NrOS: Effective Replication and Sharing in an Operating System

Ankit Bhardwaj and Chinmay Kulkarni, *University of Utah;* Reto Achermann, *University of British Columbia;* Irina Calciu, *VMware Research;* Sanidhya Kashyap, *EPFL;* Ryan Stutsman, *University of Utah;* Amy Tai and Gerd Zellweger, *VMware Research*

https://www.usenix.org/conference/osdi21/presentation/bhardwaj

This paper is included in the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation.

July 14–16, 2021

978-1-939133-22-9

Open access to the Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

NrOS: Effective Replication and Sharing in an Operating System

Ankit Bhardwaj[1], Chinmay Kulkarni[1], Reto Achermann[2], Irina Calciu[3], Sanidhya Kashyap[4], Ryan Stutsman[1], Amy Tai[3], and Gerd Zellweger[3]

[1]University of Utah, [2]University of British Columbia, [3]VMware Research, [4]EPFL

Abstract

Writing a correct operating system kernel is notoriously hard. Kernel code requires manual memory management and type-unsafe code and must efficiently handle complex, asynchronous events. In addition, increasing CPU core counts further complicate kernel development. Typically, monolithic kernels share state across cores and rely on one-off synchronization patterns that are specialized for each kernel structure or subsystem. Hence, kernel developers are constantly refining synchronization within OS kernels to improve scalability at the risk of introducing subtle bugs.

We present NrOS, a new OS kernel with a safer approach to synchronization that runs many POSIX programs. NrOS is primarily constructed as a simple, sequential kernel with no concurrency, making it easier to develop and reason about its correctness. This kernel is scaled across NUMA nodes using node replication, a scheme inspired by state machine replication in distributed systems. NrOS replicates kernel state on each NUMA node and uses operation logs to maintain strong consistency between replicas. Cores can safely and concurrently read from their local kernel replica, eliminating remote NUMA accesses.

Our evaluation shows that NrOS scales to 96 cores with performance that nearly always dominates Linux at scale, in some cases by orders of magnitude, while retaining much of the simplicity of a sequential kernel.

1 Introduction

Operating system kernels are notoriously hard to build. Manual memory management, complex synchronization patterns [36], and asynchronous events lead to subtle bugs [2–4], even when code is written by experts. Increasing CPU core counts and non-uniform memory access (NUMA) have only made it harder. Beyond correctness bugs, kernel developers must continuously chase down performance regressions that only appear under specific workloads or as core counts scale. Even so, prevailing wisdom dictates that kernels should use custom-tailored concurrent data structures with fine-grained locking or techniques like read-copy-update (RCU) to achieve good performance. For monolithic kernels, this slows development to the extent that even large companies like Google resort to externalizing new subsystems to userspace [57] where they can contain bugs and draw on a larger pool of developers.

Some have recognized that this complexity isn't always warranted. For example, wrapping a single-threaded, sequential microkernel in a single coarse lock is safe and can provide good performance when cores share a cache [67]. This approach does not target NUMA systems, which have many cores and do not all share a cache. Increased cross-NUMA-node memory latency slows access to structures in shared memory including the lock, causing collapse.

Multikernels like Barrelfish [17] take a different approach; they scale by forgoing shared memory and divide resources among per-core kernels that communicate via message passing. This scales well, but explicit message passing adds too much complexity and overhead for hosts with shared memory. Within a NUMA node, hardware cache coherence makes shared memory more efficient than message passing under low contention.

We overcome this trade-off between scalability and simplicity in NrOS, a new OS that relies primarily on *single-threaded*, sequential implementations of its core data structures. NrOS scales using node replication [28], an approach inspired by state machine replication in distributed systems, which transforms these structures into linearizable concurrent structures. Node replication keeps a separate replica of the kernel structures per NUMA node, so operations that read kernel state can concurrently access their local replica, avoiding cross-NUMA memory accesses. When operations mutate kernel state, node replication collects and batches them from cores within a NUMA node using *flat combining* [44], and it appends them to a shared log; each replica applies the operations serially from the log to synchronize its state.

The NrOS approach to synchronization simplifies reasoning about its correctness, even while scaling to hundreds of cores and reducing contention in several OS subsystems (§4.2,

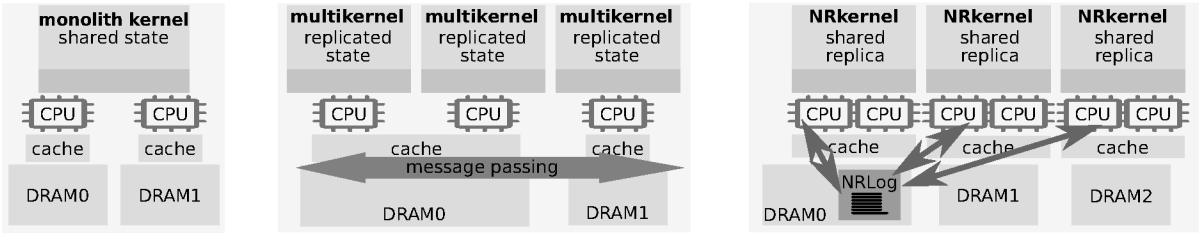


Figure 1: Architectural overview of NRkernel vs. multikernel and monoliths.

§4.4). However, node replication is not a panacea: while implementing an in-memory file system, we have encountered scenarios where frequent state mutations hinder performance. To address this challenge, we propose *concurrent node replication* (§3), which exploits operation commutativity with multiple logs and concurrent replicas to improve the performance and scalability of the file system (§4.3).

NrOS is implemented in Rust, which interplays with node replication. The Rust type system makes mutability explicit, enabling our node replication library to distinguish effortlessly between update operations that must use the log and read-only operations that can proceed concurrently at the local replica. However, we still had to solve several practical challenges, such as safely dealing with out-of-band reads by hardware, efficiently allocating buffers, and garbage collecting the log. To summarize, we make the following contributions.

1. We designed and implemented NrOS, a kernel that simplifies synchronization via node replication and runs many POSIX programs; we describe its subsystems, including processes, scheduling, address spaces, and an in-memory file system.
2. We implemented node replication in Rust, leveraging the Rust type system to distinguish mutable and read-only operations. In addition, we extended the node replication approach to exploit operation commutativity using multiple logs and concurrent replicas.
3. We evaluated NrOS on hosts with up to 4 NUMA nodes running bare metal and in virtual machines, and we compared its performance to that of Linux, sv6, and Barrelfish on file system, address space, and application-level benchmarks (LevelDB, memcached). NrOS largely outperforms conventional OSes on read-heavy workloads and on contending workloads thanks to its use of node replication.

NrOS[1] and its node replication[2] library are open source.

2 Background and Related Work

OS & Software Trends. Linux continues to be the prevalent data center OS; it uses a *monolithic kernel* (Figure 1), which shares all OS state and protects access using locks and other synchronization primitives. Despite being widely used, this model has multiple limitations. As the numbers of cores per server keeps increasing, the performance and scalability of the kernel are impacted. Its synchronization primitives, in particular, do not scale well to large numbers of cores. Moreover, architectures such as non-uniform memory access (NUMA) exacerbate scalability problems. With NUMA, each processor has lower latency and higher bandwidth to its own local memory. This causes significant performance degradation when state is shared across all processors [23].

The Linux community has reacted with point solutions to these problems, optimizing performance through fine-grained synchronization [31, 32, 36] or better locks/wait-free data structures [51, 61, 73]. However, these solutions have amplified the looming problem of complexity with the monolithic design. Correct concurrency in the kernel in the presence of fine-grained locking and lock-free data structures is hard, and it is the source of many hard-to-find bugs [26, 34, 37–39, 41, 47, 55, 79, 80].

For example, Linux has an entire class of bugs due to lockless access to lock-protected shared variables [2–4]. Use-after-free bugs are also common in Linux due to improper locking [5, 6]. These are some of the many bugs that still exist because of the heavy use of lockless or improper accesses in the Linux kernel. Such bugs would be trivially avoided in NrOS, which absolves the developer from reasoning about fine-grained locking for concurrent data structures.

The OS research community has proposed new kernel models to address these limitations [22, 30, 70, 78]. In particular, the *multikernel* model replicates kernel state and uses message passing to communicate between replicas [17], but, a decade after its introduction, multikernels have not seen widespread adoption. In part, this is because cache coherence and shared memory have not gone away (and probably will not in the foreseeable future). For example, the Barrelfish multikernel mainly uses point-to-point message passing channels between the OS nodes, avoiding use of shared memory between cores within the kernel altogether. Ultimately, this means the system needs to use n^2 messages to communicate between cores, and each core must monitor n queues; this has little benefit to offset its cost since many large groups of cores can already communicate more efficiently via shared memory access to a shared last-level cache. This also increases the number of operations that must be coordinated across cores, and some

[1] https://github.com/vmware-labs/node-replicated-kernel
[2] https://github.com/vmware/node-replication

operations require that all kernel nodes reach agreement. For example, capability operations in Barrelfish require a blocking two-phase protocol so complex that it is explicitly encoded as a large state machine, and the full Barrelfish capability system is about 8,000 lines of code not including the two domain-specific languages used for RPCs and capability operations. So, despite their scaling benefits, multikernels fail to exploit the simplicity and performance of shared memory even when it is efficient to do so.

Prior work has investigated approaches to combine the monolithic and the multikernel models [16, 74] or to apply replication for parts of the kernel [20, 25, 33, 72]. Tornado [42] and K42 [54] use clustered objects, which optimize shared state through the use of partitioning and replication. More recently, Mitosis [10] retrofitted the replication idea to page table management in the Linux kernel, and showed benefits for a wide variety of workloads. Implementing Mitosis required a major engineering effort to retrofit replication into a conventional kernel design.

Our main observation is that shared memory can be used judiciously with some replication to get the best of both worlds: a simple, elegant and extensible model like the multikernel that can run applications designed for a monolithic kernel. Based on this observation, we propose the *NRkernel*, a new multikernel model that replicates the kernel, but allows replica sharing among cores, balancing performance and simplicity.

The NRkernel is inspired by NR (§2.1), and it has at its core operation logs that provide linearizability. The logs act as global broadcast message channels that eliminate the complex state machine used by Barrelfish for consensus. What remains are simple, single-threaded implementations of data structures that apply updates from the shared log. Based on the NRkernel model, we designed and implemented NrOS, a representative system to evaluate its characteristics.

Hardware Trends. Two major hardware trends motivate the NRkernel. First, shared memory is part of every system today, and current hardware trends indicate that there will be some form of memory sharing – not necessarily with global coherence – available for many new heterogeneous components and disaggregated architectures. The industry is working on multiple specifications that will enable such sharing (*e.g.,* CXL [71], CAPI [64], CCIX [1], Gen-Z [7]). While sharing memory does not scale indefinitely as we add more cores, it works more efficiently than passing messages among a limited number of cores [27]. In such a model, a shared log and replication work well because the log can be accessed by all independent entities connected over a memory bus.

Second, main memory capacities are growing [40] and are expected to increase further with new 3D-stacked technologies [76] and the arrival of tiered memory systems comprising various types of memory, such as DRAM and SCM. Amazon already has offerings for servers with up to 24 TiB. Like other systems [49, 68], the NRkernel leverages the abundance of memory to improve performance with replication.

2.1 Node Replication (NR)

NR creates a linearizable NUMA-aware concurrent data structure from a sequential data structure [28]. NR replicates the sequential data structure on each NUMA node, and it uses an operation log to maintain consistency between the replicas. Each replica benefits from read concurrency using a readers-writer lock and from write concurrency using a technique called *flat combining*. Flat combining batches operations from multiple threads to be executed by a single thread (*the combiner*) per replica. This thread also appends the batched operations to the log using a single atomic operation for the entire batch; other replicas read the log and update their local copy of the structure with the logged operations.

NR relies on three main techniques to scale well:

(1) The operation log uses a circular buffer to represent the abstract state of the concurrent data structure. Each entry in the log represents a mutating operation, and the log ensures a total order among them. The *log tail* gives the index to the last operation added to the log. Each replica consumes the log lazily and maintains a per-replica index into the log that indicates which operations of the log have been executed on its copy of the structure. The log is implemented as a circular buffer of entries that are reused. NR cannot reuse entries that have not been executed on all replicas. This means at least one thread on each NUMA node must occasionally make progress in executing operations on the data structure, otherwise the log could fill up and block new mutating operations. Section 4 discusses how NrOS addresses this.

(2) Flat combining [44] in NR allows threads running on the same NUMA node to share a replica, resulting in better cache locality both from flat combining and from maintaining the replica local to the node's last-level cache. The combiner also benefits from batching by allocating log space for all pending operations at a replica with a single atomic instruction.

(3) The optimized readers-writer lock in NR is a writer-preference variant of the distributed readers-writer lock [75] that ensures correct synchronization between the combiner and reader threads when accessing the sequential replica. This lock lets readers access a local replica while the combiner is adding a batch of operations to the log, increasing parallelism.

NR executes updates and reads differently:

A concurrent mutating operation (update) needs to acquire the combiner lock on the local NUMA node to add the operation to the log and to execute the operation against the local replica. If the thread T executing this operation fails to acquire it, another thread is the combiner for the replica already, so T spin-waits to receive its operation's result from the existing combiner. If T acquires the lock, it becomes *the combiner*. The combiner first flat combines all operations from all threads that are concurrently waiting for their update operations to be appended to the log with a single compare-and-swap. Then, the combiner acquires the writer lock on the local replica's structure, and it sequentially executes all

unexecuted update operations in the log on the structure in order. For each executed operation, the combiner returns its results to the waiting thread.

A concurrent non-mutating operation (read) can execute on its thread's NUMA-node-local replica without creating a log entry. To ensure that the replica is not stale, it takes a snapshot of the log tail when the operation begins, and it waits until a combiner updates the replica past the observed tail. If there is no combiner for the replica, the reading thread becomes the combiner to update the replica before executing its read operation.

NR simplifies concurrent data structure design by hiding the complexities of synchronization behind the log abstraction. NR works well on current systems because the operation log is optimized for NUMA. We adopt NR's NUMA-optimized log design, but we use it to replicate kernel state.

Linearizable operation logs are ubiquitous in distributed systems. For example, many protocols such as Raft [63], Corfu [35], and Delos [15] use a log to simplify reaching consensus and fault tolerance, as well as to scale out a single-node implementation to multiple machines. Recently, the same abstraction has been used to achieve good scalability on large machines both in file systems [19] and general data structures [24, 44, 52, 58, 69]. Concurrent work has developed NUMA-aware data structures from persistent indexes [77].

2.2 NR Example

Listing 1 shows an example where a Rust standard hashmap is replicated using NR. `NRHashMap` wraps an existing sequential hashmap (line 2-4). Programs specify the read (line 7) and update (line 10) operations for the structure and how each should be executed at each replica (lines 20-31) by implementing the `Dispatch` trait.

Listing 2 shows a program that creates a single `NRHashMap` with two NR replicas that use a shared log to synchronize update operations between them. The code creates a log (line 3) which is used to create two replicas (lines 6-7). Finally, the threads can register themselves with any replica and issue operations against it (lines 14-15). NR supports any number of replicas and threads; programs must specify a configuration that is efficient for their structure and operations. For example, NrOS allocates one replica per NUMA node, and each core in a node registers with its NUMA-local replica in order to benefit from locality.

```
1  // Standard Rust hashmap node replicated to each NUMA node.
2  pub struct NRHashMap {
3    storage: HashMap<usize, usize>,
4  }
5
6  // NRHashMap has a Get(k) op that does not modify state.
7  pub enum HMReadOp { Get(usize) }
8
9  // NRHashMap has a Put(k,v) op that modifies replica state.
10 pub enum HMUpdateOp { Put(usize, usize) }
11
12 // The trait implementation describes how to execute each
13 // operation on the sequential structure at each replica.
14 impl Dispatch for NRHashMap {
15   type ReadOp = HMReadOp;
16   type UpdateOp = HMUpdateOp;
17   type Resp = Option<usize>;
18
19   // Execute non-mutating operations (Get).
20   fn dispatch(&self, op: Self::ReadOp) -> Self::Resp {
21     match op {
22       HMReadOp::Get(k) => self.storage.get(&k).map(|v| *v),
23     }
24   }
25
26   // Execute mutating operations (Put).
27   fn dispatch_mut(&mut self, op: Self::UpdateOp) ->
        Self::Resp {
28     match op {
29       HMUpdateOp::Put(k, v) => self.storage.insert(k, v),
30     }
31   }
32 }
```

Listing 1: Single-threaded hashmap transformed using NR.

```
1  // Allocate an operation log to synchronize replicas.
2  let logsize = 2 * 1024 * 1024;
3  let log = Log::<<NRHashMap as
       Dispatch>::UpdateOp>::new(logsize);
4
5  // Create two replicas of the hashmap (one per NUMA node).
6  let replica1 = Replica::<NRHashMap>::new(log);
7  let replica2 = Replica::<NRHashMap>::new(log);
8
9  // Register threads on one NUMA node with replica1.
10 let tid1 = replica1.register();
11 // Threads on other node register similarly with replica2.
12
13 // Issue Get and Put operations and await results.
14 let r = replica1.execute(HMReadOp::Get(1), tid1);
15 let r = replica1.execute_mut(HMUpdateOp::Put(1, 1), tid1);
```

Listing 2: Creating replicas and using NR.

3 Concurrent Node Replication (CNR)

For some OS subsystems with frequent mutating operations (*e.g.,* the file system) NR's log and sequential replicas can limit scalability. Multiple combiners from different replicas can make progress in parallel, but write scalability can be limited by appends to the single shared log and the per-replica readers-writer lock, which only allows one combiner to execute operations at a time within each replica.

To solve this, we extend NR to exploit operation commutativity present in many data structures [30, 45]. Two operations are *commutative* if executing them in either order leaves the structure in the same abstract state. Otherwise, the operations are *conflicting*. Like NR, CNR replicates a data structure across NUMA nodes and maintains consistency between replicas. However, CNR scales the single shared NR log to multiple logs by assigning commutative operations to different logs. Conflicting operations are assigned to the same log,

which ensures they are ordered with respect to each other. Also, CNR can use concurrent or partitioned data structures for replicas, which allows multiple concurrent combiners on each replica – one per shared log. This eliminates the per-replica readers-writer lock and scales access to the structure.

CNR transforms *an already concurrent data structure* to a NUMA-aware concurrent data structure. The original data structure can be a concurrent (or partitioned) data structure that works well for a small number of threads (4-8 threads) within a single NUMA node. This data structure can be lock-free or lock-based and may exhibit poor performance under contention. CNR transforms such a concurrent data structure to one that works well for a large number of threads (*e.g.,* 100s of threads) across NUMA nodes and is resilient to contention.

Similar to transactional boosting [45], CNR only considers the abstract data type for establishing commutativity, not the concrete data structure implementation. For example, consider a concurrent hashmap with an *insert(k, v)* operation. One might think that *insert(k, v)* and *insert(k+1, v′)* are not commutative because they may conflict on shared memory locations. However, the original data structure is concurrent and already safely orders accesses to shared memory locations; hence, these operations commute for CNR and can be safely executed concurrently.

CNR's interface is nearly identical to NR's interface, but it introduces *operation classes* to express commutativity. Implementers of a structure provide functions that CNR uses to map each operation to an operation class. These functions map conflicting operations to the same class, and each class is mapped to a log. Hence, if two conflicting operations execute on the same NUMA node they are executed by the same combiner, which ensures they are executed in order. In contrast, commutative operations can be executed by different combiners and can use different shared logs, allowing them to be executed concurrently.

Overall, CNR increases parallelism within each NUMA node by using a concurrent replica with multiple combiners, and it increases parallelism across NUMA nodes by using multiple (mostly) independent shared logs. However, ultimately every update operation must be executed at all replicas; hence, it comes at a cost, and it cannot scale update throughput beyond that of a single NUMA node. We refer to the general mechanism of replicating a data structure using operation logs as NR; when we need to explicitly distinguish cases that rely on a concurrent data structure with multiple logs (rather than a sequential one with a single log) we use the term CNR.

3.1 CNR Example

The code to use CNR to scale `Put` throughput for a replicated hashmap is almost identical to the example given in Section 2.2; it only changes in two ways. First, the structure embedded in each replica must be thread-safe, since (commutative) operations are executed on it concurrently, *i.e.,* it must implement Rust's `Sync` trait. This creates a subtle, mostly inconsequential, distinction in CNR's `Dispatch` trait because a mutable reference is not required to execute an operation on the structure; hence, Listing 1 line 27 would read `&self` rather than `&mut self`.

```
impl LogMapper for HMUpdateOp {
  fn hash(&self, nlogs: usize, logs: Vec<usize>) {
    logs.clear();
    match self {
      HMUpdateOp::Put(key, _v) => logs.push(*key % nlogs),
    }
  }
}
```

Listing 3: LogMapper implementation for update operations.

Second, CNR uses multiple logs to scale update operations; programs must indicate which operations commute so CNR can distribute commuting operations among logs. To do this, programs implement the `LogMapper` trait for their update operations (Listing 3). The program must implement this trait for read operations as well. `Get` and `Put` operations on a hashmap commute unless they affect the same key, so this example maps all operations with a common key hash to the same class and log. CNR also allows passing multiple logs to the replicas; otherwise, its use is similar to Listing 2. Some operations may conflict with operations in multiple classes, which we discuss in the next section, so a `LogMapper` may map a single operation to more than one class/log.

3.2 Multi-log Scan Operations

In addition to read and update operation types, CNR adds a scan operation type, instances of which belong to more than one operation class. These are operations that conflict with many other operations. Often these are operations that involve the shape of the structure or that need a stable view of many elements of it. Examples include returning the count of elements in a structure, hashmap bucket array resizing, range queries, or, in our case, path resolution and directory traversal for file system `open`, `close`, and `rename` operations. If these operations were assigned to a single class, all other operations would need to be in the same class, eliminating any benefit from commutativity.

Scan operations conflict with multiple operation classes, so they must execute over a consistent state of the replica with respect to all of the classes and logs involved in the scan obtained after its invocation. To obtain this *consistent state*, the thread performing the scan creates an atomic snapshot at the tails of the logs involved in the operation. Then, the replica used by the scan needs to be updated *exactly* up to the snapshot without exceeding it (unlike NR read operations, which can update past the read thread's observed log tail).

Hence, there are two challenges that CNR needs to solve for a correct scan operation: (1) obtaining the atomic snapshot of the log tails while other threads might be updating the logs;

and (2) ensuring that the replica is updated *exactly* up to the observed atomic snapshot.

The problem of obtaining an atomic snapshot is well-studied in concurrent computing [11, 14, 56]. Unlike prior solutions, which are wait-free, we designed a simple, blocking solution that works well in practice and addresses both of the above challenges simultaneously. The scan thread inserts the operation into all of the logs associated with the scan's operation classes. To ensure that two threads concurrently inserting scan operations do not cause a deadlock by inserting the operations in a different order on different logs, each thread must acquire and hold a *scan-lock* while inserting a scan operation in the logs that participate in the operation's atomic snapshot. Update threads can continue to insert their operations into the logs after the unfinished scan and without holding the scan-lock. These operations will be correctly linearized after the scan. Update threads from the same replica as the scan block when they encounter a colocated unfinished scan. With updates blocked on the replica, the scan thread can proceed to execute the operation on the replica once the replica is updated up to the scan's log entries (either by the scan thread or by combiners). After the scan has been executed at that replica, blocked threads at the replica continue with their updates.

Similar to NR read and update operations, scan operations can be of type scan-update (if the scan modifies the data structure) or scan-read (if it does not). With a scan-read operation, the operation only needs to be performed once at the replica where it was invoked; the other replicas ignore log entries for scan-read operations. Like update operations, scan-update operations must affect all replicas, but they must also be executed at each replica only when the replica is at a consistent state for that scan operation. The first combiner that encounters the scan-update operation on a replica acquires all necessary combiner locks, updates the replica to the consistent state, and executes the scan, just as is done on the replica where the scan was initiated.

Scan operations incur higher overhead than other operations, and their cost depends on how many operation classes they conflict with. In our experience, scan operations are rare, so CNR is carefully designed so that scans absorb the overhead while leaving the code paths for simpler and more frequent (read and update) operations efficient.

4 NrOS Design

We designed and implemented NrOS, a representative system for the NRkernel. The overall NrOS kernel as a whole is designed around per-NUMA node *kernel replicas* that encapsulate the state of most of its standard kernel subsystems. Kernel operations access the local replica, and state inside replicas is modified and synchronized with one another using NR, so cross-node communication is minimized (Figure 2). The collective set of kernel replicas act as a multikernel.

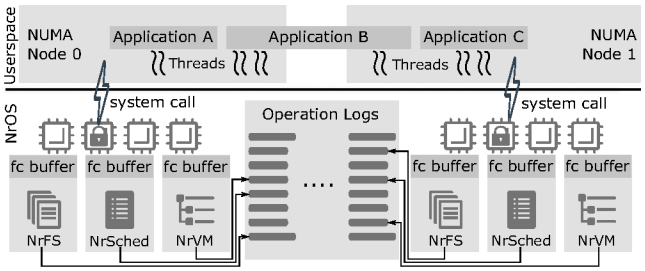


Figure 2: NrOS overview. A per-NUMA-node kernel replica services syscalls on cores on that node. Operations read local replica state; state mutating operations are replicated via NR data structures and are executed at all replicas. Each replica flat combines operations from all cores within a NUMA node, efficiently appending them to one (for NR) or multiple (for CNR) logs. Operations block until completion, ensuring linearizability.

Here, we describe the design of three of NrOS's major subsystems; all of them are replicated via NR:

NR-vMem (§4.2): A virtual memory system that replicates per-process page mapping metadata and hardware page tables for a given process.

NR-FS (§4.3): An in-memory file system that, by design, replicates all metadata and file contents.

NR-Scheduler (§4.4): A process manager that loads and spawns ELF binaries with replicated (read-only) ELF sections and makes global scheduling decisions.

In userspace, NrOS runs native NrOS programs or POSIX programs that are linked against NetBSD libOS components (§5). The libOS uses system calls for memory, process management, and file system operations. This approach can run many POSIX applications like memcached (§6.3.3), Redis, and LevelDB (§6.2.2), though it lacks `fork` and some other functionality which it would need for full POSIX support. If a process wants to write to a file or map a page, it traps into the kernel and enqueues an operation against its local replica file system or process structure and blocks until the request completes before returning from the syscall. Operations that must modify state acquire a local writer lock (if the data structure is not using CNR) before applying the operation directly. If the lock is contended, the operation is enqueued in the local flat combining buffer, then it waits until the current combiner within the node applies the operation. The logs make sure that all modifications to the state of all of the above components are replicated to the other kernel replicas through NR. Operations that only read replica state first ensure the local replica is up-to-date by applying any new operations from the NR log (if needed), and then reading from the local replica.

The kernel is also responsible for physical memory management, interrupt routing and PCI pass-through (network I/O is done directly by processes). These subsystems are not replicated using NR. Devices and interrupts are functional, but their details are outside of the scope of this paper. The full functionality provided by the kernel can be best com-

pared with a lightweight hypervisor which runs processes as containerized/isolated applications using the libOS.

NRkernel Principles. Overall NrOS represents a new point in the multikernel design space we call the *NRkernel* model, which is encapsulated by three key principles.

(1) Combining replicated and shared state. Multikernels like Barrelfish rely on per-core replicas which are prohibitively expensive; NRkernels strike a balance by maintaining a kernel replica per NUMA node; within a NUMA node cores share access to their NUMA-local replica. This maximizes the benefit of shared last-level caches while eliminating slow cross-NUMA node memory accesses. With per-NUMA-node replicas, memory consumption grows with the number of NUMA nodes rather than the number of cores.

(2) Replica consistency via an operation log. Unlike multikernels' expensive use of message passing between all pairs of cores, in NRkernels kernel replicas efficiently synchronize with shared operation logs; logging scales within a NUMA node using flat combining to batch operations across cores. The logs encode all state-changing operations for each subsystem, and ensure replica consistency while hiding the details of architecture-specific performance optimizations.

(3) Compiler-enforced memory and concurrency safety. Rust's compile-time memory-safety and thread-safety guarantees are easy to extend to a kernel's NR implementation. Its segregation of mutating and non-mutating references ensures correct, efficient code is generated where each kernel operation safely accesses the local replica when possible or is logged to synchronize replicas. Rust's `Send` and `Sync` annotations for types are a helpful mechanism to prevent putting data structures on the log that have meaning only on a local core (*e.g.,* a userspace pointer) and prevents them from ever being accessed by another core due to flat combining.

Encapsulating concurrency concerns in a single library with compiler-checked guarantees ensures most operations scale without concerns about subtle concurrency bugs. Having NR isolated in a single logical library also makes it easier to reason about concurrency correctness. For instance, in future work we plan to formally verify the NR mechanism, which would guarantee correct translation for any data structure that leverages NR for concurrency. Contrast this with a traditional kernel such as Linux, where bugs can be introduced not only in the lock library implementation (such as RCU) but especially in the way the rest of the kernel uses the library; only in 2019 did kernel developers consolidate the Linux RCU library to prevent users from mismatching locking calls [8, 59].

In the remainder of this section, we describe NrOS's subsystems, which demonstrate these principles and resolve the challenges of putting them into practice.

4.1 Physical Memory Management

Physical memory allocation and dynamic memory allocation for kernel data structures are the two basic subsystems that do not use NR. Replicated subsystems often require physical frames, but that allocation operation itself should not be replicated. For example, when installing a mapping in a page table, each page table entry should refer to the same physical page frame on all replicas (though, each replica should have its own page tables). If allocator state were replicated, each allocation operation would be repeated on each replica, breaking this. As a result, some syscalls in NrOS must be handled in two steps. For example, when installing a page, the page is allocated up front, outside of NR, and a pointer to it is passed as an argument to the NR operation. This also helps with performance; zeroing a page is slow, and it can be done before the replicated NR operation is enqueued. Operations from a log are applied serially at each replica, so this optimization eliminates head-of-line-blocking on zeroing.

At boot time, the affinity for memory regions is identified, and memory is divided into per-NUMA node caches (`NCache`). The `NCache` statically partitions memory further into two classes of 4 KiB and 2 MiB frames. Every core has a local cache `TCache` of 4 KiB and 2 MiB frames for fast, no-contention allocation when it contains the requested frame size. If it is empty, it refills from its local `NCache`. Similar to slab allocators [21], NrOS `TCache` and `NCache` implement a cache frontend and backend that controls the flow between `TCaches` and `NCaches`.

Unlike Barrelfish or seL4 [53] where all dynamic memory management is externalized to userspace, NrOS makes use of dynamic memory allocation in the kernel. For arbitrary-sized allocations, NrOS implements a simple, scalable allocator with per-core, segregated free lists of 2 MiB or 4 KiB frames. Each frame is divided into smaller, equal-sized objects. A bit field tracks per-object allocations within a frame.

Since NrOS is implemented in Rust, memory management is greatly simplified by relying on the compiler to track the lifetime of allocated objects. This eliminates a large class of bugs (use-after-free, uninitialized memory, *etc.*), but the kernel still has to explicitly handle running out of memory. NrOS uses fallible allocations to handle out-of-memory errors gracefully by returning an error to applications.

However, handling out-of-memory errors in presence of replicated data structures becomes challenging: Allocations that happen to store replicated state must be deterministic (*e.g.,* they should either succeed on all replicas or none). Otherwise, the replicas would end up in an inconsistent state if after executing an operation, some replicas had successful and some had unsuccessful allocations. Making sure that all replicas always have equal amounts of memory available is infeasible because every replica replicates at different times, and allocations can happen on other cores for outside of NR. We solve this problem in NrOS by requiring that all memory allocations for state within node replication or CNR must go through a deterministic allocator. In the deterministic allocator, the first replica that reaches an allocation request allocates memory on behalf of all other replicas too. The deterministic

allocator remembers the results temporarily, until they are picked up by the other replicas which are running behind. If an allocation for any of the replica fails, the leading replica will enqueue the error for all replicas to ensure that all replicas always see the same result. Allocators in NrOS are chainable, and it is sufficient for the deterministic allocator to be anywhere in the chain so it doesn't necessarily have to be invoked for every fine-grained allocation request. Our implementation leverages the custom allocator feature in Rust, which lets us override the default heap allocator with specialized allocators for individual data structures.

4.2 Virtual Memory Management

NrOS relies on the MMU for isolation. Like most conventional virtual memory implementations, NrOS uses a per-process mapping database (as a B-Tree) to store frame mappings which is used to construct the process's hardware page tables. NrOS currently does not support demand paging. Due to increased memory capacities, we did not deem demand paging an important feature for demonstrating our prototype. Both the B-Tree and the hardware page tables are simple, sequential data structures that are wrapped behind the node replication interface for concurrency and replication. Therefore, the mapping database and page tables are replicated on every NUMA node, forming the NR-vMem subsystem. NR-vMem exposes the following mutating operations for a process to modify its address space: `MapFrame` (to insert a mapping); `Unmap` (to remove mappings); and `Adjust` (to change permissions of a mapping). NR-vMem also supports a non-mutating `Resolve` operation (that advances the local replica and queries the address space state).

There are several aspects of NR-vMem's design that are influenced by its integration with node replication.

For example, NR-vMem has to consider out-of-band read accesses by cores' page table walkers. Normally a read operation would go through the node replication interface, ensuring replay of all outstanding operations from the log first. However, a hardware page table walker does not have this capability. A race arises if a process maps a page on core X of replica A and core Y of replica B accesses that mapping in userspace before replica B has applied the update. Luckily, this can be handled since it generates a page fault. In order to resolve this race, the page fault handler advances the replica by issuing a `Resolve` operation on the address space to find the corresponding mapping of the virtual address generating the fault. If a mapping is found, the process can be resumed since the `Resolve` operation will apply outstanding operations. If no mapping is found, the access was an invalid read or write by the process.

`Unmap` or `Adjust` (*e.g.,* removing or modifying page table entries) requires the OS to flush TLB entries on cores where the process is active to ensure TLB coherence. This is typically done in software by the OS (and commonly referred to as performing a TLB "shootdown"). The initiator core starts by enqueuing the operation for the local replica. After node replication returns it knows that the unmap (or adjust) operation has been performed at least against the local page table replica and that it is enqueued as a future operation on the log for other replicas. Next, it sends inter-processor interrupts (IPIs) to trigger TLB flushes on all cores running the corresponding process. As part of the IPI handler the cores first acknowledge the IPI to the initiator. Next, they advance their local replica to execute outstanding log operations (which forces the unmap/adjust if it was not already applied). Then, they poll a per-core message queue to get information about the regions that need to be flushed. Finally, they perform the TLB invalidation. Meanwhile the initiator invalidates its own TLB entries, and then it waits for all acknowledgments from the other cores before it returns to userspace. This shootdown protocol incorporates some of the optimizations described in Amit *et al.* [12]; it uses the cluster mode of the x86 interrupt controller to broadcast IPIs up to 16 CPUs simultaneously, and acknowledgments are sent to the initiator as soon as possible when the IPI is received (this is safe since flushing is done in a non-interruptible way).

4.3 File System

File systems are essential for serving configuration files and data to processes. NR-FS adds an in-memory file system to NrOS that supports some standard POSIX file operations (`open`, `pread`, `pwrite`, `close`, *etc.*). NR-FS tracks files and directories by mapping each path to an inode number and then mapping each inode number to an in-memory inode. Each inode holds either directory or file metadata and a list of file pages. The entire data structure is wrapped by node replication for concurrent access and replication.

There are three challenges for implementing a file system with node replication. First, historically POSIX read operations mutate kernel state (*e.g.,* file descriptor offsets). State mutating operations in node replication must be performed at each replica serially, which would eliminate all concurrency for file system operations. Fortunately, file descriptor offsets are implemented in the userspace libOS, and all NrOS file system calls are implemented with positional reads and writes (pread/pwrite), which do not update offsets. This lets NR-FS apply read operations as concurrent, non-mutating operations.

Second, each file system operation can copy large amounts of data with a single read or write operation. The size of the log is limited, so we do not copy the contents into it. Instead we allocate the kernel-side buffers for these operations and places references to the buffers in the log. These buffers are deallocated once all replicas have applied the operation.

Third, processes supply the buffer for writes, which can be problematic for replication. If a process changes a buffer during the execution of a write operation, it could result in inconsistencies in file contents since replicas could copy data

into the file from the buffer at different times. In NR-FS the write buffer is copied to kernel memory beforehand. This also solves another problem with flat combining: cores are assigned to processes (§4.4) and any core within a replica could apply an operation, but a particular core may not be in the process's address space. Copying the data to kernel memory before enqueuing the operation ensures that the buffer used in the operation is not modified during copies and is readable by all cores without address space changes.

4.3.1 Scaling NR-FS Writes

NR-FS optimizes reads so that many operations avoid the log, but other operations (`write`, `link`, `unlink`, *etc.*) must always be logged. This is efficient when these operations naturally contend with one another since they must serialize anyway and can benefit from flat combining. However, sometimes applications work independently and concurrently on different regions of the file system. For those workloads, node replication would be a limiting bottleneck as it unnecessarily serializes those operations.

To solve this, we developed CNR (§3). CNR uses the same approach to replication as node replication, but it divides commutative mutating operations among multiple logs with a combiner per log to scale performance. Others have observed the benefits of exploiting commutativity in syscalls and file systems [19, 30, 60], and CNR lets NR-FS make similar optimizations. CNR naturally scales operations over multiple combiners per NUMA node under low contention workloads that mutate state, and it seamlessly transitions to use a single combiner when operations contend.

Augmenting NR-FS to use CNR mainly requires implementing the `LogMapper` trait that indicates which log(s) an operation should serialize in (Listing 3). NR-FS hash partitions files by inode number, so operations associated with different files are appended to different logs and applied in parallel.

Some operations like `rename` may affect inodes in multiple partitions. Our current version of NR-FS handles this by serializing these operations with operations on all logs as a scan-update (§3.2). Using scans ensures that if the effect of any cross-partition operation (like `rename`) could have been observed by an application, then all operations that were appended subsequently to any log linearize after it (external consistency). We plan to experiment in the future with more sophisticated approaches that avoid serializing all operations with every such cross-partition operation.

4.4 Process Management and Scheduling

Process management for userspace programs in NrOS is inspired by Barrelfish's "dispatchers" and the "Hart" core abstraction in Lithe [65] with scheduler activations [13] as a notification mechanism.

In NrOS, the kernel-level scheduler (NR-Scheduler) is a coarse-grained scheduler that allocates CPUs to processes. Processes make system calls to request for more cores and to give them up. The kernel notifies processes of core allocations and deallocations via upcalls. To run on a core, a process allocates executor objects (*i.e.,* the equivalent of a "kernel" thread) that are used to dispatch a given process on a CPU. An executor mainly consists of two userspace stacks (one for the upcall handler and one for the initial stack) and a region to save CPU registers and other metadata. Executors are allocated lazily but a process keeps a per-NUMA-node cache to reuse them over time.

In the process, a userspace scheduler reacts to upcalls indicating the addition or removal of a core, and it makes fine-grained scheduling decisions by dispatching threads accordingly. This design means that the kernel is only responsible for coarse-grained scheduling decisions, and it implements a global policy of core allocation to processes.

NR-Scheduler uses a sequential hash table wrapped with node replication to map each process id to a process structure and to map process executors to cores. It has operations to create or destroy a process, to allocate and deallocate executors for a process, and to obtain an executor for a given core.

Process creation must create a new address space, parse the program's ELF headers, allocate memory for program sections, and relocate symbols in memory. A naive implementation might apply those operations on all replicas using node replication, but this would be incorrect. It is safe to independently create a separate read-only program section (like `.text`) for the process by performing an operation at each of the replicas. However, this would not work for writable sections (like `.data`), since having independent allocations per replica would break the semantics of shared memory in the process. Furthermore, we need to agree on a common virtual address for the start of the ELF binary, so position independent code is loaded at the same offset in every replica.

As a result of this, process creation happens in two stages, where operations that cannot be replicated are done in advance. The ELF program file must be parsed up front to find the writable sections, to allocate memory for them, and to relocate entries in them. After that, these pre-loaded physical frames and their address space offsets are passed to the replicated NR-Scheduler create-process operation. Within each replica, the ELF file is parsed again to load and relocate the read-only program sections and to map the pre-allocated physical frames for the writable sections.

Removing a process deletes and deallocates the process at every replica, but it also must halt execution on every core currently allocated to the process. Similar to TLB shootdowns, this is done with inter-processor interrupts and per-core message queues to notify individual cores belonging to a replica.

4.5 Log Garbage Collection

As described in Section 2.1, operation logs are circular buffers, which fixes the memory footprint of node replication. However, entries can only be overwritten once they have been applied at all replicas; progress in applying entries at a replica can become slow if operations are rare at that replica (*e.g.,* if cores at one replica spend all of their time in userspace).

NrOS solves this in two ways. First, if a core at one replica cannot append new operations because another replica is lagging in applying operations, then it triggers an IPI to a core associated with the lagging replica. When the core receives the IPI, it immediately applies pending log operations to its local replica, unblocking the stalled append operation at the other replica. On initialization, NrOS provides a callback to the node replication library that it can use to trigger an IPI; the library passes in the id of the slow replica and the id of the log that is low on space. Second, frequent IPIs are expensive, so NrOS tries to avoid them by proactively replicating when cores are idle. So long as some core at each replica sometimes has no scheduled executor, IPIs are mostly avoided.

Finally, some operations hold references to data outside of the log that may need to be deallocated after an operation has been applied at all replicas (*e.g.,* buffers that hold data from file system writes). If deallocation of these resources is deferred until a log entry is overwritten, then large pools of allocated buffers can build up, hurting locality and putting pressure on caches and TLBs. To more eagerly release such resources, these references embed a reference count initialized to the number of replicas, which is decremented each time the operation is applied at a replica; when the count reaches zero, the resource is released.

5 Implementation

We implemented NrOS from scratch in Rust; it currently targets the x86-64 platform. It also has basic support for Unix as a target platform, which allows kernel subsystems to be run within a Linux process and helps support unit testing. The core kernel consists of 11k lines of code with 16k additional lines for kernel libraries (bootloader, drivers, memory management, and platform specific code factored out from the core kernel into libraries). In the entire kernel codebase, 3.6% of lines are within Rust `unsafe` blocks (special blocks that forego the compiler's strong memory- and thread-safety guarantees). Most of this unsafe code casts and manipulates raw memory (*e.g.,* in memory allocators or device drivers), a certain amount of which is unavoidable in kernel code.

Node Replication. We implemented node replication in Rust as a portable library totaling 3.4k lines of code (5% in `unsafe` blocks). We made some notable changes to the state-of-the-art node replication design [28] and built CNR on top of it. Specifically, our implementation relies on Rust's generic types, making it easy to lift arbitrary, sequentially-safe Rust structures into node-replicated, concurrent structures. This is done by implementing the `Dispatch` interface in Listing 1.

Name	Memory	Nodes/Cores/Threads
2×14 Skylake	192 GiB	2x14x2 Xeon Gold 5120
4×24 Cascade	1470 GiB	4x24x2 Xeon Gold 6252

Table 1: Architectural details of our evaluation platforms.

Userspace Library. NrOS provides a userspace runtime support library (vibrio) for applications. It contains wrapper functions for kernel system calls, a dynamic memory manager, and a cooperative scheduler that supports green threads and standard synchronization primitives (condition variables, mutexes, readers-writer locks, semaphores, *etc.*).

This library also implements hypercall interfaces for linking against rumpkernels (a NetBSD-based library OS) [50]. This allows NrOS to run many POSIX programs. rumpkernel provides libc and libpthread which, in turn, use vibrio for scheduling and memory management through the hypercall interface. The hypercall interface closely matches the reference implementation of the rumprun-unikernel project [9]; however, some significant changes were necessary to make the implementation multi-core aware. The multi-core aware implementation was inspired by LibrettOS [62].

The NrOS kernel itself does not do any I/O, but it abstracts interrupt management (using I/O APIC, xAPIC and x2APIC drivers) and offers MMIO PCI hardware passthrough to applications. Applications can rely on the rump/NetBSD network or storage stack and its device drivers for networking and disk access (supporting various NIC models and AHCI based disks). The I/O architecture is similar to recent proposals for building high performance userspace I/O stacks [18, 66].

6 Evaluation

This section answers the following questions experimentally:

- How does NrOS's design compare against monolithic and multikernel operating systems?
- What is the latency, memory and replication mechanism trade-off in NrOS' design compared to others?
- Does NrOS's design matter for applications?

We perform our experiments on the two machines given in Table 1. For the Linux results, we used Ubuntu version 19.10 with kernel version 5.3.0. If not otherwise indicated, we did not observe significantly different results between the two machines and omit the graphs for space reasons. We pinned benchmark threads to physical cores and disabled hyperthreads. Turbo boost was enabled for the experiments. If not otherwise indicated we show bare-metal results.

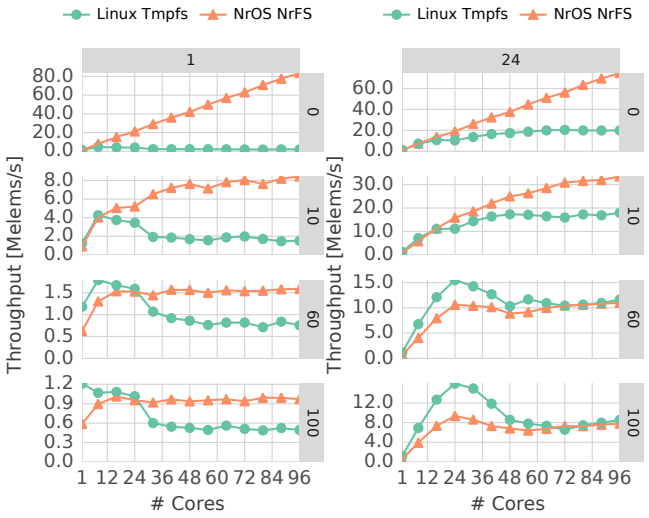


Figure 3: NR-FS read/write scalability for write ratios 0%, 10%, 60% and 100% with 1 or 24 files on 4×24 Cascade.

6.1 Baseline Node Replication Performance

We have extensively tested our Rust-based NR library in userspace on Linux using a variety of structures to compare its performance with the existing state-of-the-art NR implementation [28] and with other optimized concurrent structures. We omit these results as they are orthogonal to NrOS's contributions, but we summarize a few key results when comparing with RCU which is the most relevant comparison for NrOS.

We tested scaling a hash table on 4×24 Cascade, NR (just wrapping the unmodified, sequential `HashMap` from the Rust standard library as shown in Listing 1) outperforms other concurrent hash maps written in Rust and the *urcu* library's read-modify-write-based hash table. With 0% updates, urcu and node replication scale linearly, but urcu lags behind; NR achieves perfect NUMA locality by replicating the hash table. The NR hash table also stores elements in-place, whereas urcu stores a pointer to each element, leading to an additional de-reference. This is inherent in the urcu design since elements are deallocated by the last reader. In short, for read-only workloads, NR performs about twice as well as urcu, which is the next fastest approach we tested. Even with any fraction of read operations it performs strictly better at scale. However, we find that urcu can outperform NR when reads and writes are split between threads rather than when reads and writes are mixed on every thread. This is because RCU allows readers to proceed without any synchronization overhead whereas node replication must acquire the local reader lock.

6.2 NR-FS

We evaluate NR-FS performance by studying the impact of issuing read and write operations to files using a low-level microbenchmark and a LevelDB application benchmark.

6.2.1 Microbenchmark: NR-FS vs `tmpfs`

In this benchmark, we are interested in the file operation throughput while varying cores, files and the read/write ratio. We pre-allocate files of 256 MiB upfront. Each core then accesses exactly one file, and each file is shared by an equal number of cores when the number of cores exceeds the number of files. The cores either read or write a 1 KiB block in the file uniformly at random. This general access pattern is typical for many databases or storage systems [43, 46]. We compare against the Linux's (in-memory) `tmpfs` to minimize persistence overhead [23].

Figure 3 shows the achieved throughput for write ratios 0%, 10%, 60%, and 100%, while increasing the number of cores (x-axis). The left graphs measured the throughout if a single file is read from/written to concurrently. With $WR = 0\%$, NR-FS achieves ∼40× better read performance at max. utilization. This increase is due to replication of the file system and making reads an immutable operation; largely the benefit comes from higher available memory bandwidth (4×24 Cascade has 88 GiB/s local vs. 16 GiB/s remote read bandwidth). However, replication increases the memory consumption significantly; for 24 files, each 256 MiB, `tmpfs` uses 6.1 GiB (6 GiB data and 0.1 GiB metadata) as compared to 24.1 GiB (24 GiB data and 0.1 GiB metadata) for NR-FS. For higher write ratios, `tmpfs` starts higher as NR-FS performs an additional copy from user to kernel memory to ensure replica consistency (Section 4.3) and its write is likely not as optimized as the Linux codebase. However, the `tmpfs` throughput drops sharply at the first NUMA node boundary due to contention and increased cache coherence traffic. For $WR = 100\%$, NR-FS performs ∼2× better than `tmpfs` at max. utilization.

Discussion: With the Intel architectures used in our setting, single file access generally outperforms Linux as soon as the file size exceeds the combined L3 size of all NUMA nodes (128 MiB on 4×24 Cascade). A remote L3 access on the same board is usually faster than a remote DRAM access; therefore, replication plays a smaller role in this case. As long as the file fits in L3 or the combined L3 capacity, NR-FS has on-par or slightly worse performance than `tmpfs`. NR-FS gains its advantage by trading memory for better performance.

The right side of the figure shows the less contended case where the cores are assigned to 24 files in a round-robin fashion (at 96 cores, each file is shared by four cores). For $WR = 0\%$, NR-FS performs around 4× better than `tmpfs` due to node local accesses from the local replica. For higher write ratios (60%, 100%), `tmpfs` performs better than NR-FS on the first NUMA node. On top of the additional copy, the major reason for the overhead here is that intermediate buffers for writes in NR-FS remain in the log until all replicas have applied the operation. This results in a larger working set and cache footprint for writes than `tmpfs`, which can reuse the same buffers after every operation. We empirically verified that this is the case by making the block size smaller; with

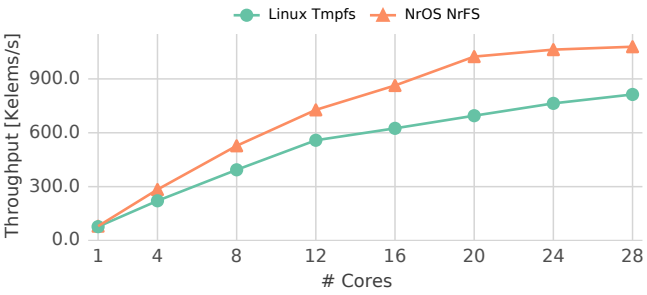


Figure 4: LevelDB readrandom throughput on 2×14 Skylake.

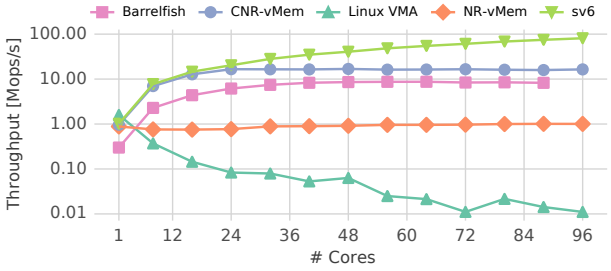


Figure 5: NrOS page insertion throughput on 4×24 Cascade in comparison with other OSes.

this change the performance discrepancy between `tmpfs` and NR-FS disappears.

After the first NUMA node, `tmpfs` throughput degrades due to contention, and cross-node memory accesses. NR-FS manages to keep the same performance as a single NUMA node. At *cores* = 96 both systems have similar throughput, but NR-FS actively replicates all writes on all 4 nodes. We omit the results using 2 to 23 files because the trend is the same: NR-FS performs much better for read-heavy workloads and the same or better for write-heavy operations. On 4×24 Cascade with 24 logs and $WR = 100\%$, NR-FS scalability stops for this benchmark with more than 24 files because of the additional CPU cycles required for active replication of all writes, the observed throughput remains constant instead.

Impact of multiple logs: Figure 3 shows the advantages of using CNR over node replication for less contended, write-intensive workloads. As discussed in (§2.1), the write performance for node replication data structures is often limited by a single combiner per replica. While it is certainly possible to build compound structures using multiple node replication instances (*e.g.,* one per file), this is typically too fine-grained, as often we have much less compute cores than files. We resolve this issue with our CNR (§3) scheme. A CNR based NR-FS performs 8× better (for $wr = 100$) than a node replication based NR-FS while preserving read performance.

6.2.2 LevelDB Application Benchmark

To evaluate the impact of the NR-FS design on real applications we use LevelDB, a widely used key-value store. LevelDB relies on the file system to store and retrieve its data, which makes it an ideal candidate for evaluating NR-FS. We use a key size of 16 Bytes and a value size of 64 KiB. We load 50K key-value pairs for a total database size of 3 GiB, which LevelDB stores in 100 files.

NR-FS outperforms `tmpfs` when running LevelDB. Figure 4 shows LevelDB throughput when running its included `readrandom` benchmark while varying core count. After *cores* = 12, contention on a mutex within LevelDB begins to affect the scalability of both systems. At *cores* = 28, LevelDB on NrOS has 1.33x higher throughput than on Linux.

6.3 NR-vMem

We evaluate the performance of NR-vMem with microbenchmarks that stress the address-space data structures under contention, and exercise the respective TLB shootdown protocols on different operating systems. Finally, we measure the impact of page table replication with memcached.

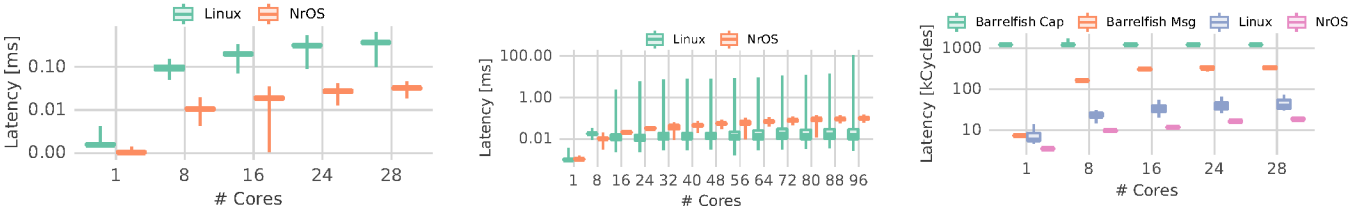
6.3.1 Map Performance

For this benchmark we compare NrOS against Linux, sv6, and Barrelfish. We allocate a backing memory object (*e.g.,* a physical memory region on NrOS, a shared memory object on Linux, a physical frame referenced by a capability on Barrelfish, and a memory-backed file on sv6) and repeatedly map the same memory object into the virtual address space of the process. The benchmark focuses on synchronization overheads; it creates the mapping and updates bookkeeping information without allocating new backing memory.

We evaluate a partitioned scenario where each thread creates new mappings in its own address space region (the only comparison supported by all OSes). We ensure that page tables are created with the mapping request by supplying the appropriate flags. sv6 does not support `MAP_POPULATE`, so we force a page fault to construct the mapping. We show throughput of the benchmark in Figure 5.

NR-vMem wraps the entire address space behind a single instance of node replication, therefore it does not scale even for disjoint regions. As this benchmark consists of 100% mutating operations, it has constant throughput – similar to the other benchmarks it remains stable under heavy contention.

Linux is very similar to NR-vMem in its design (apart from missing replication). It uses a red-black tree to keep track of mappings globally. For each iteration of the benchmark, a new mapping has to be inserted into the tree and afterwards the page fault handler is called by `mmap` to force the population of the page table. The entire tree is protected by a global lock and therefore performance decreases sharply under contention. The single-threaded performance of Linux VMA is slightly better than NR-vMem which has still room for improvement:



(a) Map latency for 2×14 Skylake.

(b) Map latency for 4×24 Cascade.

(c) Unmap latency on 2×14 Skylake.

Figure 6: Virtual memory operation (map and unmap) observed latency distributions.

For example, our current implementation zeroes page tables while holding the combiner lock.

Barrelfish (git revision `06a9f5`) tracks resources with a distributed, partitioned capability system. Updating a page table and allocating new memory corresponds to a capability operation, which may require agreement among the multikernel nodes. The Barrelfish memory management code shares a capability to the top level page table and statically partitions the virtual address space of a process (*e.g.,* a core has only capabilities to its own partition which only it can modify). So, inserts to private regions do not require any agreement. Furthermore, this design uses a single page table per process in the system; therefore, there is no overhead to synchronize replicated state.

For good scalability, we eliminated the use of $O(n)$ linked-list operations for the address-space meta-data tracking of the OS. Once we fixed those issues, Barrelfish throughput scaled with a sub-linear factor. Concurrent updates to the same partition from multiple cores are not supported. This could be implemented with delegation, by using the provided messaging infrastructure in the library OS and capability operations.

sv6 (git revision `9271b3f`) uses a radix tree data structure [29] for its mapping database. It is able to apply fine-grained locking for non-overlapping regions of the address space by atomically setting bits at each level in the radix tree. Compared to all evaluated systems, sv6 performs best for disjoint regions with near-linear scalability. A potential downside is the memory overhead, since sv6 replicates page tables on every core. It mitigates this issue with lazy construction of page tables and discarding them under memory pressure.

CNR-vMem. While NR-vMem does not scale as well as sv6 or Barrelfish for mappings in disjoint regions, it keeps a relatively complex interaction of multiple data structures as entirely single-threaded code. If better scalability is desired, we can scale NR-vMem updates by using CNR. Similar to Barrelfish, CNR-vMem partitions the address space into 512 separate regions and maps updates to partitions with different logs. CNR-vMem matches sv6' performance on the first NUMA node. Afterwards, scaling stops because of per-NUMA replication. Compared to Barrelfish, we find that CNR-vMem is more flexible: concurrent updates from multiple cores to the same partition are supported without extra implementation effort thanks to flat-combining.

Latency. To understand how the batching in node replication impacts latency, we further instrument Linux and NR-vMem by measuring the completion time for 100k requests per core. Figure 6a and 6b show the latency distributions (with the min representing p1, and max p99) for 4×24 Cascade and 2×14 Skylake. We observe slightly worse median latencies for NrOS on 4×24 Cascade but overall better tail characteristics and throughput. On the other hand, we find that NrOS has better latencies than Linux on 2×14 Skylake. Latency is directly correlated with the number of cores participating in flat combining (*i.e.,* 2×14 Skylake has only 14 cores per replica vs. 24 on 4×24 Cascade). Per-NUMA replicas offer a good compromise for memory consumption vs. throughput, but we can tune latency further by having more than one replica per node on NUMA nodes with more cores.

6.3.2 Unmap and TLB Shootdown Performance

We evaluate the scalability of unmapping a 4 KiB page by measuring the time it takes to execute the unmap system call, which includes a TLB shootdown on all cores running the benchmark program.

We compare our design to Linux and Barrelfish. Linux uses a single page table which is shared among the cores used in this benchmark. The general shootdown protocol is similar to the one in NrOS, except that NrOS has to update multiple page tables in case a process spawns multiple NUMA nodes. Barrelfish partitions control over its page table per core and uses the capability system (Barrelfish Cap) or userspace message passing (Barrelfish Msg) to ensure consistency among the replicas and coherency with the TLB. Barrelfish uses point-to-point message channels instead of IPIs.

Figure 6c shows the latency results. NrOS outperforms all other systems. Barrelfish Cap has a constant latency when using a distributed capability operation because all cores participate in the 2PC protocol to revoke access to the unmapped memory regardless of whether this memory was actually mapped on that core. Moreover, implementing the TLB shootdown using point-to-point messages (Barrelfish Msg) has a higher constant overhead compared to using x2APIC

OS	Time	Throughput	System Mem.	PT Mem.	PT Walks
NrOS 4-replicas	251 s	63 Mop/s	424 GiB	3.3 GiB	1.20 kcyc/op
NrOS 1-replica	276 s	57 Mop/s	421 GiB	840 MiB	1.54 kcyc/op
Linux	327 s	48 Mop/s	419 GiB	821 MiB	1.63 kcyc/op

Table 2: memcached on NrOS (1 and 4 replicas) and Linux running on 4×24 Cascade, comparing runtime, throughput, total system memory consumption, process page table memory and cycles spent by the page table walkers.

with broadcasting in NrOS and Linux due to sequential sending and receiving of point-to-point messages. Using more optimized message topologies could potentially help [48].

Linux should achieve better results than NrOS, especially when we spawn across NUMA since it only has to update one page table. However, the proposed changes to Linux from the literature [12] which inspired our TLB shootdown protocol have not yet been integrated to upstream Linux. We expect Linux to be comparable once early acknowledgments and concurrent shootdown optimizations become available.

6.3.3 Page Table Replication with Memcached

As a final benchmark, we measure the impact of replicated page tables on memcached. When taking into account the implicit reads of the MMU, page tables often end up being read much more than modified. memcached serves as a representative application for workloads with generally high TLB miss ratios (*i.e.,* applications with large, in-memory working sets and random access patterns).

We measure the throughput of memcached with `GET` requests (8 byte keys, 128 byte values, 1B elements) on 4×24 Cascade. Our benchmark directly spawns 64 client threads inside of the application. For this experiment, we run Linux and NrOS inside KVM because we want to have access to the performance counters, which is currently not supported on NrOS. To limit the effects of nested-paging, we configure KVM to use 2 MiB-pages, and use 4 KiB pages in both the Linux and NrOS guests.

Table 2 compares memcached running on NrOS in different configurations and Linux. Overall, the achieved throughput for NrOS (with per-NUMA replication) is 1.3× higher than Linux. To quantify the impact of page table replication on the throughput, we can configure NrOS to use a single replica for the process (NrOS 1-replica). We find that the page table replication accounts for a third of the overall improvement compared to Linux. The systems have different physical memory allocation policies, locking implementation, scheduling, and page tables *etc.*, so it is difficult to attribute the other two thirds to specific causes.

By instrumenting performance counters, we find that remote page table walks – a key bottleneck for this workload – decreased by 23% with replication. NrOS does use 4× more memory for the replicated page tables structures. In total, this still amounts to less than 1% of the total memory.

7 Conclusion and Future work

We designed and implemented NrOS, an OS that uses single-threaded data structures that are automatically adapted for concurrency via operation logging, replication, and flat combining. Our results show that the NRkernel model can achieve performance that is competitive with or better than well-established OSes in many cases.

NrOS' unique design makes it an interesting platform to explore several future directions:

Relaxing consistency. We apply node replication on relatively coarse-grained structures, which makes reasoning about concurrency easy. CNR improves performance by exploiting commutativity among mutating operations. However, we could achieve better performance by relaxing strong consistency between replicas for some operations.

Verifying correctness. NrOS might also serve as a useful basis for a verified multi-core operating system by using verification in two steps: verify the node replication transformation from a sequential data structure to a concurrent one, then verify the sequential data structures. Verifying node replication is harder, but it only needs to be done once. Verifying new sequential data structures is substantially easier.

Extending NrOS for disaggregated compute. NrOS' log-based approach with replication is most useful when systems have high remote access latencies. Thus, NrOS could be extended to work over interconnects that offer shared memory in compute clusters via Infiniband or other high-speed networks by designing a new log optimized for the interconnect.

Acknowledgments

We thank our OSDI 2020 and 2021 reviewers and our shepherd Irene Zhang for their thoughtful feedback. Ankit Bhardwaj and Chinmay Kulkarni contributed to this work as PhD students at University of Utah and during internships at VMware Research. This material is based upon work supported by the National Science Foundation under Grant No. CNS-1750558. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Chinmay Kulkarni is supported by a Google PhD Fellowship.

References

[1] CCIX. https://www.ccixconsortium.com.

[2] Concurrency bugs should fear the big bad data-race detector. https://lwn.net/Articles/816850/.

[3] Fix a data race in ext4_i(inode)->i_disksize. https://lore.kernel.org/patchwork/patch/1190562/.

[4] Fix a data race in mempool_free(). https://lore.kernel.org/patchwork/patch/1192684/.

[5] Fix locking in bdev_del_partition. https://patchwork.kernel.org/project/linux-block/patch/20200901095941.2626957-1-hch@lst.de/.

[6] Fix two RCU related problems. https://lore.kernel.org/patchwork/patch/990695/.

[7] Gen-Z Consortium. https://genzconsortium.org/.

[8] The RCU API, 2019 Edition. https://lwn.net/Articles/777036/.

[9] The Rumprun Unikernel. https://github.com/rumpkernel/rumprun.

[10] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 283–300, 2020.

[11] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993.

[12] Nadav Amit, Amy Tai, and Michael Wei. Don't shoot down TLB shootdowns! In *European Conference on Computer Systems (EuroSys)*, 2020.

[13] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 95–109, 1991.

[14] James Aspnes and Maurice Herlihy. Wait-free data structures in the asynchronous PRAM model. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 340–349, 1990.

[15] Mahesh Balakrishnan, Jason Flinn, Chen Shen, Mihir Dharamshi, Ahmed Jafri, Xiao Shi, Santosh Ghosh, Hazem Hassan, Aaryaman Sagar, Rhed Shi, et al. Virtual consensus in Delos. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 617–632, 2020.

[16] Antonio Barbalace, Binoy Ravindran, and David Katz. Popcorn: a replicated-kernel OS based on Linux. In *Proceedings of Ottawa Linux Symposium (OLS)*, 2014.

[17] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[18] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. IX: A protected dataplane operating system for high throughput and low latency. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 49–65, 2014.

[19] Srivatsa S. Bhat, Rasha Eqbal, Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scaling a file system to many cores using an operation log. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 69–86, 2017.

[20] William J. Bolosky, Robert P. Fitzgerald, and Michael L. Scott. Simple but effective techniques for NUMA memory management. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 19–31, 1989.

[21] Jeff Bonwick. The slab allocator: An object-caching kernel memory allocator. In *Proceedings of the USENIX Summer 1994 Technical Conference (USTC)*, page 6, 1994.

[22] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: An Operating System for Many Cores. In *Symposium on Operating Systems Design and Implementation (OSDI)*, page 43–57, 2008.

[23] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. In *Symposium on Operating Systems Design and Implementation (OSDI)*, page 1–16, 2010.

[24] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. OpLog: a library for scaling update-heavy data structures. Technical report, 2014.

[25] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 15(4):412–447, November 1997.

[26] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the 15th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 167–178, 2010.

[27] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message Passing or Shared Memory: Evaluating the delegation abstraction for multicores. In *International Conference on Principles of Distributed Systems (OPODIS)*, pages 83–97, 2013.

[28] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. Black-box Concurrent Data Structures for NUMA Architectures. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 207–221, 2017.

[29] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *European Conference on Computer Systems (EuroSys)*, page 211–224, 2013.

[30] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. *ACM Transactions on Computer Systems (TOCS)*, 32(4), January 2015.

[31] Jonathan Corbet. The big kernel lock strikes again, 2008. https://lwn.net/Articles/281938/.

[32] Jonathan Corbet. Big reader locks, 2010. https://lwn.net/Articles/378911/.

[33] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on NUMA systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–394, 2013.

[34] Pantazis Deligiannis, Alastair F. Donaldson, and Zvonimir Rakamaric. Fast and precise symbolic analysis of concurrency bugs in device drivers. In *International Conference on Automated Software Engineering (ASE)*, pages 166–177, 2015.

[35] Medhavi Dhawan, Gurprit Johal, Jim Stabile, Vjekoslav Brajkovic, James Chang, Kapil Goyal, Kevin James, Zeeshan Lokhandwala, Anny Martinez Manzanilla, Roger Michoud, Maithem Munshed, Srinivas Neginhal, Konstantin Spirov, Michael Wei, Scott Fritchie, Chris Rossbach, Ittai Abraham, and Dahlia Malkhi. Consistent clustered applications with Corfu. *Operating Systems Review*, 51(1):78–82, 2017.

[36] Hugh Dickins. [PATCH] mm lock ordering summary, 2004. http://lkml.iu.edu/hypermail/linux/kernel/0406.3/0564.html.

[37] Marco Elver. Add Kernel Concurrency Sanitizer (KCSAN). https://lwn.net/Articles/802402/, 2019.

[38] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[39] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Symposium on Operating Systems Design and Implementation (OSDI)*, page 151–162, 2010.

[40] Paolo Faraboschi, Kimberly Keeton, Tim Marsland, and Dejan Milojicic. Beyond processor-centric operating systems. In *Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[41] Pedro Fonseca, Rodrigo Rodrigues, and Björn B. Brandenburg. SKI: Exposing Kernel Concurrency Bugs Through Systematic Schedule Exploration. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[42] Ben Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 87–100, 1999.

[43] Sanjay Ghemawat and Jeff Dean. LevelDB, 2011.

[44] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat Combining and the synchronization-parallelism tradeoff. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364, 2010.

[45] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, page 207–216, 2008.

[46] Richard D Hipp. SQLite, 2020.

[47] Dae R. Jeong, Kyungtae Kim, Basavesh Ammanaghatta Shivakumar, Byoungyoung Lee, and Insik Shin. Razzer: Finding kernel race bugs through fuzzing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, 2019.

[48] Stefan Kaestle, Reto Achermann, Roni Haecki, Moritz Hoffmann, Sabela Ramos, and Timothy Roscoe. Machine-aware atomic broadcast trees for multicores. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 33–48, 2016.

[49] Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. Shoal: Smart allocation and replication of memory for parallel programs. In *USENIX Annual Technical Conference (ATC)*, pages 263–276, 2015.

[50] Antti Kantee. *Flexible Operating System Internals: The Design and Implementation of the Anykernel and Rump Kernels*. PhD thesis, Aalto University, 2012.

[51] Sanidhya Kashyap, Irina Calciu, Xiaohe Cheng, Changwoo Min, and Taesoo Kim. Scalable and practical locking with shuffling. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 586–599, 2019.

[52] Jaeho Kim, Ajit Mathew, Sanidhya Kashyap, Madhava Krishnan Ramanathan, and Changwoo Min. MV-RLU: Scaling Read-Log-Update with multi-versioning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 779–792, 2019.

[53] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.

[54] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W. Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, Amos Waterland, and Volkmar Uhlig. K42: Building a complete operating system. In *European Conference on Computer Systems (EuroSys)*, pages 133–145, 2006.

[55] Alexander Lochmann, Horst Schirmeier, Hendrik Borghorst, and Olaf Spinczyk. LockDoc: Trace-Based Analysis of Locking in the Linux Kernel. In *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, pages 11:1–11:15, 2019.

[56] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[57] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 399–413, 2019.

[58] Alexander Matveev, Nir Shavit, Pascal Felber, and Patrick Marlier. Read-Log-Update: A lightweight synchronization mechanism for concurrent programming. In *ACM Symposium on Operating Systems Principles (SOSP)*, pages 168–183, 2015.

[59] Paul E McKenney. A critical RCU safety property is... ease of use! In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 132–143, 2019.

[60] Changwoo Min, Sanidhya Kashyap, Steffen Maass, Woonhak Kang, and Taesoo Kim. Understanding many-core scalability of file systems. In *USENIX Annual Technical Conference (ATC)*, 2016.

[61] Ingo Molnar and Davidlohr Bueso. Generic Mutex Subsystem, 2017. https://www.kernel.org/doc/Documentation/locking/mutex-design.txt.

[62] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. LibrettOS: A dynamically adaptable multiserver-library OS. In *International Conference on Virtual Execution Environments (VEE)*, page 114–128, 2020.

[63] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, pages 305–320, 2014.

[64] OpenCAPI consortium. http://opencapi.org.

[65] Heidi Pan, Benjamin Hindman, and Krste Asanovic. Composing parallel software efficiently with Lithe. In *International Conference on Programming Language Design and Implementation (PLDI)*, 2010.

[66] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2014.

[67] Sean Peters, Adrian Danis, Kevin Elphinstone, and Gernot Heiser. For a Microkernel, a big lock is fine. In *Proceedings of the 6th Asia-Pacific Workshop on Systems (APSys)*, 2015.

[68] Iraklis Psaroudakis, Stefan Kaestle, Matthias Grimmer, Daniel Goodman, Jean-Pierre Lozi, and Tim Harris. Analytics with smart arrays: Adaptive and efficient language-independent data. In *European Conference on Computer Systems (EuroSys)*, 2018.

[69] Ori Shalev and Nir Shavit. Predictive log-synchronization. In *European Conference on Computer Systems (EuroSys)*, page 305–315, 2006.

[70] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. LegoOS: A disseminated, distributed os for hardware resource disaggregation. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 69–87, 2018.

[71] Navin Shenoy. A Milestone in Moving Data. https://newsroom.intel.com/editorials/milestone-moving-data.

[72] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 279–289, 1996.

[73] Al Viro. parallel lookups, 2016. https://lwn.net/Articles/684089/.

[74] Michael von Tessin. The Clustered Multikernel: An approach to formal verification of multiprocessor os kernels. In *Workshop on Systems for Future Multicore Architectures (SFMA)*, 2012.

[75] Dmitry Vyukov. Distributed reader-writer mutex. http://www.1024cores.net/home/lock-free-algorithms/reader-writer-problem/distributed-reader-writer-mutex, 2011.

[76] Daniel Waddington, Mark Kunitomi, Clem Dickey, Samyukta Rao, Amir Abboud, and Jantz Tran. Evaluation of Intel 3D-Xpoint NVDIMM technology for memory-intensive genomic workloads. In *Proceedings of the International Symposium on Memory Systems (MEMSYS)*, page 277–287, 2019.

[77] Qing Wang, Youyou Lu, Junru Li, and Jiwu Shu. Nap: A black-box approach to NUMA-aware persistent memory indexes. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2021.

[78] David Wentzlaff and Anant Agarwal. Factored Operating Systems (Fos): The case for a scalable operating system for multicores. *Operating Systems Review*, 43(2):76–85, April 2009.

[79] Weiwei Xiong, Soyeon Park, Jiaqi Zhang, Yuanyuan Zhou, and Zhiqiang Ma. Ad hoc synchronization considered harmful. In *Symposium on Operating Systems Design and Implementation (OSDI)*, page 163–176, 2010.

[80] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. Krace: Data race fuzzing for kernel file systems. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*, 2020.

A Artifact Appendix

Abstract

The evaluated artifact is provided as a git repository and contains the source code of NrOS, build instructions and scripts to run the OS and benchmarks used in this paper.

Scope

The artifact contains code and steps to reproduce results obtained in Figure 3, Figure 4, Figure 5 and Figure 6.

Contents

The artifact consists of NrOS, including libraries, userspace programs and benchmarks. The documentation to build and run NrOS, along with the necessary commands to run the benchmarks are written down in the `doc` folder of the repository. The document which lists the steps to execute the artifact evaluation is located at `doc/src/benchmarking/ArtifactEvaluation.md`.

Hosting

The artifact source code for NrOS is published on Github under https://github.com/vmware-labs/node-replicated-kernel.
The code used in the artifact evaluation is tagged as `osdi21-ae-v2`.

Requirements

Building NrOS requires an x86-64 system set-up with Ubuntu 20.04 LTS.
NrOS itself requires an Intel CPU (Skylake microarchitecture or later) to run. The following CPUs are known to work: Xeon Gold 5120, 6252 or 6142. For virtualized execution on these platforms, a Linux host system with QEMU (version >= 5.0.0) and KVM is required. For bare-metal execution, DELL PowerEdge R640 and R840 servers systems are known to work.