.

**The Millisecond Barrier** Persistence at second timescales may be practical for end users of desktop systems, but not for latency sensitive server applications. Server applications require persisting state to disk before responding to outside clients, which requires latency of milliseconds or lower.

Single level stores can use *external synchrony* to transparently withhold external communications until data is safely persisted on disk [181]. Previous systems that employ external synchrony demonstrate modest overheads when persisting or replicating data at millisecond granularity [99, 171, 14].

EROS and other single level stores do not enforce external synchrony because they flush data infrequently. Withholding communications for seconds would cause prohibitive overheads and even break high availability services [22].

**Hardware Trends Necessitate Revisiting the SLS** Hardware has evolved tremendously since the EROS single level store was created. EROS's storage layer focuses on using spinning disks effectively to achieve persistence in timescales of tens of seconds. Spinning disks are orders of magnitude slower than memory. The storage bandwidth constrains the amount of data the single level store can produce, limiting the frequency of persistence.

Three major hardware trends are making single level stores a practical approach to application persistence. First, NVMe flash devices over PCIe increase bandwidth and decrease latency. Second, modern processors have an aggregate PCIe bandwidth that rivals memory. Third, 5-level page tables allow applications to hold up to 128 PiB of data in an address space.

Fast checkpoint frequencies are possible because of the low performance gap between disk and memory devices, the lowest in 50 years. The aggregate IO bandwidth available can be larger than the memory bandwidth. For example, the 3rd Generation AMD EPYC CPUs have a PCIe bandwidth of $256\,\mathrm{GB\,s^{-1}}$ and a memory bandwidth of $205\,\mathrm{GB\,s^{-1}}$ [65].

**Application Checkpointing** Application checkpointing is an extensively studied problem [135, 150, 104, 63, 233, 18, 123, 222, 195]. Checkpointing serializes the running application state including the CPU state, OS state and memory into an image. The image enables recreating the running application at a later point in time or on another machine. Recreating an application is useful for debugging, surviving hardware crashes, fault tolerance and other uses.

Checkpointing is difficult because the state we need to capture is inherently complex in existing operating systems. The application memory is the majority of state we need to save in terms of size and is relatively simple to collect. The remaining state is scattered across objects throughout the userspace and the kernel.

The operating system state is small but complex and contains both userspace and kernel components to each object. For example, a `FILE` object in C contains userspace state of the file object and a POSIX file descriptor, which is an integer that is passed to system calls. The user visible kernel state includes the file offset, the mode that file was opened in, and the file type. Some additional state is available through the process file system.

An even larger amount of kernel state is inaccessible from userspace such as the filename, what processes are sharing the file descriptor, and what processes are sharing the underlying file system vnode. Applications share file descriptors, including the offset, using `fork` and UNIX domain sockets. Applications share the vnode, but not the file descriptor, by each calling `open` on the file.

Internal kernel state does not have a builtin API to query internal fields, because it contains implementation and version specific data. It also contains transient state that is invalid across machine reboots.

VM checkpointing is relatively straightforward and frequently used in commercial systems [178, 93]. Virtual machines avoid many of the OS checkpointing problems by carefully designing the virtual hardware interface between the guest operating system and the hypervisor. VM checkpointing incurs additional overhead because of saving the guest operating system in addition to the application.

Application checkpointing images are smaller than VM images but are significantly more difficult to create. The state of the art application checkpointing system is CRIU [61] that is used for container migration [191]. CRIU extends the system call interface and process file system to capture and recreate application state from userspace. CRIU's design leads to unnecessary complexity because it has to query individual state objects starting from a process or container, find sharing relationships between them, and then deduplicate state.

Similar to prior systems, CRIU and its APIs are process-centric. The checkpointing system collects and traverses the state, and then infers any sharing relationships between processes like shared memory. This extra work increases CRIU's code complexity and adds performance overhead. CRIU's compatibility with applications comes with substantial complexity; it contains over 100k SLOC, kernel patches notwithstanding. CRIU's complex design makes it difficult to expand, e.g., the developers only added support for UNIX domain sockets seven years after its initial release [34].

Table 3.1 shows a breakdown of the cost of checkpointing a Redis instance with a 500 MB working set. CRIU gathers OS state in 50 ms and copies checkpoint state in another 413 ms. The system suspends the application for the duration of the operation, leading to large *stop times* that prevent deploying external synchrony techniques. Writing the data to disk takes another 350 ms without ensuring persistence.

| Type | CRIU |
|---|---|
| OS State Copy | 49 ms |
| Memory Copy | 413 ms |
| Total Stop Time | 462 ms |
| IO Write | 350 ms |

Table 3.1: A breakdown of CRIU's checkpointing overheads for a 500 MB Redis process.

**Incremental Checkpointing** Table 3.1 illustrates typical overheads of checkpointing systems. The time spent copying application memory is the largest contributing factor to system performance. One optimization is to use incremental checkpointing, and another is to save the application data concurrently with application execution.

*Incremental checkpointing* creates checkpoints that contain only the state modified since the previous checkpoint. Most of the memory is not modified between successive checkpoints reducing the amount of state that needs saving. Memory changes are tracked using the per-page dirty bit in the MMU.

*Continuous checkpointing* optimizes incremental checkpointing further by concurrently dumping state with application execution to minimize stop time. Continuous checkpointing creates an atomic snapshot of memory and tracks changes using a copy-on-write (COW) mechanism. COW allows for applications to concurrently run with the flushing of state as COW memory cannot be modified.

There are a number of prior incremental checkpointing systems [231, 233, 172]. Many incremental checkpointing systems are used to provide transparent fault tolerance [99, 14] and improve record/replay for virtual machines [144, 7]. VAS-CRIU [231] is a research system that extends CRIU to use detachable address spaces in the kernel to implement copy-on-write tracking of memory. It does not handle complex applications that use shared memory between processes.

**The Way of the Single Level Store** We achieve better results by approaching checkpointing from the perspective of a single level store. Aurora provides persistence throughout the operating system allowing us to expose the system state the way that the OS perceives it. Aurora persists each POSIX object without duplicate work or additional processing to reconcile sharing.

Instead we make every POSIX object (e.g., a file descriptor, VM object or process) persistent in a separate on-disk object. Objects can be copied or shared between processes

to represent any relationship possible in POSIX. We call this implied relationship the *POSIX object model*, which leads to a modular and faster checkpointing system.

Existing checkpointing systems are process-centric, which has discouraged them from viewing the persistence of OS state this way. Replicating the POSIX object model in storage allows Aurora to avoid the cost of inferring sharing, as in existing checkpointing systems.

Furthermore, Aurora develops a new technique called *system shadowing* that allows concurrent memory flushing across processes while the application continues to run. We reduce the state copied while applications are quiesced to achieve millisecond timescales for persistence. Applications that use our API can further reduce the overhead to achieve microsecond timescales for persistence.

## 3.3 Using Aurora

Aurora checkpoints at the granularity of a *consistency group* – a group of processes that Aurora persists atomically. Typically a consistency group will encompass a single application or container. Consistency groups use external synchrony when communicating outside the group by buffering outgoing communications until the checkpoint is persisted. No external synchrony is necessary for processes within the same consistency group.

Aurora runs unmodified applications transparently, and provides an API to enable new functionality and persist custom applications more efficiently. Aurora persists unmodified applications by periodically checkpointing their consistency group. Users set the checkpointing frequency of each consistency group. Aurora's default frequency is 100× per second, but users can persist applications at much lower rates.

Aurora supports ephemeral processes that are part of a consistency group but are not persistent. For example, this is used for worker processes that the application can easily recreate after a crash. Aurora notifies the parent process by sending a `SIGCHLD` after a restore. To the parent, it looks as if the child exited unexpectedly.

**Command Line Interface** To better illustrate Aurora, we show a subset of the command line interface in Table 3.2. Users persist an application using the `sls attach` command to attach it to a consistency group. By default Aurora persists the application 100× per second, with external synchrony [181]. The execution history of an application is stored on-disk given adequate space. Users manually create named checkpoints with the `sls checkpoint` command, and view all application checkpoints in Aurora using `sls ps`. Users restore a previous checkpoint or resume execution after a system crash using `sls restore`.

| Command | Description |
|---|---|
| `sls attach` | Attach to a running application |
| `sls detach` | Makes a process ephemeral |
| `sls checkpoint` | Checkpoint an application |
| `sls restore` | Restore an application |
| `sls ps` | List applications in Aurora |
| `sls suspend` | Suspend application into persistent store |
| `sls resume` | Resume a suspended application |
| `sls dump` | Generate an ELF coredump |
| `sls send` | Send an application to a remote |
| `sls recv` | Receive an application from a remote |

Table 3.2: A subset of the command line interface.

Users pause and resume applications using `sls suspend` and `sls resume`. For debuggability, any checkpoint or running state can be extracted as an ELF coredump with the `sls dump` command.

Users share or migrate applications using the `sls send` and `sls recv` commands to serialize a checkpoint state or continually feed incremental checkpoints to a remote host. Flags to these commands allow the user to pipe a single checkpoint to a file, live migrate the application, or provide high availability.

**Aurora API** Table 3.3 shows the Aurora API that custom applications use to control and optimize persistence. Applications can manually initiate checkpoints, restore, or roll back checkpoint state.

Developers optimize application checkpoints by selectively excluding memory regions through `sls_mctl` and file descriptors through `sls_fdctl`. Developers make fine-grained checkpoints using `sls_memckpt` to atomically checkpoint a single memory region or `sls_journal` to synchronously update an on-disk journal.

A common pattern is to take a full checkpoint and continuously persist the memory regions that contain data. Aurora integrates partial checkpoints and journaled regions into

| Function | Description |
|---|---|
| `sls_checkpoint()` | Create a checkpoint |
| `sls_restore()` | Restore a checkpoint |
| `sls_memckpt()` | Asynchronous checkpoint of mapped region |
| `sls_journal()` | Non-temporal flush (outside checkpoint) |
| `sls_barrier()` | Wait for a checkpoint to be flushed |
| `sls_mctl()` | Include/exclude memory regions |
| `sls_fdctl()` | Control external synchrony |

Table 3.3: A subset of the Aurora application API.

the full checkpoint. After a restore, applications fix up runtime state inside of an Aurora specific signal handler.

Another way to improve performance is to selectively disable external synchrony with `sls_fdctl`. Disabling external synchrony reduces latency for outgoing communications. For example, application connections that perform read only operations (e.g., an authentication request) do not require external synchrony.

Developers wishing to suspend external synchrony for application connections that mutate state need to use the persistence APIs. For example, a database can disable external synchrony to its clients and use `sls_journal` to persist operations before sending acknowledgements.

## 3.4 The Aurora Operating System

The main goals of Aurora are correctness and performance. Aurora must correctly checkpoint the state of the application such that we can resume an execution indistinguishable from the original. Aurora must minimize the performance impact and resource overhead.

Aurora's checkpoints must be runnable after a reboot or on another machine. The system's state, e.g., the running kernel, may be different at restore time than at checkpoint time. The checkpoint contains enough of the application's userspace and kernel state to reproduce the executable state. While userspace state is self contained, Aurora extracts kernel state like file descriptors from the running kernel.

Continuous checkpointing must have low overhead to be practical. Overheads include the time to create an application checkpoint and persist it in storage. Creating a checkpoint requires stopping the application, which adds overhead. How quickly Aurora flushes checkpoints bounds checkpoint frequency. Creating and writing out the checkpoint consumes CPU, memory and IO bandwidth.

Aurora applies COW semantics in memory and on-disk to minimize application stop times and sustain high checkpointing frequencies. COW tracking of memory enables writing checkpoints to storage concurrently with application execution. On-disk COW minimizes checkpoint size while allowing constant time restores at any point in the application's execution history.

### 3.4.1 Aurora's Architecture

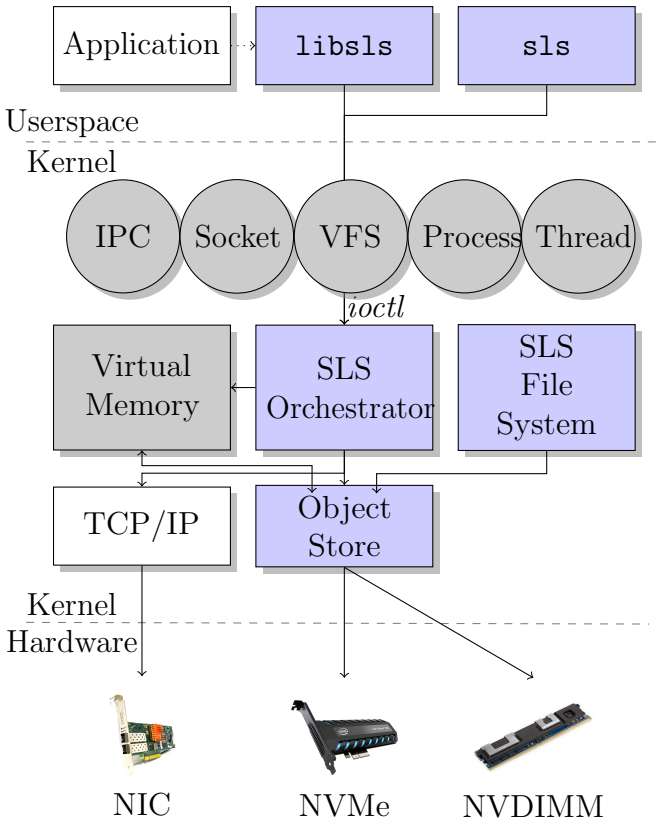



Figure 3.1: Aurora system diagram.

Figure 5.1 shows the architecture of the Aurora persistent operating system. Aurora has three major components: the SLS orchestrator, the object store, and a custom file system. Each POSIX object in the operating system (e.g., open files or SysV shared memory) contains code that continuously serializes and stores the state in the object store.

The SLS orchestrator maps kernel objects to the object store and manages checkpointing and resuming processes as described in § 3.5. The orchestrator provides serialization barriers across the entire OS to provide consistent checkpoints. It uses these barriers to momentarily pause processes and copy their state.

The orchestrator copies smaller POSIX objects, e.g., file descriptors, synchronously into memory buffers and larger objects, e.g., process memory, are tracked using copy-on-write (COW). The state will be written to storage asynchronously, while the application continues to execute. This process is repeated 100× per second with modest overhead. The orchestrator also manages restores by recreating all POSIX objects and resuming the application.

Aurora uses a novel technique called *system shadowing* (see § 3.6) to track system-wide memory changes for correctness and performance. Unlike COW techniques used by `fork`, system shadowing tracks shared memory regions. System shadowing works seamlessly with the virtual memory subsystem to accurately reproduce shared and user COW regions. System shadowing enables asynchronously writing the pages to storage, while processes continue to run.

Aurora applies *external synchrony*, which buffers external communications until all computation prior to sending the message is persistent. All applications within a single consistency group are checkpointed atomically.

The object store (see § 3.7) provides persistence for the objects that comprise each checkpoint. Each object is either a POSIX object, memory object, or file in our system. The object store uses a copy-on-write design, rather than a log structure, to provide low latency checkpoints. Aurora subsumes swap and integrates it with the object store to insert swapped pages into the next checkpoint.

The Aurora file system is a namespace into the single level store that solves problems that arise when using Aurora with conventional file systems. For example, POSIX file systems reclaim anonymous files (i.e., unlinked but open files) after a system crash. This reclamation prevents restoring applications that were using this file, so the Aurora file systems includes open file references in a hidden link count.

## 3.5 Making POSIX Persistent

Aurora approaches checkpointing as a single level store, which differentiates it from existing checkpointing systems. Aurora individually persists each POSIX object in the system and groups these objects into an application checkpoint. This new architecture allows Aurora to achieve millisecond level checkpointing frequencies with low overhead.

### 3.5.1 Basic Persistence

To persist processes we have to deal with five categories of state: process state, thread state, CPU state, memory regions, and file descriptors. Nearly every POSIX object is a file descriptor, a memory region, or both in the case of memory mapped files. Interposing on every memory access of a memory region is intractable so we use the MMU to track changes (see § 3.6).

**Quiescing Processes** Aurora must efficiently *quiesce* the system before checkpointing an application. Quiescing means stopping the application to prevent the modification of its state while checkpointing. Failing to quiesce an application opens the possibility of creating an inconsistent checkpoint that is not a valid running state.

At any point the application will be in one of three places: userspace, the kernel, or the boundary between the two. When in userspace it is impossible to serialize application state, as it is constantly changing. When in the kernel, userspace state is constant but the underlying kernel structures (e.g., files) are changing.

Aurora's first implementation used the `SIGSTOP` signal to quiesce the application, but this is an incomplete and nontransparent solution. It is incomplete because kernel operations in progress, e.g., `read` system calls, can modify application state. It is not transparent because `SIGSTOP` is visible to the application, and some system calls return `EINTR` after continuing. In addition, POSIX signals are delivered asynchronously increasing the latency of quiescing.

To avoid these problems we instead quiesce at the kernel boundary by extending the mechanism used by `fork` and `exec`. Aurora sends interprocessor interrupts (IPIs) to all cores running the application, forcing them to the boundary.

System calls that do not sleep have very low execution times, so waiting for these calls to complete does not noticeably increase stop times. Aurora interrupts and restarts system calls that sleep, forcing all threads in the kernel to the boundary. Aurora transparently restarts interrupted system calls that would return `EINTR` by rewinding the thread's userspace

program counter to right before the `syscall` instruction. The thread will reissue the call immediately after it starts executing.

**Process, Thread, and CPU State** Most POSIX objects are handled by serializing important fields of each kernel structure and reconstructing them after a crash. Aurora must also recreate the process tree, processes and threads. Apart from parent/child relationships, Aurora must also recreate the process groups and sessions that were present at checkpoint time. These groupings are used for job control, signals, and sandboxing mechanisms.

Threads have signal masks, pending signals, scheduling priorities and other state that must be recreated. Aurora copies the CPU state of each thread by copying the registers off the kernel stack and FPU/vector registers from the process structure. For processors that lazily save and restore FPU and vector state, this may require an IPI to flush the state into the process structure.

**File Descriptors** File descriptors are the most complex, and largest source of state after memory. Most POSIX objects are accessed using file descriptors including: files, sockets, interprocess communication (IPC) and devices.

File descriptors are challenging because they form complex relationships between kernel objects that are difficult to capture from userspace. While not explicit in the POSIX standard, the API implies a design that is similar across all POSIX implementations that we've examined. Objects are shared in ways that cannot be captured from userspace without exposing kernel state to it.

We use an example of a file descriptor to see how common operations lead to complex object hierarchies in the kernel. These hierarchies dictate how the resources are shared between processes, and influence the semantics of their operations. The processes themselves cannot inspect these hierarchies, so they do not know what they are sharing.

Suppose a process opens a file, then calls `fork`. The resulting processes each have a reference to the same file descriptor, so any modifications to the file descriptor by one process is visible to the other. For example, if one process reads from the file it changes the offset of the descriptor, and any reads from the other process will use the new offset.

Suppose a third process opens the same file. The new process has its own file descriptor in the kernel, with both descriptors being backed by the same vnode. The file itself is shared between all three processes, but the file offset of the third process is independent.

Aurora accurately captures sharing semantics by treating objects in the kernel, e.g., file descriptors and vnodes, as first class objects. Aurora avoids overhead when gathering the state by directly inspecting these objects in the kernel.

### 3.5.2 Persistence Challenges

Aurora provides several novel solutions to encapsulating application state. By approaching the problem from the perspective of a single level store we revisit the traditional design of checkpointing systems to achieve correctness and performance. We argue that the failure of checkpointing systems to support complex applications is due to prior attempts focusing on a process centric view of persistence.

**POSIX Object Model** Our first contribution is to reframe persistence around the *POSIX object model*. Aurora stores all POSIX objects *and internal kernel objects implied by POSIX* as first class objects in our persistent store. This greatly simplifies checkpointing correctness and improves performance over conventional systems by eliminating a lot of the code needed to handle the complexities of sharing.

During checkpointing Aurora persists each POSIX object separately into a corresponding on-disk object in Aurora's object store. For each incremental checkpoint Aurora maintains a mapping of each object's address in the kernel to a 64-bit on-disk object identifier. This structure allows Aurora to scan over all persistent objects and serialize each of them to storage exactly once. The orchestrator's job is to provide atomicity across objects belonging to a consistency group.

Going back to the file descriptor example, Aurora effortlessly distinguishes between the two cases using the POSIX object model. Aurora backs each file descriptor and vnode with its own object in storage, and gathers all state independently. Aurora restores the objects separately and links them back up to recreate sharing.

Aurora's approach contrasts with process centric views that treat file descriptors and vnodes as a single entity and fail to capture sharing relationships. Alternative approaches that infer everything from userspace or do not account for internal kernel objects result in excessive implementation complexity or unhandled corner cases.

These approaches either scan all processes or introduce backmaps to find what processes have a file descriptor open. These approaches require additional processing during checkpointing or invasive changes to introduce backmaps (n.b., System V objects make this difficult). File descriptors are difficult to track because they may be shared through `fork`, duplicated in a single process using `dup` or sent to another process through a UNIX domain socket.

**File System** The Aurora file system enables Aurora to persist file system state along with the application itself. The file system is a namespace into the single level store. Memory

mapped regions and files are treated identically in the object store unifying memory mapped files. The file system also introduces two optimizations to accelerate checkpointing.

This architecture solves edge cases that arise in conventional file systems where ephemeral state necessary for restoring is lost. One example is anonymous files (i.e., an open but unlinked file) that are widely used by software to store temporary state. Anonymous files are unlinked from the file system but are still held open by processesk. These files have no name and are reclaimed when the process exits or the machine reboots, so any information they contain is lost.

Aurora avoids the problem of unlinked files by reference counting all objects separately from the file system link counts that are used for reclamation. Individual files are state objects like any other part of an application and are referenced by Aurora's identifiers – even when the file has no path in the directory structure. Files persist while they are referenced by another on-disk object.

The Aurora file system optimizes checkpointing vnodes by referencing the inode number instead of the file path. Using the inode number avoids costly lookups in the VFS name cache and `namei` calls during the checkpoint stop time.

Aurora uses *checkpoint consistency* to optimize file system operations. The file system ignores *fsync* operations and provides consistency at checkpoint time. Aurora relies on external synchrony and the use of the APIs for correctness.

### 3.5.3 Completing Persistence

**Network Sockets** Aurora supports UDP sockets, UNIX domain sockets, and TCP sockets. For UDP sockets Aurora checkpoints the socket's address, port, options and socket buffer. UNIX domain sockets are treated the same way, but Aurora also parses the socket buffer to deal with control messages that contain credentials or file descriptors. Aurora scans the buffer for in flight control messages and checkpoints each type of supported descriptor.

Aurora checkpoints TCP listening sockets, but omits the accept queue. This omission looks to clients as if the server did not call `accept` or dropped the SYN packet. Clients will retry the SYN packet or reestablish the connection. For established connections Aurora saves the 5-tuple (source IP/port, destination IP/port and protocol), the TCP sequence number, socket options, and socket buffers.

**System Wide Identifiers** Aurora restores process identifiers (PID) and thread identifiers (TID). PIDs are used to route signals to processes, e.g., from a parent to a child. Not

restoring the PID would lead to a failure to deliver the signal. TIDs are used by the PThread library for synchronization (e.g., mutexes).

Aurora solves ID conflicts by virtualizing ID allocation. Each process/thread has two IDs: a local and a global. The local ID is the one seen at checkpoint time, while the global ID is the one Aurora allocates at restore time and is visible to the rest of the system. This allows applications to have access to checkpoint time IDs without conflicts with already running applications.

**Asynchronous IO** Aurora tracks all asynchronous IOs (AIOs) in flight and quiesces them for checkpointing. AIOs are issued by kernel threads or by the storage device itself, depending on driver support. Aurora does not record the AIOs for file system writes but rather delays marking the checkpoint as complete until they are incorporated into it. Failed AIOs require updating the checkpoint with the failure status. Aurora tracks reads and includes them in the checkpoint so that they are reissued during restore.

**Device Files** Aurora supports several devices that are mapped into process memory. The most common ones are hardware timers and clocks used to providing low latency time keeping. On x86-64 the High Precision Event Timer (HPET) is mapped read-only into the application address space. We have a whitelist of special devices that are supported by persistent processes.

Another special case is the virtual Dynamic Shared Object (vDSO) that provides optimized platform specific implementations of some system calls. On restore we inject the current platform's vDSO into the application address space. Allowing the application to resume even when hardware or software changes have altered operating system optimizations.

## 3.6 Making Persistence Fast

Aurora introduces *system shadowing*, a novel mechanism that improves continuous checkpoint performance in two ways. First, system shadowing enables incremental checkpointing by tracking the set of pages dirtied between two successive checkpoints. Second, it uses COW to write memory to storage concurrently with application execution, while retaining shared memory semantics between processes.

**The Mach VM System** To better understand system shadowing we briefly recap the FreeBSD virtual memory (VM) subsystem, which is derived from the Mach VM [199]. There is nothing preventing the adoption of system shadowing to other platforms but we will use the Mach terminology throughout the paper. Most OSes including Windows and UNIX derivatives use a Mach-like design.

Figure 3.2 shows the structure of address spaces in FreeBSD. The address spaces have two components: the *VM map* and the *physical map*. The physical map holds the hardware page tables and is a cache for the VM map. The page tables are ephemeral and recreated from the VM map as necessary.

The VM map is a list of entries mapped in a process address space. Each *VM entry* is a memory region that holds a virtual address range, permissions, and `madvise` hints. Each entry is backed by a single *VM object*.

VM objects are collections of pages that back a *VM entry*. VM objects represent different kinds of memory, i.e., anonymous, vnode, or device memory. Objects have no knowledge of permissions or virtual addresses, allowing them to be mapped in different VM maps to implement shared memory. The object is backed by physical memory lazily, i.e., only pages in use by process are populated.

FreeBSD implements copy-on-write (COW) using *object shadowing*. The VM system uses object shadowing to track which pages in a COW region have a process private copy, and which are shared between processes. With shadowing a parent object backs one *object shadow* for each process sharing the region. The pages of each shadow are private to a process, while those of the parent are shared. On a page fault the handler first looks into the shadow. It only searches the parent if the shadow has no page at the virtual offset. Pages from parent objects are always mapped read only. On a write fault, the system will create a private copy of the page in the shadow object.

The VM subsystem reverses object shadowing with *object collapsing*. Shadow objects become unnecessary when the original objects they shadow are no longer shared across processes (i.e., reference count of one). For example, if a process with a single child exits, all of the pages in the initial object are only accessible from the child. The collapse operation merges the pages of the shadow and the parent, keeping the shadow's version of the page if present.

**Checkpointing the VM** Aurora persists the entire VM object hierarchy instead of a flat view of memory. Maintaining VM objects allows Aurora to minimize the number of pages to be flushed to disk. For example, a copy-on-write mapping shared by a process and its parent results in individually storing the two COW objects and the read-only backing object. For memory mapped vnodes using the `MAP_PRIVATE` flag the object only stores the private changes.

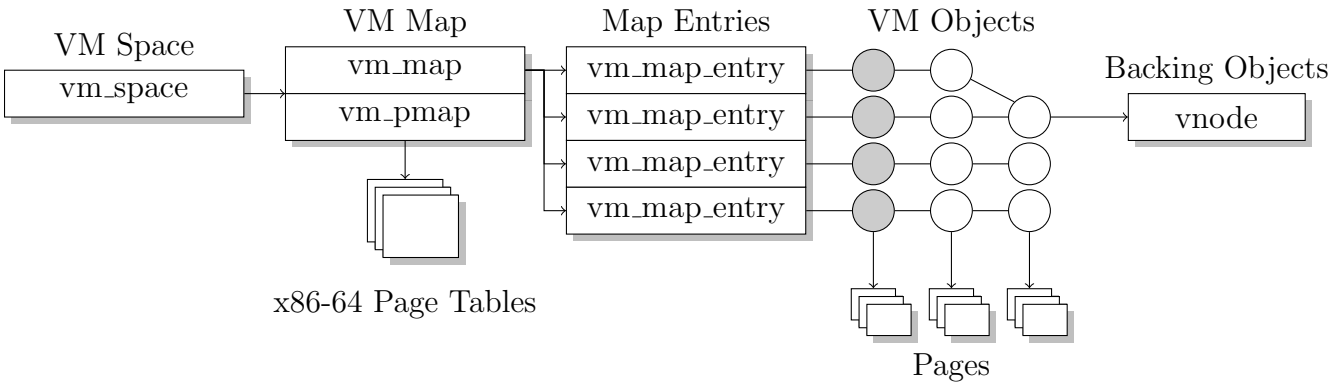



Figure 3.2: VM Memory Diagram of FreeBSD. Address spaces include page tables and a list of mapped regions. Each region is backed by mappable groups of physical pages called VM objects. VM objects represent anonymous or file memory and back each other to enforce COW semantics.

**System Shadowing** *System shadowing* is a novel mechanism that shadows memory across all applications within a consistency group. Checkpointing creates a system shadow to track incremental changes with minimal overhead.

The COW mechanism used by `fork` is insufficient for several reasons. First, `fork` applies COW semantics at the process level, while Aurora requires atomicity across a consistency group. Second, `fork` cannot shadow shared memory regions without breaking sharing. Third, `fork` does not apply to IPC VM objects.

The `fork` COW mechanism is limited as it operates on a single process and ignores shared memory. Fork assumes mappings are shared across the system or are private to a single process – there is no in between.

System shadowing works by creating one shadow for each writeable object across all address spaces in a consistency group. For shared memory regions the shadow replaces all objects. For memory mapped files the Aurora file system handles COW semantics, so there is no need to shadow the VM objects that back the files. For POSIX or SysV shared memory descriptors we introduce a backmap to update the reference in the descriptor, ensuring further mappings will use the latest shadow. System shadowing is transparent to processes and works with `fork` without any conflict.

The frequency of system shadowing would result in long chains of shadow objects if we did not collapse system shadows. Chains of shadows cause memory overhead because each object can have a private version of a page. Chains also add performance overhead because

in the worst case a page fault may traverse the entire chain.

Aurora eagerly collapses system shadows to limit the chain length to two. The first shadow is the incremental checkpoint Aurora is currently flushing to storage. The second shadow tracks changes for the next checkpoint. Once the flushing of the first checkpoint is complete, Aurora immediately collapses the shadow into the parent on the next triggered checkpoint.

The original collapse operation inserts the parent's pages into the shadow object. The original operation scales linearly with the number of unmodified or read only pages. System shadows exist for such short periods that there are few pages in each shadow object, resulting in a slower collapse operation than if the direction were reversed.

Slow collapse operations also hurt runtime performance as applications frequently fault in pages because system shadowing flushes the translation lookaside buffer (TLB). Lock contention between page faults and the collapse operation, which require locking VM objects, further increases the overhead of repopulating the MMU.

Aurora optimizes the collapse operation by reversing its direction from the shadow to the parent. Aurora moves the shadow's pages into the parent to reduce the number of pages moved between VM objects. This reduces the average cost of the collapse operation for Aurora.

**Memory Overcommitment** Applications that use the Aurora API place all data in memory. Developers do not have to deal with migrating data between memory and storage, either for persistence or memory overcommitment. Memory overcommitment evicts pages to storage when the system is under memory pressure. Aurora cannot use a conventional swap partition because the metadata is kept in memory, and lost after a crash or reboot.

Aurora has a unified zero copy data path both for checkpointing and swapping. Pages already in a checkpoint are clean and are evicted by the swapping daemon without IO. Aurora flushes dirty pages into the subsequent checkpoint to persist pages. On a page fault Aurora retrieves the most recent version of the page.

The paging policy prefers evicting clean pages unless memory pressure becomes too high. Fast checkpointing keeps most pages clean for transparent applications, but custom applications may have non-persistent regions of memory. Custom applications can use `madvise` to improve the page selection policy.

The swap integration enables *lazy restores* where pages are brought in lazily or in the background when resuming an application. Lazy restore accelerates restore times by deferring the loading of memory to execution time, allowing it to page in its working set on demand.

## 3.7 Aurora Object Store

The Aurora object store is a copy-on-write store designed to support the needs of Aurora. These needs dictate a COW design for the store that allows for high frequency checkpointing. Existing file systems either do not have checkpoints, or in the case of COW systems like WAFL [130] and ZFS [77] use slow mechanisms that take hundreds of milliseconds.

The object store creates checkpoints with low latency to support the high checkpoint frequency of Aurora. This is in part done by eliminating garbage collection overheads through COW. The series of incremental checkpoints holds the history of an application execution.

The object store's checkpoints map one-to-one to application checkpoints. Aurora prevents resuming incomplete checkpoints by finding the last complete checkpoint after a crash. Aurora prevents loading corrupted checkpoints through COW, ensuring no data is modified in-place.

The object store creates on-disk checkpoints with minimal latency to prevent throttling checkpointing speed. Aurora waits for a checkpoint to fully persist before initiating another one, so a slow object store is a bottleneck to the system.

Aurora uses a low overhead garbage collection scheme similar to WAFL and ZFS. Garbage collection in other COW file systems, such as the log structured file system [204], require periodic garbage collection that would impact the latency of checkpointing.

Aurora efficiently retains execution history of an application when desired by avoiding reclamation of prior checkpoints. Users can use the history to inspect or rewind the application image for debugging purposes. The history of an application execution is only limited by the available storage.

Objects in the Aurora store represent POSIX objects, file system objects, or memory objects. One important design choice is to make sure that both files and memory objects are represented identically. Through this representation we preserve the complex relationship between POSIX objects.

**Non-COW Objects for the Aurora API** For custom applications we provide two low-latency APIs. One uses shadowing to provide atomic checkpoints of a memory region, and the second provides a write-ahead journal. These two APIs provide different performance characteristics.

The atomic region API (`sls_memckpt`) allows Aurora to avoid whole application checkpointing and provides atomic snapshots of a memory region. Atomic region checkpointing

shadows the region's VM object. The VM object is asynchronously flushed to storage. During restore the object is composed on top of a full application checkpoint.

The journal API (`sls_journal`) provides low latency updates to a write ahead log. This API uses preallocated non-COW objects that are updated in place. We provide an append and truncate API for these journals that synchronously writes a 4 KiB page in 28 µs.

## 3.8 Implementation

We implemented Aurora on top of the FreeBSD 12.1 kernel. Our implementation consists of three kernel modules: 9398 source lines of code (SLOC) for the single level store, 4726 SLOC for the object store, and 2890 SLOC for the file system. Our user level libraries, tools and headers consist of 3250 SLOC. We also have a few thousand lines of scripts for our pre-check-in test cases and benchmarks.

The kernel has 2078 lines of changes and additions to support Aurora. This includes support for PID/TID reservations, PID/TID virtualization for jails, and our optimized VM object collapsing (see § 3.6).

**Limitations** Aurora is under development and some features are still incomplete. We are adding support for active TCP sessions and rewriting support for sending and receiving checkpoints over file descriptors to optimize for sending data over TCP and to file systems. Aurora currently does not support external synchrony as many other systems have shown the value and overheads of external synchrony [181].

## 3.9 Evaluation

We evaluate Aurora in terms of its checkpoint frequency, runtime overhead, and restore overhead. We choose these metrics because they are critical for transparent persistence. First, we evaluate Aurora's ability to achieve high checkpointing frequency. This is necessary to make checkpoint-based persistence possible in the first place. We also measure the runtime overhead of Aurora's different checkpointing APIs and mechanisms to the application. The runtime overhead is the deciding factor in whether Aurora is a viable alternative to application-specific persistence code. Finally, we measure the restore latency of Aurora to evaluate the performance of application recovery after a crash failure.

We break down Aurora's runtime overhead into its components to better understand the ways that Aurora is faster compared to conventional checkpointing systems. We split runtime overhead into application stop time, and COW-based runtime overhead. We further split the application stop time into the time it takes to gather application metadata, and

the time it takes to reapply COW to application memory. We take measurements for the application stop time that apply to all workloads, as the stop time overhead per primitive checkpointed and the COW application time per page are the same for all applications. The COW runtime overhead depends on the application, so we evaluate it by running multiple different workloads and measuring the performance overhead of Aurora.

We also compare the performance of the different persistence APIs offered by Aurora. We measure these overheads both in terms of overall stop time compared to the size of the dirty set. We then demonstrate how these APIs combine for the production-grade RocksDB database to improve overall system performance, both in terms of latency and overall throughput.

Finally, we compare Aurora with existing checkpoint-restore frameworks. We use CRIU as a comparison point because CRIU is the only checkpoint-restore system that is currently available in Linux. We evaluate Aurora both in terms of checkpoint and restore overheads. We only compare full checkpoints and not differential checkpoints because CRIU does not provide the latter. Our experiments help frame the rest of the evaluation against the state of the art in checkpoint-restore and demonstrate that our checkpoints have two orders of magnitudes lower stop times and our restores are have latency that is lower by and order of magnitude.

All benchmarks are run on a server with dual Intel Xeon Silver 4116 CPUs running at 2.1 GHz with Hyperthreading enabled and Turbo Boost disabled. The machine has 96 GiB of RAM, and four Intel Optane 900P PCIe NVMe devices striped at 64 KiB. For client-server benchmarks we use machines with identical CPU and memory connected using Intel x722 10 GbE NICs. We disabled page table isolation (PTI), which are the SPECTRE mitigations. Benchmarks are run at least three times except for the client-server benchmarks that were run five times. The error bars shown in all graphs are the standard deviation over the runs.

### 3.9.1 The Aurora File System

Before we evaluate Aurora's checkpoint restore, we first evaluate the Aurora Object Store's performance. This way, we demonstrate that our object store provides the necessary throughput to support sustained high-frequency checkpointing from the OS. As a reminder, our object store offers checkpoint consistency and creates a single on-disk COW checkpoint per Aurora checkpoint. High frequency on-disk COW checkpointing is not available in conventional file systems, which only use checkpointing to occasionally create backups and can only persist individual files (using `fsync`).

We evaluate our object store using file system benchmarks

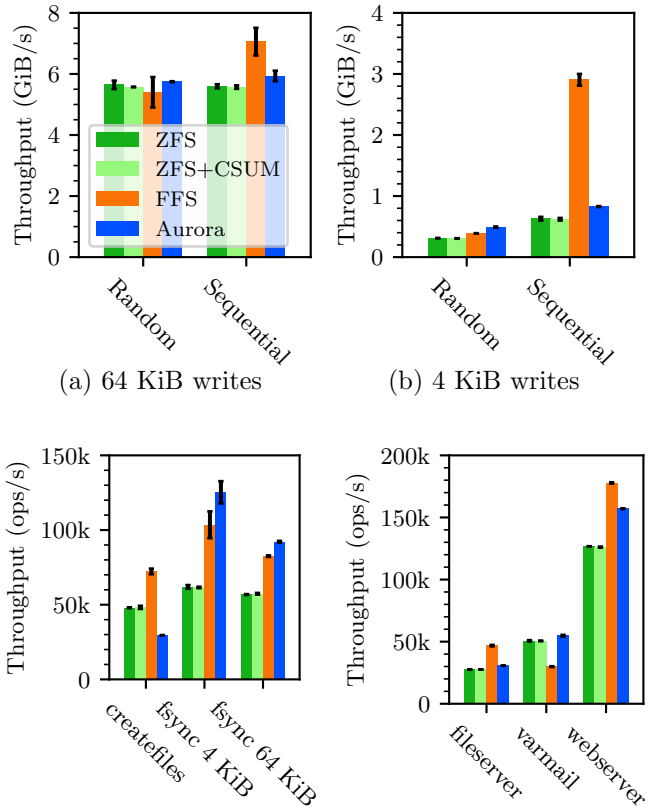



Figure 3.3: FileBench microbenchmarks that evaluate the Aurora file system to both ZFS and FFS. The Aurora file system and its checkpoint-based consistency scheme is competitive with existing file systems.

We use FileBench to evaluate the Aurora Object Store by treating it as a conventional file system. This way we can directly make apples-to-apples comparisons with existing COW file systems and demonstrate that our object store and its checkpoint consistency model is competitive in terms of throughput. The FileBench workload also allows us to to evaluate the object store's throughput when persisting checkpoints, as its write microbenchmarks effectively simulate the disk IO traffic generated by checkpointing. Other operations like file metadata operations are less critical but still important for checkpointing performance.

Our main performance metric is the throughput in terms of write operations. Checkpoints generate large IOs at a single point in time, so the overall IO time depends on the throughput

of the object store or file system used. We also measure the throughput of the object store in terms of metadata operations, namely file creation and `fsync`. These operations are critical for persistence-oriented applications, along with the aforementioned file writes. Finally, as a macrobenchmark for the object store we compare the overall performance of three file-oriented applications when using the Aurora object store and a conventional file system.

We compare the object store to ZFS (with and without checksumming) and FFS (with SU+J [169, 6] and without checksumming). ZFS provides a comparison to a similar snapshotting file system while FFS provides a baseline for traditional file systems. Each file system is configured with 64 KiB block sizes.

Figures 3.3(a) and (b) show the object store's throughput for 4 KiB and 64 KiB random and sequential writes. ZFS is slower than Aurora in both configurations because Aurora's simpler metadata updates are designed to reduce the latency of periodic checkpoints. FFS has an optimized small write path because of the use of fragments that reduce internal fragmentation [168]. As an optimization for fragments FFS delays allocations to allow blocks to be promoted to full blocks for IO.

Figures 3.3(c) show the operations per second for file creation and `write+fsync`. File creation in Aurora is unoptimized and currently requires grabbing a global lock. Aurora's use of checkpoint consistency results in faster `fsync` calls because the operation is a no-op. ZFS syncs are slower than FFS and Aurora because its COW mechanism generates complex changes to file system state, but it writes these changes to the ZFS intent log rather than generating a checkpoint.

Figure 3.3(d) shows the performance of the file systems for three benchmarks simulating a file server, a mail server, and a web server. Aurora performs similarly to the other file systems but it outperforms in varmail because of the workload uses `fsync`, which is a no-op under Aurora.

These results show that the object store performs well compared to conventional file systems. The object store offers competitive write throughput that is enough to support high checkpointing frequencies. The object store achieves this performance while offering checkpoint consistency and being able to checkpoint at a frequency that matches that of Aurora's application checkpointing. The object store can thus create a single COW checkpoint on the disk for each in-memory operation and properly support Aurora's persistence model.

### 3.9.2 Persisting POSIX Objects

Next, we measure the performance of persisting the POSIX objects that make up an application's metadata state. Applications may have large amounts of such objects at any time, and their checkpoint latency directly affects stop time, so achieving a minimal checkpoint latency for them is critical. Aurora also eagerly restores all POSIX objects at restore time, so we also consider their restore latency an important metric.

For this experiment we measure the time to checkpoint a single instance of a POSIX object into a record, and the time to restore it. These operations are measured completely in memory and do not include IO. We measure the cost of checkpointing and restoring in isolation, because our use of the POSIX object model (Section 3.5) means that checkpointing/restoring a single object is a self-contained operation whose cost is independent of the cost of checkpointing/restoring the application as a whole.

This microbenchmark helps us better understand the cost of checkpointing for our application benchmarks. The cost of each checkpointing/restoring each primitive is fixed per primitive type, because all instances of a single object type, e.g., all kqueue instances, have identical amounts of state. The latencies we measure can thus be directly used to analyze the cost checkpointing an application, by assuming that each of its POSIX object has the checkpoint/restore costs measured here. Since these operations are independent between objects of the same application, we can sum the operation costs for all POSIX objects in an application to measure the checkpoint/restore costs for its metadata in total.

Table 3.4 presents the time it takes to checkpoint and restore common POSIX objects, and serialize each object into a buffer. Most POSIX objects are small and typically involve one lock and pointer chasing, which incurs cache misses. Each object has its own challenges when checkpointing and restoring which explain the differences in latency:

- Checkpointing vnodes is fast when they reside in our object store because we only store a reference to the inode. The Aurora Object Store manages the actual data, which does not need to be accessed during checkpoint or restore.
- Shared memory checkpoint times include the time spent shadowing, but not invaliding the MMU. Shadowing involves creating a new shadow VM object for the memory mapping (Section 3.6).
- System V is more costly than POSIX shared memory because Aurora scans the global SysV namespace. We have to traverse this namespace, an array of 64K entries, to find the object before checkpointing it. POSIX objects are in contrast directly accessible through their file descriptor.

| POSIX Objects | Checkpoint | Restore |
|---|---|---|
| Kqueue w/1024 events | 35.2 µs | 2.7 µs |
| Pipes | 1.7 µs | 2.6 µs |
| Pseudoterminals | 3.1 µs | 30.2 µs |
| Shared Memory (POSIX) | 4.5 µs | 3.8 µs |
| Shared Memory (SysV) | 14.9 µs | 2.8 µs |
| Sockets | 1.8 µs | 3.6 µs |
| Vnodes | 1.7 µs | 2.0 µs |

Table 3.4: Checkpoint and restore times for POSIX objects.

- Checkpointing the Kqueue is slow because it contains 1024 individual event structures, and requires locking each structure. Most of these structures are not active, so the overhead is purely caused by traversing the data structure containing them.
- Pseudoterminals are slow to restore because they require taking locks in the device file system when recreating the virtual device.

Our results show that checkpointing POSIX objects is in the order of microseconds and therefore does not impose significant overheads in the checkpoint process even for complex applications. Since each type of object has the same amount of metadata state, the latency of each POSIX object type for all applications should match Table 3.4.

### 3.9.3 Checkpointing Memory Objects

Next, we measure the cost of checkpointing and applying COW to the memory objects of an application. The overall cost of this operation combined with the numbers in Section 3.9.2 combine to give us a measure of the stop time imposed by Aurora per checkpoint. The cost of checkpointing the memory state of an application is twofold: On a checkpoint, Aurora first gathers the metadata state of the VM objects that back the application's address space(s). Aurora then applies COW to all pages modified since the last checkpoint. The cost of gathering metadata state is a fixed overhead for a given address space and does not scale with the size of the mappings, while the cost of applying COW scales with the amount of dirty pages in the address space.

Table 3.5 shows the stop time required to persist a modified memory region for transparent checkpoints (incremental), checkpoints of a single memory region (atomic), and synchronous updates to a region (journaled). Incremental checkpoints include all memory

| Object Size | Checkpoint | Aurora API | |
|---|---|---|---|
| | Incremental | Atomic | Journaled |
| 4 KiB | 185 µs | 80 µs | 28 µs |
| 16 KiB | 185 µs | 83 µs | 32 µs |
| 64 KiB | 183 µs | 74 µs | 55 µs |
| 256 KiB | 186 µs | 81 µs | 121 µs |
| 1 MiB | 186 µs | 72 µs | 443 µs |
| 4 MiB | 226 µs | 114 µs | 1.8 ms |
| 16 MiB | 304 µs | 184 µs | 6.6 ms |
| 64 MiB | 600 µs | 492 µs | 25.9 ms |
| 256 MiB | 1.9 ms | 1.6 ms | 104.7 ms |
| 1 GiB | 6.1 ms | 6.3 ms | 417.2 ms |

Table 3.5: Checkpoint times for userspace data objects using differing modes of Aurora's API.

and OS state. Atomic checkpoints use our API to shadow a single memory object and asynchronously write the changes to storage. The journaled method uses `sls_journal` to synchronously write the data to storage.

The table shows that checkpoint stop time scales linearly with the dirty set, because of the linear time needed to mark pages copy-on-write in the x86 page tables. Atomic checkpointing is faster than full checkpointing by roughly 100 µs, which is significant for dirty sizes of up to 64 MiB. The journaling API issues synchronous writes and is the fastest strategy up to 64 KiB. For larger sizes asynchronous approaches are better for minimizing the stop time at the cost of increasing the latency for the write operation.

These results give us insight into Aurora's overheads when handling memory checkpointing. First, gathering the metadata of the address space, i.e., the address map and all its mappings, is low - about 100 µs. We can verify this by comparing the cost of incremental checkpointing with that of region checkpointing. In this benchmark both checkpointing APIs apply COW to the same number of pages, so the only difference in actual work done is that the incremental checkpointing API also checkpoints the metadata for the entire address space, while the region checkpoint only gathers the data from a single memory region. As a note, our reported application checkpointing latency for 1 GiB dirty are lower than the latency of region checkpointing for the same size due to noise in the benchmark.

The second insight is that for high frequency checkpointing, the overall overhead scales linearly or sublinearly with the dirty set. Since checkpointing more frequently means that

| | Type | firefox | mosh | pillow | tomcat | vim |
|---|---|---|---|---|---|---|
| Checkpoint | Size | 198 MiB | 24 MiB | 75 MiB | 197 MiB | 48 MiB |
| | Mem | 1.4 ms | 0.4 ms | 0.7 ms | 2.7 ms | 0.7 ms |
| | Full | 1.8 ms | 0.4 ms | 0.9 ms | 3.2 ms | 0.8 ms |
| | Incr. | 1.9 ms | 0.4 ms | 0.6 ms | 2.1 ms | 0.7 ms |
| Restore | Mem | 0.9 ms | 0.2 ms | 0.2 ms | 0.5 ms | 0.3 ms |
| | Full | 12.4 ms | 1.9 ms | 8.2 ms | 33.6 ms | 4.1 ms |
| | Lazy | 6.3 ms | 0.9 ms | 0.2 ms | 3.1 ms | 2.4 ms |

Table 3.6: Checkpoint stop times and restore times for several popular applications.

less data gets dirtied between successive checkpoints, our differential checkpointing scales well with checkpointing frequency.

These results combined with those of Section 3.9.2 demonstrate that the stop time imposed by Aurora is in the microsecond range. Even for complex applications with multiple of processes and hundreds of primitives and VM objects, the cost of is many times smaller than our target checkpointing frequency of 10 ms. This way we show that Aurora's differential checkpointing achieves our goal of keeping the overhead of continuous checkpointing low for the application.

### 3.9.4 Application Checkpointing

We now measure checkpoint and restore times for a variety of real world applications of varying amounts of complexity. Our goal is to show that Aurora's low checkpointing latency for the various POSIX and memory primitives of an application directly translate to low overall checkpointing latency both for large and small applications in terms of metadata state.

Table 3.6 shows the checkpoint stop times and restore times for a variety of popular applications, including Pillow, a popular python image manipulation framework and Tomcat, a Java application server, and the Firefox browser running in headless mode. The applications are mostly idle for the incremental checkpoints to show the lower bound. We show the stop time for memory checkpoints (i.e., not flushed to disk), full and incremental checkpoints.

Memory checkpoints differ from full checkpoints in the way they represent the image. Memory checkpointing represents each image as a hash table of records instead of fully serializing the data into a disk-appropriate format. Avoiding serializing the image reduces

checkpointing costs for more complex applications.

The simpler applications, like mosh, pillow, and vim predictably take below 1 ms both to checkpoint and restore. The reason is that these applications have simple address spaces and use up to tens of POSIX object primitives, so the low checkpointing latency for each of these objects translates to low checkpoint times.

For these benchmarks the complexity of OS state, including page tables, is the primary factor affecting stop times. OS state and virtual memory data are not correlated. For example, vim and pillow (Python) have small memory footprints, but complex OS state including hundreds of address space objects. The time to flush the checkpoint (not shown) is proportional to the resident set size of the application.

Smaller, simpler applications like vim and mosh have identical performance for all checkpointing strategies because of their trivial system shadowing overheads and small POSIX object state. For example, the tomcat server has a larger working set and therefore sees a latency reduction of 0.5 ms when using incremental checkpointing as opposed to full checkpointing.

Firefox in contrast has multiple processes and uses hundreds of POSIX objects, while also having little actual memory usage when idle in headless mode. As a result, incremental checkpoint offers little benefit. Our experiment in fact reports incremental checkpointing as slightly more expensive than full checkpointing because the difference is within margin of error.

For restore times we report the time to restore an application from an in-memory image (Mem), the time to bring an entire image from disk then restore it (Full), and the time to restore an on-disk image by only reading its metadata from disk and leaving most data nonresident (Lazy). As expected, applications with a larger working set see the most benefit from Lazy checkpointing as opposed to Full. The tomcat server in particular sees a 10×improvement in restore latency.

The pillow server also sees a 16 times restore time improvement. This is expected, as Pillow's checkpoint latency benefits from incremental checkpointing just like like the Tomcat server. This means that memory state is the dominating factor in checkpoint/restore latency, and not bringing it in at restore time removes the main bottleneck for its restore operation.

Memory restore times remove the cost of deserializing the application image at restore time into records and is therefore uniformly faster than lazy checkpointing. All applications except for pillow see latency improvements between 3.5×–8×when avoiding this overhead. Pillow is the exception because it is a single Python process with little to no POSIX objects and a large amount of address space mappings.

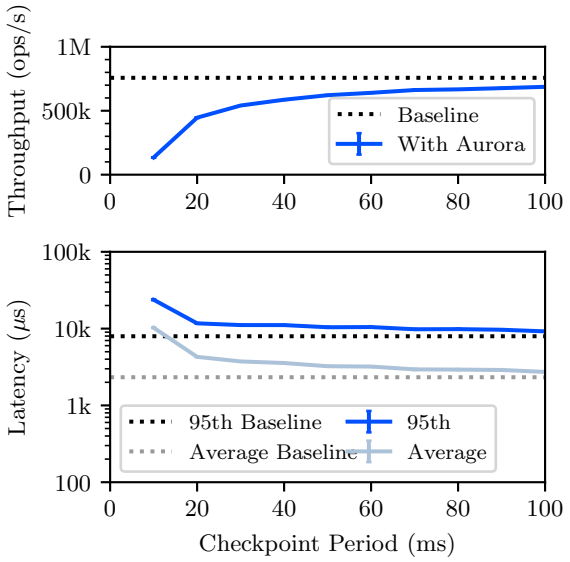



Figure 3.4: Memcached at max throughput over varying checkpoint periods.

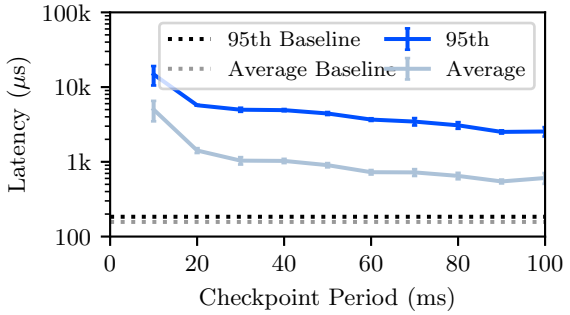



Figure 3.5: Memcached with throughput pegged at 120 k ops/s over varying checkpoint periods.

### 3.9.5 Transparent Persistence for Memcached

Next we measure the efficiency of transparent continuous checkpointing for memory-intensive applications. Our goal is to demonstrate that transparent persistence is possible even at a high granularity and even if the application is memory intensive. To this end we choose the Memcached in-memory key value store as a workload. Memcached is memory intensive while also having simple POSIX object state. Benchmarking it then lets us isolate the cost of continuously checkpointing application state.

Memcached is also designed without persistence in mind and presents a worst-case scenario for Aurora, allowing us to measure the cost of persistence for an unmodified application. Memcached uses data layouts and structures like hash tables that are suitable for in-memory computation because they do not have page granularity and do not have access locality. Since Aurora tracks changes in data at a page granularity, Memcached exhibits significant write amplification under transparent checkpointing. We thus use Memcached to test whether Aurora can provide transparent persistence even when the workload goes against its limitations.

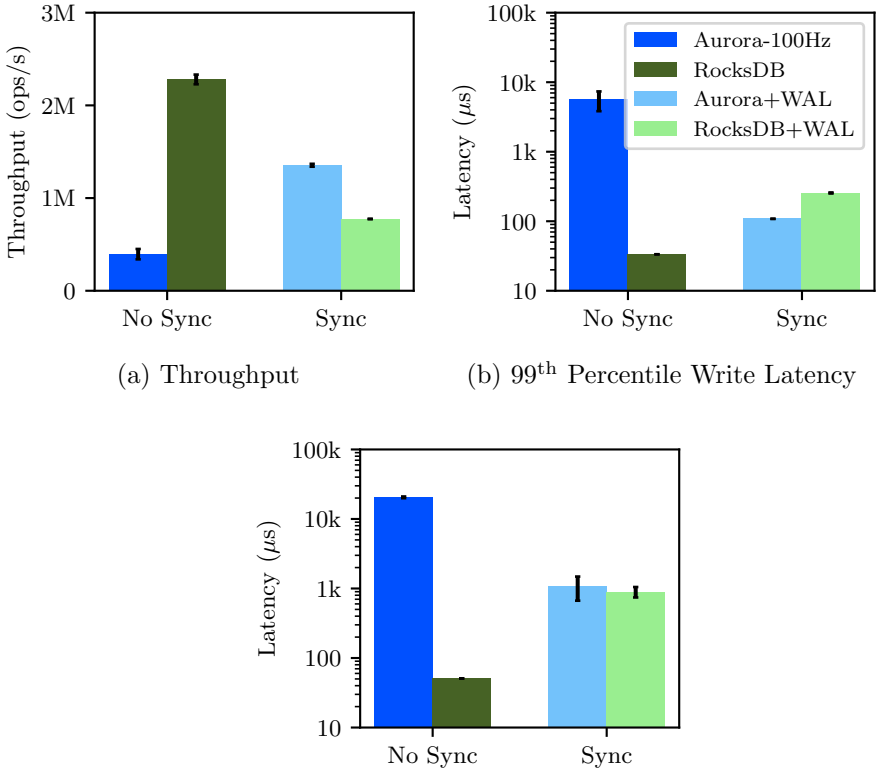
Figure 3.4 shows Memcached's throughput and latency in response to the checkpoint period. The horizontal lines show the baseline for non-persistent Memcached without Aurora. We use Aurora to transparently add persistence to Memcached with no code changes or developer effort. We use the Mutilate benchmark running the Facebook workload [74] with four machines generating load and another machine to measure latency. Each machine has 12 threads and 12 connections per thread.

Increasing the checkpoint period (i.e., fewer checkpoints per second) increases the throughput and decreases the latency of the application. Between the 10 ms and 20 ms data points checkpoint the frequency halves from 100 to 50 per second. Throughput increases proportionally and latency decreases by more than 2. For checkpoint latencies of 60 ms and up, Memcached's throughput is within 15% of the baseline.

Figure 3.5 shows Memcached's latency response for a fixed 120 k ops/s, or 15% of peak, in response to changes in checkpoint period. This graph examines the worst case for Aurora's transparent persistence. Aurora at low throughputs has a much larger effect on the overall latency of the system because of low network queuing. The baseline without persistence has an average latency of 157 µs. With transparent persistence at a 100 ms period Memcached has an average latency of 607 µs. Looking back at Figure 3.4, the latency impact of transparent persistence has a small effect once network queues begin to saturate and the base latency increases.

Our experiment has two significant results: First, transparent persistence is feasible even for completely unmodified workloads, albeit at a steep performance cost for very high checkpointing frequencies. The end-to-end metrics we show to evaluate Memcached show that overheads scale not with the checkpointing period, but with the checkpointing frequency. Decreasing the checkpointing period then causes significantly larger overheads when the period is already low.

As a result, for very high persistence granularities we must complement transparent checkpointing with mechanisms that remove the need to checkpoint at frequencies that are too high for the workload. In this thesis we provide and evaluate two such alternatives,




(a) Throughput

(b) 99th Percentile Write Latency

(c) 99.9th Percentile Write Latency

Figure 3.6: A comparison between multiple RocksDB configurations running the Prefix_dist Facebook workload. Configurations that do not provide write persistence are labeled "No Sync" while those that do are labeled "Sync".

the Aurora journaling API in the next section, and MemSnap µCheckpoints in the next chapter.

### 3.9.6 Customized RocksDB

We use RocksDB [46], a popular key-value store, to explore the performance trade-offs of using the Aurora API over an optimized persistence backend. We show that efficient customized applications can be created with minimal developer effort. In our customized RocksDB, we replaced 81k SLOC of persistence code (i.e., ∼40% of the codebase) with 109 SLOC while providing the same persistence guarantees.

The goal of this experiment is twofold. First, we demonstrate that integrating existing persistent applications with Aurora is easy and removes the need for large amounts of

persistence-related code. Second, we show that this integration offsets the limitations of purely transparent checkpointing that we highlight in the previous section. In total, our aim is to show that transparent checkpointing with the help of the right API makes persistence significantly easier even for mature applications.

RocksDB has three main data structures: The Memtable which buffers data in memory before writing data to disk, a log-structured merge tree to store data on disk, and a write-ahead log (WAL) for crash consistency [105].

We remove RocksDB's log structured merge tree and use Aurora to persist the Memtable. We replace RocksDB's WAL with Aurora's journaling APIs to store data in a non-COW region on disk. When the journal is full, RocksDB triggers an Aurora checkpoint and clears the journal.

We compare the performance of RocksDB across four configurations. The first configuration is an unmodified RocksDB instance with no persistence at all. The second is a RocksDB instance with Aurora transparently persisting the application every 10 ms. The third configuration is an unmodified RocksDB using it's built-in WAL, and the fourth is our modified RocksDB with our custom WAL.

To make the comparison fair, we size the Memtable to fit the whole database in memory, allowing reads to be serviced from memory. We use the Facebook Prefix_dist workload that uses RocksDB as a library [86].

Figure 3.6 shows the results for each configuration of RocksDB. Figure 3.6(a) shows an 83% decrease in throughput when using Aurora's transparent mode relative to the ephemeral database. Transparent checkpoints have about half the performance of RocksDB's WAL, but with weaker consistency guarantees as writes are not persisted until the next checkpoint. Using the custom API we provide the same write consistency as RocksDB with the WAL, but with 75% higher throughput than the other persistent configurations.

Figures 3.6(b) and 3.6(c) show the latency overhead of the various configurations of RocksDB. High tail latencies occur in unmodified RocksDB with Aurora because of application stop times caused by transparent checkpoints. Our custom journal achieves better $99^{th}$ percentile latency than the unmodified RocksDB WAL, but the $99.9^{th}$ percentile suffers as writes that trigger checkpoints must wait for the checkpoint to complete.

One reason for the performance difference is that Aurora the use of the application's file APIs with its checkpointing primitive. RocksDB is required to persist and manage its on-disk LSM tree as a form of checkpointing, since it persists individual operations with WAL writes, and so must periodically truncate the WAL to prevent it from growing in size indefinitely. Managing and persisting the LSM tree is complex and, as we show, has

| Type | Aurora | CRIU | RDB |
|---|---|---|---|
| OS State | 0.3 ms | 49 ms | N/A |
| Memory | 3.7 ms | 413 ms | N/A |
| Total Stop Time | 4.0 ms | 462 ms | 8 ms |
| IO Write | 97.6 ms | 350 ms | 300 ms |

Table 3.7: Comparing Aurora's full checkpoint performance to CRIU with a 500 MiB Redis instance. Aurora is two orders of magnitude faster in terms of stop time both for metadata and data copies. Aurora writes out the checkpoint 3 faster, even though CRIU does not flush to disk.

overheads that Aurora's transparent checkpointing avoids. Transparent Aurora checkpoints are thus preferable to RocksDB's own LSM tree-related code for smaller than memory workloads.

The Aurora journaling API remedies the core limitation of transparent checkpointing for this workload by persisting individual writes, while still providing completely removing file-based semantics from the application as required by an SLS. As we show, Transparent Aurora cannot replace RocksDB's WAL API without significant throughput overheads of 83%. This is consistent with the our results from Section 3.9.5 where we show that transparent checkpointing does not scale well down to very short periods for memory-bound workloads. In contrast, the Aurora journaling API provides microsecond persistence 3.9.3, and is almost a drop-in replacement for RocksDB's WAL API while also being compatible with Aurora's transparent checkpointing. Adding the Aurora journaling and only checkpointing when the journal is full reduces the checkpointing frequency to frequencies similar to those of LSM operations and reducing its cost.

### 3.9.7 Comparison with Linux CRIU

Finally, we compare the performance of Aurora with that of existing checkpointing systems. Table 3.7 compares the full checkpoint performance of Aurora versus CRIU, the state of the art checkpointing system for Linux. We use full checkpointing because CRIU does not have support incremental checkpointing for persistence.

We evaluate both systems on the time to checkpoint POSIX state, the time to gather memory state, and the time to flush the checkpoint to the disk. Checkpointing POSIX and memory state are done while the application is stopped, so they directly translate to overhead for the application. IO write time runs concurrently with the application and

places an upper bound to the checkpointing frequency because a checkpoint must completely persist before the next one starts.

We break down the checkpoint times of Aurora and CRIU for a 500 MB Redis instance. CRIU was run on Ubuntu 20.04 with the same hardware. The results show that Aurora is over 100 faster than CRIU with regards to application stop time while outperforming it by more than 3 when writing to storage. Unlike Aurora that uses system shadowing, CRIU must prolong application stop time to collect data pages. Aurora's system shadowing in contrast requires no copying while the application is stopped.

Aurora's OS state checkpointing time is also 163 ×faster than CRIU. Aurora directly gathers this state from the kernel, and its POSIX object model avoids processing (Section 3.5). In contrast, CRIU reads application state from userspace using `procfs`, processing the information returned while the application is stopped to determine the application's exact POSIX OS state.

We also compare Aurora with Redis' `fork` based RDB mechanism. RDB saves the Redis database by forking the original process and writing out the key value pairs from the child. We measure the time from the RDB save until all the keys are written out by using the `BGSAVE` and `SAVE` commands from a local client.

The RDB mechanism is than Aurora despite only saving the data rather than the whole process, taking 2.16 ×more time to gather the data. The reason is that RDB uses `fork` to create the COW copy, which requires creating a new address space apart from creating VM object shadows, including getting references to any files and pseudodevices mapped into Redis' address space from the file system or `devfs`. Aurora's system shadowing mechanism avoids these overheads.

The time to write out the data is also 3 slower than Aurora because of serialization overheads. Aurora issues IOs for the checkpoint in parallel, while Redis uses individual writes to flush out the data.

## 3.10 Related Work

**Migration and Fault Tolerance** Live migration moves applications between machines for maintenance, distributed resource management, and other applications. Migration is a well studied problem [93, 129, 234, 165]. Many techniques for live migration such as pre-copy, post-copy and hybrid approaches have been studied. Aurora's iterative checkpoints can be used to build a precopy migration mechanism.

Research distributed operating systems support transparent process migration and

failover as a major benefit [220, 90, 71]. Aurora can be used as a building block to enable migration and failover functionality in operating systems.

Checkpointing techniques used in databases [171] for fault tolerance require developer effort. Other systems rely on specialized hardware like NVRAM [175].

**Serverless Computing:** Checkpoint and restore has been used by serverless computing for snapshotting serverless functions [108, 228]. These systems snapshot initialized functions and restore them at invocation time. The checkpointing mechanisms used in serverless computing are oriented towards fast restores and do not optimize checkpointing.

**Record/Replay Systems:** Record/replay systems [89, 203, 110, 111, 166, 230, 153] record non-deterministic inputs to an application to replay the complete execution. These systems generate large amounts of data and cannot sustain recording indefinitely. Checkpointing bounds the size of the replay log by only retaining the non-deterministic inputs of the execution since the last checkpoint.

**Databases:** Databases [43, 42, 41] and key-value stores [45, 40, 46] often spend a significant amount of their code managing persistence and the paging of database state. The `fsync` and `msync` calls have subtle semantic issues depending on hardware and software configuration leading to data loss bugs in even mature projects like LevelDB [8, 10, 13, 12, 193] and PostgreSQL [44]. Aurora provides an alternative way to achieve persistence in databases, allowing for the reduction of code and developer effort.

## 3.11 Limitations and Future Work

Aurora's design focuses on transparent persistence on commodity hardware, and its limitations as a checkpointing system stem from our efforts in achieving our design goals, even at the cost of performance. Aurora also has certain drawbacks in functionality and performance that we are planning on addressing in future work. Even with these limitations, Aurora achieves its core goal of transparent persistence through high frequency checkpointing.

Aurora mostly implements transparent checkpointing and does not provide fine tuned checkpointing primitives for subsets of applications except for checkpointing a single checkpointing region. This has the effect that applications can only integrate with Aurora up to a point, and cannot avoid certain metadata and data checkpointing overheads even if developers were willing to write code for more tight integration with Aurora. While we leave the design of a more finely granular API for checkpointing as future work, we do address the design of a persistence API for data with our next work, MemSnap (Chapter 4).

More specifically, Aurora's COW mechanism is efficient for checkpointing frequencies of 10s to 100s of milliseconds, but does not scale lower. Applying COW on a dataset

as implemented in existing virtual memory subsystems requires scanning all memory sequentially to find writable pages, then turning them readable by modifying the page tables. The cost of traversing all PTEs is not noticeable until very low checkpointing periods, at which point it becomes a bottleneck. For this reason, Aurora is not able to scale beyond the low milliseconds in terms of checkpointing period. Our next work, MemSnap (Chapter 4) addresses and remedies this limitation.

Aurora currently does not implement differential checkpointing on application metadata. This can increase checkpointing time, e.g., for applications with many large files. The absence of differential metadata checkpointing does not affect performance: Metadata checkpointing is a small part of the overall checkpointing overhead, as the main bottleneck is in applying COW and writing out application data. Solving this problem for Aurora would thus not provide the ability to checkpoint at a higher frequency.

Aurora uses page-level granularity for its differential page tracking because it uses conventional TLB hardware to track its dirty set. This translates to write amplification for applications that heavily dirty memory with no spatial locality. Such applications include key-value stores built to be completely in memory. This limitation stems from Aurora's use of conventional memory management hardware. It is possible to overcome these limitations using recent advances in hardware-based memory management like hardware capabilities [243] and cache-line level dirty set tracking [85].

Aurora does not support the checkpointing of connected TCP sockets. Such sockets are restored in an invalid state to force the application to reconnect. Our reasoning is that connected sockets are used to communicate with an external client that is possibly not present at restore time, so restoring the socket does not make sense. Even if the client is present, e.g., when using Aurora for fault tolerance, restoring the socket in a disconnected state will be handled by the application: A well-formed application will see that the socket is invalid, clean it up, and possibly initiate a new connection with the remote client. Aurora also does not support applications that use asynchronous IO and `io_uring`.

Aurora depends on external synchrony to ensure checkpoint consistency. This external synchrony manifests as additional latency when communicating with external applications, so its overhead is modest when communication is low but high when communication is frequent. Such an example is for a database client doing a complex transaction, as highlighted by previous works like RemusDB [171]. In that case external synchrony becomes the main bottleneck. For this reason, we have considered setups where the application is loaded by multiple clients making single requests. This is a ubiquitous scenario in modern data centers due to the dominance of simpler key-value stores. We can lift the performance limitations of external synchrony by changing the clients to be aware of possible failures,

like in RemusDB.

Aurora assumes that the application is self-contained and is not sharing resources like anonymous pipes or shared memory with another application that is not being checkpointed. This is a reasonable assumption for modern systems that deploy applications using OS virtualization techniques like containers: Aurora treats the entire container as the application and checkpoints it correctly. However, Aurora does not guarantee that the checkpoint will be well-formed if containerization is not present. In our experience, all applications we used had clearly defined boundaries and were checkpointable.

Aurora cannot do intraprocess checkpointing transparently, that is, checkpoint part of a process. This is an inherent limitation of transparent checkpointing systems, as threads in a process share their memory and access it without generating system level events. It is therefore impossible to determine the working set of a single thread in a process. Using COW faults is insufficient, because only the first write in a page would be tracked for checkpointing, and subsequent writes by a thread on the same data would not trigger a fault.

We do not explore the performance of efficient memory overcommit for Aurora. Aurora handles memory overcommit the same way as a swapper in terms of policy and is reactive, that is, it only retrieves data from the disk when the workload attempts to access them. Handling overcommit efficiently would require both designing an API for applications to request prefetching memory mapped pages from the disk if not resident, and efficiently integrating this API with individual workloads.

## 3.12 Conclusion

We present Aurora, the first modern single level store for UNIX. Aurora is built on the observation that hardware trends have lead to persistent storage bandwidth rivaling that of memory. Aurora offers a simple and elegant solution for application and data persistence as an OS level service.

Our evaluation shows that Aurora works efficiently with a wide variety of workloads, including key-value stores but also applications like web servers and even web browsers. Aurora handles unmodified workloads with reasonable overhead and provides an API that applications can use to lower the cost of checkpointing.

Aurora shows that persistence is a useful and flexible OS level mechanism that enables a wide range of services, from debugging to serverless computing. Aurora's API enables application persistence with a fraction of the effort it would otherwise need, and enables new

research into application/OS co-design for specialized applications. We expect hardware advances like NVDIMM and FPGAs to enable even more efficient single level stores.

Aurora's source code is available at https://github.com/rcslab/aurora/ and the patched FreeBSD 12.1 kernel is available at https://github.com/rcslab/aurora-12.1.

# Chapter 4

# MemSnap: SLS µCheckpoints for Low-Latency Persistence

## 4.1 Introduction

Single level stores (SLSes) like EROS [209], Aurora [226, 225], and TreeSLS [245] promise to simplify the development of application persistence by offering an alternative to file IO. Application writers build applications completely in memory as if they never fail, and SLSes provide transparent or nearly transparent persistence by checkpointing entire applications. Applications in an SLS forego file-based storage logic and its semantics that have led to data loss even for mature systems like PostgreSQL [44].

The problem with current SLS systems is that they provide persistence with coarse granularity that leads to them supplementing their APIs with traditional journaling mechanisms. EROS persisted applications every 30 s on a spinning disk, Aurora every 10 ms using PCIe SSDs, and TreeSLS achieves persistence every 1 ms using non-volatile memory (NVM). Aurora and EROS provide a write-ahead log API to bridge the performance gap between the coarse grained checkpoints and individual application operations that require durability.

SLS WAL APIs undermine the core premise of the single level store by retaining much of the complexity of the file API. Developers must still implement WAL and checkpoint systems to achieve good performance in SLSes, same as the file API. Thus, applications retain most of the code required for crash consistency, which is prone to bugs [200, 193, 250]. Even worse, using the WAL usually leads to infrequent SLS checkpoints causing larger and

significant pause time to atomically track memory. The pause time adds tail latency to the application.

SLSes will only qualitatively change how applications are written if they truly deliver on memory semantics. We need an API that has low latency and works at a fine granularity to completely eliminate the checkpoint and WAL paradigm.

To support high performance applications with the full promise of a simplified developer model, we need a system that persists atomic checkpoints as small as a single memory page in microseconds. This would fully replace the file API abstraction with a simpler design that ensures atomicity across memory and storage in the face of system crashes. Developers would not need WALs or other approaches for correctness in their storage systems. Furthermore, it can lead to potential performance gains by eliminating the need to write data both to the WAL and to the primary store.

This paper reexamines the core mechanism behind the single level store: atomic checkpointing across virtual memory and storage. We focus on incremental checkpointing of memory regions similar to Aurora's region API. Applications work only with memory-mapped data that they modify in place and then update on disk with a single call. Our approach revisits the core idea of SLSes by providing a per-thread memory tracking and checkpointing API that outperforms file APIs and existing SLSes.

We introduce MemSnap, a new novel single level store that provides per-thread checkpoints as small as a 4 KiB page with low overhead. MemSnap is the first single level store to track individual per-thread transactions at the kernel level as opposed to the entire process without application intervention. MemSnap provides atomic checkpoints fast enough to subsume the need for the WAL by introducing new page table handling primitives in the virtual memory system. MemSnap uses the hardware TLB and fast page protection kernel code paths both to track the dirty set and prevent application threads from modifying in-flight data.

MemSnap makes the following contributions:

- Low overhead dirty set tracking for individual threads. MemSnap tracks the pages dirtied by each thread without help from the application.
- A novel per-thread checkpointing mechanism that persists dirty data into µCheckpoints with low pause times. MemSnap supports the existing SLS semantics of checkpointing all threads' data.
- A data persistence API that does not depend on a WAL. MemSnap's atomic

µCheckpoints across memory and storage are fast enough to replace storage paradigms such as WAL and checkpoint and LSM Trees.

- Three case studies that demonstrate MemSnap's performance gains and simplify the code of production grade databases: SQLite, RocksDB, and PostgreSQL.

MemSnap is faster than existing file APIs and past SLS-based approaches. MemSnap µCheckpoints have 2×-1.17× overhead for writes of 4 KiB-64 Kib over direct disk IO. For comparison, file based random write IO incurs a 9×-43× overhead. In our case studies we show that MemSnap is faster in large applications that use the file API or traditional SLSes. MemSnap increases the performance of SQLite by 5× over file APIs. For RocksDB, MemSnap achieves a 4× improvement over Aurora's checkpointing.

## 4.2 Motivation

File APIs lack atomicity in three important ways: write-tearing, multi-file writes and multi threaded isolation. First, write-tearing happens when a crash occurs during an IO that spans multiple sectors, resulting in the partial writing of the IO. The calls used by applications to flush changes to disk, i.e., `fsync` and `msync`, are not atomic on disk. Disks provide atomicity at the level of individual sectors.

Second, file systems lack the ability to atomically update multiple files. The `fsync` and `msync` calls only operate on a file or a contiguous file region.

Third, flushing changes is not atomic with respect to other threads. Modifications made by other threads to the same file will be flushed by any call to `fsync` or `msync`. There is no way to differentiate the write-sets of individual threads.

The traditional solution to solving the first two problems is through the use of a write-ahead log (WAL). Applications first persist updates to the WAL, acknowledge the original request, and then update the primary data structure in place (e.g., B+ Tree, LSM Tree). If a crash were to occur during the WAL write, the user is never acknowledged. If a crash occurs while updating the primary data structure, the WAL is used to complete the operation.

The third problem is solved by combining a variety of approaches including software level tracking of updates, checkpointing of the main data and multi version concurrency control (MVCC). This problem often leads to applications making trade-offs between design complexity and performance. For example, MVCC allows unfinished operations to reach the disk, but introduces costly garbage collection.

| Task | % Time | Task | % Time |
|---|---|---|---|
| **Userspace** | | **Kernel** | |
| Tx Memory | 18.3% | Buffer Cache | 5.1% |
| Log | 8.0% | File System | 3.1% |
| Tx Disk | 8.5% | VFS | 6.4% |
| IO Generation | 4.3% | Locking | 6.1% |
| Serialization | 1.1% | Rangelock | 2.1% |
| Other Userspace | 16.2% | Syscall | 4.4% |

Table 4.1: Breakdown of RocksDB's total execution time between userspace and the kernel. Only 18.3% of CPU time is spent on the in-memory transaction - the rest is spent on persistence across userspace and the kernel.

Flushing the WAL to the primary data structure is expensive and necessary. Applications must flush the WAL to reclaim space at some point. The flush operation depends on how the system represents data, for example the SQLite database creates a checkpoint of the database file, while the RocksDB key-value store adds a new node to its LSM tree. SQLite checkpoints generate many random IOs that increase latency, while writes to RocksDB's LSM trees are sequential but require additional IO because of background compaction operations (i.e., garbage collection).

Systems that build on `fsync` and `msync` share the same limitations as the original calls and still need WALs to function. Atomic msync [189] extends `msync` to be atomic in the presence of crashes, so an interrupted `msync` call does not corrupt data. AdvFS has the `syncv` [232] call that persists multiple files atomically using copy-on-write (COW) at the file system level. Both systems are slow, taking milliseconds to flush data to disk, require an internal WAL for consistency across multiple files, and do not address per-thread isolation.

**The Cost of Persistence** File APIs are not only lacking in atomicity, but expensive due to a tight integration with bulky subsystems such as the virtual file system (VFS). To illustrate the cost of file based persistence used by databases, we examine RocksDB [46] running the Facebook MixGraph workload [86]. RocksDB is a popular key-value store that is representative of other systems. Table 4.1 shows the CPU breakdown between components of RocksDB and the OS kernel for the workload.

RocksDB spends the majority of its CPU time on persistence instead of servicing requests. Approximately 40% of RocksDB's total time is spent writing the WAL, serializing records and issuing IO. RocksDB splits its time between userspace (56%) and the kernel (44%). Only 18.3% of userspace cycles are updating the in-memory representation, and

almost half of its userspace cycles are spent on IO-related work like logging to the WAL, creating LSM tree nodes, and garbage collecting stale data. Almost all of RocksDB's time in the kernel is IO related: taking file rangelocks, running file system-specific code, and updating the buffer cache with new file data.

The userspace IO handling code also adds to RocksDB's complexity. Approximately 40% of RocksDB's code is persistence related. This contributes to the complexity of the system and to bugs. Worst of all, bugs in the persistence code can lead to data loss and corruption [16].

Together these overheads present an opportunity for performance gains and improving the reliability of storage systems. Storage systems can benefit from consolidating the software stack through a better API.

**Single Level Stores** Single level stores (SLSes) [209, 226, 245] unify virtual memory and persistent storage to simplify application persistence. SLSes eliminate the dichotomy between memory and storage by saving incremental application checkpoints, which include the entire memory, OS and CPU state of a running application. SLSes recover from a crash by restoring the application from the on-disk image.

Checkpointing application state is complex and too slow for individual database transactions. For example, EROS checkpoints every 30 s while Aurora checkpoints every 10 ms on PCIe flash. TreeSLS [245] breaks POSIX compatibility and requires byte-addressable persistent memory DIMMs for low-latency persistence every 1 ms. All of these SLSes are still too slow and require serialization across all threads. To remedy the overhead, Aurora and EROS complement checkpointing with a WAL API, however, this undermines the single level store goal of eliminating application storage logic.

Aurora offers an intermediate API that persists single address mappings (regions), to reduce the amount of data checkpointed and avoid the overhead of OS state. However, region checkpointing still uses Aurora's underlying system shadowing mechanism, which requires the stopping of all threads resulting in a serialization point that induces high overheads. This API is still too slow for single transactions and thus requires a WAL.

Table 4.2 breaks down the performance overhead of region checkpointing for a single IO done by RocksDB for the MixGraph workload. Aurora's checkpointing is based on the "system shadowing" mechanism, which is responsible for 180 µs of latency out of 208 µs. System shadowing stops all threads, applies COW by creating a "shadow object", and resumes the threads in parallel with issuing IOs. "Shadow objects" are created to retain an atomic snapshot of memory while allowing threads to continue executing during the IO write. After the IOs complete, Aurora "collapses" the shadow object back into the base object. The operation merges the page lists of the shadow with those of the base and costs

| **Operation** | **Aurora** |
|---|---|
| Stopping Threads | 26.7 µs |
| Shadow Creation | 79.8 µs |
| Write IO | 27.9 µs |
| Shadow Collapse | 91.7 µs |
| Total | 208.1 µs |

Table 4.2: Latency breakdown for synchronous Aurora region checkpointing during RocksDB's `dbbench` benchmark. Most of the overhead comes from the shadowing mechanism that applies to entire mappings, even though the amount of dirty data is 64 KiB.

| System | Subset | Atomic | Per-Thread | ¡1 ms |
|---|---|---|---|---|
| `fsync` | No | No | No | Yes |
| `msync` | Contig. | No | No | Yes |
| atomic `msync` | Contig. | Yes | No | No |
| Aurora | Contig. | Yes | No | No |
| `memsnap` | Yes | Yes | Yes | Yes |

Table 4.3: Comparison between different persistence mechanisms. `fsync` and `msync` flush an entire file or region's dirty set without atomicity. Atomic `msync` requires contiguity and does not track the per-thread dirty set. Aurora is not microsecond-scale and does not track the dirty set per thread. MemSnap combines strong guarantees, performance and per-thread set tracking.

90 µs because its latency is proportional to the mapping's size. The frequency of executing region checkpointing is limited by the total time including collapsing.

**MemSnap: Our Approach** MemSnap is a novel approach to the single level store, which runs on commodity PCIe SSDs and subsumes the need for a WAL by addressing the three aforementioned problems of the file API with microsecond level performance.

MemSnap provides an API for persisting in-memory data to the disk without file operations or a WAL. MemSnap tracks the per-thread dirty set of the application at a page granularity, rather than a global view of the dirty set that requires serializing across all threads. The result is an API that fully delivers on the single level store vision of writing applications that solely modify memory and do not require additional logic for atomicity and persistence, and addresses the limitations of existing systems (Table 4.3).

| Command | Description |
|---|---|
| `int msnap_open(char *name, void **addr, size_t len, int flags)` | Create or open a region for snapshotting and return id |
| `epoch_t msnap_persist(int md, int flags)` | Persist the snapshot synchronously or asynchronously |
| `int msnap_wait(int md, epoch_t epoch)` | Wait for the snapshot to persist |

Table 4.4: The MemSnap API. Users either explicitly register page-aligned regions of memory into the snapshot or implicitly track them throughout execution. MemSnap atomically persists these pages. Threads that should block until the data persists do so using `msnap\_persist` with the `MS\_SYNC` flag or `msnap\_wait`.

## 4.3 Design

MemSnap replaces file I/O based persistence with a fast and intuitive API shown in Table 4.4. Users create/open memory regions that are used to hold the main dataset they wish to persist atomically. Users modify data in these regions then persist it with a single call. Applications do not track their dirty pages/blocks explicitly as MemSnap does this transparently.

Applications use `msnap_open` to open or create new memory regions. The call maps the region into the process address space and returns a descriptor `md`. Similar to POSIX shared memory descriptors, these are opaque descriptors, not files. Each MemSnap mapping has a unique address where it is mapped every time. This ensures that pointers in the persistent data set are valid across reboots. We use the high end of the address space for MemSnap mappings, for a total address space of 32 PiB for machines with 5-level page tables.

Applications use `msnap_persist` to atomically persist modifications, then continue executing. `msnap_persist` is synchronous by default, but is callable with an `MS_ASYNC` flag that makes the call return after initiating the IO. The thread can later wait for the IO to complete using `msnap_wait`.

Flags to `msnap_persist` also specify which part of the dirty set it should persist. Passing a region descriptor to the call persists only the pages belonging to the corresponding region. Passing a descriptor of `-1` persists all modifications across all regions. `msnap_persist` by default persists modifications made by the calling thread unless passed the `MS_GLOBAL` flag to persist modifications made by all threads.

On a `msnap_persist` call MemSnap creates a µCheckpoint to persist the dirty pages required by the call. To track dirty data MemSnap uses a page-granularity copy-on-write (COW) storage mechanism and the hardware TLB. MemSnap stores the data on its COW object store. Using MemSnap, application threads can safely modify memory even if it is currently being flushed. Applications use page-granularity locking or equivalent

mechanisms only for concurrency, while MemSnap handles the atomicity of persistence-related IO. MemSnap's asynchronous mode lets a thread unlock the data in memory after `msnap_persist` to unblock other transactions.

Our design introduces three novel mechanisms that span the virtual memory and storage subsystems. First, the usage of a custom page handling technique that uses the hardware TLB and allows for the transparent tracking of dirty data for each thread. Second, our unified COW mechanism, which bridges the virtual memory and storage systems by transparently performing page table modifications in response to concurrent accesses to flushing data. Third, a fast COW storage system that solely provides direct IO and namespace requirements for our persisted objects.

**Hardware Assisted Dirty Set Tracking** Our dirty page tracking mechanism tracks the dirty set of each application thread individually across all memory mappings. We use the TLB hardware to track page writes by read protecting pages and trapping writes using a custom page fault handler.

MemSnap uses minor write faults to track dirty pages. All pages in a MemSnap memory region are initially read-only. MemSnap marks the region as writable (but not COW) and the PTE (page table entry) of each page as readable. This mapping configuration is unique to MemSnap and does not interfere with existing code paths in the VM fault handler or with swapping. When an application writes to a page, it takes a minor page fault. The page fault handler appends the dirty page to a thread-local list if the page is not already being tracked. MemSnap tracks the working set at the physical page level to avoid the overheads of tracking the write set of each mapping separately, e.g. using shadowing (Section 4.2).

The minor write fault has a lower cost than a regular COW fault because no page copy is necessary. The overall cost of a write fault is a small part of the total cost of persistence. Every page faulted is eventually committed to the disk, which is a much larger cost than the write fault itself.

MemSnap constructs the µCheckpoint using the threads' page list. The `msnap_persist` call goes through the list to create and issue a scatter/gather IO (i.e., vectored IO).

MemSnap reapplies read protection to each flushed page after writing, so subsequent writes to a page will attach it to a new µCheckpoint. `msnap_persist` reapplies read protection efficiently by going through the thread-local page list of the caller and marking the PTEs corresponding to the page as read-only. To handle multiprocess applications, MemSnap uses the page's physical-to-virtual mappings to find all page tables that contain it and modify their PTE. For small checkpoints MemSnap then issues a TLB shootdown for the dirty pages, or invalidates the entire TLB for larger working sets.

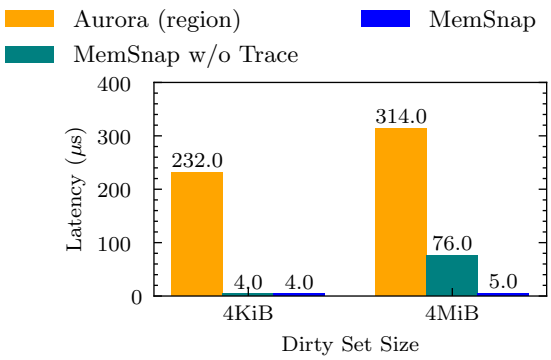



Figure 4.1: Comparison of the three techniques for marking pages as read only. Traversing the mapping's page tables is expensive even for small dirty sets, and traversing the page table for each page adds overhead for large dirty sets. MemSnap's trace buffer is both fast and scalable.

MemSnap's design minimizes the time necessary for reapplying read protections. MemSnap does not stop other threads to read protect the dirty set because it does not modify the address mappings themselves, unlike existing `fork`-based read-protection routines that directly manipulate the address space. The dirty per-thread page list also means that MemSnap only reapplies protection to the pages added to the checkpoint instead of the entire address space. This is done instead of the default routine which scans the entire page table region for the mapping to find dirty data.

To reduce the costs of traversing the page table, MemSnap records the physical address of the PTE during the page fault and stores it in a thread-private trace buffer. The OS is guaranteed not to move the PTE entry so the mapping stays stable. When reapplying page protections after a µCheckpoint we avoid traversing the page tables from the root by locking them, traversing the trace buffer and directly modifying the stored PTEs. This optimization eliminates the need for multiple page table traversals to reapply COW due to dirty pages being sparsely spread throughout the table.

We show the performance gains of our design in Figure 4.1. The baseline system traverses the page tables of a 1 GiB memory mapping to find and protect the dirty pages, and has large overheads even when protecting a single 4 KiB data page. Moving to a page-based approach to page protection reduces the cost of protecting a single page but its cost scales for larger dirty sets like 4 MiB. MemSnap's trace buffer reduces the cost of page protection to almost nothing.

MemSnap manages contention for hot pages through a per-page COW mechanism.

Avoiding locking the pages in the µCheckpoint is important to prevent contention with userspace threads, e.g., for the root of a tree data structure.

MemSnap sets a new "checkpoint in progress" flag in the physical page structure (`vm_page`) to signify the page as busy. Once complete, MemSnap returns to the calling thread. All threads can continue to read and write pages as they are being flushed to the disk. Writes to pages that have the checkpoint in progress flag trigger a COW path that duplicates the original page and updates the mapping and page tables to point to the new copy. This allows MemSnap to create an atomic checkpoint, while avoiding the cost of copy-on-write operations on all pages. Pages that do not have the checkpoint in progress flag are handled by the dirty set tracking fault code path.

**Persisting MemSnap Regions** MemSnap uses a COW object store to persist µCheckpoints. The API of the store is similar to the internal APIs of popular COW file systems [77, 136, 201, 55], and these file systems are potential backends for the MemSnap mechanism if their API is extended to expose lightweight object checkpoints. Our object store is a simple implementation of the minimum viable design required to support MemSnap.

The store uses COW radix trees instead of the more complex, read optimized, COW B-Trees used by many storage systems [202]. The object store optimizes for random access patterns and small object sizes and heavily prioritizes writes over reads. Radix trees work well for the write patterns generated by the workload because they are block based and do not suffer from the extent fragmentation problems that B-Trees have if snapshotted frequently.

The object store has a key-value interface and is not burdened by the traditional file system API or POSIX standards. The object store does not integrate with the buffer cache and instead does direct IO, deferring the management of memory pressure to the VM page cache. The store only allows memory mapping the data and does not support file IO.

Each object maintains its own logical history. An on disk object contains a monotonic epoch number that increments after each µCheckpoint that the object is a part of. This allows for restores independent of the state of the object store. Further, it allows for µCheckpoints to occur concurrently with each other as objects are not tied to some global epoch.

MemSnap's OS/workload co-design reduces the cost of persistence close to the latency of the disk without any optimizations from the user at all. Table 4.5 shows the cost of persistence for a single 64 KiB transaction done by the RocksDB workload we presented in Section 4.2. An `msnap_persist` call from RocksDB takes 51.4 µs, only 7 µs more than the latency of 44 µs for a direct disk IO of the same size. Most of the additional latency is in

| Operation | Overhead |
|---|---|
| Resetting Tracking | 5.1 µs |
| Initiating Writes | 6.5 µs |
| Waiting on IO | 39.7 µs |
| Total | 51.4 µs |

Table 4.5: Breakdown of an `msnap\_persist` call for 64 KiB worth of dirty pages. The call protects the pages to reset tracking in 5.1 µs, and spends 39.7 µs waiting for the IO to complete. Minimizing COW application time quickly frees up the modified pages to be accessed and modified by other threads and thus avoids contention.

resetting tracking to the pages by modifying their PTEs, a fundamental cost for dirty page tracking. The other 45.2 µs are spent on the actual IO, same latency as a direct IO.

## 4.4 MemSnap-based Crash Consistency

We now present how to adapt an application to the MemSnap API. First, we describe how MemSnap replaces the file interface while leaving its ACID guarantees intact. We then enumerate the key properties that the in-memory data structures must adhere to. Finally, we show how these properties combine with the data structures' existing locking mechanisms to guarantee on-disk consistency.

MemSnap modifies the lowest level of the database's software stack and leaves most components unmodified. Databases split their logic into a storage engine that manages the data across memory and files, and an upper layer that implements abstractions like tables and transactions. The upper layers manage high level data modifications and transactional locking and do not directly interact with on-disk data, instead using the storage engine for serialization and disk access. Enabling MemSnap only changes the storage engine's persistence calls and leaves the upper-layer logic as-is.

We modify the storage engine to use MemSnap regions instead of files. The unmodified storage engine stores the data in files for persistence, and also caches part of it in an in-memory data structure, e.g., a B-tree, for fast access. We expand the in-memory data structure to hold all data and place it in MemSnap persistent regions, removing the files.

MemSnap flushes write transactions by tracking the changes it makes to the data structure's memory, and writing the changes as a µCheckpoint to disk. The unmodified storage engine handles writes by first logging the data in a WAL, then modifying the

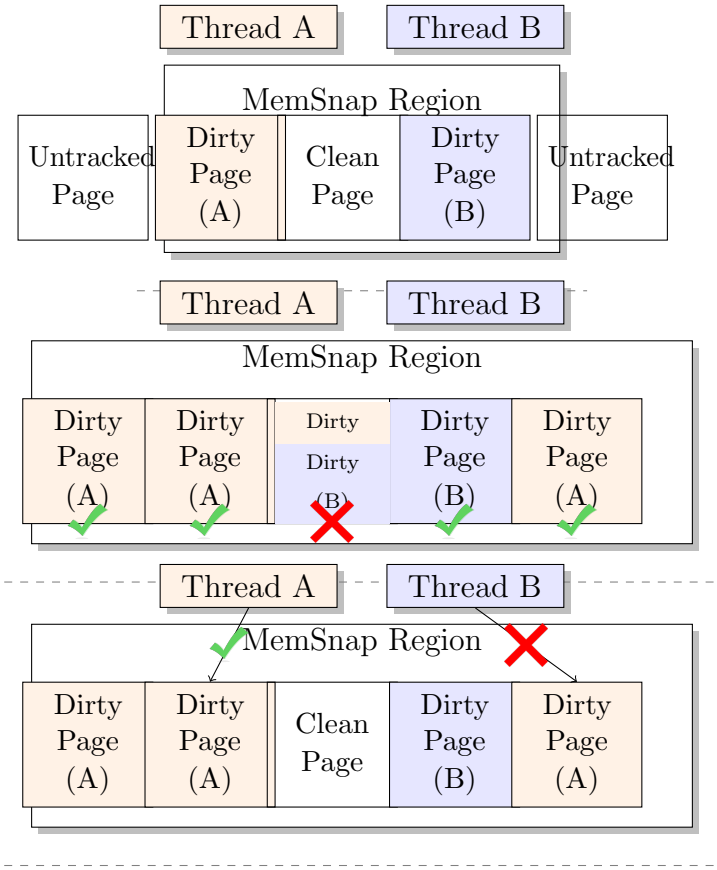



Figure 4.2: The three conditions required for MemSnap-based persistence. ① MemSnap persists only dirty pages that are in a MemSnap region. ② Two threads cannot simultaneously dirty different parts of the same page. ③ A page dirtied by a thread cannot be dirtied by another until it is flushed out.

in-memory copy. We replace WAL writes with MemSnap's dirty page tracking, and replace WAL `fsync` with `msnap_persist` calls.

We do not modify the storage engine's concurrency control. We call `msnap_persist` where `fsync` was previously called. The calls provide the same persistence guarantees, so the surrounding code's assumptions remain valid.

**Persisting the In-Memory Representation** MemSnap uses its COW-based dirty page tracking to determine the contents of each transaction's corresponding µCheckpoint. For a µCheckpoint to correctly update the on-disk copy, it must include all the data

structure pages dirtied by the transaction that created it. The µCheckpoint must not include any pages dirtied by other concurrently executing transactions to avoid persisting uncommitted data through a crash.

An example is a B-tree updated by a write transaction with new keys. The corresponding µCheckpoint must persist the new key-value pairs, and any B-tree nodes dirtied by the operation. The µCheckpoint must also not persist any changes made to the B-tree by other transactions.

The in-memory data structures must adhere to three properties to ensure MemSnap's correctness (Figure 4.2):

① All data falls within MemSnap persistent regions for compatibility with our API. We enforce this property by allocating the in-memory data structure from MemSnap regions.

② Two key-value pairs or data structure nodes cannot be on the same OS page. By nodes we refer to the individual components of the data structure, e.g., a B-tree node or a skip list node. We ensure this property by making each node or key value pair of a data structure page-aligned, if this is not already the case. Two transactions with non-overlapping write sets will then write to non-overlapping pages.

③ A node cannot be simultaneously modified by multiple active transactions. When a transaction modifies a key, it must lock it to prevent others from writing to it until it is persisted. This is already done by some storage engines, e.g., SQLite. However, some data structures like RocksDB's skip lists require additional fine-grained locking (Section 4.7.2). This extra locking does not change data structure semantics.

These properties combined ensure that we update the on-disk dataset in a *serializable* fashion, i.e., our persistence operations form a series of transactions applied to the image one by one. Property ② ensures that MemSnap's page tracking adds to a thread's working set all the data it has dirtied, and never includes unrelated dirty data. Property ③ ensures that pages added to the dirty set were initially clean and cannot be written until the transaction commits to the disk, so the µCheckpoint does not include other transactions' changes. MemSnap's `msnap_persist` calls thus write out all data structure modifications done by a single transaction.

MemSnap applications restore after a crash by mapping persisted data back into memory. The new instance first recreates components e.g., transaction managers, allocators, lost during the crash. The instance then retrieves a list of all MemSnap regions in an application, along with their address in memory. It recreates these mappings and uses memory accesses to page in the data. All pointers in the mappings that point to persistent data are still valid after the crash because regions are always mapped at the same address. The application also updates stale pointers to lost volatile state.

**MemSnap Region Crash Consistency** All MemSnap regions are crash consistent due to our COW object store. Each MemSnap region is an object in the store and its data is stored in a radix tree. The object store uses COW to update data without overwriting it. Every µCheckpoint first writes the data to newly allocated space. Because of COW the object store then creates a copy of the leaf node that points to the new data, then a copy of that node's parent that points to the new node, and so on until it writes a new tree root. Changes commit after the new root persists.

Region data is consistent after a crash. Interrupted transactions and µCheckpoints do not persist and the application recovers from the last completed µCheckpoint.

## 4.5 Scope and Limitations

MemSnap handles persistence and does not address memory overcommit. The file API's semantics for memory-mapped data is problematic both for persistence, and for tiering larger-than-memory datasets between disk and memory [98]. MemSnap leaves the question of a memory-mapped overcommit API out of scope, much like the other SLSes, because it is orthogonal to its core goal of efficient persistence. MemSnap does not limit the design of new overcommit APIs. The lack of an overcommit API however, limits our MemSnap case studies to datasets that fit in memory.

MemSnap is designed with page-aligned data structures in mind and causes write amplification with data structures that use small key-value pairs. MemSnap flushes at a 4 KiB page granularity regardless of how much data in the page is actually dirty, incurring disk write amplification. MemSnap also requires transactions to lock the dataset at a page granularity. Buffer caches already do this, but for systems like key-value stores, changes are required to either use a single per-page write lock for all key-value pairs in the same page, potentially losing parallelism, or placing each key-value pair in its own page, causing space amplification.

## 4.6 Evaluation

**Implementation** MemSnap takes up 2.5 KSLOC of kernel code for tracking pages, initiating the IOs and enforcing COW on the pages being written out. We implement these mechanisms on top of FreeBSD 12.3. We built MemSnap's storage system is 3 KSLOC with 1 KSLOC for radix trees while the rest is for the object store's logic.

We wrote 500 SLOC for SQLite, 400 SLOC for RocksDB, and 1 KSLOC for PostgreSQL, porting each to MemSnap. MemSnap makes redundant 7 KSLOC in SQLite, 40 KSLOC in RocksDB, and 10 KSLOC in PostgreSQL.

In this section we analyze MemSnap's performance and show why it is faster than file-based persistence mechanisms and Aurora's checkpointing. First we characterize the `msnap_persist` call and compare its performance against `fsync` and Aurora's region checkpointing. We then show that MemSnap has major benefits for three real world case studies: SQLite, RocksDB, and PostgreSQL. We present the databases and the changes necessary to use MemSnap, and evaluate their benefits in terms of performance and code complexity.

For all benchmarks we use a dual Intel Xeon Silver 4116 CPUs (Skylake-SP) running at 2.1 GHz with 96 GiB of memory. For storage, we use two Intel 900P PCIe SSDs striped together in 64 KiB blocks.

**Comparing Persistence APIs** MemSnap persists data at a lower latency than the file API and region checkpointing. We compare `msnap_persist` against `fsync` and Aurora by using them to persist writes with access patterns that we see in our case studies. We test `fsync` under both the FFS and ZFS file systems to show that `fsync`'s overheads are consistent across file systems. FFS [168] uses soft updates [169] and journaling for consistency, while ZFS uses copy-on-write. We show that MemSnap has lower latency than Aurora's region checkpointing even though both checkpoint the same amount of data.

Table 4.6 shows the latency of different persistence APIs for sequential and random writes. The microbenchmark flushes dirty data using `msnap_persist`, `fsync`, or direct disk IO. Random write patterns are seen during database checkpointing. Sequential write patterns are prevalent during write ahead logging and LSM-Tree writes. We also measure direct IO to the striped disks with one outstanding IO.

We evaluate MemSnap with the random IO workload and measure the synchronous and asynchronous persistence latency. Asynchronous latency is the CPU time spent on reapplying page protections to each dirty page. Synchronous latency is the total end-to-end time for persisting a transaction to the disk. We evaluate `msnap_persist` only with random writes as it persists modifications to primary data made in-place.

Table 4.6 shows that MemSnap outperforms both FFS and ZFS in sequential and random workloads. This is due to MemSnap's optimized dirty page tracking and COW object store, which translates random object updates into sequential writes on disk. MemSnap's random IO latency (sequential on disk) outperforms our direct disk IO measurement for large writes because having more outstanding IOs results in better saturation of the disk's throughput.

**MemSnap vs. Aurora** Figure 4.3 shows that MemSnap is up to 60× faster than Aurora's application checkpoints and 7× faster than its region checkpoints. We show this by synchronously persisting a randomly distributed dirty set using MemSnap and Aurora's application and region checkpointing then measure the total latency of the call. We keep all

| Size (KB) | Disk (µs) | fsync (µs) Sequential | fsync (µs) Random | memsnap Random (µs) |
|---|---|---|---|---|
| | | FFS—ZFS | FFS—ZFS | Sync.—Async. |
| 4 | 17 | 70—64 | 156—232 | 34—6 |
| 8 | 18 | 79—71 | 252—371 | 36—6 |
| 16 | 22 | 89—80 | 464—706 | 41—6 |
| 32 | 31 | 111—134 | 828—1.4K | 48—6 |
| 64 | 44 | 134—137 | 1.9K—2.9K | 50—6 |
| 128 | N/A | 164—204 | 4.3K—7.8K | 70—9 |
| 256 | N/A | 218—347 | 8.8K—11.7K | 112—13 |
| 512 | N/A | 338—672 | 12.6K—15.6K | 168—23 |
| 1024 | N/A | 581—937 | 17.9K—18.2K | 297—36 |
| 2048 | N/A | 1.1K—1.7K | 23.5K—20.2K | 552—57 |
| 4096 | N/A | 1.9K—3.0K | 33.7K—30.9K | 1.0K—108 |

Table 4.6: The latency of persistence for `fsync`, `msnap\_persist`, and Aurora. We use `fsync` after writing out a random or sequential series of 4 KiB pages and compare its latency with that of `msnap\_persist` for a random IO pattern of the same size. MemSnap outperforms disk writes for large write sizes because disk writes always have one outstanding IO.

data in a single mapped region because region checkpoints cannot handle dirty sets spread across multiple mappings.

Region checkpointing provides data persistence like MemSnap does, but uses object shadowing and collapsing to manage tracking the memory region. Aurora's application checkpointing overhead is an order of magnitude larger than region checkpointing because it must protect the entire address space and has larger post-checkpoint cleanup operations that add latency. MemSnap has 70% the latency of region checkpointing as it avoids the overhead of system shadowing.

## 4.7 Case Studies

To show how MemSnap is a general purpose tool for persistence we apply our API across three databases: SQLite, RocksDB, and PostgreSQL. Databases are unique applications as they stress many edge cases of a persistence API while demanding performance. These databases are chosen as they represent different database designs. SQLite is a read optimized

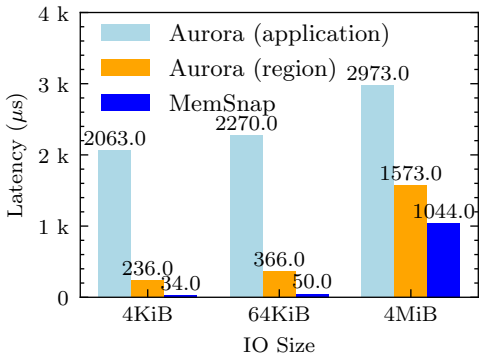



Figure 4.3: Comparison of MemSnap with Aurora's region and application checkpointing. We measure the latency of a synchronous persistence operation for randomly distributed dirty sets of different sizes. MemSnap outperforms Aurora's region checkpointing by 7× for small IOs because it processes individual pages.

single writer SQL database. RocksDB is a write optimized key/value store that uses LSM-Trees. PostgreSQL is a multi process system that uses multi version concurrency control (MVCC) for highly concurrent transactions.

### 4.7.1 Case Study: SQLite

The MemSnap integration with SQLite shows how the MemSnap mechanism speeds up database persistence without changing the design or requiring major modifications to the database. SQLite is a popular [58] database that is embedded into larger applications to provide data persistence. We present SQLite before and after adding MemSnap, show the equivalence of the two versions, and compare performance.

SQLite's architecture follows the general layered designed outlined in Section 4.4. The database models each table as a B-tree with a key-value pair for each row, and the database's upper layer translates SQL statements down to API calls on the B-tree's keys. The API includes locking and persistence. For example, upper layers isolate transactions by locking keys to prevent uncommitted values from being visible.

The SQLite storage engine implements the B-tree API and moves data between the WAL and the database (DB) file. The storage engine `mmap`()s both the DB file and the WAL into memory. The WAL doubles as a cache for the DB file and services reads and writes. The engine also includes a lock manager for the WAL and DB file's contents.

Baseline SQLite writes B-tree nodes dirtied during a transaction to the WAL, and calls

`fsync` on it when the transaction ends. When the WAL becomes large from all the logged data, SQLite checkpoints by copying its contents into the DB file. SQLite `fsync`s both files, then truncates the WAL.

To integrate MemSnap with SQLite, we developed an SQLite plugin that replaces file IO with MemSnap equivalents. The plugin is loaded at runtime, requiring no modification or recompilation of SQLite. The plugin is 347 SLOC and replaces the standard 4.8 KSLOC Unix file module.

MemSnap SQLite persists changes directly in the DB file and skips persisting the WAL with `fsync`. In our system, the storage engine backs the WAL in volatile memory and the database with a MemSnap persistent region. During a commit we move the transaction's dirty set from the WAL to the persistent region, and flush with `msnap_persist`.

Our change retains the storage engine and lock manager's logic. We still use the "WAL" as a cache, and when we flush the transaction data to the persistent region we reuse the calls that baseline SQLite uses to flush the WAL to the DB file during a checkpoint. To the upper layers of SQLite, the MemSnap plugin semantically is identical to a checkpoint occurring after every transaction. However, checkpointing frequency only impacts performance and does not degrade transaction semantics or crash consistency.

Our system satisfies all properties in Section 4.4:

① We have moved the main database from a DB file to a persistent region. The main database is already accessed through memory-mapping, so this change does not affect the rest of the code.

② SQLite already stores data in database page-aligned chunks, and locks them at the same granularity. SQLite thus already prevents two transactions from dirtying the same page. We configure the size of each database page to be 4 KiB, to make SQLite lock data at the same granularity with which MemSnap tracks the dirty set.

③ SQlite already uses the lock manager to uphold its ACID guarantees and crash consistency. Our changes do not affect the lock manager, so MemSnap's µCheckpoints do not include uncommitted modifications by other threads.

**Evaluation** We first compare MemSnap to the baseline by measuring the number and latency of persistence related system calls made during the `dbbench` workload. `dbbench` generates up to 1M keys with 128 byte values. Key/value pairs are batched sequentially or randomly into write transactions ranging from 4 KiB to 1 MiB in size until 2 million total key value pair writes have been performed. Checkpointing is configured to occur every 4 MiB worth of dirty data as is the default [60].

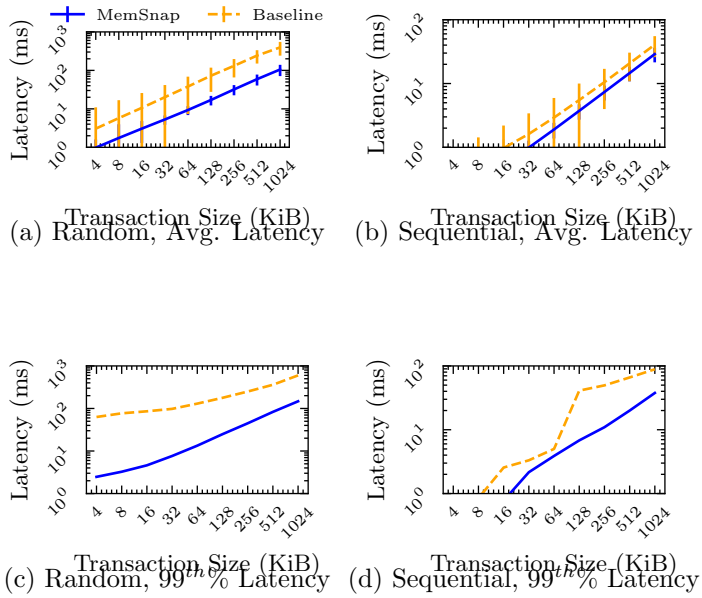



Figure 4.4: Performance of MemSnap vs the WAL+Checkpoint baseline for transaction sizes from 4 KiB to 1 MiB. The actual IO done is larger than the transaction size because of write amplification. The latency for both configurations scales linearly with transaction size, but MemSnap is faster both for average and tail latency across all transaction sizes. MemSnap has a fixed per-4 KiB page latency across all transaction sizes, while the baseline's latency rises for smaller sizes.

Table 4.7 shows the latency and number of system calls for MemSnap versus the file API baseline. We show the average call latency and total calls made for each system call over several transaction sizes. SQLite makes one `fsync` call on every transaction that ranges from 4 KiB to 1 MiB in size, and an fsync call on every checkpoint after 4 MiBs of writes.

The table shows that MemSnap does fewer and faster calls than the file API, lowering the cost of persistence. The number of MemSnap calls is equal to the number of transactions, while the number of `fsync` calls is equal to the number of transactions and checkpoints made during the benchmark.

The number of checkpoints made by the baseline is larger due to write amplification of the WAL. SQLite implements the WAL such that when any block is dirtied through a write, the block is appended to the WAL. This means every 128-byte value written is

| Transaction Size | memsnap | | fsync | | write | | read | |
|---|---|---|---|---|---|---|---|---|
| | Latency | Total Ops | Latency | Total Ops | Latency | Total Ops | Latency | Total Ops |
| **Random IO** | | | | | | | | |
| 4 KiB | 152.2 µs | 63.1 K | 1137.3 µs | 67.4 K | 6.7 µs | 7584.1 K | 2.9 µs | 2846.8 K |
| 64 KiB | 1621.0 µs | 3.9 K | 10024.4 µs | 6.2 K | 6.7 µs | 6703.2 K | 2.9 µs | 2729.7 K |
| 1024 KiB | 17521.8 µs | 0.2 K | 41521.7 µs | 0.9 K | 6.7 µs | 4127.1 K | 2.9 µs | 2046.8 K |
| **Sequential IO** | | | | | | | | |
| 4 KiB | 51.0 µs | 59.4 K | 155.6 µs | 63.5 K | 6.7 µs | 1174.7 K | 3.1 µs | 94.6 K |
| 64 KiB | 97.7 µs | 3.5 K | 471.4 µs | 3.8 K | 6.7 µs | 247.3 K | 2.9 µs | 90.2 K |
| 1024 KiB | 564.8 µs | 0.2 K | 3429.1 µs | 0.3 K | 6.7 µs | 176.1 K | 2.8 µs | 88.0 K |

Table 4.7: The number and cost of persistence-related system calls for both MemSnap and the baseline. MemSnap only requires the `msnap\_persist` call that is both less frequent and cheaper than `fsync`. The baseline in contrast uses WAL and checkpoint that generates a significant amount of file IO.

| **Baseline** | **%CPU** | **MemSnap** | **%CPU** |
|---|---|---|---|
| **Random IO** | | | |
| userspace | 1.58% | userspace | 10.51% |
| fsync | 29.15% | memsnap | 9.31% |
| write | 30.34% | memsnap flush | 8.57% |
| read | 3.92% | page faults | 30.90% |
| **Wall clock time** | **175s** | | **35.4s** |
| **Sequential IO** | | | |
| userspace | 24.63% | userspace | 60.34% |
| fsync | 14.33% | memsnap | 3.15% |
| write | 26.62% | memsnap flush | 2.74% |
| read | 2.51% | page faults | 10.38% |
| **Wall clock time** | **12.5s** | | **7.2s** |

Table 4.8: CPU usage and total `dbbench` execution time for MemSnap and the baseline. MemSnap runs 2× to 5× faster than the baseline while doing a similar amount of work in userspace.

amplified to a 1 KiB write into the WAL resulting in more checkpoints. Every write to a block causes a write to the WAL even during the same transaction.

We show the average call latency for different transaction sizes in Table 4.7. Write

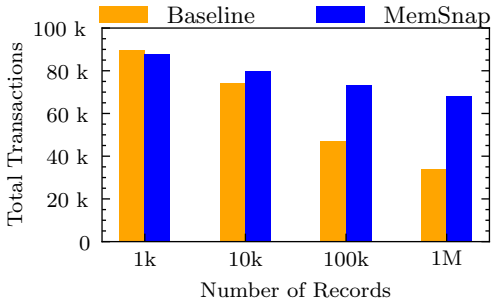



Figure 4.5: TATP benchmark. The baseline slightly outperforms MemSnap for small database sizes by because of its WAL index, but heavily degrades in performance for larger databases.

amplification to the WAL causes the total IO to exceed the transaction size and further increases the performance benefit of MemSnap. MemSnap's cost scales linearly with the transaction size, with 64 KiB transactions having 16× the cost of a 4 KiB transaction. `fsync` calls are on average an order of magnitude slower compared to MemSnap for 4 KiB transactions, because the average `fsync` latency is skewed by checkpoint `fsyncs`.

Both systems use `memcpy` to update the in-memory data and file IO is only ever done for the baseline. The `write` and `read` calls have low latency as they are serviced from the OS buffer cache, but are numerous and so add significant latency. MemSnap does not do `write`/`read` system calls.

Figure 4.4 shows average and $99^{th}$ percentile latency. MemSnap has 4 times lower latency with low variance, while the baseline's variance increases as smaller transactions incur less latency while checkpointing latency remains constant and skews the average. The performance gap is even larger for random transactions. MemSnap does sequential IO even for random in-memory writes, while the baseline's IO patterns are affected by the writes' sequentiality. All configurations are subject to write amplification as values are 128 bytes long, but pages and FS blocks are both 4 KiB.

**CPU Usage** MemSnap-based SQLite increases throughput because it reduces kernel time. Table 4.8 shows the CPU usage breakdown of the `dbbench` workload. The table shows that MemSnap spends 10% of its CPU time in userspace for random IO, and 60% for sequential. This higher ratio is because aggregating the dirty page set into a µCheckpoint takes less than 1% of total CPU time (i.e., `memsnap`).

MemSnap's speedups over the baseline are due to cheaper persistence-related calls in SQLite's critical path. The baseline spends 25% of its CPU time on writes to the WAL,

and 15-30% of its CPU time on the `fsync` calls required for persistence. MemSnap's cost in contrast mainly stems from the page faults required to track the dirty set, about 30% of CPU time for random access and 10% for sequential due to locality.

**TATP Benchmark** We demonstrate MemSnap throughput benefits using the TATP database workload [59]. TATP is used by SQLite's authors to evaluate its performance for real-world database workloads [116]. These workloads have a 80% read, 20% write transaction mix and span across multiple tables. All write transactions commit synchronously to the disk. We run the benchmark for 60 seconds using different database sizes (1 K to 1 M records as are typical [116]), and measure the total transactions done.

MemSnap consistently outperforms the baseline with the difference increasing for larger database sizes. MemSnap sees a 23% reduction in throughput as it increases the database size from 1 k to 1 M records, while the baseline system sees a 63% reduction. The baseline system's `fsync` costs increases with the resident size of the mapping and not just the dirty set. This behavior is consistent across OSes including Linux and FreeBSD. MemSnap's overhead is independent of the resident size of the mapping and so its throughput stays more consistent. Both systems lose throughput because larger databases include userspace overheads, e.g, SQLite's internal B-Trees become slower because their height increases.

### 4.7.2 Case Study: RocksDB

The main challenge of integrating RocksDB with MemSnap is its in-memory skip list data structure, which was not designed to be written to the disk. This is in contrast to our other workloads where MemSnap persists the data under the same on-disk format as the original database. We optimize the skip list for MemSnap and ensure crash consistency.

MemSnap replaces RocksDB's WAL-and-checkpoint mechanism. RocksDB distributes data between a WAL file, in-memory skip lists called MemTables, and *SSTable* files. `Put` calls in RocksDB write new KV pairs to the WAL, persist them with `fsync`, then add them to the skip list. When the WAL gets too large, RocksDB checkpoints the skip list by serializing it into a new file. Our goal is to remove WAL logging and serialization and use MemSnap-based persistence.

We remove the WAL and use MemSnap to directly persist the skip list, merging both their functionality into a single data structure. Skip lists are ordered singly linked lists where each node holds one KV pair. Nodes have extra *skip pointers* that skip ranges of keys, speeding up random searches.

To satisfy Property ①, we move the skip list from volatile memory to a persistent MemSnap region. We also add an `msnap_persist` call at the end of the skip list's `Insert`

call made by RocksDB's `Put`. Our changes retain the persistence semantics of the `Put` call as `Insert` happens after the WAL logging and within the same critical section.

Changes to the skip list retain RocksDB's transaction semantics. The skip list's own internal locking only preserves its own integrity, not the database's ACID guarantees. RocksDB implements range queries and transactions on top of RocksDB's snapshot versioning and lock management mechanisms respectively, and we do not modify either.

We adjust the skip list's internal locking to prevent threads from being able to overwrite unpersisted modifications, satisfying Property ③. The skip list's `Insert` operation updates linked list pointers in existing nodes using compare-and-set (`CAS`). To prevent an uncommitted node from being modified before the `msnap_persist` call we replace the `CAS` operation with a per-node spinlock. This introduces minimal performance overhead, in the order of a few dozen cycles.

As an optimization, we do not persist skip pointers because we only need to guarantee the crash consistency of the underlying linked list. For each write we then persist only the new node and its previous in the list. Skip pointers speed up searches, and act as an index on top of the linked list structure. We can recreate this index after a crash by traversing the restored linked list.

Finally, we adjust the size of each node to be compatible with MemSnap's page tracking and satisfy property ②. We adjust the node size to 4 KiB to align them with MemSnap's page tracking mechanism. This makes it possible to track each KV pair separately using our dirty page tracking at the cost of additional write amplification.

**Crash Recovery** Interrupted µCheckpoints are automatically cleaned up after restoring. MemSnap-RocksDB recovers after a crash by restoring the MemSnap region from the on-disk state. The object store updates its data using COW, so interrupted µCheckpoints cannot overwrite persisted data.

RocksDB creates a single µCheckpoint for each transaction commit, so the on-disk data includes no uncommitted writes. We use RocksDB's `WriteCommitted` configuration that writes data to the MemTable only when committing a transaction, using a single `MultiPut` to write all changes.

To restore, the RocksDB instance retrieves the on-disk regions in the object store and maps these regions to their specified in-memory location to ensure that the linked list pointers are valid. It then traverses the linked list nodes to recompute skip pointers. The restored RocksDB skip list is consistent and includes all data committed before the crash.

We run a test to ensure that in-memory transactions remain properly atomic, isolated, and crash consistent. The test starts with an initialized database containing 100 k random

key-value pairs and running 20 threads. Each thread creates a transaction that randomly selects 100 keys and increments each of their values, and repeats 100 k times. We then verify the consistency of the dataset by summing all values.

We re-run the above test to verify that the database is crash consistent, by crashing with a kernel panic during the test. We load the on-disk data to a new instance and verify that the values sum up to the correct amount, by referencing acknowledged transactions that occurred before the crash.

**Effects of Removing the LSM Trees** Our design removes the on-disk *LSM tree* that RocksDB uses to tier data on the disk, by placing all data in a single MemTable (i.e skip list). This choice was made because the LSM tree's main advantage is automatic tiering for larger-than-memory workloads that we do not consider in this work. Removing the on-disk LSM tree has the advantages that it reduces application complexity and removes the need for compaction.

MemSnap integration does not fundamentally require using a single MemTable or removing the LSM tree. Alternative designs can periodically swap out MemTables to generate multiple smaller on-disk regions and tier them into an LSM tree in the same way that the baseline creates an LSM tree out of SSTable files.

**Evaluation** We compare a MemSnap-RocksDB system with a system that uses Aurora's region checkpointing. The Aurora system stores all MemTable data in a single mapping and issues a `checkpoint` after each write. On a checkpoint call Aurora stops all application threads and atomically creates an incremental checkpoint of the region, then resumes all threads except for the caller and synchronously flushes the data. We modify Aurora to synchronously persist region checkpoints so that region checkpointing provides identical guarantees to MemSnap and baseline RocksDB.

We evaluate MemSnap with RocksDB using the Meta's MixGraph workload. MixGraph replicates usage patterns seen in large-scale deployments and is commonly used to evaluate RocksDB [86]. The workload is composed of 84% `Get`, 14% `Put`, and 3% `Seek` requests and represent queries and updates to Facebook's social media graph. For this workload RocksDB persists writes synchronously. We fill the database with 20 M 48-byte keys - 100-byte value pairs (3 GiB). Keys are chosen uniformly, while writes are chosen using a generalized Pareto distribution. We run with 12 threads, and measure total throughput, average and tail latency.

We benchmark our system using three different configurations. The first uses MemSnap for persistence after every write to the MemTable. The second uses the unmodified RocksDB code that uses a WAL for single write persistence and creates a new SSTable file every

| Configuration | Metric | | |
|---|---|---|---|
| | **Kops** | **Avg(µs)** | **99th(µs)** |
| `memsnap` | 420.7 | 138.9788 | 239.62 |
| **Baseline+WAL** | 388.0 | 162.7709 | 248.44 |
| `Aurora` | 91.8 | 751.9326 | 4.2K |

| System Call | Metrics | |
|---|---|---|
| | **Latency (µs)** | **Total Count** |
| `memsnap` | 51.4 | 208.1K |
| `fsync` | 63.1 | 190.4K |
| `write` | 19.4 | 190.6K |
| `checkpoint` | 204 | 88.6K |

Table 4.9: RocksDB latency comparison between MemSnap RocksDB, a WAL-based baseline, and Aurora's region checkpointing. MemSnap provides 3× lower latency than Aurora as it persists data with a single call. MemSnap's costs scale down with the dirty set, unlike Aurora's region checkpoints.

64 MiB worth of writes. The third configuration uses Aurora's region checkpointing which checkpoints after every write to the database.

Table 4.9 compares the overall performance of the three systems and shows how MemSnap's low latency translates to better total throughput. MemSnap has significantly lower latency than the baseline system, both in the average and $99^{th}$ percentile. The lower latency directly translates to higher overall throughput for the workload because persistence-related overheads are the main bottleneck. The table also demonstrates that Aurora's region-based persistence has unacceptable overheads for multi threaded applications. Checkpointing an address space region with Aurora reduces overall throughput by 75% as region checkpointing includes a fixed-cost overhead that does not scale down with the dirty set.

Table 4.9 explains the performance differences between the three systems by measuring the number and latency of persistence-related calls. Persistence-related calls make up for the majority of end-to-end operation latency, while the total number of operations done heavily correlates with the overall throughput of the benchmark. MemSnap uses a single `msnap_persist` call to persist the modified data, while the baseline uses a `write` call immediately followed by an `fsync`, and Aurora uses a `checkpoint` call. The `msnap_persist` call has an average latency of 51 $\mu$s compared to the cumulative latency of `write` and `fsync` of 82.5 $\mu$s. The `fsync` call is costlier than `msnap_persist` because of its more complex code

| Operation | Time (µs) MemSnap | Aurora |
|---|---|---|
| Waiting for Calls | N/A | 26.7 |
| Applying COW | 5.1 | 79.8 |
| Flush IO | 46.3 | 27.9 |
| Removing COW | N/A | 91.7 |
| Total | **51.4** | **208.1** |

Table 4.10: Breakdown of MemSnap vs Aurora's persistence cost. Aurora's region COW tracking causes 80% of its total latency. MemSnap's page tracking avoids these overheads. MemSnap also avoids contention between concurrent persistence operations compared to Aurora.

path that includes traversing the buffer cache and initiating the write in the file system.

Figure 4.10 shows MemSnap is 4× faster than Aurora's `checkpoint`. This is because the cost of region checkpointing scales with the size of the memory region and the cleanup routines must run after the IO completes. These routines are associated with Aurora's "system shadowing" COW mechanism and are unavoidable. Aurora also only supports one outstanding checkpoint at a time for each region resulting in serializing calls from multiple threads even if their working sets are independent. RocksDB avoids contention in Aurora by also taking advantage of flat-combining but still experiences an average of 26.7 µs in stall time per checkpoint.

### 4.7.3 Case Study: PostgreSQL

We evaluate the effectiveness of MemSnap with PostgreSQL, a widely used database with over 25 years of development. PostgreSQL is highly configurable and optimized and is unique among our case studies as it is a multiprocess application with a complex buffer cache.

The buffer cache is a shared memory region, which consists of 8 KiB blocks that are associated with a file and offset. Each buffer is described by a buffer descriptor, which holds metadata and a logical pointer to the file block. The buffer cache manager persists buffers through an OS agnostic file interface that uses OS-specific persistence calls (e.g., `fsync`).

For MemSnap integration we must convert the underlying file interface to MemSnap regions, and modify the buffer cache to point directly to these regions. This removes the need for PostgreSQL to first write to the WAL and stage writes in its volatile buffer cache

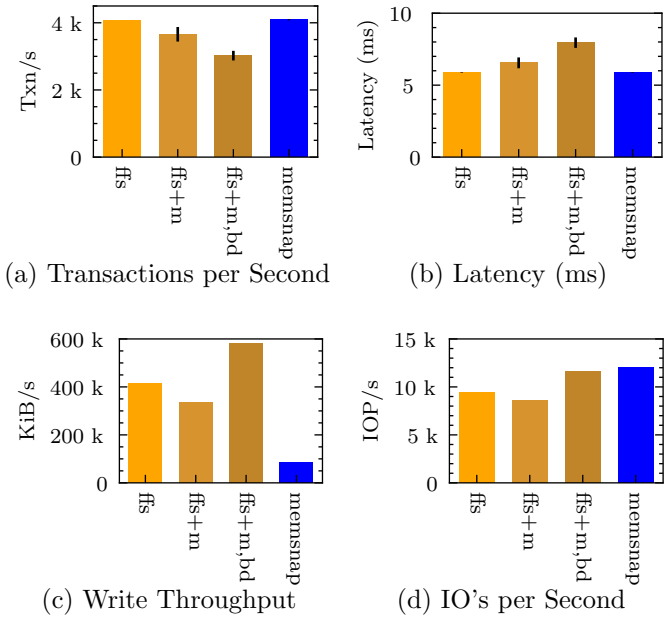



Figure 4.6: Performance of four variations of PostgreSQL running a TPC-C benchmark: Baseline FFS, FFS with `mmap`'d files, FFS with `mmap` and also directly writing data instead of using the buffer cache (i.e., bufdirect, labeled "bd"), and MemSnap.

region.

We now show the changes required for the storage engine to satisfy MemSnap's three properties, and how these changes retain the ACID guarantees of PostgreSQL.

① To uphold the first property we replace files with MemSnap regions. We modify PostgreSQL's file module and file data structure to include a MemSnap region. We further modify read and write to perform memory copies to the region versus using file related system calls. We then change PostgreSQL's buffer cache to directly point to these new memory regions instead of using its volatile shared memory.

② Page isolation between writes is satisfied within PostgreSQL due to locking semantics and PostgreSQL's use of MVCC semantics. MVCC means PostgreSQL never directly modifies values and instead appends changes to the buffer. Changes that are a part of an uncommitted transaction can be flushed without risking data corruption, allowing a transaction to commit other transactions' writes safely. This means MemSnap can persist buffer pages even if another transaction thread is currently still working on them.

③ PostgreSQL upholds the key modification property due to the same MVCC mechanism that upholds property ②. Key data is never directly modified by a transaction and always appended, so a transactions' uncommitted writes cannot be overwritten by subsequent transactions.

We disable the appending of data to the WAL, by disabling the "full page writes" configuration option within PostgreSQL. As described in Section 4.3 our on-disk COW mechanism allows us to safely write directly to MemSnap regions, which represent table data, to ensure crash consistency in the event of a power failure. In case of a crash we use the last persisted copy of the region in the store.

**Recovery** PostgreSQL starts by fetching files required to initialize the database. Our changes modify the file module to instead use MemSnap related calls. MemSnap persisted regions are named after the same path used by PostgreSQL.

After restoring, MemSnap-PostgreSQL maps all MemSnap regions to their original fixed memory address. PostgreSQL then runs its existing recovery logic, since the difference between regions and files is transparent to the upper layers.

**Evaluation** We evaluate PostgreSQL by using the sysbench TPC-C benchmark, a 50% write, online transaction processing (OLTP) benchmark. The benchmark is scaled to 150 warehouses (around 30 GiB). The TPC-C benchmark was run for 2 minutes, run with 24 connections, and all processes were pinned to the first CPU socket to reduce NUMA effects. The sysbench threads were pinned to the other socket.

We configured PostgreSQL using a performance tuning guide [56] and used a 48 GiB buffer cache. Huge pages are disabled, as this resulted in a performance decrease, likely due to the random write workload of TPCC [20].

To better show MemSnap's performance improvements, we demonstrate that modifying PostgreSQL's data in place using existing OS APIs for persistence incurs significant performance penalties. We do this with two additional systems: the first ("ffs-mmap"), which directly maps table data and second ("ffs-mmap-bufdirect"), which uses directly mapped table data and allows for the direct modification of the data over using buffer pages. Through these intermediate variations we show data that corroborates the historical observation that directly mapping data incurs performance penalties [98], in contrast to a buffer cache.

Our last system replaces all `mmap` persistence API calls with MemSnap calls ("ffs-mmap-bufdirect"). We see in Figure 4.6 that rather then incurring a 25% decrease in transactions per second, we see a 1.5% increase. This can be attributed to an overall reduced write throughput and the lower latency of MemSnap.

As seen in Figure 4.6, the overall disk throughput as reported by disk statistics during the running of the benchmark was reduced by 80% when compared to the baseline for MemSnap. This reduction is due to two reasons: First, the block size by default in PostgreSQL is 8 KiB, resulting in larger amounts of write amplification for smaller writes to the WAL and when checkpointing. Second, checkpointing flushes of buffers in the buffer cache are no longer required as MemSnap modifies data in place further reducing write amplification. For example, a 4 KiB dirty page within standard PostgreSQL can result in 16 KiBs of writes. The 16 KiBs comes from the block being written to the WAL and then to the file directly.

There is an increase in overall IO per second by 26% for MemSnap. This is because MemSnap requires creating an IO for every table object modified during every transaction. The baseline aggregates writes into one IO to the WAL. Buffer flushes occur more rarely.

A major advantage MemSnap brings to PostgreSQL is a simplification of the storage stack. For example, PostgreSQL's buffer/file subsystems account for over 10 KSLOC not including the hundreds of various call sites, and logic within the other subsystems. Furthermore, data is no longer managed in four separate forms: the userspace buffer cache, the kernel buffer cache, the WAL, and all file objects. Rather, the storage stack can be collapsed into in memory versions of tables with serialization points at transaction commits, greatly reducing code and likelihood of bugs.

Furthermore, the MemSnap system is the bottom end of what is achievable with better kernel supplied virtual memory abstractions. Further optimizations would require significant

engineering effort as PostgreSQL is a highly complex system, with a storage backend of 600 KSLOC.

An example is the transition states required by PostgreSQL to notify users when a buffer is currently being read in. These states and surrounding policy is no longer required as paging itself by the OS is able to perform this task transparently.

## 4.8 Related Work

Checkpointing systems like Remus [99] use continuous differential checkpointing for transparent fault tolerance of VMs. Project PBerry [85] eliminates the overheads of page-level dirty set tracking and can benefit many checkpointing systems but requires specialized hardware.

File system extensions [189, 232, 155, 136, 247, 140, 196, 210] extend file semantics or extend buffer cache usability, however are designed solely within the storage subsystems. Other systems focus solely on improving virtual memory systems or paging mechanisms [187, 69, 215, 176, 186].

Numerous works [192, 241, 170, 188, 131] attempt to eliminate persistence bugs by reducing the complexity associated with transactional systems and databases. These works demonstrate the difficulty of building persistence as an application developer, and motivate our work on MemSnap.

Some systems remove the overheads of persistence by delaying the externalization of side effects until commit time [181, 91, 180]. Rethink the sync [181] uses this to speed up operations like file moves that require multiple commit points. These techniques are used in SLSes to gain throughput at the cost of extra operation latency.

Recent key-value stores have novel architectures aimed towards high throughput, random access-friendly SSDs. The Kvell KV store [156] uses a shared-nothing architecture and avoids sorting data on disk to minimize CPU usage and maximize throughput. Kvell+ [157] expands this system to optimize long-lived operations like scans. SplinterDB [94] minimizes the CPU and IO write amplification costs of compaction for small keys using $STB^{\varepsilon}$ trees to reduce writes during compaction. MemSnap is compatible with these systems as it only changes applications' persistence-related logic and leaves operations like range scans and compactions unmodified.

## 4.9 Conclusion

In this paper we presented MemSnap, a system for efficient persistence using an architecture inspired by single level stores. MemSnap provides clear persistence semantics, allowing users to directly modify their main dataset in memory, reducing code complexity while outperforming both WAL-and-checkpoint and application checkpointing techniques. We introduce our unified COW mechanism, which is necessary to perform atomic µCheckpoints at sub-millisecond latency. Our experiments show that MemSnap has a 14% overhead over direct disk IO, compared to file APIs that are 43× slower. We also show speed ups in databases like RocksDB of 8.5% and PostgreSQL of 1.5%, while reducing code complexity by up to 40%.

# Chapter 5

# Metropolis: An SLS-Based Serverless Invoker for Fast Cold Starts

## 5.1 Introduction

The serverless paradigm provides scalable infrastructure on demand, fulfilling the long-standing vision of utility computing [97]. Developers write serverless code (i.e., functions) and upload them to cloud providers. Providers deploy instances of these functions inside sandboxes (e.g., VMs or OS containers) to service client requests (i.e., invocations). Providers assumes the responsibility of infrastructure management (hardware and software) and function autoscaling. Recent studies show most functions are shorted lived and often execute in under 1 s [208] leading to sandbox creation being a significant source of overhead. Optimizing on-demand sandbox creation is critical for efficient serverless.

A popular optimization for sandbox creation is to restore an already initialized function from an on-disk snapshot [79]. Serverless sandboxes like Amazon's Firecracker [66] require 100 ms to boot and even more to initialize the function runtime. For example, a Python script that imports NumPy, one of Python's most popular libraries, takes 270 ms to set up, more than the execution time of 25% of serverless functions [208]. Various systems have optimized the use of VM snapshots within serverless [228, 84, 72, 212].

Snapshotting is underexplored for OS container based sandboxes, which suffer from startup times that are as long as the function execution time.

OS containers are increasingly used in both public and private serverless infrastructure. In public clouds, containers are used to colocate functions in the same VM to share specialized hardware such as GPUs and TPUs [53, 121]. Private and hybrid clouds use

containers to run serverless functions on owned infrastructure using products like Oracle Cloud Functions and Redhat Openshift Serverless [54, 50, 29, 49, 48, 37, 36, 127, 23]. However, none of these existings systems use snapshotting techniques to accelerate function invocations.

A few research systems [237, 213] use conventional checkpoint/restore (CR) approaches, but do not explore the unique benefits of OS containers to improve performance. These systems use CRIU, the standard Linux checkpoint/restore mechanism, to restore serverless instances. However, these systems do not investigate ways to optimize CRIU for serverless applications.

Containers allow us to go beyond typical snapshotting techniques through the use of OS introspection. Container snapshots can be manipulated during restore in ways that are not possible with VMs that appear as a black box. This fine-granularity allows for optimizations within serverless typically unavailable to other approaches, such as implicit dependency sharing and the manipulation of kernel objects in restored instances. We explore the use of OS introspection to optimize serverless container invocations.

We present Metropolis, a system that enables fast and efficient serverless invocations through container-based snapshots. Metropolis creates fine-grained images from container snapshots which describe containers as sets of OS resources, enabling faster restores. Metropolis consists of a kernel module that provides and manipulates the fine-grained container image using provided checkpointing and introspection mechanisms. The Metropolis invoker uses these images and manipulates restored state to minimize invocation latency, memory usage, and IO.

Metropolis uses the Aurora single level store (SLS) [226] as a checkpointing system that enables introspection. SLSes structure application images as groups of individually serialized OS objects (e.g., libraries, mappings, and process state). SLSes restore by deserializing the images within the kernel. However, Aurora is designed for application persistence and does not support serverless programming models. Aurora lacks an optimized restore path and does not implement any of the Metropolis mechanisms to support fast invocation of serverless containers.

Metropolis uses fine grained OS introspection on OS containers to enable submillisecond container restores and fast function executions. First, Metropolis supports fine-grained sharing of dependencies between images on-disk and in-memory. Second, sharing reduces the footprint of each function's image and reduces the amount of IO needed to bring in an image into memory. Third, Metropolis further uses OS introspection to automate the snapshotting of serverless functions, regardless of runtime, without arbitrary waiting. Fourth, Metropolis optimizes network communication through our *socket handoff* technique

that injects a connected socket into a restoring function to bypass the builtin reverse proxy most local host invokers implement.

Metropolis reduces a major overhead common to snapshotting systems in which the cold start latency is proportional to the working set size of the function during its execution as this memory must be read in from disk. Metropolis does this through fine-grained sharing of dependencies on disk and in memory between functions without using a common base image. Metropolis caches common dependencies in memory, improving performance by minimizing IO during the functions execution. This technique, called *collaborative warmups*, provides up to 3× better execution times than alternative approaches (e.g., image overlays or function traces). Even seemingly unrelated functions benefit from collaborative warmups. For example when a JSON function runs before a video processing one, it results in a 100 ms speed-up to video processing (or a 25% reduction).

Sharing reduces the image size on-disk allowing Metropolis to cache many images on local disks rather than on cloud storage, which others have shown is beneficial [228, 147]. The effective image size is only the private code and data of that function's container and everything else is shared transparently. Metropolis can store 700 k no-op Python functions per terabyte of storage.

Metropolis uses an automatic snapshot creation mechanism and an invocation mechanism for efficiently backing new containers with image data. We observe that serverless functions always listen for incoming clients after initialization allowing us to snapshot them immediately after setup without userspace code. Our mechanism waits for functions to listen for clients, then transparently checkpoints them. This technique handles arbitrary runtimes and languages.

Metropolis uses OS introspection for other optimizations including *socket handoff*. Socket handoff directly connects function invocations with the client, allowing it to avoid the overhead of the builtin reverse proxy present in most invokers. This optimization reduces communication latency by up to 45%.

Our evaluation shows the performance benefits of OS support for snapshotting serverless containers(§ 5.5.1). Metropolis restores a running Python interpreter in 500 µs, 10× faster than CRIU-based systems. Metropolis achieves 5000 functions per second on a single socket. Compared to state-of-the-art VM-based solutions Metropolis achieves up to 7× faster application response times due to its sharing mechanism.

## 5.2 Background

*Cold starts* occur when a serverless function is run infrequently requiring restarting the

service before processing requests. Cold starts are expensive as startup times are on the order of seconds. For example, a study of Azure's serverless infrastructure showed that a Python sandbox, which takes longer than the execution time of %40 of all functions run on Microsoft Azure [208]. This study further shows that cold starts are common as most functions are called rarely [208].

The main source of overhead when invoking a cold function from an on-disk snapshot is paging in the application memory during restore and execution. *Pagein* is the process of reading in pages from disk (or a cloud store) into the memory of a process. Invoking the snapshot requires restoring the sandbox state, retrieving the image and paging in its memory. Function image size scales with sandbox size, so optimizing the pagein process is important for good performance. A common technique is to selectively pagein all memory you expect to be used at once. Any unexpected memory accesses during the course of a function's execution will result in IO as this memory must be fetched from disk during runtime.

**Paging Strategies.** The pagein of image data is either *eager*, i.e., at invocation time, or *lazy*, i.e., on demand at runtime. Eager paging can be faster than lazy due to better IO locality, however, depends on the predictability of the function's working set to avoid overhead. Excessive paging wastes time doing IO during restore as unneeded data is paged, this also increases the memory use an instance. On the other hand, inaccurately predicting the working set leads to costly pagein during function execution.

Some systems improve performance by attempting to predict the working set of a function during its execution. One such technique is Record and Prefetch (REAP) [228], which uses the fact that instances of the same function often have similar working sets, and traces all pages accessed during a test invocation to a file. REAP eagerly pages in the trace file at invocation time using sequential IO.

However, VM snapshots are large and include the function's dependencies and sandbox state, which is unshareable between function images. Even small functions have large images by virtue of including interpreter state and common dependencies. For example, each Firecracker function has its own gRPC package, server and Python interpreter with standard dependencies. This leads to even small functions having a large on-disk and in-memory footprint which reduces local node capacity.

In general, snapshotting is expensive in terms of disk usage, with REAP trace files adding to this overhead. Table 5.1 shows a breakdown of the space overhead for a Hello World benchmark using REAP with Firecracker VMs. Firecracker snapshots are 256 MiB in size, equal to the VM memory size. However, Hello-World only has 1 KiB of Python code that is unique to the application, while the rest is common image data. Function state

| | **Firecracker** |
|---|---|
| Snapshot | 256 MiB |
| VM State | 12 KiB |
| Docker Image | 65 MiB |
| REAP Trace | 7.9 MiB |
| Disk Total | 329 MiB |
| Memory Total | 166 MiB |

Table 5.1: Breakdown of the Disk Usage for a Firecracker image and a HelloWorld application. The application is a less than 1 KiB of Python.

accounts for a major part of both the image and the trace.

**Reducing Image Size.** Reducing image size is crucial for invocation latency because of two major effects. First, reduction through sharing and/or deduplication results in shared state remaining warm, meaning less IO during subsequent restores [84]. Second, smaller images result in more images being able to be stored on local disk, which is orders of magnitude faster to read from than cloud storage [228]. For example, the latency to retrieve a small object from AWS S3 is 8 ms or higher [147]. For short lived serverless functions, 8 ms adds significant overhead to the invocation time.

SEUSS [84] uses differential snapshotting to reduce cold start latency. Implemented for serverless unikernels, SEUSS minimizes disk IO during invocations by using the same common in-memory image as a base for all functions with private data to the function in an on-disk diff. Function images are thus composed by stacking the in-memory base with the on-disk function private data. Functions shares dependencies only if it is imported in the language interpreter included in the base snapshot.

Common in-memory base images for OS containers lead to complex functions slowing simpler ones, which does not occur with VMs. In SEUSS, adding dependencies adds little overhead as VMs have nearly constant restore times proportional to memory size. This is because hypervisors view VM's as a list of memory pages. Containers are far more complex as the operating system views processes not as just pages of memory, but also the kernel objects that represents and aggregates these pages together. This view results in checkpoint/restore times for OS containers being proportional to the complexity of the process.

To illustrate this, we implemented SEUSS's strategy within Metropolis and used two

functions as an example, a no-op Python function compared to that of a video processing one using OpenCV [143]. We found that while a no-op Python function's address space takes only 50 µs to create, we need 3000 µs for that of the video processing workload. Using a common base for both functions leads to the no-op benchmarks taking over 3000 µs to restore.

**OS Container Snapshotting.** Container Frameworks typically use CRIU [17], the popular Linux checkpoint restore system, to create function images. Such systems include Function Prebaking [213], which builds a snapshotting invoker based on CRIU, and Medes [206], which involves actively scaning memory to deduplicate memory within images. These systems which depend on conventional checkpointing mechanisms, like CRIU, are inherently limited by the speed with which the framework can deserialize images into running containers.

Deserializing a language interpreter like Python into a running application involves restoring the in-kernel state of the container, including all processes, threads, and individual mappings. The costs of restoring these primitives heavily penalizes cold starts due to the complexity of in-kernel container state. For example, a simple "hello world" Python program includes about 500 discrete address space mappings. For this reason, systems like Catalyzer consider OS container checkpointing fundamentally slow [108].

**SLS: A Better Approach to Checkpointing.** Single level stores (SLS) [209] unify volatile and persistent storage by continuously and incrementally checkpointing application state. Application objects like file descriptors or memory regions are stored as separate objects in the underlying object store. From the perspective of the SLS an on-disk file and a memory mapped object are the same. SLSes treat volatile memory as a cache for system state. Applications are persistent from a user's point of view and continue executing after a crash as if nothing happened.

Aurora [226] views checkpointing from the internal kernel perspective to faithfully checkpoint and restore any and all OS state, rather than the userspace perspective used by conventional checkpointing systems. Each kernel object is checkpointed and restored separately facilitating the manipulation and inspection of checkpoint state.

**Metropolis: Our Approach to Serverless Containers.** Metropolis uses the Aurora single level store's checkpointing system with persistence disabled and develops a new set of optimizations that accelerate restore and execution of serverless functions. This requires a significant engineering effort in optimizing the restore path, which is not performance critical for single level stores.

Metropolis introduces fine-grained sharing to save resources on-disk and in-memory while reducing cold start latency by reducing IO. Metropolis benefits from the view that

dependencies are independent objects in the SLS and uses introspection to achieve this fine grained sharing. Unlike previous systems, Metropolis achieves an implicit common warm image without added OS state complexity.

Metropolis adds *lightweight cloning* to SLS objects to enable sharing. On-disk objects have copy-on-write semantics so functions can safely share dependencies without being able to modify each other's data. Similarly, when the object is mapped in memory the pages are shared copy-on-write, and any private modification breaks sharing to ensure data integrity. This occurs at the page granularity and is transparent to the running function instances.

The performance benefits of sharing subsumes any need for paging strategies, because function private state is typically small and often part of the working set. The remaining working set is in shared objects (code or data) that benefit from sharing. Hence, the *collaborative warmup* technique amortizes pagein cost for shared objects across all functions using the object.

Metropolis further uses OS introspection for novel optimizations. For example, to mitigate networking costs Metropolis injects an already connected socket into a restoring function to allow direct communication to the client. This skips the need for the invoker to behave as a reverse proxy and the invoker uses OS to monitor the function's execution.

## 5.3 Design

### 5.3.1 Overview

Metropolis has three main design goals:

**Low Latency:** Servicing user requests with low latency is key for serverless frameworks. As serverless function chains grow they become more sensitive to high tail latencies [101], limiting their size. Low latency is crucial to enable developers to decompose applications into functions without incurring large overheads.

**Transparency and Generality:** Optimizations and mechanisms should support arbitrary runtimes and languages. Metropolis should snapshot and invoke serverless containers without the developer writing code for either operation.

**High Image Density:** High on-disk image density improves resource use on the local node and indirectly benefits performance. On-disk density minimizes overall application response latency by reducing the high latency IO done to cloud storage. Ensuring high image density in the local node increases the overall efficiency of the node.

Metropolis is a single node platform for fast serverless invocations It is comparable to invokers like containerd [47], and provides an API to cloud level orchestrators for

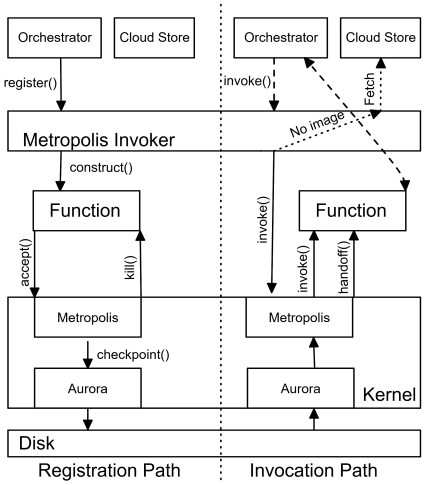



Figure 5.1: Overview of the Metropolis platform. The invoker on the local machine registers new functions by setting up the container instance, which automatically triggers a checkpoint after it is setup. On invocations it retrieves the image from the cloud if needed, restores it, and hands off a socket to it to directly receive arguments from the orchestrator. Metropolis applies uses introspection and sharing (§ 5.3) during the restore process to accelerate sharing.

invoking functions on the node. Metropolis delegates the placement, scheduling and routing of functions to orchestrators like Kubernetes and OpenWhisk. The cloud orchestrator communicates with an individual instance of the Metropolis invoker. Figure 5.1 shows the overall process of how functions are used within Metropolis.

Metropolis uses FreeBSD jails for OS containers, extended to conform to the OCI standard [31]. Metropolis natively supports Linux and FreeBSD containers, and arbitrary language runtimes. Metropolis runs FreeBSD binaries natively, and Linux applications using the Linux system call emulation layer [3]. This allows the deployment of Linux containers on FreeBSD in the same way Joyent [52] provides Linux containers on OpenSolaris. Functions either use Metropolis-provided language runtimes or provide their own.

Metropolis' uses an introspection-oriented design for performance, generality, and density. We use a fine-grained image format which describes functions as sets of OS primitives, and which allows us to use introspection during snapshotting and invocation. We take advantage of code patterns common in serverless to transparently trigger the snapshot creation process (§ 5.3.2). On invocations Metropolis individually restores each dependency and maps it into the function's address space (§ 5.3.3). Metropolis uses introspection

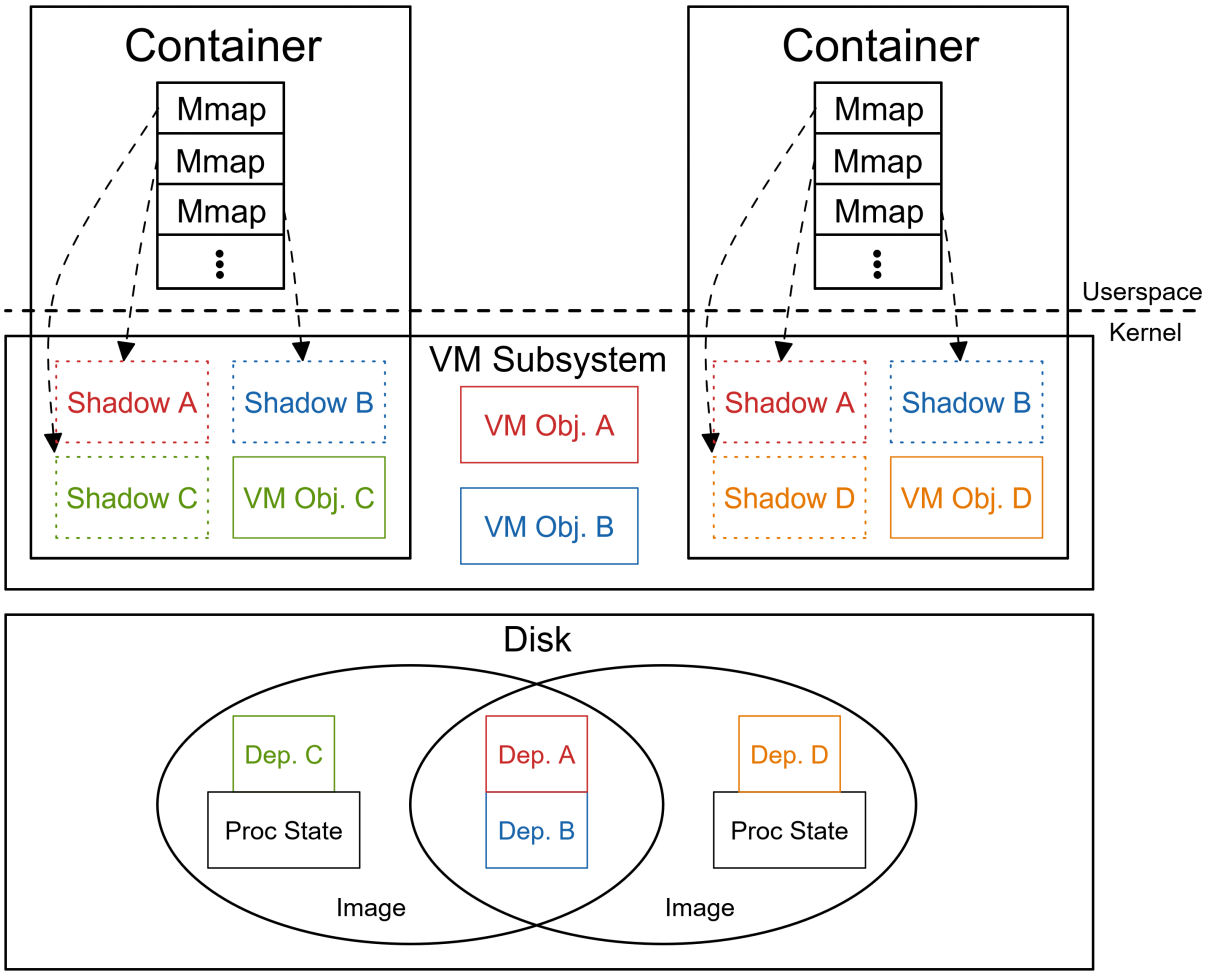



Figure 5.2: The Metropolis sharing mechanism.

to safely share dependencies between functions (§ 5.3.4). Functions share the read-only pages of dependencies, providing them with enough locality to keep them in memory and *speeding up each other's invocations*. Finally, Metropolis uses introspection to optimize the restore paths of OS primitives like vnodes and sockets for serverless by overloading their deserialization procedures (§ 5.3.5).

### 5.3.2 Triggering Snapshotting/Invocations

Metropolis uses introspection to register and invoke functions without coordinating with function instances or requiring a custom runtime. Programmers write a function as they would a serverful application. Metropolis functions are runnable as stateful servers without modification. Newly invoked instances start executing as if they just connected to a client, and receive their arguments through a connected socket. Metropolis thus simplifies the function development process and does not require adding runtime dependencies to the image for communicating with the invoker.

The invoker uses introspection when registering a function to snapshot the instance when it has finished setting up. We observe that serverless functions share a common code pattern, and use it to trigger snapshotting immediately when the instance is ready. Functions first set up their runtime, then wait for incoming clients. For this reason Metropolis treats an `accept` call from the container as a sign that the instance is fully set up. Metropolis ignores `accept` calls made on Unix domain sockets and IPs bound to localhost so as to not interfere with IPC done during setup.

Metropolis tracks calls by overloading the invocation's private system call vector. The FreeBSD kernel associates a system call vector with each process, and uses these vectors to emulate the ABI of other platforms. For example, FreeBSD runs unmodified Linux binaries by backing them with a system call vector which implements the Linux ABI. Metropolis uses its own system call vector with the `accept` system call replaced with code to call the snapshotting API.

Metropolis uses userspace logic to handle functions which do not call `accept` immediately. Some runtimes, like Node.JS, use `poll`, `select`, or `epoll/kqueue` to monitor a socket file descriptor, and call `accept` once a connection is ready. The Metropolis invoker forces such runtimes to call `accept` at registration time by calling `connect` to the serverless function's listening socket. The Metropolis kernel component relays to the invoker which network address to connect to by interposing on the function's `bind` calls.

Metropolis interposes `accept` calls in the kernel to ensure that tracking is general, efficient, and simple to implement. Metropolis handles runtimes like Go and Rust which issue system calls directly instead of using `libc`, and so cannot be intercepted with library interposition. We do not modify existing kernel paths, as we only use the Metropolis system call vector with serverless functions. Metropolis handles functions which use either TCP or UDP sockets to receive requests. For functions using UDP Metropolis uses the interposition mechanism to trigger snapshotting by tracking `recv` calls instead of `accept`.

### 5.3.3 Fast Address Space Creation

Metropolis safely backs function address spaces with image data using a copy-on-write (COW) mechanism to prevent modifications to the image. The underlying SLS normally destroys application images when restoring the application as it directly maps the image into the restored processes' address spaces. Metropolis adapts an existing COW primitive used by `fork` for COW between a function instance and its image. Instances retrieve image data by triggering the unmodified kernel page fault handler.

Metropolis avoids existing mapping techniques like CRIU's `userfaultfd` because they incur large overheads on minor page faults. Handling page faults in userspace has a latency of 6 *us*, not including the time it takes to copy over data from the image. By comparison, the kernel page fault handler has a total latency of 1 *us*. Even a no-op Python function takes 1000 minor faults during execution, so the overhead from minor page faults quickly adds up.

We build on top of existing virtual memory abstractions to implement the Metropolis invoker. FreeBSD's Mach-derived [64] VM subsystem does not directly attach pages to address space mappings. It instead backs mappings with groups of mappable physical pages called VM objects. The kernel maps data in a process by associating a VM object with an address range in the process' address space. The page fault handler populates the process' page table on a fault by finding the address mapping for the address, accessing the backing VM object, then retrieving the physical page. Metropolis images use the VM object abstraction to represent function data as a set of VM objects both on-disk and in-memory. Metropolis brings in function data by triggering paging from the image the VM object.

Metropolis applies COW between the function and the image through object shadowing, a mechanism used internally by `fork`. In object shadowing, top-level objects called *shadows* directly back process mappings and are themselves backed by a *parent* object. The parent holds all COW shared pages, while the shadow holds pages with data private to the process. On a page fault, the handler first searches the shadow VM object then the parent. On a page write, the handler creates a copy of the page and adds it to the shadow for the process to write to. Metropolis uses parent objects to represent function mappings in the image as parent object data is never modified.

Metropolis uses introspection to back function instances with image data in a COW fashion efficiently with minimal overhead at invocation time. Metropolis creates a parent object for each VM object in the image, and a shadow for each function invocation. The objects are initially fully unpopulated as to not to waste IO by preemptively paging in data. Metropolis always pages in data to the parent object, and then creates a function-private copy on demand. Metropolis makes use of the Aurora pager to access the image on disk.

Metropolis applies COW with minimal overhead at invocation time and runtime. Metropolis creates shadows, parent objects, and mappings concurrently at invocation time for minimal overhead compared to using `fork` and `mmap`. The invocation uses the kernel page fault handler for minor page faults during execution, leading to minimal overhead. There are no inconsistencies in the invocation's VM state, and to the kernel it looks as if we set up the instance from scratch without snapshotting.

Metropolis uses the fine-grained mapping mechanism for function autoscaling. A parent VM obejct can have multiple shadows, so VM objects across instances of the same function share parent objects underneath. Instances thus share image data using COW, and if one instance brings in a page then other instances may reuse it to avoid disk IO. Metropolis shares through VM object shadowing, so no page table sharing takes place.

Metropolis uses the pager to deploy paging strategies common in serverless snapshotting. For eager paging Metropolis forces the pager to bring in in all image data from the disk at restore time. For lazy restores Metropolis uses the pager to retrieve data on-demand during execution.

We use Metropolis' fine-grained image format to implement sophisticated paging policies with minimal effort. We implement the record-and-prefetch (REAP [228]) algorithm which traces function executions to identify the working set of a function. Metropolis lazily invokes a function, and tracks the pages which the pager brings in during execution. Metropolis tracks page faults without adding latency to the running function. It then eagerly brings in these pages on subsequent invocations to avoid major page faults.

### 5.3.4 Collaborative Warmups

Metropolis uses introspection to expand shadowing so that if functions share a dependency, they COW share the dependency's image in memory. Sharing dependencies between functions significantly improves invocation latency because common dependencies are used often enough to keep them in memory. Individual functions do not exhibit good locality, so keeping whole images in memory is impractical. Metropolis' fine-grained image format allows it to cache only popular image components, e.g., dependencies, instead of whole images.

Metropolis uses object shadowing to share common dependencies in a fine-grained manner. COW sharing memory is normally only possible using `fork`, which works on the whole address space and incurs significant latency. Metropolis uses object shadowing to back shadows in multiple instances with the same parent object, so that the instances share the parent object's data. Object shadowing works with VM objects, allowing Metropolis to apply COW between functions for a single dependency.

Sharing allows functions to *warm each other up* using the same parent object to page in image data. The pager brings in image data from the disk and into the parent object, so if two instances request the same page only the first one will trigger an IO. Metropolis keeps or evicts function data using the VM page cache to track how warm each image page is. Commonly used dependencies like libraries are kept warm by multiple invocations, while function-private data quickly becomes cold and is evicted.

Sharing provides significant performance benefits because serverless functions are small, and dependencies are thus a major part of the total code. Many dependencies are also large and popular, e.g., NumPy for Python functions. Popular dependencies like OpenCV or even libc are even used by functions written in different languages through bindings. On an invocation Metropolis thus need only bring in private function data, which requires minimal IO.

Metropolis implements sharing on the disk by expanding function image semantics to share components, allowing components to be referenced by multiple images. Aurora images are self-contained to make checkpointing large stateful applications simpler. Aurora restores few applications at once, and that only after a power failure. Metropolis invokes functions constantly, executes invocations concurrently, and does not depend on fast checkpoints.

Metropolis sharing reduces both memory and disk usage, improving the overall performance of the node. Increasing memory density allow Metropolis to run more functions concurrently in a single node without risking paging. High on-disk density ensures that Metropolis rarely retrieves image data from the cloud store, which induces large latency.

Metropolis collaborative warmups work at a fine granularity without using a common base image for functions. Common base images prevent the developer from specifying their own custom root file system, and allow sharing only for dependencies already in the image. Using a base image for sharing also does not support multiple conflicting versions of the same dependency, which Metropolis effortlessly handles. Our techniques are not specific to container approaches. Paravirtualization enables sharing in virtualized environments using techniques similar to Disco [82].

### 5.3.5 Using Introspection

**Socket Handoff.** Socket handoff is a technique for reducing the overhead which the instance incurs when downloading large arguments. With this technique Metropolis reduces latency using introspection to connect invocations to the cloud orchestrator. The orchestrator sends large arguments directly to the function and thus bypasses the local invoker. Argument passing incurs significant latency costs, so optimizing it with socket handoff reduces overall function latency.

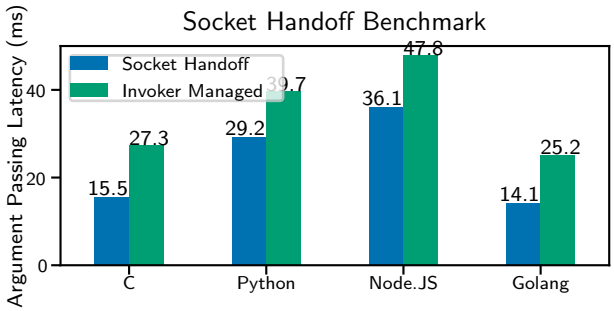



Figure 5.3: Performance gains for the socket handoff optimization for simple functions echoing a 6MB argument. Removing the invoker from the path between the orchestrator and the function reduces communication overhead by 45% for compiled languages (C, Go) and 25% for the Python interpreter and Node.JS JIT.

The orchestrator triggers socket handoff when it connects to a machine with a local Metropolis invoker. The invoker waits on a listening socket for incoming requests from the orchestrator. Once a request arrives, the invoker creates both a new connected socket and a new function instance. Metropolis passes the connected socket directly to the new instance, which then receives its arguments directly.

We implement socket handoff using introspection to pass the connected socket into the instance. Metropolis snapshots functions in the middle of an `accept` call. When it restores an instance it passes it ownership of the connected socket by attaching it to its file descriptor table. It then adjusts the instance's thread state to pass the file descriptor to the connected socket to userspace. Metropolis does this by passing the file descriptor as a return value to the thread which was waiting on the socket for arguments. The thread returns to userspace as if it finished an `accept` call, complete with the file descriptor of the new socket..

Socket handoff allows the local Metropolis invoker to track incoming requests and gather statistics. Functions close the connected socket every time they finish execution, and persist to serve warm invocations. On a subsequent request the invoker uses socket handoff again to create a new connection and again pass it to the function as a return value to `accept`. The invoker thus still counts incoming requests for autoscaling and load balancing. Establishing a connection on every new invocation does not add significant latency because the connection is between the orchestrator and the local invoker.

We demonstrate the performance improvements of socket handoff in Figure 5.3. We run

four language echo servers each written in a different runtime, and send an argument from a client using TCP either through the invoker or directly using socket handoff. The server echoes the argument back to the client after receiving it. The argument is 6MB in size, the maximum amount AWS allows for direct argument passing. We measure the latency from the client.

Our results show that socket handoff provides a significant performance improvement over passing function arguments through the invoker. For the C and Golang which are combined there is a 45% reduction in latency, while for the Python and Node.JS the overhead reduction is 25%. Interpreted and JIT'ed runtimes incur overheads during warmup, which also explains why they have larger latency overall compared to C and Go. The latency which argument passing adds is large, 27.3 $ms$ when passing through the invoker and 15.5 $ms$ with socket handoff.

**Introspection on Restore Paths.** Metropolis uses the ephemeral nature of serverless functions to make operations like restoring memory mapped files faster. Mapping vnodes is traditionally an expensive operation relative to rest of the restore path. It require multiple operations, including opening the vnode and updating the access time on-disk.

Metropolis' ephemeral file system mappings allow us to avoid unnecessary restore operations. Metropolis uses introspection to replace files opening and maping routines with fast paths that only take a vnode reference. These optimizations improve the Python helloworld benchmark by an order of magnitude from 3.2 ms to 0.3 ms.

### 5.3.6 Security Performance Considerations

Metropolis' threat model depends on whether the provider collocates containers belonging to different users in the same (physical or virtual) machine. The threat model in a cloud which collocates user containers is an attacker who cannot escape the containerization mechanism. Users who to deploy serverless functions on infrastructure they own have as a threat model an attacker who cannot escape a virtual or physical machine.

Using interimage sharing mechanism with collocated users allows different users to infer whether library pages are in memory and thus have been recently used. Metropolis' default behavior is to not share security-critical dependencies, e.g., OpenSSL. Metropolis allows programmers to further break sharing for sensitive dependencies (§ 5.3). The Metropolis sharing mechanism is otherwise analogous to `fork`, and does not share page tables.

## 5.4 Implementation

Metropolis is built as a kernel module on FreeBSD 12.3 in addition to the Aurora code. Metropolis includes the shadowing mechanism, system call interposition layer, and code for sharing and collaborative warmups written in 2.8k SLOC of kernel code. Metropolis contains 1.2k SLOC that implement eager, lazy and REAP's paging strategies. Metropolis adds a 200 SLOC kernel patch to expose fast vnode mapping. Metropolis' userspace invoker consists of 675 SLOC.

## 5.5 Evaluation

Our evaluation focuses on two aspects of invocation performance: latency and density. We quantify the benefits of, *fast invocations*, *collaborative warmups*, and *socket handoff*. We break down latency into restore latency, paging latency, and execution time to explain the performance trade-offs made by different systems. Throughout the evaluation we use FunctionBench [143], a popular serverless benchmark that provides representative workloads.

We analyze Metropolis' invocation latency from memory and compare it to that of CRIU. Metropolis restores outperform those of CRIU by an order of magnitude and add minimal overhead to the invocation. We compare paging strategies with collaborative warmups, and measure their effect when running similar functions together. Instances of different functions warm each other up and provide a significant reduction in latency.

We measure image density both on disk and in memory. On-disk images include the entire file system state and the resumable function image. We show how Metropolis minimizes the size of on-disk images using sharing. In-memory images include only the function image and any files accessed during execution. We compare with the differential checkpointing method used by SEUSS.

We compare Metropolis' restore latencies to those of Catalyzer, is the state of the art in snapshot-based invocations. Metropolis outperforms Catalyzer's `sfork` by up to 3×. We further compare Metropolis against VM based systems using vHive.

For all measurements we use a server with dual Intel Xeon Silver 4116 CPUs (Skylake-SP) running at 2.1 GHz, 96 GiB of memory, and an Intel X722 10 Gb network card. The machine has hyperthreading enabled and turbo boost disabled. We use two 280 GB Intel 900P NVMe SSDs in a striped configuration with a 64 KiB stripe size as our storage. All experiments are the average of five or more runs and error bars are included where possible.

| | C | Python | NodeJS |
|---|---|---|---|
| Process creation | 45 $\mu s$ | 43 $\mu s$ | 312 $\mu s$ |
| Address space setup | 87 $\mu s$ | 227 $\mu s$ | 481 $\mu s$ |
| File table setup | 121 $\mu s$ | 122 $\mu s$ | 256 $\mu s$ |
| Other Metropolis | 119 $\mu s$ | 110 $\mu s$ | 119 $\mu s$ |
| Execution | 501 $\mu s$ | 1397 $\mu s$ | 5892 $\mu s$ |
| Retired instructions | $0.7M$ | $17.6M$ | $28.5M$ |
| iTLB misses | $0.2K$ | $1.1K$ | $7.1K$ |
| dTLB load misses | $0.3K$ | $5.1K$ | $9.7K$ |
| dTLB store misses | $0.1K$ | $0.9K$ | $1.1K$ |
| L2 misses | $2.4K$ | $63.0K$ | $39.6K$ |
| User time (us) | 0 | 6456 | 10536 |
| System time (us) | 510 | 2944 | 8035 |
| Minor faults | 30 | 1063 | 1028 |
| Major faults | 0 | 0 | 0 |

Table 5.2: Shows the breakdown of total response time into restore time and execution time for helloworld across three language runtimes.

### 5.5.1 In-memory Invocation Times

Our first experiment shows that deserializing the image and creating the container adds minimal overhead to the end-to-end response time. We break down the end-to-end latency of three echo servers (i.e., helloworld from FunctionBench) written in C, Python, and Node.js. In all experiments we measure the end-to-end response time from a remote machine. We restore the function from a serialized in-memory image to isolate the cost of paging in the memory of the function.

Table 5.2 shows the latency for functions written in C, Python, and JavaScript. By comparing these three servers we quantify the restore and execution overheads caused by language runtimes. C is compiled and has a minimal runtime making it the fastest of the three. Python is interpreted and the process address space includes state related to the language interpreter. Node.js is a very complex runtime with 12 threads, 250 memory mapped regions, and 20 open pipes, kqueues and file descriptors.

Table 5.2 is broken into three parts. The first part breaks down the latency of the restore path until the function starts executing. The second part contains several microarchitectural performance counters including L2 misses and TLB misses to better understand execution

latency. The last part contains OS accounting of user time, system time and page faults as measured by `getrusage`. For these simple workloads, the time spent in the kernel is from fixing up page tables and responding to the client through the socket.

The C function is the fastest of the three, with an end-to-end latency of around 800 $\mu s$. Restoring takes less than half that time, with most of it spent recreating the address space and open file descriptors. OS metrics show that the function spends all of its time in the kernel, because it executes `read` then `write` after resuming execution. The function completes after executing a few thousand instructions in userspace, which is far below a microsecond.

Python's larger address space and runtime lead to slower function restores ($\sim$500 $us$). The Python interpreter's larger code size and working set causes a $\sim$150 $us$ increase to restore latency. Conversely, the execution time is 3$\times$ that of the C function because the interpreter executes more code and thus induces more cache and TLB misses.

Node.js is the slowest and most complex application of the three. It takes 1.2 $ms$ to restore Node.js, while its execution time is 5.9 $ms$. Even though the function is 3.5$\times$ slower than the Python function, both functions have a similar amount of TLB and cache misses. Python has 1.5$\times$ the L2 misses fo Node.js. The amount of instructions executed also does not merit the added execution time. The slowdown in Node.js is because of the disproportionate amount of iTLB misses compared to code size caused by JIT compilation.

The results show Metropolis is not bottlenecked by the restore mechanism. The mechanism scales well with function complexity, and the main overhead lies in function execution. Metropolis' snapshot invocation performance is sufficient to not contribute to the overhead.

**Invocation Latency Comparison to CRIU.** Figure 5.4 compares the restore times of the Metropolis and CRIU invokers for five language runtimes and for the Functionbench workloads. We snapshot each running server after its setup is complete, then measure the time it takes for CRIU and Metropolis to invoke it from the image. We use two configurations, one where the invoker retrieves image from a ramdisk, and one where it retrieves it from backing disk storage. We use eager restores for all configurations, which incurs paging related overhead at invocation time to compare the paging and deserialization costs of invocations.

Metropolis consistently outperforms CRIU by 12.5$\times$–20$\times$ because of its mapping mechanism and introspection-based restore path. Metropolis invokes faster from on-disk images than CRIU does from a ramdisk. For CRIU, image deserialization is as costly as retrieving the image from the disk and dominates invocation time for small functions. Metropolis deserialization costs are below a millisecond for most workloads except Node.JS and

`video_processing`. For these benchmarks, deserialization cost is 10% the total invocation time, so paging still dominates.

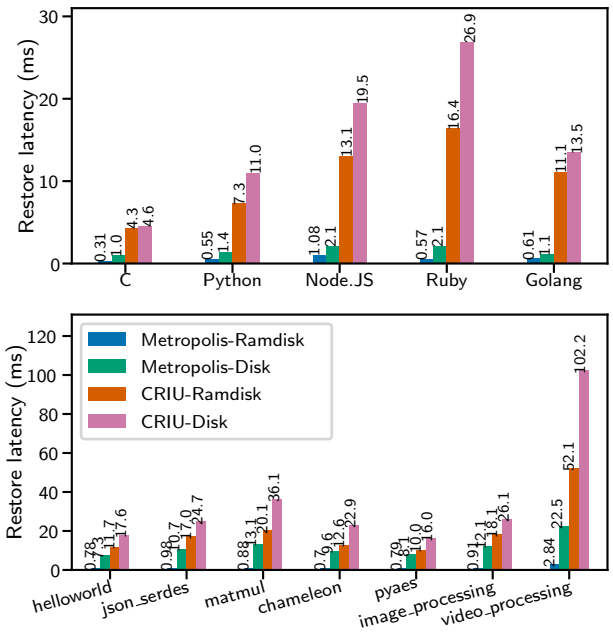



Figure 5.4: Metropolis and CRIU invocation times from images on a ramdisk and physical disk.

### 5.5.2 Paging Strategies

With deserialization times minimal, the major remaining source of restore overhead comes from paging in the memory of the application (i.e., anonymous memory, binaries, libraries, and data). Metropolis eliminates most of the paging latency using *collaborative warmups*.

This second experiment explores two trade-offs in the design space that we can make to optimize paging. First, paging in immediately or lazily paging in as the workload needs. Serverless is unlike other workloads because functions have a short lifespan. There is little benefit from lazy paging strategies, if the function uses most of the memory. Lazily paging often pays an increased overhead because the process incurs page fault latencies and does fewer smaller IOs than bringing in the entire process.

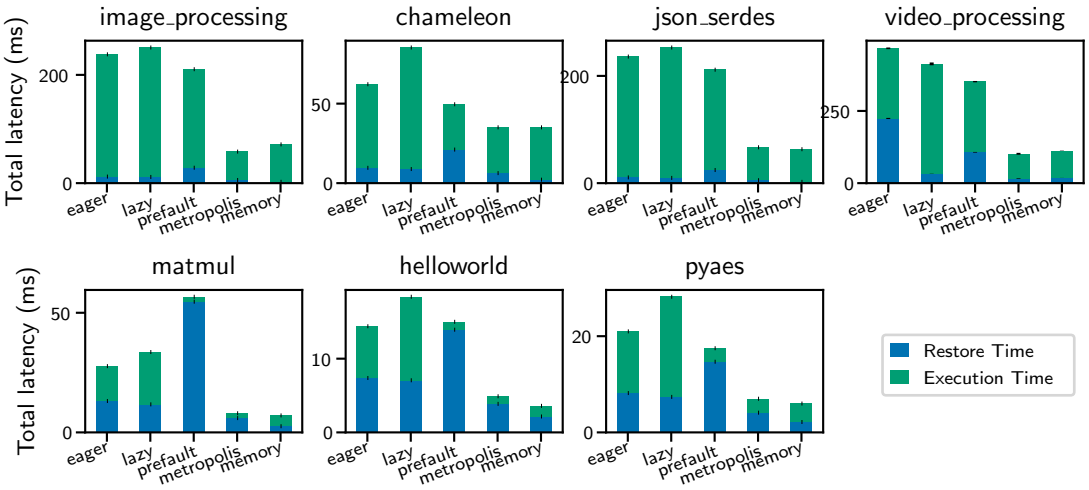



Figure 5.5: Paging Strategies vs Execution times. Sharing is performant compared to tracing as it avoids IO for the working set.

The only reason we may see benefits is when the function does not use it's entire image, which brings us to our second trade off. Selectively subsetting the image to bring in only the pages we need. Prefaulting strategies do this by tracing the function execution once and only paging in the used pages in subsequent calls.

We compare the effectiveness of different paging strategies for invocations from on-disk images for efficient cold starts. We measure the application response time of the FunctionBench workloads from a remote host for each paging strategy. Strategies include: *Eager* that pages in all memory during restore time; *Lazy* that does not page in at restore time, and pages in as needed during execution; *Prefault* uses a trace to preemptively page in data used on prior executions (i.e., a reimplementation of REAP [228]); and *Metropolis* that pages in anonymous pages at restore time and depends on collaborative warmups for vnode pages. We compare these strategies with restores from in-memory images, which do not do any paging. We flush the buffer cache between invocations, except for the Metropolis and memory strategies.

Figure 5.5 shows the results for each benchmark broken into restore and execution time. As we outlined earlier lazy restores are often the slowest, because of the short lifespans coupled with the need to page in a large fraction of the image. Each page fault incurs additional overhead in the page fault handler to bring in a few pages at a time. The function will be suspended by the operating system until the page is brought in, which adds to the application response time.

| $2^{nd}$ \ $1^{st}$ | **helloworld** | **pyaes** | **matmul** | **chameleon** | **json_serdes** | **image_proc.** | **video_proc.** |
|---|---|---|---|---|---|---|---|
| helloworld | 4.5 *ms* | 6.0 *ms* | 6.0 *ms* | 6.0 *ms* | 6.0 *ms* | 5.9 *ms* | 6.3 *ms* |
| pyaes | 8.5 *ms* | 6.7 *ms* | 8.3 *ms* | 8.0 *ms* | 8.2 *ms* | 8.4 *ms* | 8.3 *ms* |
| matmul | 47.5 *ms* | 47.0 *ms* | 8.4 *ms* | 43.1 *ms* | 43.2 *ms* | 44.2 *ms* | 11.0 *ms* |
| chameleon | 39.6 *ms* | 39.3 *ms* | 36.4 *ms* | 34.1 *ms* | 35.9 *ms* | 36.1 *ms* | 35.9 *ms* |
| json_serdes | 202.3 *ms* | 202.7 *ms* | 197.3 *ms* | 199.5 *ms* | 66.9 *ms* | 71.8 *ms* | 71.1 *ms* |
| image_proc. | 198.1 *ms* | 199.1 *ms* | 193.3 *ms* | 194.3 *ms* | 67.3 *ms* | 57.0 *ms* | 66.1 *ms* |
| video_proc. | 459.7 *ms* | 463.9 *ms* | 427.6 *ms* | 458.7 *ms* | 336.3 *ms* | 333.2 *ms* | 101.3 *ms* |

Table 5.3: Shows the pairwise benefit of collaborative warmups across any two benchmarks in FunctionBench. The biggest benefit is for workloads warming up other instances of themselves, which share all libraries and most memory.

Generally eagerly paging in the entire image is better than lazy because of the characteristics of serverless environments. The `matmul` and `video_processing` workloads have a large application image with only a portion required for execution resulting in slower eagar restores.

Prefaulting often performs similar to eager restores for the same reason that much of image is used. Metropolis function images do not hold much unused function state like VM snapshots do. Large file system IO sizes prefetching amplify the total amount of data brought in when paging in libraries. Prefaulting thus avoids only a minor amount of IO for Metropolis images.

Metropolis is close to the performance of the in-memory restores, because the collaborative warmups reduces the total IO beyond whats possible with REAP's tracing. Strategies like collaborative sharing which keep libraries warm are critical for high performance.

**Benefits of Collaborative Warmups.** We breakdown the pairwise benefit of collaborative warmups across any two FunctionBench workloads. In each case, we flush the buffer cache, then invoke a warmup function eagerly. We then invoke a different function and measure its application response time. Each invocation brings in anonymous pages, and warms up the cache for vnodes. If the vnode is already populated, the operation is a no-op.

Table 5.3 shows that even very different functions benefit from collaborative warmups. The functions do not share major dependencies, e.g., only `video_processing` uses OpenCV. Functions are invoked faster than eager restores in the presence of other non-trivial functions. For example, executing `image_processing` after `json_serdes` confers a 2× speedup. The source of this speedup is the inclusion of system libraries and the python runtime. Shared objects like `libc.so` are almost universal and reduce IO significantly.

| Function | Full Size | Full per TB | Metropolis Size | Metropolis per TB | % Saving |
|---|---|---|---|---|---|
| helloworld | 4.9 MiB | 214.0 *k* | 1.5 MiB | 712.0 *k* | 69.9% |
| pyaes | 18.5 MiB | 56.6 *k* | 14.9 MiB | 70.4 *k* | 19.6% |
| chameleon | 24.3 MiB | 43.1 *k* | 17.1 MiB | 61.4 *k* | 29.9% |
| json_serdes | 39.5 MiB | 26.6 *k* | 18.9 MiB | 55.8 *k* | 52.3% |
| image_proc. | 131.5 MiB | 7.9 *k* | 19.6 MiB | 53.5 *k* | 85.2% |
| video_proc. | 3.0 GiB | 0.35 *k* | 32.6 MiB | 32.1 *k* | 98.9% |

Table 5.4: Image sizes and density per TB of storage for FunctionBench functions, using full images versus Metropolis images. The sizes include the function snapshot and all dependencies. Metropolis' on-disk sharing scheme reduces the size of function images by up to 98.9%.

### 5.5.3 Disk Density

Table 5.4 shows the space consumed by Metropolis image and the density we can pack of a given FunctionBench application per tebibyte. Metropolis images consume a fraction of the total amount of file system state accessible to the image. We reduce image size by 30–99% for up to 100× higher density, depending on the function's dependencies. The most complex applications (machine learning and video processing) have the highest savings.

All the dependencies of the FunctionBench workloads use common packages, e.g., OpenCV, Scikit, and Pandas, which are likely to be shared in any real production system. Metropolis shares these packages between images, so each has a constant overhead for the entire machine.

### 5.5.4 Sustained Invocation Throughput

Figure 5.6 shows the scalability of invocations per core of helloworld across two SSDs on a single socket. This is our worst case application as the helloworld function executes for just over 4 ms. Disk restores consume about 80% of the bandwidth of the SSDs before becoming latency bound. Memory restores contend on coarse grained locks that we have not optimizes, but we still saturate 20 cores with helloworld. Any other function would easily saturate the machine.

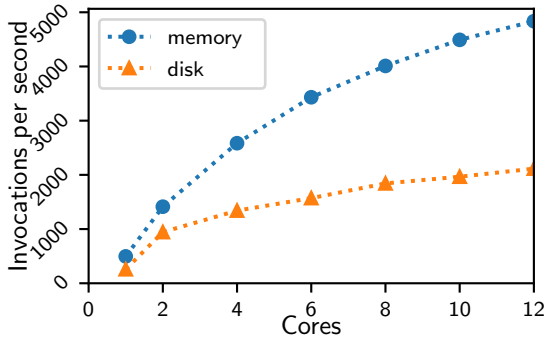



Figure 5.6: Invocations per second on a single machine with two PCIe SSDs. The disk restores achieve $\tilde{8}0\%$ of the SSD throughput, because of some latency bound operations in our storage code. The memory case seems to be limited but requires 12 cores just processing helloworld to saturate.

### 5.5.5 Comparison to Other Systems

Finally, we compare against three state of the art systems. First, we compare against a reimplementation of SEUSS's delta checkpoints on Metropolis. Second, we compare against vHive with REAP running on Linux and show even after removing virtualization overheads we outperform them confirming our reimplementation of REAP. Third, we compare Catalyzer's sfork against Metropolis' fully cached restores.

**SEUSS's Delta Checkpoints.** In this experiment we demonstrate that incremental checkpointing is not effective in saving space for container based serverless functions. Incremental checkpoints are well established for unikernels, which hold OS and even file system related data in the function image. Only a small fraction of these images is actually taken up by the functions themselves. Metropolis in contrast produces lean images that mostly hold by function specific data.

Figure 5.7 shows the number of pages included in each checkpoint for full checkpoints, SEUSS's incremental checkpoints and Metropolis. Incremental checkpointing is effective for small functions where most of the state is from the Python interpreter.

The effectiveness of incremental checkpointing decreases as functions become larger, which implies they contain more state private to that function. The larger state includes imported python libraries, native dynamic libraries and initial data in the function. Often in these benchmarks, the dependencies constitute the largest part of the address space, which reduces the benefit of sharing the initial runtime checkpoint.

Metropolis sharing is more effective, but has similar problems. The savings are more

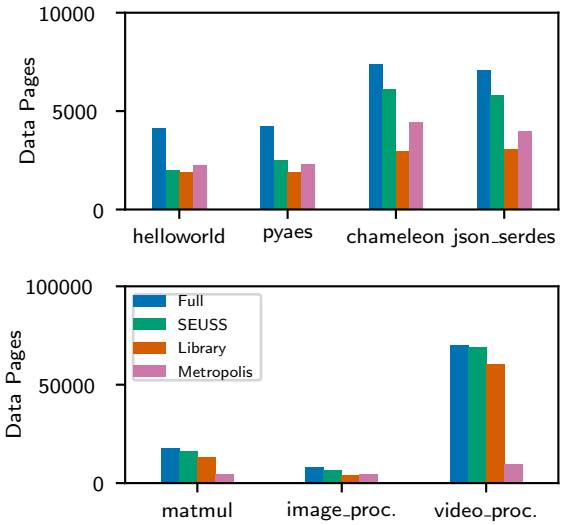



Figure 5.7: The size of a function's address space in the image. Metropolis' passive sharing technique measurably reduces the total checkpoint size. Delta checkpointing is effective for trivial functions (`helloworld`, `pyaes`), but it cannot reduce the size of large shared libraries in larger workloads like `video_processing`.

significant for small functions, and less for larger ones. The effectiveness of Metropolis sharing only decreases because of the relative increase of the total working set. Metropolis does not share anonymous pages, so its efficiency is not affected by package imports dirtying intepreter related pages.

**Metropolis vs. vHive/REAP.** Figure 5.8 shows a breakdown of the invocation latency for Metropolis and vHive with REAP. In this experiment we omit `video_processing` and `matmul`, because vHive did not have a runnable VM for them. We break down the total application response latency into its components to better understand the overheads.

vHive uses Firecracker [66] to run serverless functions inside of microVMs. vHive introduces REAP (Record and Prefetch) that traces a function's working set during an execution. vHive saves it to disk and eagerly restores it on subsequent invocations.

Metropolis optimizes image deserialization to achieve submillisecond restore times, and application response times for helloworld and pyaes of an order of magnitude lower than vHive. vHive in contrast, suffers from the fixed latency cost of restoring the host-side microVM state. Setting up hypervisor state adds significant fixed latency to the process regradless of benchmark size. The VMM setup operation together with creating the

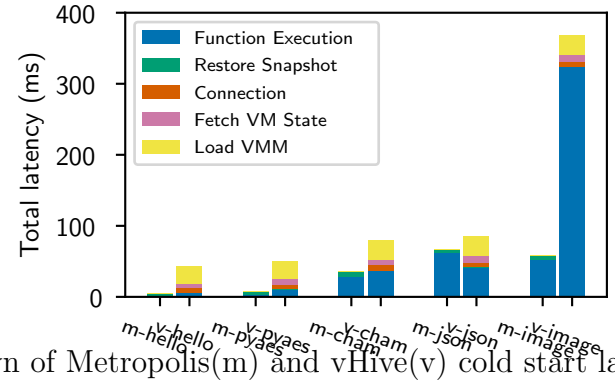



Figure 5.8: Breakdown of Metropolis(m) and vHive(v) cold start latency for five FunctionBench workloads. Metropolis has better performance both at invocation and execution time, except for `json\_serdes` for which Metropolis is using an outdated version of the package.

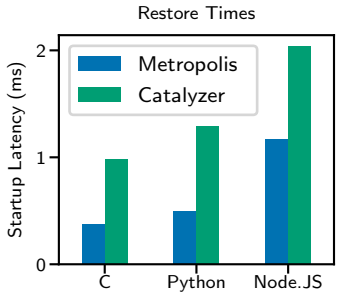



Figure 5.9: Cold starts for Metropolis snapshot restore modes. Metropolis consistently outperforms Catalyzer by 2–3× due to its optimized deserialization path.

connection between host and guest dominate total latency for smaller benchmarks.

Metropolis' restore time is similar to vHive fetching the function state from the disk, because of the similarity in working sets. Execution times Metropolis and vHive are similar, with Metropolis taking 5 ms less than vHive. This cost is due to vHive having to bring in more state that Metropolis shares many of the shared libraries and runtime state without needing to do IOs or page faults.

Metropolis outperforms vHive on larger benchmarks such as `image_rotate`. Long benchmarks have more opportunity for sharing and unpredictable access patterns across invocations, making prefaulting ineffective. The vHive outperforms on the JSON serdes benchmark because it is using a newer version of the serdes package.

**Catalyzer sfork vs. Metropolis's Cached Restores.** Finally, we compare our

restore times against Catalyzer, the fastest known serverless framework in the literature. We specifically compare against Catalyzer's sfork mechanism that restores from a fully cached in memory image. The sfork mechanism caches initialized kernel data structures and uses them to create new instances from memory with. We compare against the fully cached Metropolis restore case. Figure 5.9 shows the results for Hello World programs written in C, Python, and Node.js.

The results suggest that Metropolis restores are faster than Catalyzer when doing so from memory. Catalyzer faces gVisor specific obstacles that add overhead to its `sfork` primitive, like serializing and deserializing thread state to implement multithreaded `fork`. Metropolis solves these issues at the system instead of the language level.

As Catalyzer is closed source, we note that they're results where on hardware superior to ours. The Catalyzer sfork does no IO so disks play no role. It was run an 4-core/8-thread Intel i7-7700 CPU running at 3.6 GHz versus our Intel Xeon 4116 CPU running at 2.1 GHz. Metropolis is not multithreaded and is does not benefit significantly from the increased CPU cache. Catalyzer benefits from the clock speed advantage of their hardware.

## 5.6 Related Work

OS containers are well established for serverless sandboxing. SOCK [183] expands on the OpenLambda [127] framework to optimize container creation throughput and latency. SOCK solves the Linux-specific scalability problems and focuses on Python. The authors report latencies of 100 ms for no-op functions when not using Zygotes, compared to Docker's 130 ms. RunD [51] similarly solves scalability issues for Linux with setting up Kata containers, and optimizes VM sandbox snapshot restores.

Function Zygotes are a popular technique that creates new functions by forking them out of template containers instead of constructing them from scratch. This technique improves invocation latency by skipping parts of the initialization process and is applicable to OS containers and sandboxes like gVisor [1]. SOCK uses Zygotes with popular dependencies already in the container to avoid having to install them at invocation time. This method further lowers invocation time to 10 ms. Catalyzer [108] brings the Zygote technique to gVisor sandboxes for restore times of 1.2 ms for Python no-op functions. Pagurus [160] turns containers of finished functions back into Zygotes to avoid creating.

Snapshot based restores have been optimized for various languages and runtimes. Replayable Execution [237] provides snapshot-based invocations for Java-based functions. Faasm [211] uses snapshotting for WASM based functions. Fireworks [212] optimizes snapshotting for functions with JIT runtimes. FaasSnap [72] optimizes Firecracker-based

snapshot restores and expands on the REAP algorithm [228] for faster cold starts.

Various work uses optimized runtimes to improve execution performance for warm functions. Nightcore [137] enables microsecond latencies for functions using language based isolation. Medes [206] introduces deduplicated serverless containers for storage density, which are running containers whose data Pocket [147] provides a fast ephemeral storage stack for analytics-oriented functions.

## 5.7 Conclusion

We presented Metropolis, a serverless framework that uses sharing to improve function density and cold start times for container based serverless functions. Metropolis shares common dependencies between functions in memory and on-disk to improve density in the node. By maintaining sharing across snapshots we improve startup latency by reducing overall IO needed. Our experiments show that sharing is a powerful technique that simultaneously optimizes cold start invocations, memory and disk density.

# Chapter 6

# Future Work

This thesis presents the Aurora Single Level Store UNIX OS which focuses on persistence and application deployment, but as we have described SLSes have many more potential applications. Aurora, MemSnap and Metropolis handle transparent checkpointing, persistence for data-oriented applications like databases, and deployment, and do not deeply investigate application domains like fault tolerance, debugging, and novel APIs. This work serves as a base on top of which it is possible to build systems for these application domains, while also supporting the use cases we support right now.

One future direction is adapting Aurora for transparent fault tolerance. This requires streaming checkpoints into a remote machine instead of local disks, and also requires efficiently restoring complex applications with large amounts of memory. Fault tolerance also requires a network wide mechanism for making the move of the application from one machine to another transparent to clients. This requires that connected sockets stay connected.

A similar future direction is efficient application migration between machines. Migration has been a standard technique for virtual machines, and has been possible for applications through systems like CRIU. A potential system that applies techniques from these systems to Aurora and combines it with its differential checkpointing technique can outperform existing solutions. The opportunity in such a system is namely using techniques like precopy/postcopy migration and soft working set tracking together with differential checkpoints to minimize overall overhead and migration time.

Another use case is using Aurora for differential debugging. Future work on Aurora will investigate the performance of its on-disk checkpointing restore, but used to store a large amount of fine grained checkpoints instead of persistence. Such a system would require

investigating pipelining checkpoints, keeping multiple COW copies of the working set in memory, and deciding how to distribute checkpoint data between memory and the disk to minimize checkpointing latency. Such future work would be more efficient than existing checkpoint restore for time travelling debugging that only handle single-process applications and keep the checkpoints completely in memory [172].

Future work on Aurora involves integrating it with record-replay systems for debugging and fault tolerance. Aurora's checkpoint restore offers the opportunity to remove the core limitation of record-replay systems that they must store the entire log from the beginning of the application's execution, with no way to compact it. Aurora's checkpoint-restore can provide such compaction, replacing parts of the log with checkpoints and allowing record-replay to run indefinitely. This combination of record-replay and checkpoint-restore would require determining how often checkpointing should happen, and whether using differential checkpointing is more efficient than record-replay during some parts of the application's execution.

We are planning on exploring the integration of Aurora's checkpoint and restore operations with programming language runtimes. Persistence in programming languages has either been implemented in terms of generic object management [174] or as interfaces to specialized hardware, e.g., persistent memory [190]. Future Aurora work will in contrast investigate providing persistence at the language level not just for data, but also for executable application state.

The above research direction includes using Aurora to speculative execution mechanisms similar to `setjmp/longjmp` or software transactional memory for applications. This mechanism would require investigating how to track state changes at a fine granularity in an application, and quickly roll them back with a restore operation. Implementing such a mechanism would provide users with a speculative execution and rollback mechanism that imposes no limitations on the operations that can be done while speculating, unlike existing transactional memory mechanisms.

Future work for MemSnap involves building a new database that is designed around MemSnap's persistence API. We have currently applied MemSnap exclusively to existing databases, whose locking disciplines and layout code have not been build with MemSnap's limitations in mind. A novel design built around MemSnap's persistence model will both demonstrate the full potential of this data SLS, and provide design feedback on how to more efficiently adapt existing databases to MemSnap.

We are also planning on using MemSnap to provide SASOS [102] and SLS [209] data models on a conventional Unix OS for the first time. Past work on persistent object models was done on top of research OSes, limiting their real world applicability. MemSnap runs on

top of FreeBSD and can integrate with widely used applications, while also providing a persistence API on top of which we can build persistent object models. It thus provides the opportunity to explore the applicability of these persistent object models for conventional applications.

We will be expanding Metropolis to use checkpointing for stateful serverless computing. Serverless functions currently provide at least once execution guarantees: Serverless functions may be terminated at any moment by the provider to remove load from a host. We plan on using checkpointing on Metropolis functions to load balance serverless functions across hosts on demand, without killing and restarting a function instances. The challenges of this work would be minimizing migration time for fast load balancing, while also avoiding at least once semantics.

# Chapter 7

# Conclusion

This thesis presents Aurora, a UNIX-compatible single level store OS. The base Aurora system provides SLS semantics using its kernel application checkpoint-restore (CR) operation, and explores the use of SLS semantics for transparently providing persistence to unmodified applications. The MemSnap persistence API expands Aurora's semantics to provide a low-latency persistence API for high-performance applications, and allowing these applications to combine efficient persistence with memory-mapped IO. Finally, Metropolis expands Aurora's application restore mechanism with mechanisms that speed up serverless invocations, taking advantage of the similarity between serverless workloads to reduce cold start overheads.

One takeaway of this thesis is that SLSes are both tractable and useful. During the last decade, the latency gap between memory and device latency has greatly decreased to the point that high-frequency transparent persistence is now feasible with proper OS support. At the same time, providing persistence at the application level is a difficult task that SLSes can help avoid.

Another takeaway is that persistence as a property should be provided by the entire system, not just by file systems. Both Aurora and MemSnap achieve high performance because they use kernel and virtual memory mechanisms to track, flush and protect data while in flight. Presenting persistence as a first-class primitive in the OS is the main design principle behind Aurora's efficiency.

Simlarly, Aurora shows that CR is a primitive that is best provided by the kernel itself. CR amounts to introspection on the applications' in-kernel state, but is normally done by userspace daemons using inefficient interfaces like `proc`. Aurora shows that an in-kernel CR service is both simpler and more efficient than equivalent userspace approaches.

Finally, Metropolis is a demonstration of the value of CR as an operation, independently of persistence. Metropolis optimizes Aurora's restore path for serverless computing, using restore operations to deploy functions from prepared images. Metropolis achieves this by adapting Aurora's differential CR logic to take advantage of the similarity between serverless function images by deduplicating their common state, and so reducing serverless invocation times.

# References


[1] gvisor: Application kernel for containers. https://github.com/google/gvisor. Accessed: 2022-07-03.

[2] Kata containers. https://katacontainers.io/. Accessed: 2022-07-03.

[3] Linux binary compatibility. https://wiki.freebsd.org/Linuxulator. Accessed: 2022-07-03.

[4] Overlay file system. https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html. Accessed: 2022-07-03.

[5] FileBench. https://web.archive.org/web/20080723182431/www.solarisinternals.com/wiki/index.php/FileBench, July 2008.

[6] Journaled Soft-updates. https://www.mckusick.com/softdep/suj.pdf, May 2010.

[7] Replay Debugging – Try it Today! https://blogs.vmware.com/workstation/2010/01/replay-debugging-try-it-today.html, Jan 2010.

[8] Issue 261623: Unrecoverable chrome.storage.sync database corruption. https://bugs.chromium.org/p/chromium/issues/detail?id=261623, July 2013.

[9] CRIU Performance research. https://www.criu.org/Performance_research, December 2015.

[10] Panic: leveldb/table: corruption on data-block. https://forum.syncthing.net/t/panic-leveldb-table-corruption-on-data-block/2526, April 2015.

[11] Compaction priority in RocksDB. https://smalldatum.blogspot.com/2016/02/compaction-priority-in-rocksdb.html, February 2016.

[12] Corruption on data-block while synchronising. https://ethereum.stackexchange.com/questions/1159/corruption-on-data-block-while-synchronising, February 2016.

[13] Db corruption observed with powerloss #333. https://github.com/google/leveldb/issues/333, January 2016.

[14] VMware vSphere: What's New - Availability Enhancements. http://www.slideshare.net/muk_ua/vswn6-m08-avalabilityenhancements, Jan 2017.

[15] *CARP(4) FreeBSD Kernel Interfaces Manual*, 12.0-RELEASE edition, July 2018.

[16] PostgreSQL's fsync() suprise. https://lwn.net/Articles/752063/, April 2018.

[17] CRIU website. https://www.criu.org/Main_Page, April 2019.

[18] DragonFly On-Line Manual Pages. https://www.dragonflybsd.org/cgi/web-man, April 2019.

[19] Mozilla Kraken JavaScript Benchmark. https://wiki.mozilla.org/Kraken, April 2019.

[20] PostgreSQL and Huge Pages. https://wiki.postgresql.org/images/7/7d/PostgreSQL_and_Huge_pages_-_PGConf.2019.pdf, 2019.

[21] Sysbench Benchmark Tool. https://dev.mysql.com/downloads/benchmarks.html, April 2019.

[22] Tuning Failover Cluster Network Thresholds. https://techcommunity.microsoft.com/t5/failover-clustering/tuning-failover-cluster-network-thresholds/ba-p/371834, March 2019.

[23] Apache Openwhisk. https://openwhisk.apache.org/, December 2020.

[24] AWS Lambda. https://aws.amazon.com/lambda/, December 2020.

[25] Azure Functions. https://azure.microsoft.com/en-us/services/functions/, December 2020.

[26] Bastille BSD. https://openwhisk.apache.org/, December 2020.

[27] Cloud Computing Service. https://cloud.google.com/, December 2020.

[28] Docker Website. https://docker.com, December 2020.

[29] IBM Cloud Functions. https://cloud.ibm.com/functions/, December 2020.

[30] IO Cage. https://iocage.io/, December 2020.

[31] Open Container Initiative. https://opencontainer.org/, December 2020.

[32] Quickstart Guide on pot. https://people.freebsd.org/~pizzamig/pot/QuickStart.html, December 2020.

[33] What are Durable Functions? https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview?tabs=csharp, December 2020.

[34] CRIU Release 3.6. https://criu.org/Download/criu/3.6, January 2021.

[35] DragonFly On-Line Manual Pages. https://man.dragonflybsd.org/, January 2021.

[36] Fission: Serverless Functions for Kubernetes. https://fission.io/, April 2021.

[37] Fn Project The Container Native Serverless Framework. https://fnproject.io/, April 2021.

[38] ibm-functions/composer. https://github.com/ibm-functions/composer, April 2021.

[39] Intel Optane SSD DC P4800X Series. https://www.intel.ca/content/www/ca/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-ssd-series/optane-dc-p4800x-series/p4800x-1-5tb-aic.html, May 2021.

[40] LevelDB Source Repository. https://github.com/google/leveldb, January 2021.

[41] MongoDB: The most popular Database for Modern Apps . https://www.mongodb.com/, January 2021.

[42] MySQL Website. https://www.mysql.com/, January 2021.

[43] PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/, January 2021.

[44] PostgreSQL's fsync() surprise. https://lwn.net/Articles/752063/, January 2021.

[45] Redis Website. https://www.redis.io, January 2021.

[46] RocksDB — A persistent key-value store. https://www.rocksdb.org, January 2021.

[47] containerd An industry-standard container runtime. https://containerd.io/, October 2022.

[48] Home - Knative. https://knative.dev/docs/, October 2022.

[49] Official Red Hat OpenShift Documentation. https://docs.openshift.com/, October 2022.

[50] Oracle Cloud Infrastructure. https://cloud.oracle.com, October 2022.

[51] RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, Carlsbad, CA, July 2022. USENIX Association.

[52] SmartOS - Joyent. https://www.tritondatacenter.com/smartos, October 2022.

[53] What is Azure Container Instances? https://learn.microsoft.com/en-us/azure/container-instances/container-instances-overview, October 2022.

[54] What is Dell APEX? https://www.wwt.com/article/what-is-dell-apex, October 2022.

[55] bcachefs: The COW filesystem for Linux that won't eat your data. https://www.sqlite.org/wal.html, August 2023.

[56] Chapter 14. Performance Tips. https://www.postgresql.org/docs/current/performance-tips.html, August 2023.

[57] HC-tree. https://sqlite.org/hctree/doc/hctree/doc/hctree/index.html, April 2023.

[58] SQLite: Most Widely Deployed and Used Database Engine. https://www.sqlite.org/mostdeployed.html, August 2023.

[59] TATP Benchmark. https://tatpbenchmark.sourceforge.net/TATP_Description.pdf, August 2023.

[60] Write-Ahead Logging. https://www.sqlite.org/wal.html, August 2023.

[61] DragonFly On-Line Manual Pages: checkpoint(). https://www.dragonflybsd.org/cgi/web-man?command=checkpoint&section=ANY, January 2025.

[62] Cristina L. Abad, Edwin F. Boza, and Erwin van Eyk. Package-Aware Scheduling of FaaS Functions. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 101Ž013106, New York, NY, USA, 2018. Association for Computing Machinery.

[63] Hazim Abdel-Shafi, Evan Speight, and John K Bennett. Efficient User-Level Thread Migration and Checkpointing on Windows NT Clusters. In *Windows NT 3rd Symposium (Windows NT 3rd Symposium)*, Seattle, WA, July 1999. USENIX Association.

[64] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. 1986.

[65] Advanced Micro Devices, Inc. AMD EPYC 7003 Processors (Data Sheet). https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf, 2021.

[66] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, 2020.

[67] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[68] Wail Y. Alkowaileet, Sattam Alsubaiee, and Michael J. Carey. An lsm-based tuple compaction framework for apache asterixdb. *Proc. VLDB Endow.*, 13(9):1388–1400, may 2020. https://doi.org/10.14778/3397230.3397236.

[69] 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.

[70] Mohamed Alzayat, Jonathan Mace, Peter Druschel, and Deepak Garg. Groundhog: Efficient request isolation in faas. In *Proceedings of the Eighteenth European Conference*

*on Computer Systems*, EuroSys '23, page 398–415, New York, NY, USA, 2023. Association for Computing Machinery.

[71] Thomas E Anderson, David E Culler, and David Patterson. A case for NOW (networks of workstations). *IEEE micro*, 15(1):54–64, 1995.

[72] Lixiang Ao, George Porter, and Geoffrey M. Voelker. Faasnap: Faas made fast using snapshot-based vms. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 730–746, New York, NY, USA, 2022. Association for Computing Machinery.

[73] Apollo Computer Inc. Domain/OS Design Principles. Technical report, 1989.

[74] 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.

[75] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Makalu: Fast recoverable allocation of non-volatile memory. In *ACM SIGPLAN Notices*, volume 51, pages 677–694. ACM, 2016.

[76] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. Twizzler: a Data-Centric OS for Non-Volatile Memory. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 65–80. USENIX Association, July 2020.

[77] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.

[78] Marc Brooker, Adrian Costin Catangiu, Mike Danilov, Alexander Graf, Colm MacCartaigh, and Andrei Sandu. Restoring uniqueness in microvm snapshots. *arXiv preprint arxiv:2102.12892*, 2021.

[79] Marc Brooker, Adrian Costin Catangiu, Mike Danilov, Alexander Graf, Colm MacCárthaigh, and Andrei Sandu. Restoring uniqueness in microvm snapshots. *CoRR*, abs/2102.12892, 2021.

[80] Marc Brooker, Adrian Costin Catangiu, Mike Danilov, Alexander Graf, Colm MacCarthaigh, and Andrei Sandu. Restoring uniqueness in microvm snapshots. *arXiv preprint arXiv:2102.12892*, 2021.

[81] Edouard Bugnion, Vitaly Chipounov, and George Candea. Lightweight snapshots and system-level backtracking. In *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*, 2013.

[82] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running Commodity Operating Systems on Scalable Multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[83] Sebastian Burckhardt, Badrish Chandramouli, Chris Gillum, David Justo, Konstantinos Kallas, Connor McMahon, Christopher S Meiklejohn, and Xiangfeng Zhu. Netherite: Efficient execution of serverless workflows. *Proceedings of the VLDB Endowment*, 15(8):1591–1604, 2022.

[84] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[85] Irina Calciu, Ivan Puddu, Aasheesh Kolli, Andreas Nowatzyk, Jayneel Gandhi, Onur Mutlu, and Pratap Subrahmanyam. Project PBerry: FPGA Acceleration for Remote Memory. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 127Ž2013135, New York, NY, USA, 2019. Association for Computing Machinery.

[86] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 209–223, Santa Clara, CA, February 2020. USENIX Association.

[87] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 433–452, New York, NY, USA, 2014. ACM.

[88] Ryan Chard, Tyler J Skluzacek, Zhuozhao Li, Yadu Babuji, Anna Woodard, Ben Blaiszik, Steven Tuecke, Ian Foster, and Kyle Chard. Serverless supercomputing: High performance function as a service for science. *arXiv preprint arXiv:1908.04907*, 2019.

[89] Yunji Chen, Shijin Zhang, Qi Guo, Ling Li, Ruiyang Wu, and Tianshi Chen. Deterministic Replay: A Survey. *ACM Comput. Surv.*, 48(2):17:1–17:47, September 2015.

[90] David Cheriton. The V Distributed System. *Commun. ACM*, 31(3):314–333, March 1988.

[91] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 228–243, New York, NY, USA, 2013. Association for Computing Machinery. https://doi.org/10.1145/2517349.2522726.

[92] 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*, NSDI'05, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[93] 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*, NSDI'05, pages 273–286, USA, 2005. USENIX Association.

[94] Alexander Conway, Abhishek Gupta, Vijay Chidambaram, Martin Farach-Colton, Richard Spillane, Amy Tai, and Rob Johnson. SplinterDB: Closing the bandwidth gap for NVMe Key-Value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 49–63. USENIX Association, July 2020.

[95] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. New York, NY, USA, 2010. Association for Computing Machinery.

[96] Fernando J Corbato. A Paging Experiment with the Multics System, 1968.

[97] Fernando J Corbató and Victor A Vyssotsky. Introduction and Overview of the Multics System. In *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, pages 185–196, 1965.

[98] 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.

[99] 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.

[100] S.H Dahlby, G.G Henry, D.N Reynolds, and P.N Taylor. System/38 - A high-level machine. In *IBM System/38 Technical Developments*, 1978.

[101] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56:74–80, 2013.

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

[103] Alex Depoutovitch and Michael Stumm. Otherworld: giving applications a chance to survive os kernel crashes. In *Proceedings of the 5th European conference on Computer systems*, pages 181–194, 2010.

[104] William R. Dieter and James E. Lumpp. User-Level Checkpointing for LinuxThreads Programs. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, page 81–92, USA, 2001. USENIX Association.

[105] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Evolution of Development Priorities in Key-value Stores Serving Large-scale Applications: The RocksDB Experience. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 33–49, 2021.

[106] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. *ACM Trans. Storage*, 17(4), oct 2021. https://doi.org/10.1145/3483840.

[107] Fred Douglis and John K Ousterhout. *Process migration in the Sprite operating system*. Computer Science Division, University of California, 1987.

[108] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. Catalyzer: Sub-Millisecond Startup for Serverless Computing

with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 467–481, New York, NY, USA, 2020. Association for Computing Machinery.

[109] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 45–59, 2020.

[110] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2003.

[111] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution Replay of Multiprocessor Virtual Machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, page 121–130, New York, NY, USA, 2008. Association for Computing Machinery.

[112] Elmootazbellah Nabil Elnozahy, David B Johnson, and Willy Zwaenepoel. Measured performance of consistent checkpointing. In *Proceedings of the Eleventh Symposium on Reliable Distributed Systems*, 1992.

[113] Xiyang Feng, Guodong Jin, Ziyi Chen, Chang Liu, and Semih Salihoğlu. Kùzu graph database management system.

[114] Henrique Fingler, Amogh Akshintala, and Christopher J Rossbach. Usetl: Unikernels for serverless extract transform and load why should you settle for less? In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 23–30, 2019.

[115] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association.

[116] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. Sqlite: Past, present, and future. *Proc. VLDB Endow.*, 15(12):3535–3547, aug 2022.

[117] Qi Gao, Weikuan Yu, Wei Huang, and Dhabaleswar K Panda. Application-transparent checkpoint/restart for MPI programs over InfiniBand. In *2006 International Conference on Parallel Processing (ICPP'06)*, pages 471–478. IEEE, 2006.

[118] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A Bender, Abhishek Bhattacharjee, et al. Mosaic pages: Big tlb reach with small pages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 433–448, 2023.

[119] Kinshuk Govil, Dan Teodosiu, Yongqiang Huang, and Mendel Rosenblum. Cellular disco: resource management using virtual clusters on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 18(3):229–262, 2000.

[120] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. In-memory performance for big data. 2014.

[121] Runsheng Guo, Victor Guo, Antonio Kim, Josh Hildred, and Khuzaima Daudjee. Hydrozoa: Dynamic hybrid-parallel dnn training on serverless containers. In D. Marculescu, Y. Chi, and C. Wu, editors, *Proceedings of Machine Learning and Systems*, volume 4, pages 779–794, 2022.

[122] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *OSDI*, volume 8, pages 193–208, 2008.

[123] Paul H Hargrove and Jason C Duell. Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In *Journal of Physics: Conference Series*, volume 46, page 494. IOP Publishing, 2006.

[124] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. *ACM Trans. Comput. Syst.*, 30(3), aug 2012. https://doi.org/10.1145/2324876.2324878.

[125] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. Mungi: A distributed single address-space operating system. In *Proceedings of the 17th Australasian Computer Science Conference*, pages 271–80, 1993.

[126] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[127] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver, CO, June 2016. USENIX Association.

[128] Poul henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *In Proc. 2nd Intl. SANE Conference*, 2000.

[129] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-Copy Live Migration of Virtual Machines. *SIGOPS Oper. Syst. Rev.*, 43(3):14–26, July 2009.

[130] Dave Hitz, James Lau, and Michael Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[131] Yige Hu, Zhiting Zhu, Ian Neal, Youngjin Kwon, Tianyu Cheng, Vijay Chidambaram, and Emmett Witchel. Txfs: Leveraging file-system crash consistency to provide acid transactions. *ACM Trans. Storage*, 15(2), may 2019. https://doi.org/10.1145/3318159.

[132] Zhuo Huang, Song Wu, Song Jiang, and Hai Jin. Fastbuild: Accelerating docker image building for efficient development and deployment of container. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 28–37, 2019.

[133] Theo Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9(2):111–120, 1984. https://doi.org/10.1016/0306-4379(84)90020-6.

[134] Intel Corporation. 3rd Gen Intel Scalable Processors for IoT Product Brief. https://www.intel.com/content/dam/www/central-libraries/us/en/documents/3rd-gen-xeon-sp-for-iot-product-brief.pdf, 2021.

[135] G.J. Janakiraman, J.R. Santos, D. Subhraveti, and Y. Turner. Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 260–269, 2005.

[136] 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.

[137] Zhipeng Jia and Emmett Witchel. Nightcore: Efficient and scalable serverless computing for latency-sensitive, interactive microservices. 2021.

[138] Terry Jones, Michael J. Brim, Geoffroy Vallee, Benjamin Mayer, Aaron Welch, Tonglin Li, Michael Lang, Latchesar Ionkov, Douglas Otstott, Ada Gavrilovska, Greg Eisenhauer, Thaleia Doudali, and Pradeep Fernando. Unity: Unified memory and file space. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, ROSS '17, New York, NY, USA, 2017. Association for Computing Machinery.

[139] Poul-Henning Kamp. *JAIL(8) FreeBSD System Manager's Manual*, 12.0-RELEASE edition, October 2018.

[140] 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.

[141] Sanidhya Kashyap, Changwoo Min, Byoungyoung Lee, Taesoo Kim, and Pavel Emelyanov. Instant OS Updates via Userspace Checkpoint-and-Restart. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 605–619, 2016.

[142] Kimberly Keeton. The Machine: An Architecture for Memory-Centric Computing. In *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, volume 10, 2015.

[143] 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.

[144] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging Operating Systems with Time-Traveling Virtual Machines. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*, Anaheim, CA, April 2005. USENIX Association.

[145] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association.

[146] 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.

[147] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, October 2018. USENIX Association.

[148] Aasheesh Kolli, Vaibhav Gogte, Ali Saidi, Stephan Diestelhorst, Peter M Chen, Satish Narayanasamy, and Thomas F Wenisch. Language-level persistency. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 481–493, 2017.

[149] Oren Laadan and Serge E Hallyn. Linux-CR: Transparent Application Checkpoint-Restart in Linux. In *Linux Symposium*, volume 159. Citeseer, 2010.

[150] Oren Laadan and Jason Nieh. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *USENIX Annual Technical Conference*, pages 323–336, 2007.

[151] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 1–12, New York, NY, USA, 2009. Association for Computing Machinery.

[152] C.R. Landau. The checkpoint mechanism in KeyKOS. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems*, pages 86 – 91, 10 1992.

[153] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.*, 36(4):471–482, April 1987.

[154] Peter Alan Lee, Thomas Anderson, Peter Alan Lee, and Thomas Anderson. *Fault tolerance*. Springer, 1990.

[155] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. Virtual-memory assisted buffer management. *Proc. ACM Manag. Data*, 1(1), may 2023. https://doi.org/10.1145/3588687.

[156] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery.

[157] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell+: Snapshot isolation without snapshots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 425–441. USENIX Association, November 2020.

[158] Henry M Levy. *Capability-Based Computer Systems*. Digital Press, 2014.

[159] K Li, JF Naughton, and JS Plank. Real-time, concurrent checkpointing for parallel programs. In *2nd ACM SIGPLAN Symposium on Principles and Practice of Parall el Programming (PPoPP'90)*, pages 79–88.

[160] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. Help rather than recycle: Alleviating cold startup in serverless computing through Inter-Function container sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 69–84, Carlsbad, CA, July 2022. USENIX Association.

[161] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 49–64, Berkeley, CA, USA, 2016. USENIX Association.

[162] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P Stevenson. Si-tm: Reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 383–398, 2014.

[163] Chen Luo and Michael J Carey. Lsm-based storage techniques: a survey. *The VLDB Journal*, 29(1):393–418, 2020.

[164] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 2182013233, New York, NY, USA, 2017. Association for Computing Machinery.

[165] Ali José Mashtizadeh, Min Cai, Gabriel Tarasuk-Levin, Ricardo Koller, Tal Garfinkel, and Sreekanth Setty. XvMotion: Unified Virtual Machine Migration over Long Distance. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 97–108, Philadelphia, PA, June 2014. USENIX Association.

[166] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards Practical Default-On Multi-Core Record/Replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 693–708, New York, NY, USA, 2017. Association for Computing Machinery.

[167] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. *SIGPLAN Not.*, 52(4):693–708, April 2017.

[168] 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, August 1984.

[169] Marshall Kirk McKusick and Gregory R. Ganger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem. In *1999 USENIX Annual Technical Conference (USENIX ATC 99)*, Monterey, CA, June 1999. USENIX Association.

[170] 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.

[171] Umar Farooq Minhas, Shriram Rajagopalan, Brendan Cully, Ashraf Aboulnaga, Kenneth Salem, and Andrew Warfield. Remusdb: Transparent high availability for database systems. *The VLDB Journal*, 22(1):29–45, February 2013.

[172] Armando Miraglia, Dirk Vogt, Herbert Bos, Andy Tanenbaum, and Cristiano Giuffrida. Peeking into the Past: Efficient Checkpoint-Assisted Time-Traveling Debugging.

In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 455–466, 2016.

[173] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. Aries: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, mar 1992. https://doi.org/10.1145/128765.128770.

[174] Ronald Morrison, Alfred L Brown, Raymund Carrick, Richard CH Connor, Alan Dearle, and Malcolm P Atkinson. The napier type system. In *POS*, pages 3–18. Springer, 1989.

[175] Dushyanth Narayanan and Orion Hodson. Whole-System Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 401–410, New York, NY, USA, 2012. Association for Computing Machinery.

[176] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, dec 2003. https://doi.org/10.1145/844128.844138.

[177] David L Nelson and Paul J Leach. The architecture and applications of the apollo domain. *IEEE Computer Graphics and Applications*, 4(4):58–66, 1984.

[178] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 25–25, Berkeley, CA, USA, 2005. USENIX Association.

[179] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative execution in a distributed file system. *SIGOPS Oper. Syst. Rev.*, 39(5):191–205, oct 2005. https://doi.org/10.1145/1095809.1095829.

[180] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, page 308–318, New York, NY, USA, 2008. Association for Computing Machinery. https://doi.org/10.1145/1346281.1346321.

[181] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Trans. Comput. Syst.*, 26(3), September 2008.

[182] Edward Oakes, Leon Yang, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Pipsqueak: Lean lambdas with large libraries. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 395–400, 2017.

[183] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, 2018.

[184] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The design and implementation of zap: A system for migrating computing environments. *ACM SIGOPS Operating Systems Review*, 36(SI):361–376, 2002.

[185] 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.

[186] Anastasios Papagiannis, Giorgos Saloustros, Pilar González-Férez, and Angelos Bilas. An Efficient Memory-Mapped Key-Value Store for Flash Storage. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 490–502, New York, NY, USA, 2018. Association for Computing Machinery. https://doi.org/10.1145/3267809.3267824.

[187] 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.

[188] Daejun Park and Dongkun Shin. Stackable transactional file system using kernel-level wal. *IEEE Access*, 10:110088–110099, 2022. https://doi.org/10.1109/ACCESS.2022.3214521.

[189] 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.

[190] Steven Pelley, Peter M Chen, and Thomas F Wenisch. Memory persistency. *ACM SIGARCH Computer Architecture News*, 42(3):265–276, 2014.

[191] Simon Pickartz, Niklas Eiling, Stefan Lankes, Lukas Razik, and Antonello Monti. Migrating LinuX containers using CRIU. In *International Conference on High Performance Computing*, pages 674–684. Springer, 2016.

[192] 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.

[193] 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, 2014. USENIX Association.

[194] Eduardo Pinheiro. Epckpt—a checkpoint utility for the linux kernel. *Online at: http://www. cs. rochester. edu/u/edpin/epckpt*, 2002.

[195] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *Usenix Winter 1995 Technical Conference*, 1995.

[196] 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.

[197] Michael L Powell and Barton P Miller. Process migration in demos/mp. *ACM SIGOPS Operating Systems Review*, 17(5):110–119, 1983.

[198] Mark Raasveldt and Hannes Mühleisen. Duckdb: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 1981–1984, New York, NY, USA, 2019. Association for Computing Machinery. https://doi.org/10.1145/3299869.3320212.

[199] Richard Rashid, Avadis Tevanian, Michael Young, David Golub, Robert Baron, David Black, William Bolosky, and Jonathan Chew. Machine-Independent Virtual Memory

Management for Paged Uniprocessor and Multiprocessor Architectures. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, page 31–39, Washington, DC, USA, 1987. IEEE Computer Society Press.

[200] 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.

[201] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[202] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Trans. Storage*, 9(3), aug 2013.

[203] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999.

[204] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15, New York, NY, USA, 1991. Association for Computing Machinery.

[205] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, pages 1–8, 2018.

[206] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 714–729, New York, NY, USA, 2022. Association for Computing Machinery.

[207] Martin Schulz, Greg Bronevetsky, Rohit Fernandes, Daniel Marques, Keshav Pingali, and Paul Stodghill. Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs. In *Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 38. IEEE Computer Society, 2004.

[208] Mohammad Shahrad, Rodrigo Fonseca, Inigo 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. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 205–218. USENIX Association, July 2020.

[209] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 170–185, New York, NY, USA, 1999. ACM.

[210] 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.

[211] Simon Shillaker and Peter Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. *arXiv preprint arXiv:2002.09344*, 2020.

[212] Wonseok Shin, Wook-Hee Kim, and Changwoo Min. Fireworks: A fast, efficient, and safe serverless framework using vm-level post-jit snapshot. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 663–677, New York, NY, USA, 2022. Association for Computing Machinery.

[213] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. Prebaking functions to warm the serverless cold start. In *Proceedings of the 21st International Middleware Conference*, pages 1–13, 2020.

[214] Dimitrios Skarlatos, Umur Darbaz, Bhargava Gopireddy, Nam Sung Kim, and Josep Torrellas. Babelfish: Fusing address translations for containers. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 501–514, 2020. https://doi.org/10.1109/ISCA45697.2020.00049.

[215] Dimitrios Skarlatos, Apostolos Kokolis, Tianyin Xu, and Josep Torrellas. Elastic cuckoo page tables: Rethinking virtual memory translation for parallelism. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1093–1108, New York, NY, USA, 2020. Association for Computing Machinery. https://doi.org/10.1145/3373376.3378493.

[216] Alan Clifton Skousen. Sombrero: a very large single address space distributed operating system. Master's thesis, Arizona State University, 1994.

[217] Frank G. Soltis. *Fortress Rochester: The Inside Story of the IBM iSeries*. May 2001.

[218] Vikram Sreekanti, Chenggang Wu Xiayue Charles Lin, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *arXiv preprint arXiv:2001.04592*, 2020.

[219] E. Steketee, Wei Ping Zhu, and P. Moseley. Implementation of process migration in amoeba. In *14th International Conference on Distributed Computing Systems*, pages 194–201, 1994.

[220] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, and Sape J. Mullender. Experiences with the amoeba distributed operating system. *Commun. ACM*, 33(12):46–63, December 1990.

[221] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. Sequoia: Enabling quality-of-service in serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 311–327, 2020.

[222] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the Condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[223] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight {OS} Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 199–212, 2018.

[224] Alexander Thomasian. Two-phase locking performance and its thrashing behavior. *ACM Trans. Database Syst.*, 18(4):579–625, dec 1993. https://doi.org/10.1145/169725.169720.

[225] 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. https://doi.org/10.1145/3458336.3465285.

[226] 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.

[227] Emil Tsalapatis, Ryan Hancock, Rakeeb Hossain, and Ali José Mashtizadeh. Memsnap µcheckpoints: 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*, ASPLOS '24, page 622–638, New York, NY, USA, 2024. Association for Computing Machinery.

[228] 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)*. ACM, 2021.

[229] Geoffroy Vallee, Thomas Naughton, Hong Ong, and Stephen L Scott. Checkpoint/restart of virtual machines based on xen. In *Proceedings of the High Availability and Performace Computing Workshop (HAPCW 2006), Santa Fe, New Mexico, USA*, 2006.

[230] Kaushik Veeraraghavan, Dongyoon Lee, Benjamin Wester, Jessica Ouyang, Peter M. Chen, Jason Flinn, and Satish Narayanasamy. DoublePlay: Parallelizing Sequential Logging and Replay. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 15–26, New York, NY, USA, 2011. ACM.

[231] Ranjan Sarpangala Venkatesh, Till Smejkal, Dejan S. Milojicic, and Ada Gavrilovska. Fast In-Memory CRIU for Docker Containers. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, page 53–65, New York, NY, USA, 2019. Association for Computing Machinery.

[232] 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.

[233] Dirk Vogt, Armando Miraglia, Georgios Portokalidis, Herbert Bos, Andy Tanenbaum, and Cristiano Giuffrida. Speculative Memory Checkpointing. In *Proceedings of the 16th Annual Middleware Conference*, Middleware '15, pages 197–209, New York, NY, USA, 2015. Association for Computing Machinery.

[234] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, pages 254–265, Berlin, Heidelberg, 2009. Springer-Verlag.

[235] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *SIGOPS Oper. Syst. Rev.*, 39(5):148ž2013162, October 2005.

[236] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The locus distributed operating system. *ACM SIGOPS Operating Systems Review*, 17(5):49–70, 1983.

[237] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[238] Ashton Webster, Ryan Eckenrod, and James Purtilo. Fast and Service-preserving Recovery from Malware Infections Using CRIU. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1199–1211, Baltimore, MD, 2018. USENIX Association.

[239] Tim Wilkinson and Kevin Murray. Extensible, flexible and secure services in angel, a single address space operating system. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, volume 2, pages 755–758. IEEE, 1995.

[240] Tim Wilkinson, Tom Stiemerling, Peter E Osmon, Ashley Saulsbury, and Paul Kelly. Angel: A proposed multiprocessor operating system kernel. In *European Workshop on Parallel Computing*, pages 316–319. Barcelona, Spain, 1992.

[241] 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.

[242] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, NSDI'07, pages 17–17, Berkeley, CA, USA, 2007. USENIX Association.

[243] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In

*2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 457–468, 2014.

[244] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[245] 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.

[246] Bjoern A. Zeeb. *VNET(9) FreeBSD Kernel Developer's Manual*, 12.0-RELEASE edition, July 2018.

[247] Yang Zhan, Alex Conway, Yizheng Jiao, Nirjhar Mukherjee, Ian Groombridge, Michael A. Bender, Martin Farach-Colton, William Jannen, Rob Johnson, Donald E. Porter, and Jun Yuan. Copy-on-abundant-write for nimble file system clones. *ACM Trans. Storage*, 17(1), jan 2021. https://doi.org/10.1145/3423495.

[248] Haoran Zhang, Adney Cardoza, Peter Baile Chen, Sebastian Angel, and Vincent Liu. Fault-tolerant and transactional stateful serverless workflows. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1187–1204, 2020.

[249] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. Kappa: A Programming Framework for Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 328–343, 2020.

[250] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing databases for fun and profit. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, page 449–464, USA, 2014. USENIX Association. https://dl.acm.org/doi/10.5555/2685048.2685083.

[251] Hua Zhong and Jason Nieh. Crak: Linux checkpoint/restart as a kernel module. Technical report, Citeseer, 2001.