TreeSLS: A Whole-system Persistent Microkernel with Tree-structured State Checkpoint on NVM

Fangnuo Wu[1,2], Mingkai Dong[1,2], Gequan Mo[1], and Haibo Chen[1,2]
[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University
[2]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China

Abstract

Whole-system persistence promises simplified application deployment and near-instantaneous recovery. This can be implemented using single-level store (SLS) through periodic checkpointing of ephemeral state to persistent devices. However, traditional SLSs suffer from two main issues on checkpointing efficiency and external synchrony, which are critical for low-latency services with persistence need.

In this paper, we note that the decentralized state of microkernel-based systems can be exploited to simplify and optimize state checkpointing. To this end, we propose TreeSLS, a whole-system persistent microkernel that simplifies the whole-system state maintenance to a capability tree and a failure-resilient checkpoint manager. TreeSLS further exploits the emerging non-volatile memory to minimize checkpointing pause time by eliminating the distinction between ephemeral and persistent devices. With efficient state maintenance, TreeSLS further proposes delayed external visibility to provide transparent external synchrony with little overhead. Evaluation on microbenchmarks and real-world applications (e.g., Memcached, Redis and RocksDB) show that TreeSLS can complete a whole-system persistence in around 100 μs and even take a checkpoint every 1 ms with reasonable overhead to applications.

CCS Concepts: • **Software and its engineering → Operating systems**; **Checkpoint / restart**; • **Computer systems organization → Reliability**; *Secondary storage organization.*

Keywords: Single-Level Store, Microkernel, Non-volatile Memory, Checkpoint/Restore, Transparent Persistence

ACM Reference Format:
Fangnuo Wu, Mingkai Dong, Gequan Mo, and Haibo Chen. 2023. TreeSLS: A Whole-system Persistent Microkernel with Tree-structured State Checkpoint on NVM. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3600006.3613160

1 Introduction

Prevailing operating systems run on main memory and store data persistently in storage via files, following a convention formed in early systems where memory is fast, byte-addressable but volatile, and disks are non-volatile but slow with block interfaces. Consequently, applications move data back and forth between a two-tiered memory-storage hierarchy. Such a design may lead to bugs in complicated data persistence operations, e.g., crash consistency bugs in file systems [50, 67, 79] and in applications using these file systems [58], as well as performance degradation of data swapping between memory and storage [13].

Single-level store (SLS) is proposed to liberate applications from the complicated and error-prone data persistence, while providing near-instantaneous recovery. Unlike traditional systems exposing storage via file systems, SLS suggests using checkpointing to extend the memory layer downwards to include the disks, managing data (both permanent and ephemeral) and system state together with transparent persistence. As a result, applications execute and store data in the "single-level" memory while the operating system automatically persists data. Several systems in the past decades [12, 33, 40, 48, 62, 66, 68, 71, 74] have proposed the design and implementation of SLS on traditional storage devices. A recent system called Aurora [72, 73] further demonostrates how SLS designed for fast NVMe devices can mitigate the performance issues.

Despite the performance improvement of storage devices, the use of SLS is still limited by its *high performance overhead* and *external synchrony issue* [54] (further explained in §2.4). Existing SLSs eliminate the complexity of writing persistent applications by providing an illusion of single-level storage on top of runtime memory (DRAM) and storage (disks) with software checkpointing. Though such an illusion hides the significant differences of runtime memory and storage in both access speed and access granule (byte vs. block), it exacerbates the performance overhead caused by write amplifications, and increases the risk of data loss due to limited checkpoint frequency.

On the other hand, modern applications heavily rely on immediate persistence of external visible operations, i.e., external synchrony, to provide services to external entities like a client. Existing SLSs provide custom APIs for applications to ensure external synchrony. However, nontrivial efforts[1] are required to modify applications to accommodate the APIs for correct data persistence, which contradicts SLS's original intention of liberating applications from error-prone persistence operations.

The emergence of fast, byte-addressable non-volatile memory (NVM) presents promising features as a storage device to implement SLS with minimized performance overhead. NVM combines storage-like durability with DRAM-like byte-addressability and access performance, forming a single-level storage device that enables *fast and direct manipulation of persistent data*.

Unfortunately, even with a single-level storage device, implementing an efficient SLS still faces challenges since running on persistent memory does not make the whole system persistent naturally. Although recent techniques like Intel eADR [3] can guarantee the eventual persistence of data in CPU cache, data stored in CPU registers and device registers can still be lost upon power failures. As a result, software techniques like checkpointing are still required to guarantee the persistence of consistent whole-system state. Unlike traditional checkpointing techniques that prepare consistent state in memory and flush to storage in batch, NVM acts as both runtime memory and storage, making efficient checkpointing even more complicated. Meanwhile, as *external synchrony* becomes more prevalent in modern applications, it becomes necessary to provide *transparent external synchrony*, without forcing applications to use the error-prone journaling mechanism.

In this paper, we propose TreeSLS, an efficient SLS on NVM with whole-system persistent microkernel. TreeSLS builds on the principle of microkernel, which keeps kernel functionalities as minimal as possible and pushes most system services, e.g., drivers, network stacks and file systems, to the user space. Based on the observation that state-of-the-art microkernels [41, 66] pervasively uses capabilities [26] to manage kernel objects (like seL4's capability deviation tree), TreeSLS organizes whole-system state in a capability tree for efficient incremental state checkpointing. It further designs an in-kernel failure-resilient checkpoint manager to manage the use of non-volatile memory. The checkpoint manager uses journaling to protect itself from failures, and whole-system checkpoints are taken on the tree to ensure a runtime view is available to run applications while a consistent view is always persisted to deal with unexpected power failures.

TreeSLS proposes NVM-oriented checkpointing on the capability tree to reduce the stop-the-world pause time. To further reduce the runtime overhead caused by page faults and memory copying, TreeSLS proposes *hybrid copy* to track hot pages and conduct cross-DRAM/NVM migration and speculative page copy.

With the extreme high checkpoint frequency (e.g., one checkpoint per millisecond) enabled by efficient checkpointing, TreeSLS also provides application-transparent external synchrony via delayed external visibility. Such an approach offloads the complexity of external synchrony to the system services such as drivers, while liberating applications from writing efficient and correct persistence code.

We evaluate TreeSLS on a series of well-known applications to show that TreeSLS can successfully and efficiently checkpoint the whole system every 1 ms under various scenarios. We also conduct experiments with memory servers (e.g., Redis and Memcached) and to demonstrate that real-world applications running on TreeSLS can benefit from the simplified persistence model and fast whole-system checkpointing. TreeSLS can achieve up to 2.2× throughput of Redis with original AOF function enabled, and achieves 2.4× and 2.5× throughput of Aurora's journaling API and RocksDB's WAL, respectively.

In summary, the paper makes the following contributions.

- The first NVM-based single-tier SLS that organizes whole system state in a capability tree and the checkpoint manager under the microkernel architecture.
- Efficient whole-system checkpointing consisting of *NVM-oriented checkpoint methods* to reduce the stop-the-world checkpointing time and *hybrid copy* to reduce runtime overhead.
- Transparent external synchrony with delayed external visibility based on the high checkpointing frequency enabled by TreeSLS.

The source code and further information about TreeSLS are at https://ipads.se.sjtu.edu.cn/projects/treesls.html.

2 Background and Motivation

2.1 Memory-storage Hierarchy

As shown in Figure 1, there have been various approaches to providing persistence and crash consistency for applications. Prevailing operating systems run in volatile main memory (DRAM) and move data to non-volatile storage devices (disks) to guarantee persistence. Applications also run with runtime data in DRAM and leverage filesystem APIs to persist and retrieve data in storage. However, data movement between DRAM and persistent storage is costly and (de-)serializations are required when the data formats in

[1]The Aurora paper reports that 109 SLOC additions are sufficient for RocksDB. However, such small code changes are largely due to the existing logging mechanism in the RocksDB design.

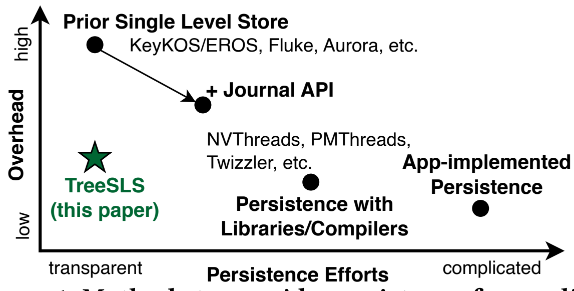


Figure 1. Methods to provide persistence for applications. TreeSLS aims to provide low-overhead, transparent persistence for applications.

DRAM and storage are different. Further, applications need to guarantee the persistence of application data and its crash consistency via mechanisms such as journaling, which are usually complicated and error-prone.

2.2 Single-Level Store

The single-level store (SLS) model [12, 40, 71] offers an alternative approach to providing a single-tier storage abstraction for applications, where every object in the system can be uniformly accessed. It extends the memory layer downwards to include the disk level and removes the file system abstraction. With this persistence provided, programmers can write applications with no persistence-related code under the assumption that the system never crashes. Existing applications designed for memory can also gain persistence support transparently with SLS. For example, in-memory key/value stores (e.g., Memcached) in cache servers, which serve as fast caches to the backend database, can benefit from the transparent persistence of SLS to avoid hours of warm-up time [31, 49] after a reboot caused by power failures. Due to SLS's promise of simplified programming and transparent recovery, much work [12, 15, 23–25, 33, 38, 48, 62, 63, 65, 66, 68, 71, 74] has treated all objects as in-memory and transparently convert objects between in-memory and on-disk representations. However, the performance of such work is highly limited by the devices available at that time and their checkpoint can only be taken at minute-level intervals. Recently, the high-performance storage devices brings up the SLS concept again [52, 72, 73].

2.3 Limitations of Existing SLSs

Though named as single-level store, most prior SLSs actually build upon the two-tiered architecture of runtime memory and storage, and provide the single-level persistence illusion with software checkpointing. Despite the performance improvement of storage devices, these SLSs still suffer from limitations due to the two-tiered architecture.

Figure 2 shows the architecture of two representative systems: (a) EROS [66], a capability-based microkernel with system-wide checkpointing to provide transparent persistence; (b) Aurora [73], a modern SLS for UNIX OSs on fast NVMe devices. These SLSs need several additional cache

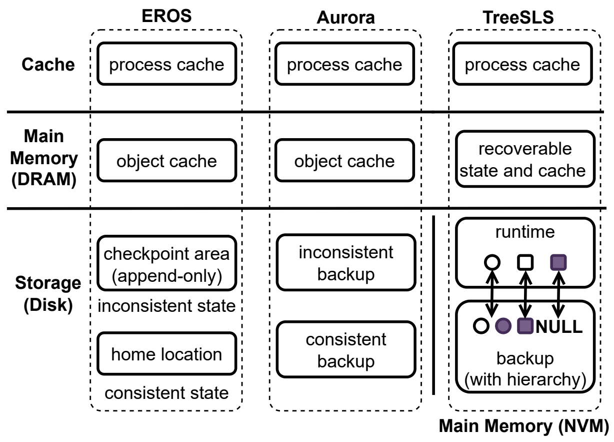


Figure 2. Architectures of prior SLSs and TreeSLS. Prior SLSs build an illusion of single-level storage upon the volatile runtime and persistent disks or SSDs. TreeSLS builds on the single-tier NVM acting as both runtime memory and storage. DRAM can store recoverable state and cache frequently accessed pages in NVM.

layers upon non-volatile storage devices since CPUs cannot directly access on-disk data. Objects in EROS are cached at two levels of abstraction: process cache and object cache. Both layers are write-back caches of the on-disk objects and are flushed when the objects they cache are invalidated. In Aurora, though using modern high-performance storage, objects still need to be put in DRAM and flushed to persistent devices. *The unmatched interface of runtime memory and storage* leads to inefficiency in two aspects:

- *Write-amplification incurs performance degradation.* CPUs manipulate runtime data (e.g., objects) in bytes, while data are persisted to storage in blocks. Thus, a small change in runtime data will cause the persistence of a whole block, which causes write amplification. Although EROS leverages an append-only checkpoint area and Aurora adopts batching, the write amplification still causes non-trivial overhead .
- *Limited checkpoint frequency exposes more data loss at risk.* As accessing storage is much slower than DRAM, prior SLSs take a checkpoint by stopping the world and copying dirty data to dedicated DRAM buffers. Such data is flushed to storage asynchronously by background threads which run concurrently with applications after the checkpointing. Such asynchronous checkpointing reduces the stop-the-world pause time, but excludes *immediate persistence* and greatly limits the checkpointing frequency. Since the checkpoint is incomplete before all dirty data is persisted, the next checkpoint cannot be taken. As all non-persisted updates after a checkpoint will be lost upon power failures, limited checkpoint frequency is likely to cause more data loss. This further complicates the approach to the external synchrony issue (§2.4).

2.4 External Synchrony

External synchrony [54] refers to the synchronization of state inside/outside the SLS system. In current SLSs, updates after the latest checkpoint are subject to data loss until the next checkpoint is taken. Such a data loss window may be acceptable for many applications decades ago when SLS was first proposed, such as word processing applications [42]. However, it is insufficient for the correctness of applications exposing state to external systems, which are very common in modern data centers. For example, a key-value store server needs to guarantee the persistence of a stored key-value pair once it has responded to clients. Databases are expected to guarantee the persistence of modifications once the transaction commitment is replied to users.

To handle such *external synchrony* issues and make such applications correct, prior SLSs provide additional mechanisms (usually journaling APIs) for applications to use. This, however, brings the applications back to the complexity of writing persistence code correctly, contradicting SLSs' original intention.

2.5 Rethinking SLS in the Context of NVM

The emerging non-volatile memory (NVM) technology combines DRAM-like byte-addressability and accesses performing with storage-like durability and large capacity. It is a good match for SLS as its promising features make SLS more efficient: CPUs can now easily manipulate data on non-volatile storage devices directly at byte-granularity.

However, simply equipping servers with NVM cannot guarantee the persistence of all transient state upon failures. With the availability of techniques like Intel eADR [3], data in the CPU cache can be eventually flushed to NVM and thus persisted in case of power failures. However, data in CPU registers and device registers can still be lost. Consequently, although NVM provides a single-level device for SLS, running systems on NVM does not make the whole system persistent naturally, and implementing an efficient SLS still faces several challenges.

- *Checkpointing is still required and has to be optimized for NVM to maintain consistency between runtime data and checkpoint data.* Since data in registers is ephemeral, software checkpointing is still required to guarantee the persistence of consistent whole-system state. However, unlike existing SLSs that prepare consistent state in memory and flush to storage in batch, an SLS on NVM can leverage the single-level device (i.e., NVM) as both runtime memory and storage. This reduces unnecessary data movements (especially movements for persistence) and write-amplification issues. In exchange, to ensure a consistent checkpoint is always available, the relation between runtime data and checkpoint data needs to be carefully maintained, as the same page can be used by the runtime and checkpoint simultaneously.

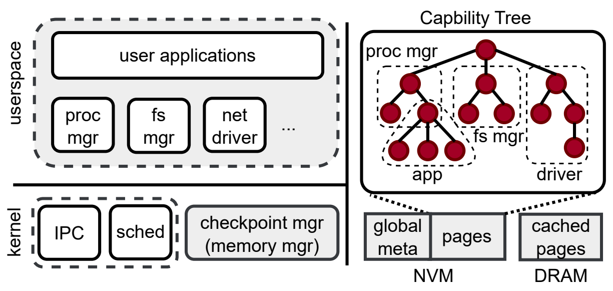


Figure 3. Architecture of TreeSLS. TreeSLS is an NVM-based microkernel SLS with hot pages cached on DRAM. TreeSLS's persistent state consists of a failure-resilience checkpoint manager managing the space, and a capability tree encapsulating whole-system state.

- *Transparent external synchrony is required for modern applications.* Modern applications heavily rely on external synchrony to provide services to others. It is cumbersome to force applications to accommodate the customized APIs for external synchrony, as writing correct persistence code with these APIs is still non-trivial and error-prone. Transparent external synchrony can add persistence transparently to existing and new applications, even if they are designed and implemented with no persistence in mind.

3 Overview of TreeSLS

TreeSLS is a single-level store system that exploits the properties of microkernel architecture and NVM to provide whole-system persistence.With its high-frequency checkpoint, immediate persistence and transparent external synchrony, TreeSLS takes a large step towards practical SLS systems.

Figure 3 shows the architecture of TreeSLS. TreeSLS adopts the microkernel architecture that minimizes kernel functionalities (e.g., IPC, scheduler, checkpoint manager) and puts most system services to the user space. Thanks to the use of NVM, TreeSLS essentially eliminates the distinction between persistent and ephemeral devices and is thus a single-tier single-level store.

For efficient whole-system persistence, TreeSLS needs to address two issues: 1) how to efficiently capture the whole-system state; 2) how to efficiently checkpoint the whole-system state. TreeSLS exploits *the capability tree* and *the checkpoint manager* to address them accordingly.

The Capability Tree. Capability-based systems [9, 32, 41, 46–48, 64, 70] usually group all capabilities into a *capability derivation tree*. System resources are represented by objects and a capability is an object reference with a set of access rights.

We observe that the capability tree essentially captures all state of the running systems (excluding the NVM space allocation state managed by the checkpoint manager itself). Hence, to achieve whole-system persistence, TreeSLS can essentially follow the capability tree to checkpoint all such state during checkpointing. Furthermore, the capability tree is a better abstraction for SLS since checkpointing a tree structure is simpler and more straightforward than building SLS on monolithic kernels, which requires special designs on complicated kernel objects or POSIX objects. For example, taking a checkpoint of file systems in a monolithic kernel requires finding FD tables, dentry-cache, and inode-cache, and preserving relations among these structures. In comparison, a microkernel usually maintains these structures in user-space file system services. The checkpoint procedures do not need to know such structures and their relations and can treat them as normal runtime data of applications.

Further, TreeSLS may also leverage the runtime state of the capability tree for efficient incremental checkpointing, i.e., by skipping state intact since the last checkpoint. Some derived state of other kernel services (*IPC* and *scheduler*) does not need to be persisted, as TreeSLS can recover such state from the capability tree, e.g., adding all threads to the scheduler's queue.

Figure 4 illustrates a graphical representation of the capability tree and Table 1 shows the detailed information of each capability-referred object. Every user-space process (application or system server) consists of a sub-tree in the capability tree (as shown in Figure 3). All allocated system resources can be reached from the *Root Cap Group*. TreeSLS thus leverages a tree-structured state checkpoint approach based on the capability tree (§4).

The Checkpoint Manager. The checkpoint manager is responsible for taking checkpoints and managing NVM space for the runtime objects and checkpoints. As an NVM allocator, the checkpoint manager uses a buddy system to manage all NVM resources in TreeSLS. Both the runtime data and checkpoints are stored in the space allocated by the checkpoint manager. Slab systems are also used to facilitate the allocation of small fixed-sized objects. Metadata of both the buddy system and slab systems are stored on NVM (the *global metadata* area in Figure 3).

To avoid bootstrapping issues (i.e., checkpointing state of the checkpoint manager) and facilitate efficient access to the capability tree, the checkpoint manager is designed as a standalone in-kernel module, whose state is not checkpointed. However, since its state is critical metadata of the whole system, the checkpoint manager needs to be failure-resilient to recover from power failures at arbitrary times. As structures of the checkpoint manager are already stored on NVM, TreeSLS only needs to prevent in-flight operations from corrupting state of the checkpoint manager in case of failures. Thus, TreeSLS leverages redo/undo journaling to maintain the crash consistency of the checkpoint manager. On reboot after a power failure, TreeSLS first applies the journal to recover the checkpoint manager to a consistent state, and then recover the whole system with the checkpoints.

An Overview of Checkpoint/Restore Procedure. Figure 5 shows the overall whole-system checkpoint/restore

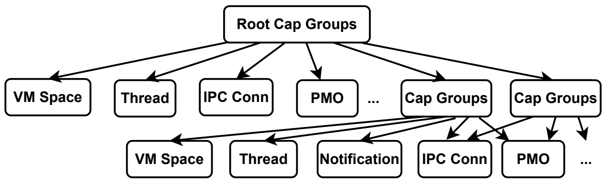


Figure 4. The capability tree of TreeSLS. All system resources (capability-referred objects) are grouped into a capability tree in TreeSLS. Every user-space process is made up of a subset of the capability tree, and checkpointing the capability tree is equal to checkpointing the whole system.

Table 1. List of capability-referred objects in TreeSLS.

Object	Description
Cap Group	a group of capabilities
Thread	thread (state & scheduling context)
VM Space	a list of virtual memory regions
PMO	a set of physical memory pages
IPC Connection	for processes communication
Notification	for synchronization (like semaphores)
IRQ Notification	a hardware signal sent to the processor

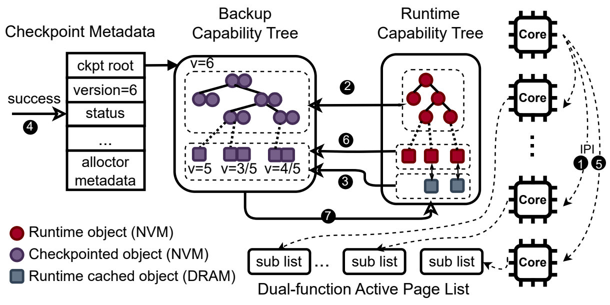


Figure 5. The checkpoint/restore procedure.

procedure. ❶ A leader CPU core sends IPI requests to all other cores to force them into a *quiescent* state. Notably, interrupts are disabled in the kernel space, so the IPI will not interrupt a core modifying object state in the kernel. ❷ After all cores respond, the leader takes a checkpoint of the current *runtime capability tree* to form a *backup capability tree*. This step does not copy user-space memory pages. Instead, all these pages are marked as read-only in the page table. ❸ In parallel to the leader core checkpointing the capability tree, other cores speculatively copy a certain set of page objects, which is further explained in §4.3. ❹ Atomically mark the checkpoint as complete and increment the global version number of checkpoints, which is a commit point making the new checkpoint in effect. This state is maintained in a global metadata area on NVM. ❺ The leader core sends IPI requests to other cores to inform them to resume execution. ❻ At runtime, page faults are triggered when processes modify their memory pages. In the handler, the memory page will be duplicated to the backup capability tree, finishing the copy-on-write procedure. ❼ The restore procedure rolls back the whole system by reviving state of the backup capability tree to restore the runtime capability tree. The only state not captured by the tree is that of the checkpoint manager. During the recovery, malloc/free operations after the last checkpoint are identified and rolled back by comparing system's state at crash with the last checkpoint's state. The latest malloc/free might be partially conducted, resulting in corruption of the buddy/slab system's state. TreeSLS uses journaling to ensure the atomicity of the in-flight allocator operations.

Correctness. The checkpoint is taken with the system in a quiescent and consistent state. All cores except the leader core are interrupted either from the user space or at the boundaries of syscalls. Thus, no core is actively modifying the kernel state, allowing TreeSLS to checkpoint consistent kernel state. All system state is captured by the capability tree, except for the state of the checkpoint manager, which is protected by journaling. The integrity of the capability tree and the consistency of checkpointed state guarantee the correctness of the checkpoint procedure.

4 Tree-structured State Checkpoint

4.1 Checkpointing the Capability Tree

To checkpoint the capability tree, TreeSLS duplicates all objects in the tree to form a backup capability tree. Since an object can be referred by multiple cap groups, TreeSLS maintains a *capability object root (ORoot)* structure for each unique object to avoid redundant checkpointing. ORoot records the addresses of the runtime object and the corresponding backup objects (if present), and each object contains the field pointing to its ORoot. With ORoot, TreeSLS can quickly find the runtime object or the backup objects of any given object.

Next, we describe how each kind of object is checkpointed to the backup capability tree. The specific checkpoint strategy for each object primarily follows several considerations. First, small-sized and frequently updated objects (e.g., Thread) are directly copied to the new checkpoint during checkpointing since the copying is quick. Second, large-sized and slowly changing objects (i.e., memory pages) are asynchronously copied during runtime. Third, objects that can be rebuilt (e.g., page tables) are not included in the checkpoint, which trades restore time for faster checkpointing.

Cap Group. A cap group is an array of capabilities; each capability consists of a pointer to the runtime object and the access rights. To checkpoint a cap group, TreeSLS first allocates space for the cap group in the backup capability tree, and copies the capabilities to the backup cap group. For convenience, the backup capability stores the pointer to the corresponding ORoot, instead of the backup object.

For each capability, TreeSLS finds the runtime object and the corresponding ORoot. If the corresponding ORoot is absent, it means that the runtime object is newly created and TreeSLS will initialize the ORoot for it. By inspecting the backup object in the ORoot, TreeSLS knows whether the runtime object has been checkpointed in this round of checkpointing. If not, TreeSLS recursively checkpoints the object according to its type.

Thread. To checkpoint a Thread object, TreeSLS allocates space and copies the thread context (e.g., registers and scheduling state) to the backup tree. As all CPU cores are trapped in the kernel when taking the checkpoint, all state of user-space threads has been consistently saved on NVM. Thus, we can safely copy them to the backup tree.

IPC Connection, Notification and IRQ Notification. These objects are used for inter-process communication and synchronization. We directly copy them to the backup capability tree.

VM Space and Page Tables. VM Space records a list of accessible virtual memory regions and a page table structure for the space. Each virtual memory region is backed by a physical memory object (PMO). To checkpoint VM Space, TreeSLS duplicates the list of virtual memory regions to the backup tree, and ignores the page table structure as the page tables can be rebuilt after recovery.[2] During a recovery, an empty page table is created for each process. Afterwards, page accesses from applications will trigger page faults and the handler will use the fault address to find the physical page from the recovered VM Space's virtual memory region and its corresponding PMO, and add the mapping to the page table. To further improve the efficiency, TreeSLS puts the page tables on DRAM as they do not need to be persisted. Also, TreeSLS will reuse the virtual memory region list for VM Space in subsequent checkpoints.

PMO and Memory Pages. PMO records a set of physical memory pages organized by a radix tree. PMO is the most special object, since it is usually large in volume and possibly only a subset of pages are modified. To checkpoint a PMO, TreeSLS duplicates the radix tree to the backup capability tree, and handles memory pages differently.

During the first checkpoint of a new process, TreeSLS marks all pages as *read-only* in the page table. A page fault will be triggered when a page is modified. In the page fault handler, TreeSLS will duplicate this page and update the pointer in the backup PMO's radix tree. In the subsequent checkpoint, pages are marked as read-only again to track subsequent modifications.

Note that the asynchronous copying of pages does not delay the persistence of checkpoint: the checkpoint of PMO is persisted once the radix tree is copied to the backup tree, since the runtime pages on NVM can be used after power failures. Similar to VM Space, TreeSLS reuses the radix tree in subsequent checkpoints to avoid constructing the tree from scratch.

[2]We use "recovery" and "restore" interchangeably in the paper.

Other State. Some state is not managed in the capability tree, but still contributes to the overall system integrity and is necessary for checkpointing. This includes components like kernel buffers and copy-on-write related bits in the page table. We identify and include them in the capability tree as special nodes.

4.2 Consistency with Versioning

To handle unexpected power failures, TreeSLS needs to guarantee that there is always a consistent checkpoint. Existing approaches use runtime pages as the volatile cache and maintain two persistent pages as the backups, which consumes unnecessary memory and incurs unnecessary memory copies.

Considering that physical memory objects (PMOs) are the objects with a large number of memory pages, TreeSLS leverages the runtime page as one backup and allocates at most one additional backup for checkpoints. For each other object, TreeSLS maintains two backups (i.e., checkpoints) besides the runtime copy. In both cases, TreeSLS attaches a version number for each backup and maintains a global version number to guarantee consistency efficiently.

As shown in Figure 6(a), we use a *checkpointed radix tree* to maintain the hierarchy of pages for each checkpointed PMO. The leaf node of the tree is represented by a structure called *checkpointed page (CP)*, which maintains the version number and the address of the *backup page*. Unlike other objects having two backups, each page in PMO has zero (address=NULL) or one backup page in the checkpointed radix tree, as NVM enables runtime pages to be used in the consistent checkpoint.

Specifically, supposing the global version number is 5, three cases can happen upon failures (Figure 6(a)): ❶ The backup version number is equal to the global version number, meaning that we saved data A to backup with version=5 in the page fault handler and the page was then modified to A'. Thus, the page should be restored with the content of the backup page (A in this example). ❷ A smaller backup version number infers that data in the runtime page (B') have not been modified since the last checkpoint; thus, we should recover using data in the runtime page (B'). ❸ An empty backup means that the page is never modified and thus not checkpointed, and we will recover with data in the runtime page (C).

In summary, TreeSLS only needs to recover pages whose backup version number is equal to the global version number, and keep other pages unchanged during the recovery.

4.3 Checkpointing Memory Pages with Hybrid Copy

Besides reducing the stop-the-world checkpointing overhead, TreeSLS needs to reduce the runtime overhead of SLS, especially in high-frequency checkpointing scenarios. As we evaluate in §7.4 and show in Figure 10, when the checkpoint is taken at the interval of 1 ms, the runtime overhead is substantial and most runtime overhead is caused by page fault handling and the page copying in the handler.

4.3.1 Methods to Copy Pages.

To reduce the overhead caused by page copying, we studied four possible methods (shown in Figure 7).

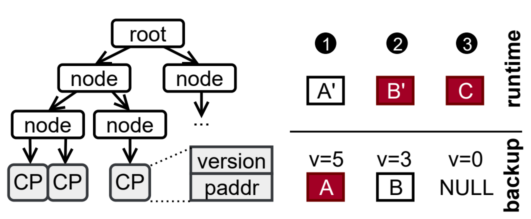
- *Stop-and-copy* copies all modified pages to the backup in the stop-the-world (STW) checkpointing. It is the simplest method, but will stop the world for a long time to copy all pages.
- *Speculative stop-and-copy* speculatively copies dirty pages before the STW checkpointing is taken, to reduce the STW time. If the speculatively copied pages are modified again before the checkpointing, these pages need to be copied again during STW checkpointing.
- *Copy-on-write*. *Stop-and-copy* and *speculative stop-and-copy* guarantee the backup is generated before the STW checkpointing completes; thus, runtime pages are free to be modified after the STW checkpointing. The *copy-on-write* method, however, marks the page as read-only and delays copying the page to the backup until the page is about to be modified. It moves the overhead from the checkpointing to the later runtime, with the cost of page fault handling and possible lock contentions among page faults. Note that for DRAM-based systems, this method will delay the availability of checkpoints since runtime data will be lost upon failures; for TreeSLS, the checkpoint is ready for failures after the STW checkpointing since runtime data are persisted on NVM.
- *Speculative copy-on-write* speculatively copies pages that are likely to be modified before page faults are triggered as in *copy-on-write*. A successful speculation will avoid page faults and move page copying overheads out of the critical path. A false speculation introduces unnecessary page copying, wasting CPU cycles and possibly contending running applications.

4.3.2 Hybrid Copy.

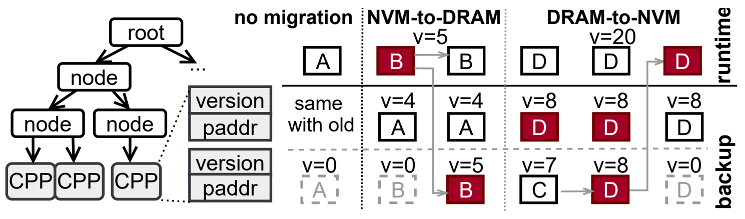
To move runtime page faults and page copying out of the critical path, we propose the *hybrid copy* method, which combines multiple methods and DRAM/NVM migration according to two observations.

First, as most applications have access locality, a set of hot pages is modified nearly in every checkpoint. We can migrate these hot pages to DRAM for faster access and *stop-and-copy* them as they are likely to be modified in the next checkpoint. Copying hot pages before they trigger copy-on-write is a form of speculative copy. Second, during the stop-the-world checkpointing, CPU cores not involved in the main checkpointing procedure can conduct *stop-and-copy* operations in parallel without introducing extra overhead.

With *hybrid copy*, TreeSLS identifies hot pages and migrates (i.e., copies) them to DRAM. TreeSLS checkpoints these hot pages on DRAM with *stop-and-copy* and the rest



(a) Checkpointed Radix Tree



(b) Extended Checkpointed Radix Tree

Figure 6. Checkpointed structures for memory pages. (a) shows the structure for original *copy-on-write*-only checkpointing method (show in §4.2). (b) shows the extended structure for *hybrid* checkpointing method (show in §4.3.3). The red color indicates this page has the correct version belonging to the current checkpoint.

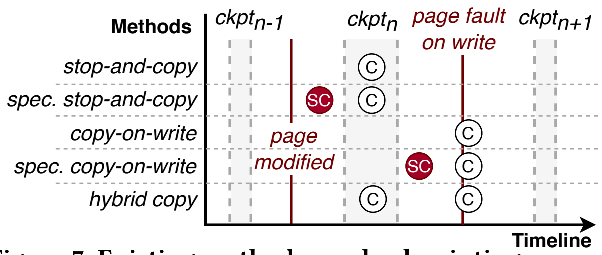


Figure 7. Existing methods on checkpointing memory pages. "C" stands for memory page copy and "SC" represents speculative page copy. The *hybrid copy* in TreeSLS leverages the idea of *speculative copy-on-write* and additionally uses *stop-and-copy* for hot pages that are migrated to DRAM.

pages on NVM with *copy-on-write*. Note that *hybrid copy* echos the high-level idea of *speculative copy-on-write*: predicting pages that are likely to be modified and copying them before the actual *copy-on-write*.

TreeSLS introduces a *dual-function active page list* to track hot pages and implement the migration. When a page fault is triggered, we increase the page's *hotness* value, and append the page to the list when its *hotness* exceeds the threshold. During checkpointing (step ❸ in Figure 5), all cores except the leading core will traverse a sub-list of the active page list. In the traverse, dirty DRAM pages will be copied to complete *stop-and-copy*. Additionally, newly appended pages since the last checkpointing are migrated to DRAM. Pages that have not been accessed for excessive times will be migrated back to NVM and removed from the list. *Hotness* values of these removed pages will be cleared.

4.3.3 Extended Versioning. To support *hybrid copy*, the structure of the checkpointed radix tree needs to be modified as DRAM-cached pages need two backups: one for in-flight checkpointing and the other for a consistent checkpoint. As shown in Figure 6(b), TreeSLS extends the original *checkpointed page (CP)* structure to *checkpointed page pairs (CPP)* structure to maintain two backups.

When no migration happens, TreeSLS uses the first pointer to point to the backup page and the second to the runtime page. Both pages are stored in NVM.

During the NVM-to-DRAM migration, TreeSLS allocates a DRAM page and copies runtime page's data to it. TreeSLS then updates the runtime page table to let the DRAM page become the runtime page. TreeSLS sets the version of the runtime page in NVM to the global version so that it becomes the latest backup page. Then, the two NVM backup pages can be used alternatively to save the DRAM runtime page's data during the checkpointing.

For the DRAM-to-NVM migration, as the migration happens only when the page has not been modified for several checkpoints, it is guaranteed that the runtime page data are identical to one of the backup page data. TreeSLS makes sure that the second backup page contains the latest data by copying from the runtime page if necessary. We set the second backup's version to zero and update the runtime page table to let the second backup page become the runtime page.

The migration is resistant to failures using the following rules in recovery: if a backup's version is equal to the global version, this backup is used for recovery; otherwise, if the second backup version is zero, the second backup is used; otherwise, the backup with a higher version is used. Such a rule is compatible with the rules before, when the runtime page is treated as the second backup with version zero.

5 Transparent External Synchrony

To support external synchrony, an SLS should make sure that the state changes caused by a request are persisted before sending responses to external systems. With high-frequency checkpointing, TreeSLS archives this by delaying external visible operations (e.g., sending network packets) until a checkpoint is taken. This can be implemented transparently to applications by allowing user-space services (e.g., network drivers) to register a *checkpoint callback*, which will be invoked at the end of each checkpointing, and a *restore callback*, which is invoked at the end of recovery. TreeSLS also provides an *eternal PMO*, which is a special kind of PMO that will not be rolled back during recovery.

Figure 8 shows how TreeSLS can support external synchrony in the network driver with modified ring buffers (e.g., queues in NVMe). Taking the send queue as an example, the ring buffer and the three pointers are stored in eternal PMOs. To send a message (*msg_2*), the message is appended to the ring buffer and the *writer* is updated (Figure 8(a)). The message is not available to be sent. When a checkpoint is taken, the checkpoint callback of the user-space network

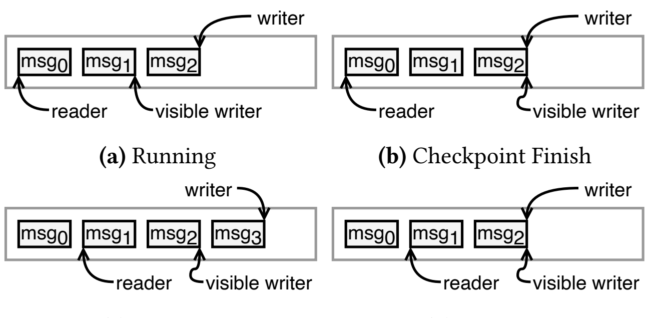


Figure 8. Ring buffer for external synchrony.

driver is invoked and will update *visible writer* (Figure 8(b)) to indicate that msg_2 can be sent since all state it depends on has been checkpointed.

In case of power failures, the restore callback is invoked and executes actions based on the state of the ring buffer at the crash (as the ring buffer is stored on an eternal PMO, its state is not rolled back). The driver can discard new messages since the last checkpoint (msg_3 in the Figure 8(c)) as the application sending the message is rolled back and will re-send the message. Note that the *reader* pointer is not rolled back since the message has already been sent to the hardware. The driver needs to check whether this message is actually sent by the hardware. The driver also needs to record hardware configurations in eternal PMOs so that it can restore the hardware state after recovery, and can resume communication with external systems.

TreeSLS's transparent external synchrony delays communications with external systems till the next checkpoint. Thus, the checkpointing frequency needs to be sufficiently high, to avoid adding too much delay to the communication. Though TreeSLS requires the drivers to be modified using TreeSLS's APIs, applications can gain the external synchrony with no modification. In current TreeSLS implementation, we implemented the external synchrony in a network server that handles communications between clients and servers on the same machine. We are working on implementing the external synchrony in real network drivers.

6 Implementation

We implemented TreeSLS on top of ChCore [10, 19], which is an educational multicore microkernel that supports POSIX APIs through musl-libc [11]. Like prior microkernels, ChCore adopts capability-based access control. ChCore contains ~21 k LOC in the kernel space, including IPC, scheduling, and memory management. The OS services in the user space include a process manager, file system servers and drivers.

TreeSLS adds ~5 k LOC in the kernel space, including ~3.3 k LOC in a module for checkpoint/restore, ~1.2 k LOC in memory allocator, and modifications to other modules (IPI, syscall, etc.). In the user space, we modify the network server to implement external synchrony (~500 LOC).

7 Evaluation

In this section, we evaluate TreeSLS with microbenchmarks and several applications to answer the following questions:

- Does TreeSLS function well in various scenarios? (§7.2)
- How much time does a checkpoint take? (§7.3)
- How do checkpoints in TreeSLS affect the performance of running applications? (§7.4)
- How do real applications perform on TreeSLS? (§7.5)

7.1 Environment Setup

We run all experiments on a machine with dual Intel® Xeon® Gold 6330 CPUs with eADR support. CPU frequency is fixed at 2.0 GHz with Hyperthreading enabled and Turbo Boost disabled. TreeSLS runs on one NUMA node, with 256 GiB DDR4 DRAM and 1 TiB Intel® Optane™ Persistent Memory.

We choose several well-known applications from various domains in the evaluation, including computing applications (Phoenix-2.0 test suite [60]), in-memory key-value stores (Redis-6.0.8 [6] and Memcached-1.6.21 [5]), persistent key-value stores (LevelDB-1.23 [4] and RocksDB-6.6 [7]), and databases (SQLite3 [8]). We compare TreeSLS with Linux Kernel 5.4 and Aurora [73].

7.2 Functional Tests

We tested self-implemented simple test programs (*hello world*, *ping-pong* and *simple key-value stores*) and the real-world applications listed in §7.1. We manually crash and reboot the system while running these programs. After reboot, these programs can continue running with expected behaviors, indicating that TreeSLS functions well.

7.3 Stop-the-world Checkpointing

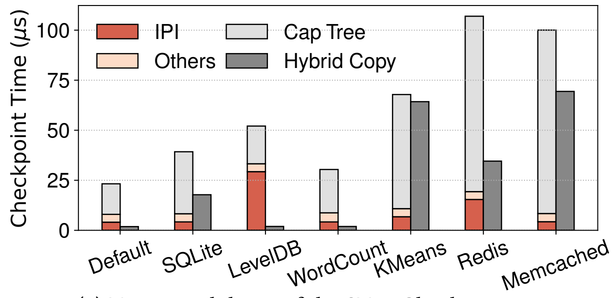
Stop-the-world (STW) checkpointing is a major source of performance overhead in TreeSLS. In this subsection, we will first demonstrate the time of taking a checkpoint on different types of objects; we will then examine the overall overhead of STW checkpointing when running a workload, and how different types of objects contribute to this overhead.

As overhead may vary with different running workloads, we tested different workloads and detailed the object count and memory usage of each workload in Table 2. The *default* workload is the system running no additional workloads, i.e., only the system services are running in the system. We show *default* for reference and object counts of all other workloads are relative to *default*. It is worth mentioning that an application's checkpoint size (*Ckpt*) is much smaller than its runtime memory consumption (*App*), since NVM's persistence allows TreeSLS to use runtime pages in the checkpoint as long as they are not changed since the checkpoint.

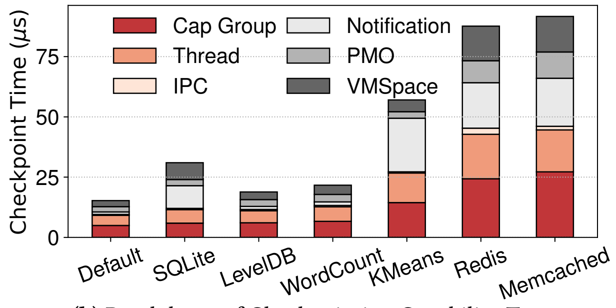
All experiments running in this subsection are configured with 1000 Hz checkpointing, i.e., taking a checkpoint per millisecond. SQLite and LevelDB are single-threaded, SQLite conducts a mixed read/insert/update/delete benchmark, while LevelDB utilizes the fillbatch workload in

Table 2. Details of different workloads. *Default* is the system running with no workloads. Object counts in other workloads are relative to *default.* Thanks to NVM's persistence, TreeSLS can include runtime pages in the checkpoint and thus an application's checkpoint size (*Ckpt*) is smaller than its runtime memory consumption (*App*). No IRQ object appears during the test.

Workload	Object Composition (Count)						Size (MiB)	
	C.G.	Thread	IPC	Noti.	PMO	VMS	App	Ckpt
A. Default	6	27	9	7	71	6	n/a	n/a
B. SQLite	+1	+4	+3	+0	+14	+1	1015	161
C. LevelDB	+1	+5	+3	+2	+18	+1	230	77
D. WordCount	+1	+12	+3	+8	+31	+1	238	160
E. KMeans	+1	+12	+3	+9	+24	+1	15	134
F. Redis	+2	+77	+60	+6	+262	+2	129	316
G. Memcached	+2	+42	+19	+17	+154	+2	303	187



(a) Time Breakdown of the STW Checkpointing



(b) Breakdown of Checkpointing Capability Tree

Figure 9. Breakdown of checkpointing.

dbbench. WordCount and KMeans are 8-threaded with 100 MiB dataset and 10 k data points for each. Redis and Memcached execute SET benchmark with 8-threaded clients (clients were also checkpointed), while Memcached itself operates with 4 threads.

Breakdown of Checkpoint. Figure 9(a) shows the time of taking an incremental STW checkpoint under various workloads and the breakdown. Two bars are given for each workload: the left bar indicates the time used by the main checkpointing procedure, including handling IPIs (*IPI*), checkpointing the capability tree (*Cap Tree*) and others (*Others*); the right bar shows the maximal time used by other cores doing the hybrid copy, which is in parallel with the main checkpointing procedure. With no workload (*Default* in the figure), the STW time is as low as ~25 μs. Two single-threaded workloads, *SQLite* and *LevelDB*, have similar low time consumption, and more complex multi-threaded applications finish checkpointing in around 100 μs.

Figure 9(b) further breaks down the time of checkpointing the capability tree according to object types. Most objects can be quickly copied during the STW checkpointing as their sizes are small. Checkpointing Cap Group and Thread is costly for workloads with a large number of objects and threads. VM Space's checkpointing also contributes to the overall time as it involves marking all newly-changed pages as read-only.

Table 3. Checkpoint/Restore time of a single object.

Time (μs)	Incr Ckpt		Full Ckpt		Restore	
	Min	Max	Min	Max	Min	Max
C.G.	0.82	3.28	2.87	17.67	5.65	20.04
Thread	0.15	0.29	0.56	1.41	1.33	1.49
IPC	0.03	0.05	0.08	0.47	0.10	0.19
Noti.	0.10	1.45	0.05	0.20	0.10	0.18
PMO	0.03	0.03	842.91	4082.75	18.95	123.67
VMS	0.41	1.68	144.28	180.50	6.47	26.57

Checkpoint/Restore of a Single Object. Table 3 further presents the time of checkpointing and restoring a single object of different types in Figure 9(b). During the first two rounds of checkpointing, a complete object snapshot is taken, which involves memory allocation and initial object structure building. Subsequent checkpoints are incremental and reuse many of the already established object structures. Thus, we show the full checkpoint time (*Full Ckpt*) and incremental checkpoint time (*Incr Ckpt*) separately in the table. For both checkpoints, we give the minimal and the maximal time we collected from all workloads.

Thanks to the microkernel minimizing in-kernel state, the longest time to incrementally checkpoint an object is 3.28 μs and most objects can be incrementally checkpointed at nanosecond-scale. The full checkpoint time is longer because it involves constructing the backup object from scratch. For example, a full checkpoint of PMO takes up to 4 ms to save the radix trees of multiple files (i.e., the 100 MiB data file used in the Phoenix benchmark.). The full time is acceptable as it only occurs at the beginning. The time of restoring is also acceptable.

The cost of checkpointing and restoring certain types of objects (Cap Group, PMO, and VM Space) is related to the object sizes. For example, because the Cap Group object maintains a table of capabilities, having a large table would increase the cost of checkpointing and restoring Cap Group. This also applies for PMOs and VM Spaces due to their radix trees and virtual memory region lists.

7.4 Runtime Overhead

Besides the overhead of creating checkpoints, TreeSLS introduces page faults and page copying during the normal

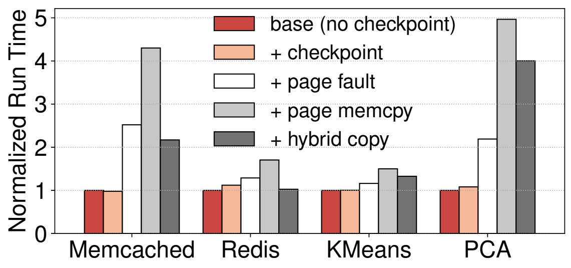


Figure 10. Breakdown of runtime overhead and effect of hybrid memory checkpoint.

Table 4. Effect of hybrid memory checkpoint.

	Memcached	Redis	Kmeans	PCA
# of runtime page faults	182	11	9	279
# of dirty cached pages	156	89	197	34
# of cached pages	395	241	431	253
Ratio of page faults eliminated	46%	89%	95%	11%
Dirty rate in cached pages	40%	33%	37%	13%

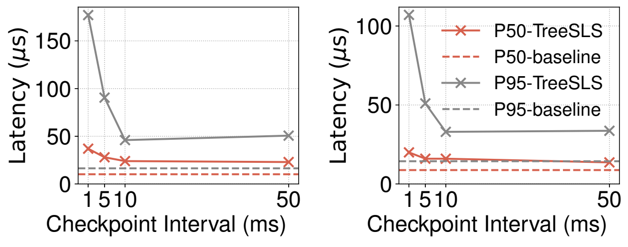
execution of applications. In this section, we demonstrate the overall performance overhead to applications.

Hybrid Memory Checkpoint. Figure 10 breaks down the runtime overhead. We choose several memory-intensive workloads. The configuration of each workload is the same with §7.3. PCA is also 8-threaded and performs on a matrix with 1 k rows and 1 k columns. The sophisticated STW checkpointing (*+checkpoint*) brings marginal overhead. Most overhead comes from the page fault handling (*+page fault*) and page copy (*+page memcpy*). The hybrid memory checkpoint (*+hybrid copy*) does reduce the overhead by up to 49%.

The recall and precision of the hybrid copy are further detailed in Table 4. Taking Memcached as an example, 395 pages are cached in DRAM as hot pages; 40% (156 pages) of these pages are actually modified between two checkpoints. On the other hand, 46% (156 pages) of all 338 modified pages are marked as hot pages and speculatively copied during the STW checkpointing; the rest 182 pages cause page faults. The result shows that hybrid copy effectively caches hot pages and reduces page faults in runtime.

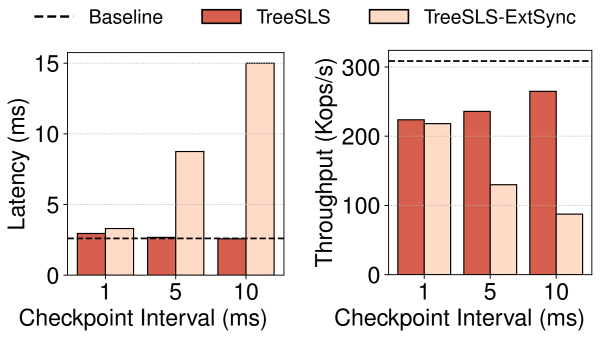
Checkpoint Frequency. Figure 11 presents the P50 and P95 latency of SET/GET operations on 10 million keys sent from an 8-threaded client to an 8-threaded Memcached server. Taking checkpoints increases the operation latency. When the checkpoint interval is less than 10 ms, the latency increases as the checkpoint interval decreases. The client and server employ a machine-local, UDP-like communication, leading to μs-scale latencies, and TreeSLS with a 1 ms checkpoint interval introduces an extra latency of 11–160 μs.

External Synchrony. Figure 12 gives the runtime overhead caused by external synchrony. The test involves 50 clients concurrently setting 10 million 1024-byte keys to a Redis server. To mitigate the performance impact caused by clients blocking and waiting for a reply, each client sends



(a) SET **(b)** GET

Figure 11. Latency overhead of Memcached SET and GET with TreeSLS's different checkpoint frequencies.



(a) P50 Latency **(b)** Throughput

Figure 12. Runtime overhead of Redis SET benchmark with or without external synchrony support.

32 requests in a batch at a time. The results reveal that delaying sending responses to clients increases latency by approximately one checkpoint interval. Despite increasing the concurrency, the blocking of clients on the critical path still negatively impacts the overall throughput.

7.5 Real-world Applications

TreeSLS provides an effective alternative for servers lacking inherent persistence guarantees (e.g. Memcached) to easily achieve strong persistence guarantees without self-implementing mechanisms. To show TreeSLS's effectiveness, we compare checkpoint overhead in TreeSLS with customized persistence mechanisms like the write-ahead logging (WAL) for in-memory key-value stores and on-disk log-structured merge (LSM) tree structures.

By taking checkpoints every 1 ms, TreeSLS can seamlessly and transparently persist applications with only a 1 ms delay in latency (to support external synchrony). Since the client's concurrency becomes a bottleneck when enabling external synchrony, i.e., clients in benchmarks block and wait for replies, the following tests were conducted with external synchrony disabled.

7.5.1 In-memory Key-Value Stores. We use the YCSB [22] benchmark to evaluate the performance of using TreeSLS to persist Redis transparently. Four configurations are used in the evaluation: Redis with no persistence guarantee on TreeSLS (*TreeSLS-base*) and Linux (*Linux-base*); Redis transparently persisted by TreeSLS with 1 ms checkpointing

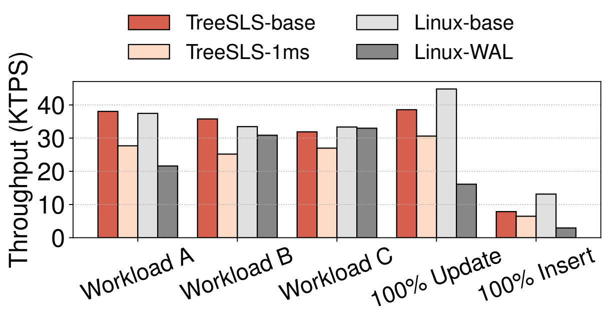


Figure 13. Runtime overhead of YCSB on Redis.

(*TreeSLS-1ms*), and Redis persisted by its original WAL on Linux (*Linux-WAL*). For fairness, Redis on Linux is built with musl-libc, and Ext4-DAX [2] is used to store the WAL on Intel® Optane™ PM for *Linux-WAL*.

Figure 13 shows the throughput of the four configurations on different workloads. For the write-intensive workload (100% Update and 100% Insert), *TreeSLS-1ms* causes 18% to 21% degradation of throughput, while *Linux-WAL* reduces the throughput by 64% to 78% compared with *Linux-base*. *TreeSLS-1ms*'s absolute throughput is 1.9× to 2.2× of *Linux-WAL*. This is because *Linux-WAL* writes an operation log in the WAL for each write operation, and such an extra write on the critical path slows down the performance. In comparison, in the checkpointing of TreeSLS, newly created pages are checkpointed by being marked as copy-on-write without actual page copying, and repeatedly changed pages between checkpoints are copied only once.

For workload with 50% read and 50% write (*Workload A*), *TreeSLS-1ms* still performs better than *Linux-WAL* by 28%. Compared with the corresponding baselines, *TreeSLS-1ms* causes 27% overhead, and *Linux-WAL* performs worse by 42%. For read-intensive workloads (95% read in *Workload B* and 100% read in *Workload C*), *TreeSLS-1ms* performs worse than *Linux-WAL*, since *Linux-WAL* records nothing for read operations, while TreeSLS still needs to make system-level checkpoints, causing performance degradation of 30% (*Workload B*) and 15% (*Workload C*), respectively.

7.5.2 Persistent Key-Value Stores.

RocksDB [7] uses a combination of in-memory Memtables and on-disk log-structured merge (LSM) trees to store data, and relies on a WAL for crash consistency. Both the on-disk LSM trees and WAL can be replaced by TreeSLS. NVM's large capacity makes it possible to hold a large Memtable in memory and use high-frequency checkpointing for persistence.

Several configurations are used: RocksDB with no persistence guarantee (*Aurora-base* and *TreeSLS-base*), RocksDB with WAL on DRAM (*Aurora-base-WAL*), RocksDB persisted by SLS systems with transparent checkpoint mechanism (*Aurora-5ms*, *TreeSLS-5ms* and *TreeSLS-1ms*) or custom APIs (*Aurora-API*). As TreeSLS's base microkernel and FreeBSD perform differently, both baselines are tested. We setup Aurora to use DRAM as storage, and set Aurora's checkpoint interval to 5 ms. A smaller checkpoint interval cannot reduce the actual time between checkpoints since it takes 5–7 ms to persist the checkpoint. The actual time between checkpoints will be as long as 100 ms if SSD is used as storage.

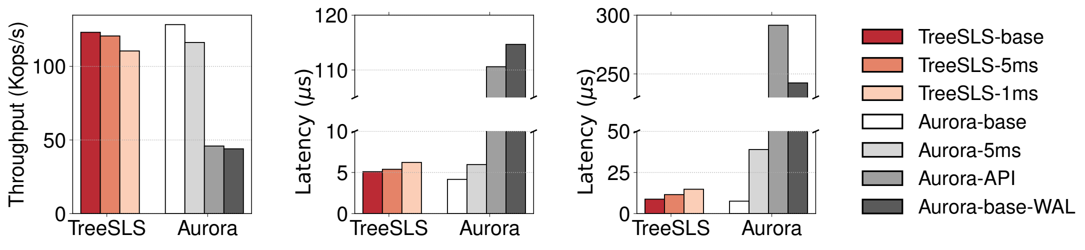
Figure 14 gives the throughput and the write latency of running different configurations with the Facebook Prefix_dist [16] workload. With TreeSLS taking checkpoints every 1 ms (*TreeSLS-1ms*), the throughput drops by 10% compared to the baseline (*TreeSLS-base*); the latency increases by 22% (P50) and 69% (P99), respectively. For TreeSLS taking checkpoints every 5 ms (*TreeSLS-5ms*), the throughput decreases by 2% and the latency rises by 6% (P50) and 32% (P99), respectively. Aurora, with 5 ms checkpoint intervals, presents a 9% throughput overhead and latency increases of 43% (P50) and 4.2× (P99), respectively, compared to *Aurora-base*. The absolute performance of *TreeSLS-1ms* is worse than *Aurora-5ms*, because the baseline of TreeSLS is slower than Aurora's FreeBSD due to the usage of different libc. Besides, to ensure external synchrony, a 1 ms latency needs to be added to each operation in *TreeSLS-1ms*. In contrast, 5–10 ms latency needs to be added for *Aurora-5ms*, including up to 5 ms waiting for the next checkpoint to be taken and 5 ms for the checkpoint to be flushed to storage.

Our transparent checkpoint also achieves 2.4× and 2.5× throughout of Aurora's journaling API (*Aurora-API*) and RocksDB's WAL (*Aurora-base-WAL*), respectively. The reason is that RocksDB is write-intensive and many writes are directly applied to the in-memory structure with persistence guaranteed by TreeSLS. As we have demonstrated, the checkpoint size is smaller than many applications' runtime consumption, and as a result, we eliminate the need for double-write (i.e., writes to the application data and WAL) for these pages, which is inevitable when using logging mechanisms.

8 Discussion

Limitations. While TreeSLS provides low-overhead and externally-synchronized whole-system persistence, different applications in the system and different data within an application may have varying persistence requirements, TreeSLS's uniformed persistence may result in the persistence of unnecessary data.

Data Reliability. Data stored in NVM are critical for TreeSLS to work correctly. Data corruption or hardware failures in NVM can break the integrity of TreeSLS's data and prevent TreeSLS's execution. To enhance the data reliability, TreeSLS can maintain multiple versions of checkpoints and recover to earlier checkpoints if the latest checkpoint is corrupted. TreeSLS can also maintain replications for each objects in TreeSLS during checkpointing. This requires storing multiple copies of the same checkpoint and consumes more space. In alternative to replications, TreeSLS can further adopt erasure coding to reduce space consumption, at the cost of using additional computing resources.



(a) Throughput **(b)** P50 Write Latency **(c)** P99 Write Latency **(d)** Legend

Figure 14. Runtime overhead of RocksDB with Facebook's Prefix_dist workload.

Memory Over-commitment. As NVM devices have a high capacity, we didn't consider out-of-memory issues in the current TreeSLS design. To support memory over-commitment, we can add a cold page list to track cold pages and evict them to secondary storage, such as SSDs and disks, when the system is under memory pressure.

Extending to Eidetic System. Eidetic system refers to a system with the ability to recover to any past state [27]. TreeSLS can be extended to maintain multiple versions of the system's lifetime, as we have already enabled version maintenance through the ORoot interface. With this, TreeSLS can provide interfaces for listing all versions and allow users to quickly navigate through arbitrary versions in the execution history, which offers numerous advantages, particularly in the context of debugging. Maintaining multiple backups will not include additional work on the critical path, but requires more space. The ORoot interface can also allow multiple checkpoints to share the same object to save space.

9 Related Work

Single-Level Store. The design of TreeSLS draws inspiration from early single-level store (SLS) systems. The idea of putting all information on a single-level layer was proposed by Atlas [40], Multics [12], IBM System/38 [71], as well as some single-address space operating systems [18, 34, 45, 69] decades ago. Our high-level idea is to achieve system-wide checkpoints by leveraging techniques similar to KeyKOS/EROS [33, 66], Aurora [73] as well as many other SLSs [48, 68, 74]: freezing the execution and take a checkpoint of all system state except for user memory pages, while employing copy-on-write (CoW) during runtime to persist user memory pages. The integrity of our persisted system state is ensured through the capability tree, which has been proposed by earlier capability-based microkernels [33, 66] and Barrelfish/DC [80]. However, TreeSLS exploits NVM by eliminating the distinction between persistent and ephemeral devices and further leverages the capability tree for efficient, incremental state checkpointing, and thus is much more efficient than prior systems.

Application Persistence on NVM. Several prior studies have used NVM to simplify application persistence. For example, NVM libraries are proposed to enable applications to either rewrite themselves to use the provided interfaces [21, 77] or rely on the compiler and runtime to automatically add durability semantics [17, 37, 78].

The Machine [39] and Twizzler [13] provide alternative data-centric OS abstractions to enable uniform access on hybrid DRAM-NVM systems, which, however, are not transparent to applications. WSP [52] and Zhuque [36] provide fully transparent system-level or application-level persistence by putting everything on NVM and flushing transient state (processor registers and caches) to NVM only on failures. They, however, mandate sophisticated hardware customization.

State Checkpointing. State checkpointing has been widely used in various domains, such as deterministic record/replay systems [28, 43, 44, 53, 55, 57], VM migration [20, 30, 35, 75], and transparent process migration [1, 14, 29, 51, 56, 59, 61]. To checkpoint memory pages, these systems utilize different approaches (e.g., stop-and-copy [29, 56], incremental copy-on-write[1, 14, 59] and speculative copy [76]), as discussed in §4.3.1. Inspired by them, TreeSLS incorporates a hybrid page checkpointing approach that exploits the characteristics of our hybrid DRAM/NVM system and leverages idle cores to perform tasks like speculative copying during the stop-the-world checkpointing.

10 Conclusion

This paper proposed TreeSLS, a persistent microkernel on NVM that simplifies the whole-system state maintenance to a capability tree and a failure-resilience checkpoint manager. Evaluation shows that TreeSLS can complete a whole-system persistence in around 100 μs and take a checkpoint every 1 ms with acceptable performance overhead.

Acknowledgments

We are grateful to our shepherd Gernot Heiser for the detailed and insightful suggestions that significantly improved the paper. We thank the anonymous SOSP reviewers for their constructive suggestions, and Shiwei Tan for contributing to an early prototype of TreeSLS. This work was supported in part by the National Natural Science Foundation of China (No. 61925206, 62141219 and 62132014). Corresponding authors: Mingkai Dong (mingkaidong@sjtu.edu.cn) and Haibo Chen (haibochen@sjtu.edu.cn).

References

[1] 2021. *CRIU*. https://www.criu.org/Main_Page

[2] 2021. *Direct Access for files*. https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[3] 2021. *eADR: New Opportunities for Persistent Memory Applications*. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

[4] 2021. *LevelDB*. https://dbdb.io/db/leveldb

[5] 2021. *Memcached - a distributed memory object caching system*. https://memcached.org/

[6] 2021. *Redis*. https://redis.io/

[7] 2021. *RocksDB Home Page*. https://www.rocksdb.org

[8] 2021. *SQLite*. https://sqlite.org/

[9] 2021. *Zircon*. https://fuchsia.dev/

[10] 2023. ChCore Lab v2. https://gitee.com/ipads-lab/chcore-lab-v2.

[11] 2023. *Musl Libc*. https://musl.libc.org/

[12] A. Bensoussan, C. T. Clingen, and R. C. Daley. 1972. The Multics Virtual Memory: Concepts and Design. *Commun. ACM* 15, 5 (May 1972), 308–318.

[13] Daniel Bittman, Peter Alvaro, Pankaj Mehra, Darrell D. E. Long, and Ethan L. Miller. 2020. Twizzler: a Data-Centric OS for Non-Volatile Memory. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 65–80.

[14] Edouard Bugnion, Vitaly Chipounov, and George Candea. 2013. Lightweight Snapshots and System-Level Backtracking. In *Proceedings of the 14th USENIX Conference on Hot Topics in Operating Systems* (Santa Ana Pueblo, New Mexcio) *(HotOS '13)*. USENIX Association, 23.

[15] Roy Campbell, Garry Johnston, and Vincent Russo. 1987. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *SIGOPS Oper. Syst. Rev.* 21, 3 (July 1987), 9–17.

[16] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*. USENIX Association, 209–223.

[17] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. 2014. Atlas: Leveraging Locks for Non-Volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (Portland, Oregon, USA) *(OOPSLA '14)*. Association for Computing Machinery, New York, NY, USA, 433–452.

[18] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. 1994. Sharing and Protection in a Single-Address-Space Operating System. *ACM Trans. Comput. Syst.* 12, 4 (November 1994), 271–307.

[19] Haibo Chen and Yubin Xia. 2023. *Operating System: Principles and Implementation* (1 ed.). China Machine Press.

[20] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. 2005. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (NSDI '05)*. USENIX Association, 273–286.

[21] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. NV-Heaps: Making Persistent Objects Fast and Safe with next-Generation, Non-Volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. Association for Computing Machinery, 105–118.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, 143–154.

[23] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabeu-Auban, P. W. Hutto, M. Y.A. Khalidi, and C. J. Wilkenloh. 1990. Design and implementation of the clouds distributed operating system. *Computing systems* 3, 1 (December 1990), 11–46.

[24] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. 1994. Grasshopper: An Orthogonally Persistent Operating System. *Comput. Syst.* 7, 3 (June 1994), 289–312.

[25] Alan Dearle and David Hulse. 2000. Operating System Support for Persistent Systems: Past, Present and Future. *Softw. Pract. Exper.* 30, 4 (April 2000), 295–324.

[26] Jack B. Dennis and Earl C. Van Horn. 1966. Programming Semantics for Multiprogrammed Computations. *Commun. ACM* 9, 3 (March 1966), 143–155.

[27] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI '14)*. USENIX Association, USA, 525–540.

[28] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. 2009. DMP: Deterministic Shared Memory Multiprocessing. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, 85–96.

[29] William R. Dieter and Jr. James E. Lumpp. 2001. User-Level Checkpointing for LinuxThreads Programs. In *2001 USENIX Annual Technical Conference (USENIX ATC 01)*. USENIX Association.

[30] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. 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* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, 467–481.

[31] Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhruv Mátáni, Josh Metzler, Fahim Ul Haq, and Janet Wiener. 2014. Fast Database Restarts at Facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, 541–549.

[32] David B. Golub, Randall W. Dean, Alessandro Forin, and Richard F. Rashid. 1990. UNIX as an Application Program. In *USENIX Summer*.

[33] Norman Hardy. 1985. KeyKOS Architecture. *SIGOPS Oper. Syst. Rev.* 19, 4 (October 1985), 8–25.

[34] Gernot Heiser, Kevin Elphinstone, Stephen Russell, and Jerry Vochteloo. 1994. Mungi: A Distributed Single-Address-Space Operating System. In *Proceedings of the 17th Australasian Computer Science Conference (ACSC)*. Christchurch, New Zealand, 271–80.

[35] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. 2009. Post-Copy Live Migration of Virtual Machines. *SIGOPS Oper. Syst. Rev.* 43, 3 (July 2009), 14–26.

[36] George Hodgkins, Yi Xu, Steven Swanson, and Joseph Izraelevitz. 2023. Zhuque: Failure is Not an Option, it's an Exception. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA). USENIX Association, 833–849.

[37] Terry Ching-Hsiang Hsu, Helge Brügner, Indrajit Roy, Kimberly Keeton, and Patrick Eugster. 2017. NVthreads: Practical Persistence for Multi-threaded Applications. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 468–482.

[38] D. Hulse. 1995. On Page-Based Optimistic Process Checkpointing. In *Proceedings of the 4th International Workshop on Object-Orientation in Operating Systems (IWOOOS '95)*. IEEE Computer Society, USA, 24.

[39] Kimberly Keeton. 2015. The Machine: An Architecture for Memory-Centric Computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers* (Portland, OR, USA) *(ROSS '15)*. Association for Computing Machinery, Article 1, 1 pages.

[40] T.D. Kilburn, B.G. Edwards, M.J. Lanigan, and F.H. Summer. 1962. One-Level Storage System. *IRE Transactions on Electronic Computers* 11 (April 1962), 223–235.

[41] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220.

[42] C.R. Landau. 1992. The checkpoint mechanism in KeyKOS. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems (IWOOOS '92)*. 86–91.

[43] Leblanc and Mellor-Crummey. 1987. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.* C-36, 4 (1987), 471–482.

[44] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. 2009. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, New York, NY, USA, 49–60.

[45] Ian Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. 1996. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas in Communications* 14 (1996), 1280–1297.

[46] Henry M. Levy. 1984. *Capability-Based Computer Systems*. Butterworth-Heinemann, USA.

[47] Jochen Liedtke. 1993. Improving IPC by Kernel Design. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 175–188.

[48] Jochen Liedtke. 1993. A persistent system in real use-experiences of the first 13 years. *Proceedings Third International Workshop on Object Orientation in Operating Systems* (1993), 2–11.

[49] Virendra J. Marathe, Margo Seltzer, Steve Byan, and Tim Harris. 2017. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *Proceedings of the 9th USENIX Conference on Hot Topics in Storage and File Systems* (Santa Clara, CA) *(HotStorage '17)*. USENIX Association, 4.

[50] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI '18)*. USENIX Association, 33–50.

[51] S.J. Mullender, G. van Rossum, A.S. Tananbaum, R. van Renesse, and H. van Staveren. 1990. Amoeba: a distributed operating system for the 1990s. *Computer* 23, 5 (1990), 44–53.

[52] Dushyanth Narayanan and Orion Hodson. 2012. Whole-System Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (London, England, UK) *(ASPLOS XVII)*. Association for Computing Machinery, New York, NY, USA, 401–410.

[53] Satish Narayanasamy, Cristiano Pereira, and Brad Calder. 2006. Recording Shared Memory Dependencies Using Strata. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, 229–240.

[54] Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn. 2006. Rethink the Sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, USA, 1–14.

[55] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. 2009. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (Washington, DC, USA) *(ASPLOS XIV)*. Association for Computing Machinery, 97–108.

[56] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. 2002. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *5th Symposium on Operating Systems Design and Implementation (OSDI '02)*. USENIX Association.

[57] Soyeon Park, Yuanyuan Zhou, Weiwei Xiong, Zuoning Yin, Rini Kaushik, Kyu H. Lee, and Shan Lu. 2009. PRES: Probabilistic Replay with Execution Sketching on Multiprocessors. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, 177–192.

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

[59] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li. 1995. Libckpt: Transparent Checkpointing under UNIX. In *USENIX 1995 Technical Conference (USENIX 1995 Technical Conference)*. USENIX Association, New Orleans, LA.

[60] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating MapReduce for Multi-core and Multiprocessor Systems. 2007 IEEE 13th International Symposium on High Performance Computer Architecture, 13–24.

[61] Richard F. Rashid and George G. Robertson. 1981. Accent: A Communication Oriented Network Operating System Kernel. *SIGOPS Oper. Syst. Rev.* 15, 5 (December 1981), 64–75.

[62] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. 1980. Pilot: An Operating System for a Personal Computer. *Commun. ACM* 23, 2 (February 1980), 81–92.

[63] John Rosenberg and David Abramson. 1985. MONADS-PC - a capability-based workstation to support software engineering.

[64] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michel Gien, Marc Guillemont, Frédéric Herrmann, Claude Kaiser, Sylvain Langlois, and Will Neuhauser. 1991. Overview of the CHORUS ® Distributed Operating Systems.

[65] Jonathan S. Shapiro and Jonathan Adams. 2002. Design Evolution of the EROS Single-Level Store. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATC '02)*. USENIX Association, 59–72.

[66] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. 1999. EROS: A Fast Capability System. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles* (Charleston, South Carolina, USA) *(SOSP '99)*. Association for Computing Machinery, 170–185.

[67] Helgi Sigurbjarnarson, James Bornholt, Emina Torlak, and Xi Wang. 2016. Push-Button Verification of File Systems via Crash Refinement. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI '16)*. USENIX Association, 1–16.

[68] Espen Skoglund, Christian Ceelen, and Jochen Liedtke. 2001. Transparent Orthogonal Checkpointing Through User-Level Pagers. In *Proceedings of the 9th International Workshop on Persistent Object Systems.*

Lillehammer, Norway, 201–215.

[69] Alan Skousen and Donald Miller. 1999. Using a single address space operating system for distributed computing and high performance. In *the 18th IEEE International Performance, Computing and Communications Conference (IPCCC '99)*. 8–14.

[70] Till Smejkal, Adam Lackorzynski, Benjamin Engel, and Marcus Völp. 2015. Transactional IPC in Fiasco.OC. *OSPERT 2015* (2015), 19.

[71] Frank G. Soltis. 1996. *Inside the AS/400.* Twenty Ninth Street Press.

[72] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. 2021. The Aurora operating system: revisiting the single level store. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. ACM, 136–143.

[73] Emil Tsalapatis, Ryan Hancock, Tavian Barnes, and Ali José Mashtizadeh. 2021. The Aurora Single Level Store Operating System. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, 788–803.

[74] P. Tullmann, J. Lepreau, B. Ford, and M. Hibler. 1996. User-Level Checkpointing through Exportable Kernel State. In *Proceedings of the 5th International Workshop on Object Orientation in Operating Systems (IWOOOS '96)*. IEEE Computer Society, USA, 85.

[75] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, 559–572.

[76] Dirk Vogt, Armando Miraglia, Georgios Portokalidis, Herbert Bos, Andy Tanenbaum, and Cristiano Giuffrida. 2015. Speculative Memory Checkpointing. In *Proceedings of the 16th Annual Middleware Conference (Middleware '15)*. Association for Computing Machinery, New York, NY, USA, 197–209.

[77] Haris Volos, Andres Jaan Tack, and Michael M. Swift. 2011. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. Association for Computing Machinery, 91–104.

[78] Zhenwei Wu, Kai Lu, Andrew Nisbet, Wenzhe Zhang, and Mikel Luján. 2020. PMThreads: persistent memory threads harnessing versioned shadow copies. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 623–637.

[79] Junfeng Yang, Can Sar, and Dawson Engler. 2006. EXPLODE: A Lightweight, General System for Finding Serious Storage System Errors. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Seattle, Washington) *(OSDI '06)*. USENIX Association, 131–146.

[80] Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. 2014. Decoupling Cores, Kernels, and Operating Systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI '14)*. USENIX Association, USA, 17–31.