The Aurora Single Level Store Operating System

Emil Tsalapatis
emil.tsalapatis@uwaterloo.ca
RCS Lab, University of Waterloo
Waterloo, Canada

Ryan Hancock
krhancoc@uwaterloo.ca
RCS Lab, University of Waterloo
Waterloo, Canada

Tavian Barnes
tbarnes@uwaterloo.ca
RCS Lab, University of Waterloo
Waterloo, Canada

Ali José Mashtizadeh
ali@rcs.uwaterloo.ca
RCS Lab, University of Waterloo
Waterloo, Canada

Abstract

Applications on modern operating systems manage their ephemeral state in memory and persistent state on disk. Ensuring consistency between them is a source of significant developer effort and application bugs. We present the Aurora single level store, an OS that eliminates the distinction between ephemeral and persistent application state.

Aurora continuously persists entire applications with millisecond granularity to provide persistence as an OS service. Aurora revists the problem of application checkpointing through the lens of a single level store. Aurora supports transparent and customized applications. The RocksDB database using Aurora's APIs achieved a 75% throughput improvement while removing 40% of its code.

CCS Concepts: **• Computer systems organization → Reliability**; *Secondary storage organization*; **Dependable and fault-tolerant systems and networks**; **• Software and its engineering → Operating systems**.

Keywords: single level store, transparent persistence, checkpoint/restore

ACM Reference Format:
Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. 2021. The Aurora Single Level Store Operating System. In *ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21), October 26–29, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3477132.3483563

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 [17, 53]. 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 [59]. The IBM compilers semi-automatically insert API calls into the application for persistence. Research systems like EROS also worked by providing a custom API [58].

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%.

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 [59]. 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 [58] and KeyKOS [41] 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 [51]. Previous systems that employ external synchrony demonstrate modest overheads when persisting or replicating data at millisecond granularity [8, 29, 47].

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 [11].

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 GB/s and a memory bandwidth of 205 GB/s [21].

Application Checkpointing. Application checkpointing is an extensively studied problem [10, 20, 30, 35, 38, 40, 54, 61, 65]. 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 [28, 50]. 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.

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

Table 1. A breakdown of CRIU's checkpointing overheads for a 500 MB Redis process.

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 [9] that is used for container migration [52]. 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 [12].

Table 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.

Incremental Checkpointing. Table 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 [48, 64, 65]. Many incremental checkpointing systems are used to provide transparent fault tolerance [8, 29] and improve record/replay for virtual machines [3, 39]. VAS-CRIU [64] 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 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.

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 2. A subset of the command line interface.

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 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 [51]. 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`.

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 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

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. A subset of the Aurora application API.

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 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.

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.

4.1 Aurora's Architecture

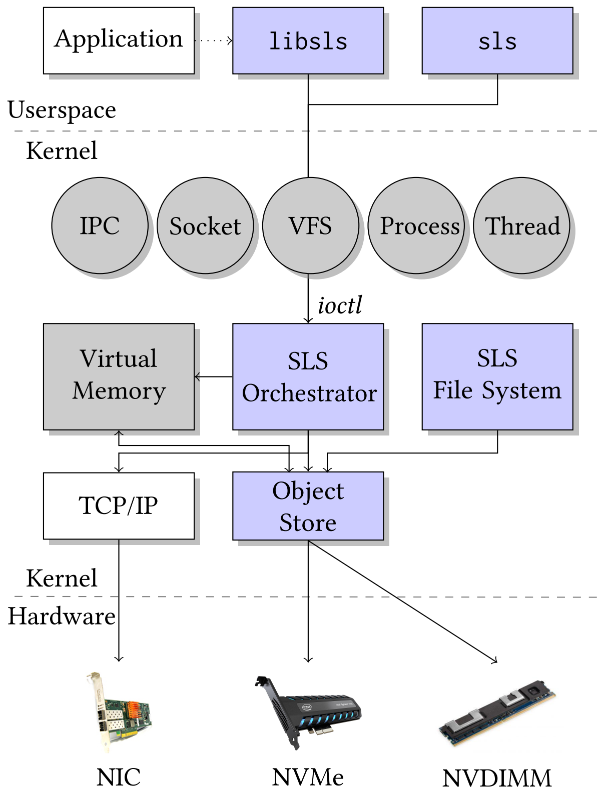


Figure 1. Aurora system diagram.

Figure 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 § 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 § 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 § 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.

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.

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 § 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.

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.

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.

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 [55]. 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 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.

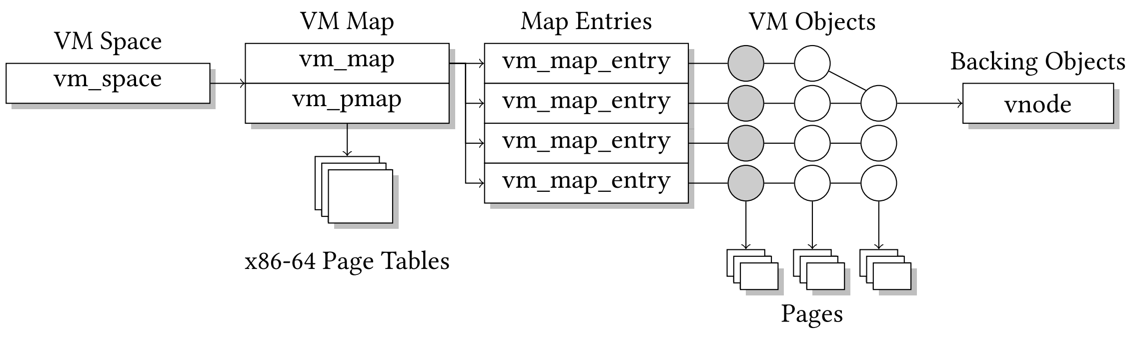


Figure 2. VM Memory Diagram of FreeBSD. Address spaces include page tables and a list of mapped regions. Each region is backed by mappable groups of physical pages called VM objects. VM objects represent anonymous or file memory and back each other to enforce COW semantics.

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.

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 [37] and ZFS [24] 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 [57], 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.

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 § 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 [51].

9 Evaluation

We evaluate the performance of Aurora with microbenchmarks and several applications. First, we evaluate the performance of the object store using FileBench [1]. We then evaluate the overhead of checkpointing POSIX objects within Aurora. We also measure the overhead for transparently checkpointing Memcached, a popular key-value store, and several other popular applications. We show the performance possible when using the Aurora API with the RocksDB key-value store. Finally, we use Redis, another key-value store, to compare Aurora to CRIU, a state of the art checkpoint system in Linux, and Redis' own builtin checkpointing.

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.

9.1 Aurora Object Store

We use FileBench to benchmark the file system and the object store. The benchmarks show that the Aurora file system, running at a 10 ms period, performs well compared to other file systems such as ZFS and FFS.

We compare the object store to ZFS (with and without checksumming) and FFS (with SU+J [2, 46] 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(a) and (b) show the object store's throughput for 4 KiB and 64 KiB random and sequential writes. We measure the write performance because it helps us understanding the rest of our benchmarks. 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 [45]. As an optimization for fragments FFS delays allocations to allow blocks to be promoted to full blocks for IO.

Figures 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(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.

9.2 Persisting POSIX Objects

Table 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. Checkpointing vnodes is fast when they reside in our object store because we only store a reference to the inode. Shared memory checkpoint times include the time spent shadowing, but not invaliding the MMU. System V is more costly than POSIX shared memory because Aurora scans the global SysV namespace. Checkpointing the Kqueue is slow because it contains 1024 event structures and requires locking each structure. Pseudoterminals are slow to restore because they require taking locks in the device file system when recreating the virtual device.

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 4. Checkpoint and restore times for POSIX objects.

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

9.3 Checkpointing Memory Objects

Table 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 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.

9.4 Application Checkpointing

Table 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

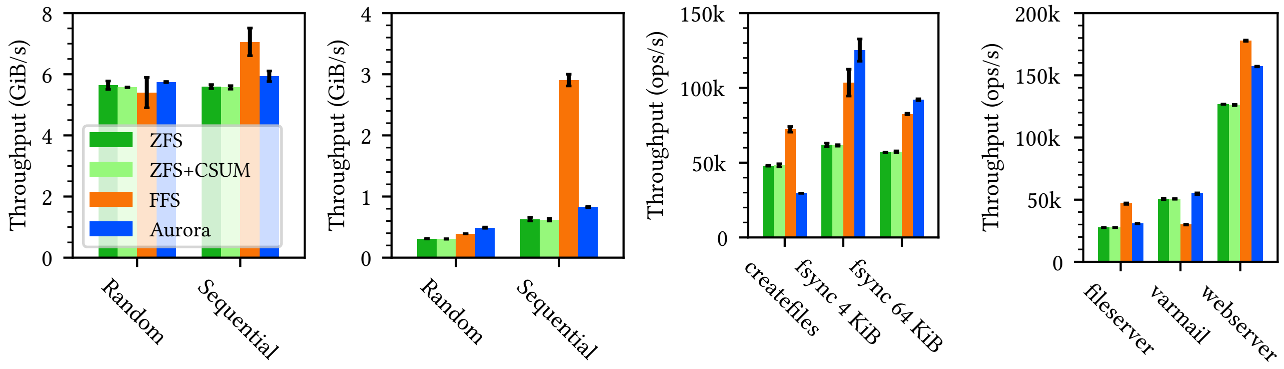


Figure 3. FileBench microbenchmarks comparing the Aurora file system to both ZFS and FFS

	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 6. Checkpoint stop times and restore times for several popular applications.

application server. 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. We show restores from memory and disk. Disk restores can either be full or lazy. Lazy restores only restore the minimal OS state needed to resume the application.

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.

9.5 Transparent Persistent for Memcached

Figure 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 [23] with four machines generating load and another machine

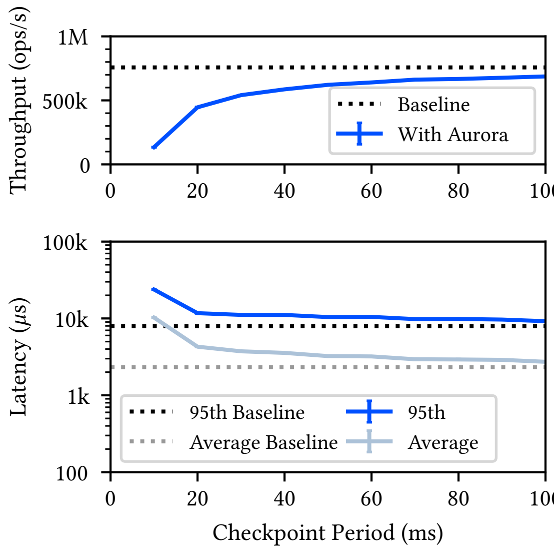


Figure 4. Memcached at max throughput over varying checkpoint periods.

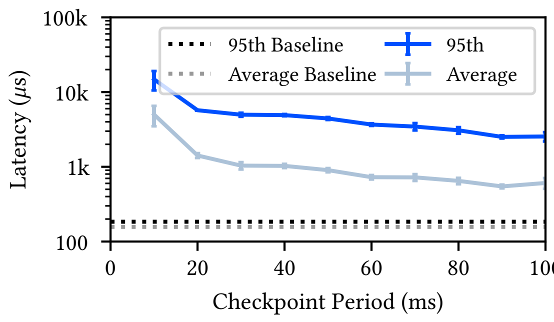


Figure 5. Memcached with throughput pegged at 120 k ops/s over varying checkpoint periods.

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×.

Figure 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 4, the latency impact of transparent persistence has a small effect once network queues begin to saturate and the base latency increases.

9.6 Customized RocksDB

We use RocksDB [19], 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.

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 [31].

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 WAL is full, RocksDB triggers an Aurora checkpoint and clears the WAL.

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 [25].

Figure 6 shows the results for each configuration of RocksDB. Figure 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.

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 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.

Figures 6(b) and 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 WAL achieves better 99$^{\text{th}}$ percentile latency than the unmodified RocksDB WAL, but the 99.9$^{\text{th}}$ percentile suffers as writes that trigger checkpoints must wait for the checkpoint to complete.

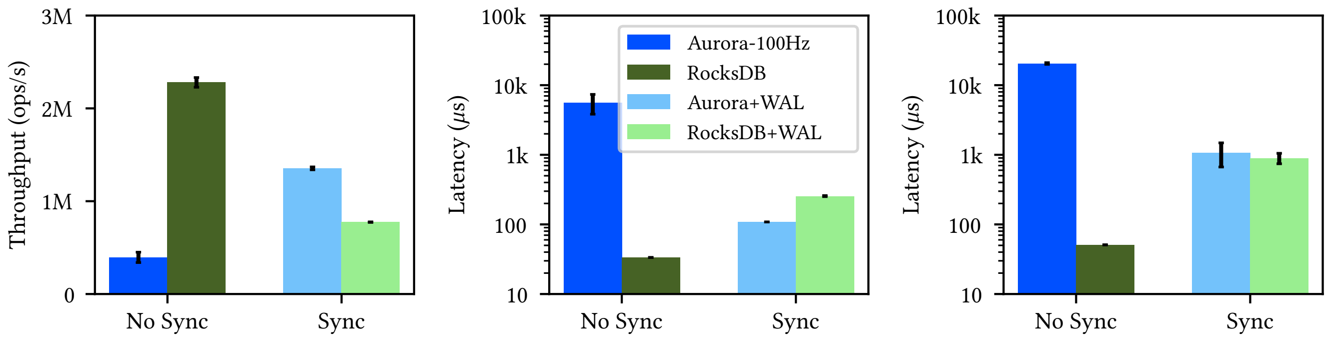
9.7 Comparison with Linux CRIU

Table 7 compares the full checkpoint performance of Aurora versus CRIU, the state of the art checkpointing system for Linux. 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.

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 slower than Aurora despite only saving the data rather than the whole process. The time to write out the data is also 3× slower than Aurora because of serialization overheads.



(a) Throughput **(b)** 99th Percentile Write Latency **(c)** 99.9th Percentile Write Latency

Figure 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”.

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 [28, 36, 43, 50, 66]. 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 [22, 27, 60]. Aurora can be used as a building block to enable migration and failover functionality in operating systems.

Checkpointing techniques used in databases [47] for fault tolerance require developer effort. Other systems rely on specialized hardware like NVRAM [49].

Serverless Computing: Checkpoint and restore has been used by serverless computing for snapshotting serverless functions [32, 62]. 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 [26, 33, 34, 42, 44, 56, 63] 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 [14–16] and key-value stores [13, 18, 19] 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 [4–7, 53] and PostgreSQL [17]. Aurora provides an alternative way to achieve persistence in databases, allowing for the reduction of code and developer effort.

11 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.

Acknowledgments

We thank Donald Porter, our shepherd, and the anonymous reviewers for their valuable feedback. We also thank Samer Al-kiswany, Haoyu Gu, Ihab Ilyas, Tamer Özsu, Bernard Wong and Oscar Zhao for their insight during discussions on the project. We thank Hongbo Zhang for his work on the initial Aurora prototypes.

References

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

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

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

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

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

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

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

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

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

[10] Dragonfly on-line manual pages : sys_checkpoint(2). https://man.dragonflybsd.org/?command=sys_checkpoint&section=2, April 2019.

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

[12] CRIU Release 3.6. https://criu.org/Download/criu/3.6, January 2021.

[13] LevelDB Source Repository. https://github.com/google/leveldb, January 2021.

[14] MongoDB: The most popular Database for Modern Apps . https://www.mongodb.com/, January 2021.

[15] MySQL Website. https://www.mysql.com/, January 2021.

[16] PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/, January 2021.

[17] PostgreSQL's fsync() surprise. https://lwn.net/Articles/752063/, January 2021.

[18] Redis Website. https://www.redis.io, January 2021.

[19] RocksDB | A persistent key-value store. https://www.rocksdb.org, January 2021.

[20] 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.

[21] Advanced Micro Devices, Inc. AMD EPYC 7003 Processors (Data Sheet). https://www.amd.com/system/files/documents/amd-epyc-7003-series-datasheet.pdf, 2021.

[22] T.E. Anderson, D.E. Culler, and D. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, 1995.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] David Cheriton. The V Distributed System. *Commun. ACM*, 31(3):314–333, March 1988.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] Paul H Hargrove and Jason C Duell. Berkeley lab checkpoint/restart (BLCR) for Linux clusters. In *Journal of Physics Conference Series*, volume 46, pages 494–499, 2006.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] Oren Laadan and Jason Nieh. Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems. In *USENIX Annual Technical Conference*, pages 323–336, 2007.

[41] C.R. Landau. The Checkpoint Mechanism in KeyKOS. pages 86 – 91, 10 1992.

[42] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.*, 36(4):471–482, April 1987.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the sync. *ACM Trans. Comput. Syst.*, 26(3), September 2008.

[52] 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.

[53] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, October 2014. USENIX Association.

[54] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing Under Unix. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 18–18, Berkeley, CA, USA, 1995. USENIX Association.

[55] 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, pages 31–39, Washington, DC, USA, 1987. IEEE Computer Society Press.

[56] Michiel Ronsse and Koen De Bosschere. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. Comput. Syst.*, 17(2):133–152, May 1999.

[57] 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.

[58] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A Fast Capability System. *SIGOPS Oper. Syst. Rev.*, 34(2):21–22, April 2000.

[59] Frank G Soltis. *Fortress Rochester: The Inside Story of the IBM iSeries*. 29th Street Press, 2001.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

[66] 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.