# The Aurora Operating System

## Revisiting the Single Level Store

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

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

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

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

## 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, yet still a source of significant bugs in mature applications. We present the Aurora single level store (SLS), an OS that simplifies persistence by automatically persisting all traditionally ephemeral application state. With recent storage hardware like NVMe SSDs and NVDIMMs, Aurora is able to continuously checkpoint entire applications with millisecond granularity.

Aurora is the first full POSIX single level store to handle complex applications ranging from databases to web browsers. Moreover, by providing new ways to interact with and manipulate application state, it enables applications to provide features that would otherwise be prohibitively difficult to implement. We argue that this provides strong evidence that manipulation and persistence of application state naturally belong in an operating system.

## CCS CONCEPTS

**• Software and its engineering → Operating systems**; **• Computer systems organization → Secondary storage organization**; **Reliability**; **Dependable and fault-tolerant systems and networks**; **• Information systems → Storage architectures**.

## KEYWORDS

single level stores, transparent persistence, snapshots, checkpoint/restore

**ACM Reference Format:**
Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. 2021. The Aurora Operating System: Revisiting the Single Level Store. In *Workshop on Hot Topics in Operating Systems (HotOS '21), June 1-June 3, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3458336.3465285

## 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 [13, 41]. Instead, applications solely use memory 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 [45] 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 [46].

Aurora revisits the single level store with three main contributions. First, we depend on improvements in hardware to achieve performance and functionality including new flash storage and large virtual address spaces. Second, we develop an architecture designed to support both unmodified and modified POSIX applications. Third, we expand the concept of a single level store with new primitives for the manipulating execution state to enable novel applications.

Aurora also accurately captures application state by treating all POSIX primitives (e.g., Unix domain sockets, System V shared memory, and file descriptors) as first class objects, rather than as parts of a process. This allows Aurora to handle applications composed of processes that share memory or files in arbitrary ways, without duplicating work or leaving edge cases unhandled. Using this approach, Aurora supports complex programs like the Firefox web browser, the RocksDB key value store, and the Mosh remote shell.

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.

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

## 2 BACKGROUND

Single level stores have existed for decades both in industry and academia [20, 23, 32, 45, 46]. These systems simplified software development by providing persistence, but were not POSIX compatible and not transparent. Developer effort was required for correctness and performance. They used incremental checkpointing [51] to persist applications at regular intervals with runtime and/or application specific hooks. These systems were severely limited by the speed of storage devices at the time, e.g., the EROS research OS spent a large effort on masking spinning disk latency [45].

Existing persistent-by-default designs like The Machine [31] and Twizzler [17] are not transparent and depend on special hardware. The Machine was an attempt to build a supercomputer based on memristors, while Twizzler is an OS that uses only NVDIMMs for storage. These systems break compatibility with existing systems in that they depend on the byte addressability of persistent storage. Single level stores like Aurora conversely use regular DRAM and disks, and hide the distinction between the two from the application.

Aurora makes a key observation that device bandwidth and latency has improved to rival the memory bus. Modern CPUs can provide an aggregate PCIe bandwidth up to 256 GB/s, more than that of memory [16]. New Intel 3D XPoint SSD's reduce IO latency to 10 $\mu s$ [8], within two orders of magnitude of memory. The combination of high bandwidth and low latency makes transparent persistence possible without needing byte addressability.

Popular checkpoint/restore mechanisms have been used for scientific computing to recover from failures and migrate workloads [29, 42, 47]. These systems do not checkpoint frequently enough to provide transparent persistence and the resulting checkpoints are not self contained.

Checkpointing of virtual machines (VM) has enjoyed a lot of popularity and applications. VMs package the application and all dependencies into a portable checkpoint. Live migration and incremental checkpointing have enabled distributed resource management, fault tolerance and other applications [5, 24, 36, 38].

Containers, which have less overhead than virtualization, have traditionally lacked these features. Providing the same functionality for containers enables the same distributed resource management and fault tolerance applications. Systems like CRIU [6], the standard for Linux container migration [40], piece together application state by querying the kernel through system calls and the `proc` file system.

While CRIU's performance is tolerable for migration, its overheads are prohibitive for other applications including transparent persistence. Even research systems that optimize memory checkpointing in CRIU have failed to reduce overheads enough [50]. Furthermore, CRIU is incredibly complex, requiring 7 years to properly add UNIX socket support [7].

Aurora is different from previous systems in two ways. First, rather than checkpointing objects exposed at the system call boundary it provides persistence throughout the OS. This includes internal kernel state, the file system, and the virtual memory subsystem. Second, Aurora treats all POSIX primitives as first class objects, persisting and replicating the

POSIX object model as seen by the kernel. Aurora's design enables broad application support comparable to CRIU with a far simpler design.

*Serverless Computing and Scale Out:* In the serverless paradigm, developers write small stateless *functions* that run in the cloud. Invoking a function involves creating a new container or VM and starting the application, an operation that adds significant latency.

Frameworks and language based approaches have reduced invocation latencies by colocating functions [26, 52], or by restoring initialized application checkpoints. Catalyzer [25] achieves submillisecond restore times from in memory checkpoints using a combination of virtualization and kernel based techniques unique to the gVisor application kernel.

Aurora's restore mechanism enables autoscaling and fast starts using OS containers. Aurora's restore times from disk rival the state of the art because of lazy restores (see § 3) and cooperative warm ups (see § 4). The object store also deduplicates otherwise unrelated checkpoints on disk for higher storage density.

*Debugging and Speculation:* Record/replay systems [21, 27, 28, 33, 35, 44, 49] record non-deterministic inputs to an application to replay the complete execution. Recordings can be large and checkpointing has been a common technique for bounding the storage overhead. Aurora's low overhead checkpointing makes record/replay practical in production, enabling developers to capture an application moments before a crash.

Limited forms of checkpoint/restore are used for application level speculation, which restores application memory to a prior state. This operation has been proposed as an OS service implemented using hardware virtualization for speculative execution and efficient backtracking [19]. Rollbacks have also been used for application security, namely intraprocess isolation [34].

*Databases:* Databases [10–12] and key-value stores [9, 14, 15] often spend a significant amount of their code managing persistence and 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 [1–4, 41] and PostgreSQL [13]. Databases also implement memory tracking in software, which duplicates the work of the hardware MMU and the operating system.

Aurora gives developers fine-grained control over persistence behavior for modified applications. An area of ongoing work is developing a richer API for applications to communicate with the OS to further optimize paging and persistence.

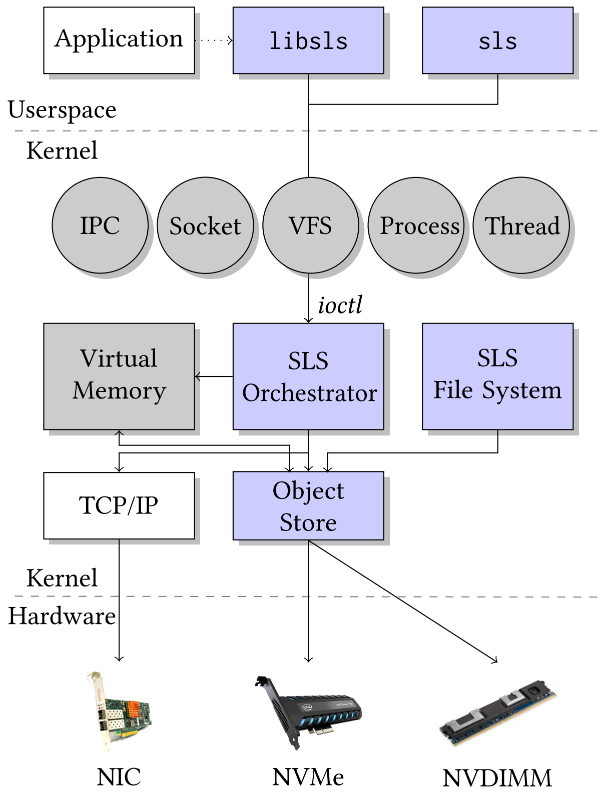

**Figure 1: Basic system diagram**

## 3 THE AURORA OPERATING SYSTEM

Figure 1 shows the architecture of the Aurora persistent operating system. Aurora has three components: the SLS orchestrator, the object store, and a custom file system. Each POSIX object in the operating system (e.g., socket pairs, POSIX shared memory or System V message queues) contains code that continuously serializes and stores the state in the object store. Each object is serialized independently, and contains enough user and kernel state to recreate the object on restore.

The SLS orchestrator maps kernel objects to the on-disk store and manages the checkpoint and resume operations. Aurora provides persistence for individual processes, process trees or containers. The orchestrator provides serialization barriers across the entire OS to provide consistent application-wide checkpoints. All processes are momentarily paused and remaining unflushed state is copied into memory buffers or tracked using copy-on-write (COW). These updates are flushed asynchronously to disk. In our current prototype this occurs up to 100× per second with modest overhead. The orchestrator also manages restores by recreating all POSIX objects and resuming applications.

Aurora uses an optimized and custom COW mechanism for its applications. The standard COW scheme used by `fork` breaks shared memory semantics by forcing each process within an application to create their own private copy on a write. The OS normally prohibits marking shared pages as COW to avoid this problem. Aurora instead modifies FreeBSD's Mach derived VM subsystem [43] to create a new page shared between all processes on a copy-on-write fault, while Aurora flushes the original page.

Aurora optimizes checkpoints by only flushing dirtied data on every checkpoint. It implements *incremental checkpointing* using our custom COW mechanism that tracks writes between successive checkpoints. It thus never flushes the same page twice for shared memory or COW memory regions. On restore, Aurora faithfully reproduces the entire memory hierarchy to preserve page deduplication.

The object store simplifies synchronizing memory and file system checkpoints. The snapshot operations of popular file systems are too slow to keep up with the SLS orchestrator. The object store does hundreds of checkpoints per second using a lower overhead COW layout than that of WAFL [30] and ZFS [18] that snapshot less frequently. The COW layout enables in-place garbage collection without needing to rewrite incremental checkpoints.

Applications are placed into a *persistence group* attached to one or more backing devices. Locally persistent applications are backed by NVMe flash or NVDIMMs when available. Applications can also be remotely persisted through the network backend to another host. For debugging and speculative execution applications can use a local memory backend to store ephemeral checkpoints. Aurora allows for attaching multiple backends at the same time, e.g., sending an application's incremental checkpoints to both a local disk and a remote machine for replication.

The Aurora file system provides a file API into the object store that enables functionality and fixes limitations in POSIX file systems. Users can create zero copy snapshots and clones of a container including process and file system state. The file system must handle special edge cases that are not normally considered in a typical POSIX file system.

An example of an edge case is unlinked but open files (i.e., anonymous files). In POSIX file systems, these files would be reclaimed after a crash, preventing application restoration. We solve this by maintaining an on-disk *open reference count* storing the number of persistent virtual file system vnodes.

Integrating swap with Aurora optimizes restores by lazily paging application memory. Aurora restores the minimal application state, including the OS and CPU state, with memory effectively swapped out. Applications fault in their working set during execution. Aurora uses the clock page replacement algorithm [22] to optimize restore by eagerly paging in the hottest pages to avoid excessive page faults. When pages are swapped out due to memory pressure they are incorporated into the subsequent checkpoint.

| Command | Description |
|---|---|
| `sls persist` | Add an application to a persistence group |
| `sls attach` | Attach a persistence group to a backend |
| `sls detach` | Detach a persistence group from a backend |
| `sls checkpoint` | Checkpoint an application |
| `sls restore` | Restore an application from an image |
| `sls ps` | List applications in Aurora |
| `sls send` | Send an application to a remote |
| `sls recv` | Receive an application from a remote |

**Table 1: A subset of the command line interface available to users.**

| Function | Description |
|---|---|
| `sls_checkpoint()` | Create an image |
| `sls_restore()` | Restore a checkpoint |
| `sls_rollback()` | Roll back state to last checkpoint |
| `sls_ntflush()` | Non-temporal flush (outside checkpoint) |
| `sls_barrier()` | Wait for a checkpoint to be flushed |
| `sls_mctl()` | Include/exclude memory regions |
| `sls_fdctl()` | Enable/disable external consistency |

**Table 2: A subset of the API available to developers from the Aurora library.**

## 3.1 Command Line Interface

To help illustrate Aurora we show the usage of a subset of the command line interface as shown in Table 1. Users begin by enabling transparent persistence of an application using the `persist` command. This registers all OS state that is part of the application with the orchestrator. By default the application is persisted 100× per second. The host and each container have their own persistence group. The `attach` and `detach` commands allow the user to register backends with persistence groups.

Users can manually create named checkpoints with the `checkpoint` command and view all application checkpoints in Aurora using `ps`. Users can restore to a point in time or resume execution of a persistence group after a system crash using `restore`. Aurora uses free space on-disk to provide a short execution history as incremental checkpoints.

Users can easily share or migrate applications using the `send` and `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 to give to another user, live migrate the application, or provide fault tolerance.

### 3.2 Aurora API

Table 2 shows the Aurora API that custom applications use to control and optimize persistence. Applications can manually initiate checkpoints, restore, or roll back checkpoint state.

Applications control persistence using `sls_barrier` as a persistence barrier for a given thread and `sls_ntflush` to initiate a low latency flush of an append-only log to a storage medium. These applications require custom code during restore to repair data structures based on the log.

Applications can also control the behavior of memory and other resources. Using `sls_mctl`, applications can selectively include/exclude memory regions and hint to Aurora the best policy for lazy restore. Developers can control the external consistency per file descriptor using `sls_fdctl`.

External consistency [39] is enforced when a communications spans a persistence group boundary (including remote hosts). Any data transmitted on a file descriptor are buffered until the corresponding checkpoint is persisted on disk to prevent other machines from seeing state that could be lost in a crash. If the remote application can handle observing such state, the developer can disable external consistency to improve latency.

## 4 APPLICATIONS OF AURORA

In this section we show how Aurora's abstractions enable very different applications including serverless computing and scale out, debugging and speculation, and databases.

*Serverless Computing and Scale Out.* Aurora can be used to optimize serverless warm starts using its lazy restore, combined with its ability to distribute and scale function runtimes. Serverless function warm starts are similarly implemented by restoring a container with the function already initialized. Scaling out amounts to repeatedly restoring an already checkpointed application.

Aurora's COW design maximizes function density in persistent storage by deduplicating shared runtime memory between different functions. The object store represents each function as a small delta over the runtime container's checkpoint. All functions share this data, allowing machines to potentially hold billions of functions.

Similar functions share unmodified pages, increasing function density and memory efficiency. This sharing causes instances to *warm each other up*: an instance faulting a page into memory shares it with the rest using COW. Recent research [48] shows that the working set of many workloads is almost identical between invocations, leading Aurora to eliminate major faults for popular functions.

*Debugging and Speculation.* Aurora can also provide improvements to application debugging. Aurora creates periodic checkpoints of a running application that can later be inspected with a debugger or executed. We can use this to build a type of time travel debugger [37] or, since new incremental checkpoints leave old ones intact, to bisect the history to find violations of invariants. Repeatedly restoring from the same image can uncover nondeterministic failures that do not manifest on every execution. We regularly used this while developing Aurora itself.

Aurora integrates with record/replay systems to bound record log size by only keeping the records since the last checkpoint. On a failure, the application is rolled back to this checkpoint and replays the remaining log. Developers can thus witness the last seconds before a crash on a production machine with a very small disk and CPU overhead compared to standalone RR.

Aurora's rollback primitive allows apps to implement speculative execution for increased performance. For example, a client sending data to a server can execute assuming that the server received it, saving a round trip's worth of time. If the transfer ends up failing, the client rolls back to before it sent the data. Aurora notifies the client of the rollback, allowing it to try a more conservative code path.

*Databases.* Aurora aims to provide a clean memory management interface between database applications and the OS. Databases communicate application and workload specific memory management policies to Aurora, and use checkpointing to trigger data transfers to and from storage. Aurora allows applications to checkpoint data without associated execution state, providing an explicit persistence primitive that does not suffer from the semantic complexities of file and memory syncing.

Aurora's APIs provide a drop in replacement for common persistence mechanisms found in key value stores. We use Aurora's persistent log (`sls_ntflush`), manual checkpoints (`sls_checkpoint`) and barriers (`sls_barrier`), to replace existing persistence mechanisms in RocksDB that uses a log structured merge tree and Redis that uses `fork` for checkpoints with a write ahead log. In the case of Redis our initial port is already faster with less code.

## 5 PRELIMINARY RESULTS

Aurora is a fully working system that we continue to develop, comprising ~19k SLOC on top of FreeBSD 12.1. We have been testing the system on a dual 2.1 GHz Intel Xeon Silver 4116 CPU (Skylake-SP) with 96 GiB of RAM, four Intel Optane 900P NVMe drives and an Intel X722 10 GbT network interface card. We continue to explore how to improve APIs for different use cases and improve performance by optimizing various system components.

We have evaluated Aurora's fitness for the use cases we outlined by measuring the checkpoint and restore times of two workloads: Redis, a key value store, with a working set

| **Checkpoint** | **Full** | **Incremental** |
|---|---|---|
| Metadata copy | 267.9 $\mu s$ | 239.7 $\mu s$ |
| Lazy data copy | 5145.9 $\mu s$ | 711.1 $\mu s$ |
| Application stop time | 5413.8 $\mu s$ | 950.8 $\mu s$ |

**Table 3: Stop time breakdown when checkpointing a Redis instance with a 2 GiB working set. Aurora flushes data in the background concurrently with application execution.**

of 2 GiB, and a smaller hello world application. Redis represents heavier applications that use Aurora for persistence or debugging purposes, and the hello world app represents serverless functions.

We checkpointed Redis both by copying its entire address space, and by using incremental checkpointing. The results in Table 3 show that, while the cost of grabbing metadata is the same in both cases, lazily copying the address space using COW is 7× faster with incremental checkpointing, for a total stop time below 1 *ms*. In neither case does Redis stop to wait for the data to reach storage, due to Aurora's external consistency semantics.

Application overhead includes the stop time for each checkpoint and the cost of servicing COW faults while the application runs. Most of the stop time is spent applying COW tracking through page table manipulations. Checkpointing frequency is bounded by the speed with which Aurora can flush incremental checkpoints to disk.

Next, we restored the same Redis instance after bringing the image to memory, as we would do when using Aurora for debugging. According to the results in Table 4, restoring takes around 755 $\mu s$, two thirds of which are spent recreating the address space. No memory is copied, since Aurora uses COW semantics to share pages between the image and the running application.

Finally, we measured the potential improvement in serverless function startup time by restoring a small workload both from memory and disk. Table 4 includes the results, which show that the only extra latency when restoring from backing storage is due to bringing in the function metadata from the object store. In both cases, restore times are well under a millisecond. Restoring metadata state for disk restores is slightly faster, because reading in the checkpoint implicitly restores some application state.

| **Restore** | **Redis** | **Serverless** | **Serverless** |
|---|---|---|---|
| Backend | Memory | Memory | Disk |
| Object Store Read | N/A | N/A | 322.7 $\mu s$ |
| Memory state | 494.4 $\mu s$ | 144.6 $\mu s$ | 122.6 $\mu s$ |
| Metadata state | 261.1 $\mu s$ | 240.4 $\mu s$ | 206.9 $\mu s$ |
| Total latency | 755.5 $\mu s$ | 454.4 $\mu s$ | 652.2 $\mu s$ |

**Table 4: Restore time breakdown for a Redis instance with a 2 GiB working set from an in-memory image, and a serverless workload from an in-memory image and from disk. Reading data from the object store implicitly restores some state, causing lower memory and metadata restore times.**

## 6 CONCLUSION

We present Aurora, the first modern single level store, and the first to provide a complete POSIX interface. Aurora's performance characteristics show that, due to recent storage hardware advances, SLSes are once again a practical approach to OS design. SLSes provide both a simple and elegant solution for application and data persistence, and a flexible tool for manipulating execution state in new and interesting ways.

We have applied Aurora to a wide variety of domains, including serverless computing and scale out, debugging and speculation, and databases. In these domains, the combination of simple but efficient persistence and extremely fast checkpoints and restores allows us to provide both performance benefits and new functionality while reducing application complexity. Many of its benefits are available to applications without any modifications at all; additional benefits are available through a simple but powerful API.

We believe that Aurora will enable creating new and complex applications with a fraction of the effort they would otherwise need, and will pave the way for new research into application/OS co-design.

## ACKNOWLEDGMENTS

The authors thank Haoyu Gu, Samer Al-Kiswany, Bernard Wong and Oscar Zhao for fruitful discussions during Aurora's development. We would also like to thank the anonymous HotOS reviewers for their valuable feedback. The authors would like to thank Hongbo Zhang for his work on Aurora's initial prototype. This research is supported by an NSERC Discovery grant, Waterloo-Huawei Joint Innovation Lab grant, and NSERC CRD grant.

## REFERENCES

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

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

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

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

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

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

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

[8] 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.

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

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

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

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

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

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

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

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

[17] 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.

[18] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. Technical report, 2003.

[19] 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.

[20] 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.

[21] 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.

[22] Fernando J Corbato. A paging experiment with the multics system, 1968.

[23] 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.

[24] 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.

[25] 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*, pages 467–481, 2020.

[26] Vojislav Dukic, Rodrigo Bruno, Ankit Singla, and Gustavo Alonso. Photons: Lambdas on a Diet. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 45âĂŞ59, New York, NY, USA, 2020. Association for Computing Machinery.

[27] 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. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (Copyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading)*, OSDI '02, page 211âĂŞ224, USA, 2002. USENIX Association.

[28] 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, pages 121–130, New York, NY, USA, 2008. ACM.

[29] 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.

[30] 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.

[31] Kimberly Keeton. The Machine: An Architecture for Memory-Centric Computing. In *Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, volume 10, 2015.

[32] 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.

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

[34] 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.

[35] 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 693âĂŞ708, New York, NY, USA, 2017. Association for Computing Machinery.

[36] 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.

[37] 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. IEEE, 2016.

[38] 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.

[39] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, page 1âĂŞ14, USA, 2006. USENIX Association.

[40] 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.

[41] 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.

[42] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. Libckpt: Transparent Checkpointing under Unix. In *Usenix Winter 1995 Technical Conference*, 1995.

[43] 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 31âĂŞ39, Washington, DC, USA, 1987. IEEE Computer Society Press.

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

[45] 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.

[46] Frank G. Soltis. *Fortress Rochester: The Inside Story of the IBM iSeries*. May 2001.

[47] 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.

[48] 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.

[49] 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.

[50] 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 53âĂŞ65, New York, NY, USA, 2019. Association for Computing Machinery.

[51] 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, page 197âĂŞ209, New York, NY, USA, 2015. Association for Computing Machinery.

[52] 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.