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*

OSDI'21

Presented by Yuchao Qian

Outline

- Background & Overview
- Node Replication
- NrOS Design
- Evaluation
- Conclusion

Background

- **Increasing CPU core count**
- **Non-uniform memory access (NUMA)**
- **Elaborate concurrent data structures**
  - fine-grained locking
  - read-copy-update (RCU)
  - good performance but increased complexity

Background

- **Big kernel lock works for microkernel[1]**
  - does not target NUMA
- **Multikernel: per-core kernels, communicating via message passing[2]**
  - scales well
  - too much complexity and overhead for hosts with shared mem

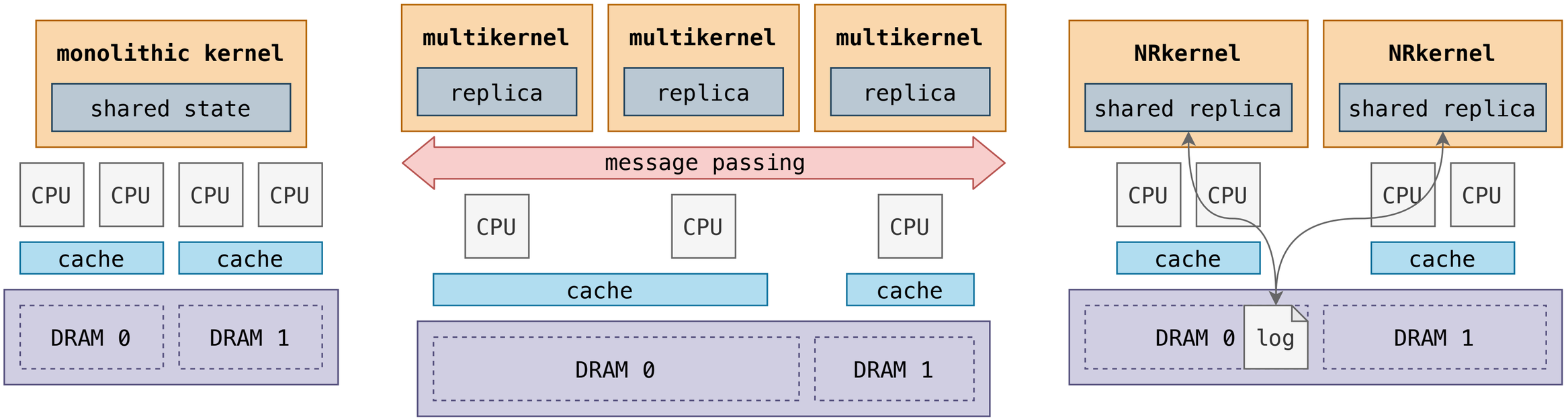
1. For a Microkernel, a big lock is fine. APSys'15.
2. The Multikernel: A New OS Architecture for Scalable Multicore Systems. SOSP'09.

Overview

- **NRkernel**
  - node replication[1]
  - kernel state replica per NUMA node
  - read local replica concurrently
  - mutate by shared operation log, serially
- **NrOS: an NRkernel**

1. Black-box Concurrent Data Structures for NUMA Architectures. ASPLOS'17.

Overview


monolithic kernel
shared state
CPU
CPU
CPU
CPU
cache
cache
DRAM 0
DRAM 1
multikernel
replica
multikernel
replica
multikernel
replica
message passing
CPU
CPU
CPU
cache
cache
DRAM 0
DRAM 1
NRkernel
shared replica
NRkernel
shared replica
CPU
CPU
CPU
CPU
cache
cache
DRAM 0
log
DRAM 1

Outline

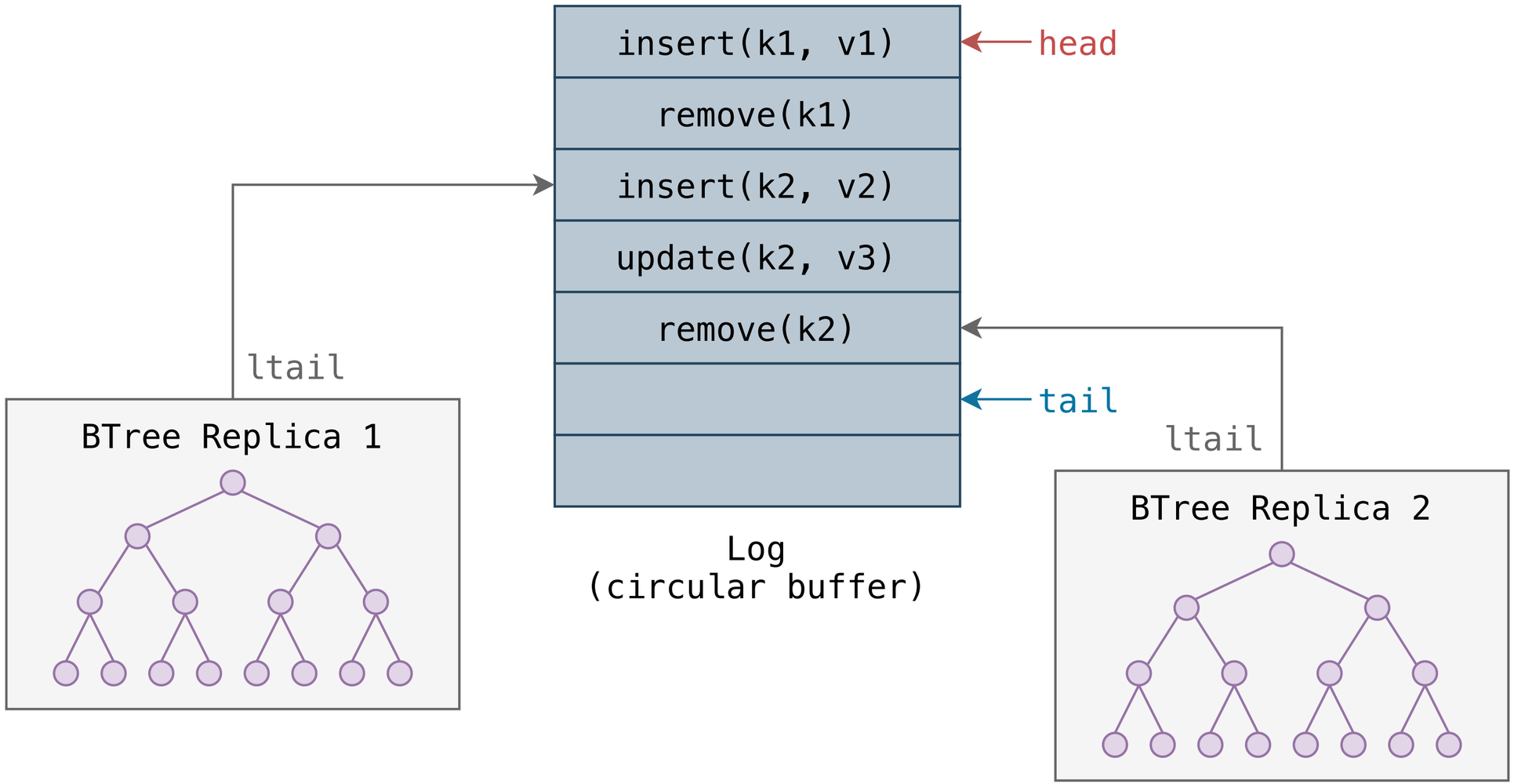
- Background & Overview
- Node Replication
- NrOS Design
- Evaluation
- Conclusion

Node Replication (NR)

Turn a sequential data struture into a linearizable NUMA-aware concurrent data structure.

- **Replica on each NUMA node**
- **Operation log**
- **Flat combining**
- **Optimized readers-writer lock**

Operation Log



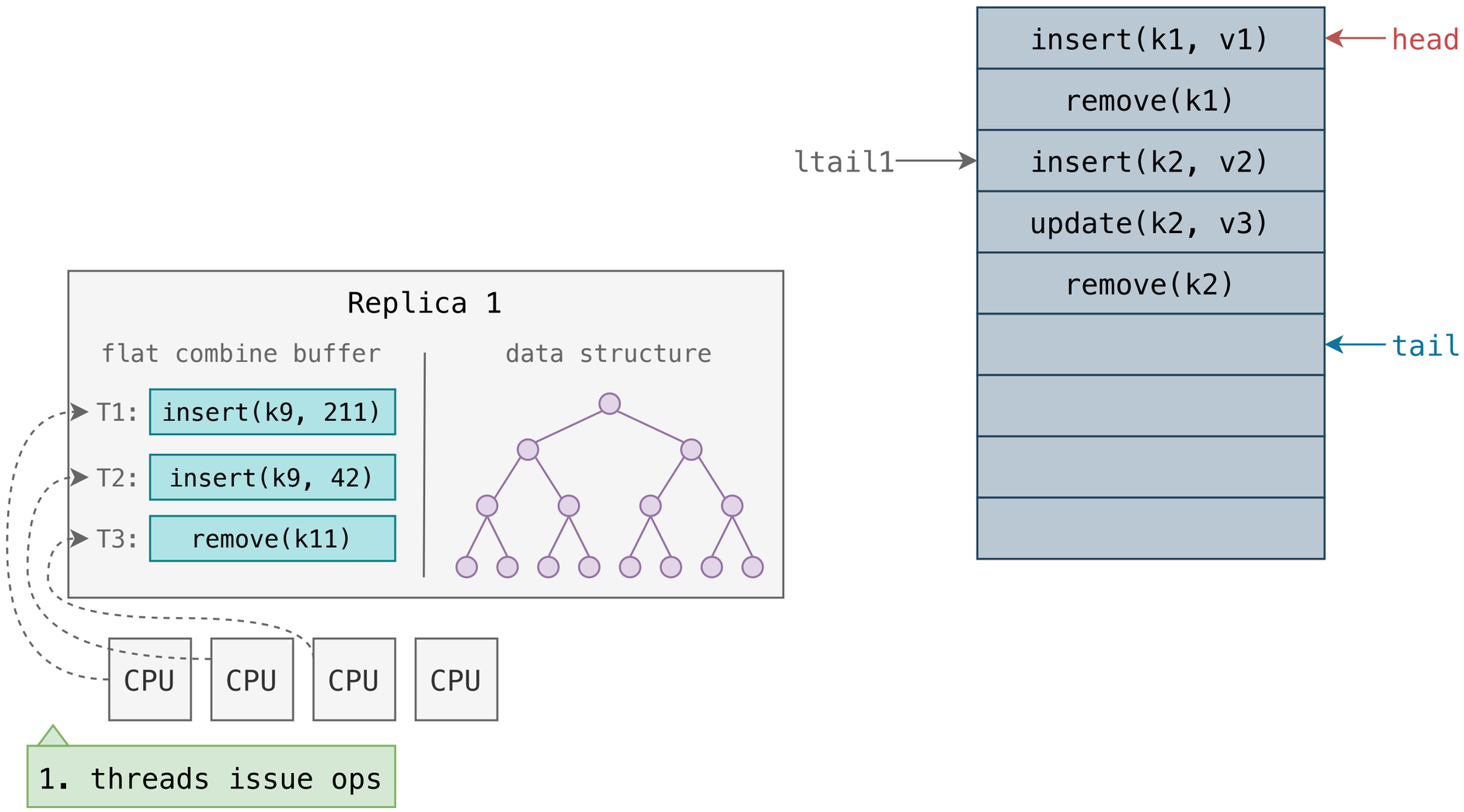
Flat Combining

- **One combiner per NUMA node**
  - batch appending operation logs
  - batch executing mutations
- **Lower cost, better cache locality**

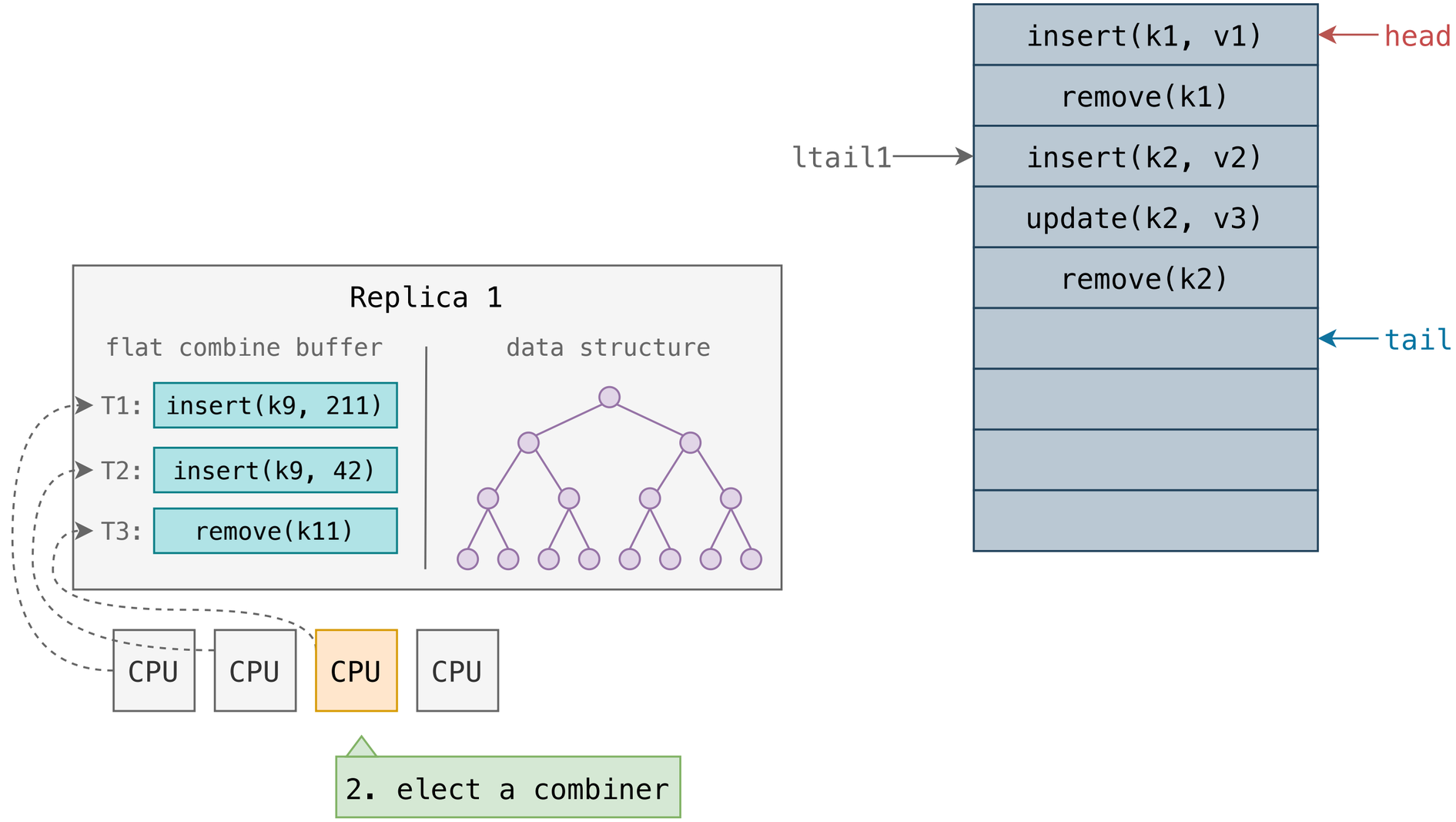
Optimized Readers-Writer Lock

- **Protect concurrent read/write on local NUMA node**
- **Writer-preference**
- **Split writer lock and combiner lock**
  - allow parallel reading and combining

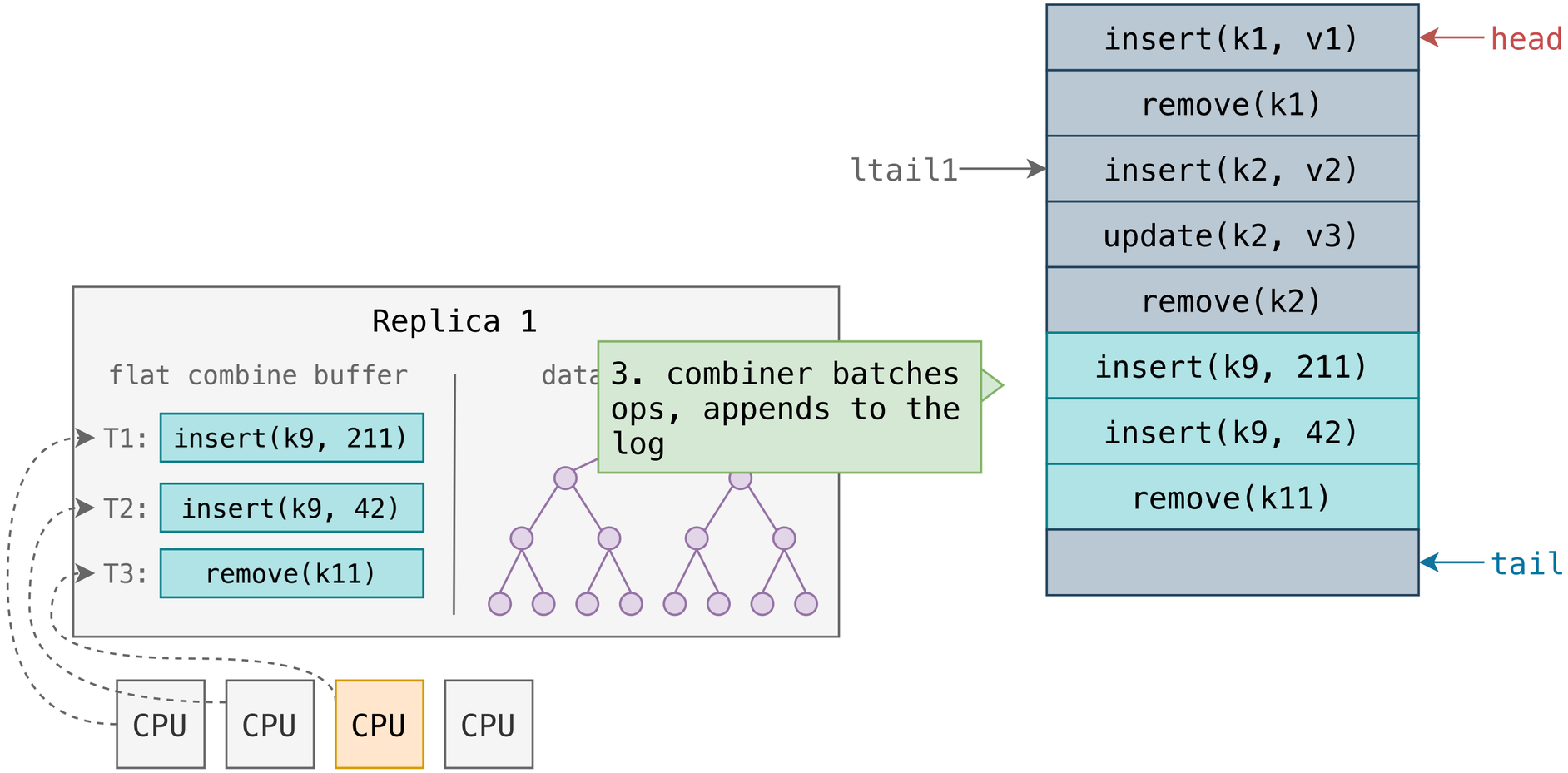
Example: Update (mutating)



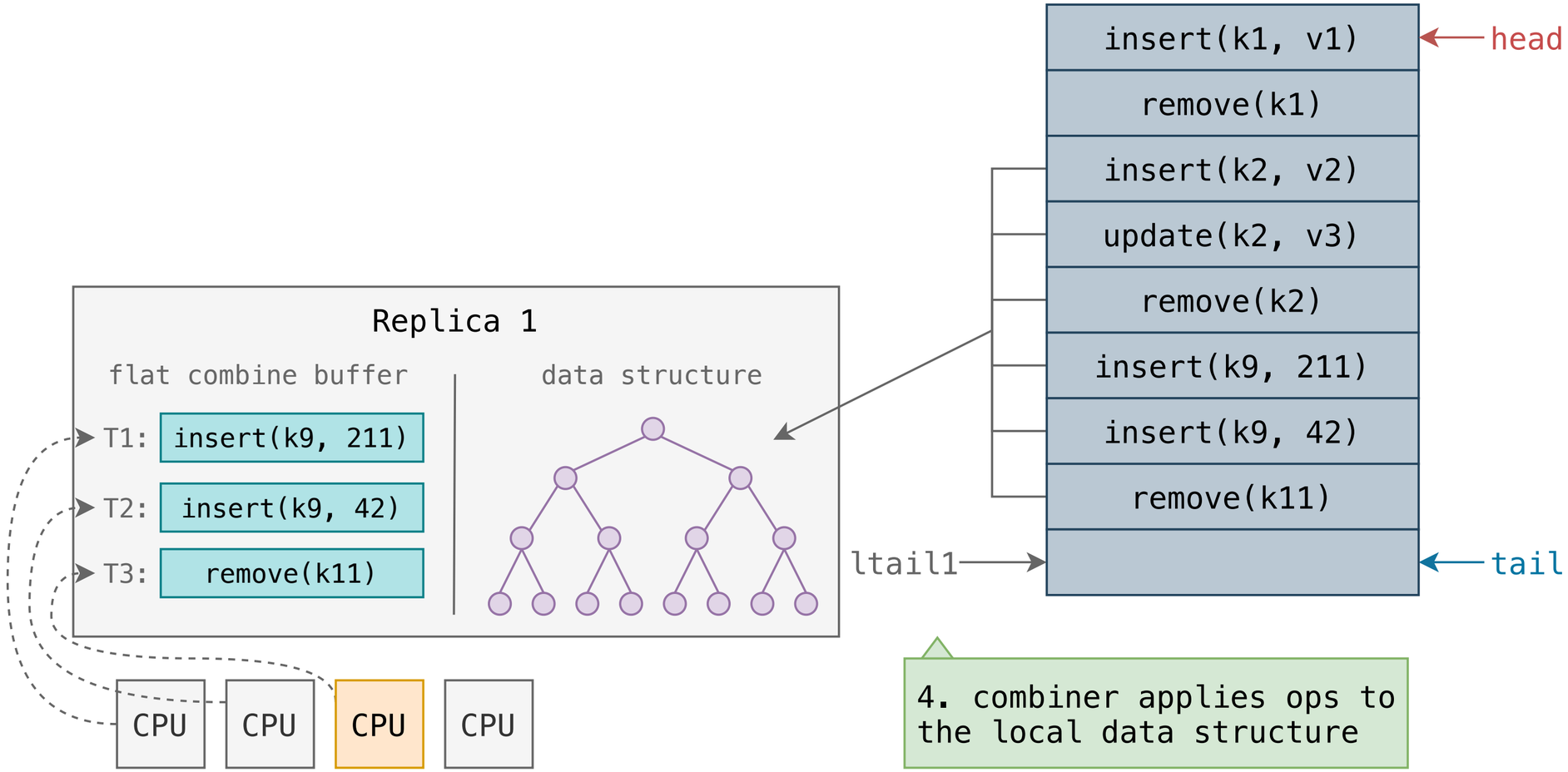
Example: Update (mutating)



Example: Update (mutating)



Example: Update (mutating)



Code Example: Replicate a HashMap

```
struct NrHashMap { storage: HashMap<u64, u64> }

enum WriteOp { Put(u64, u64) }
enum ReadOp { Get(u64) }

impl Dispatch for NrHashMap {
    type ReadOperation = ReadOp;
    type WriteOperation = WriteOp;
    type Response = Option<u64>;

    fn dispatch(&self, op: Self::ReadOperation) -> Self::Response {
        match op {
            ReadOp::Get(key) => self.storage.get(&key).map(|v| *v),
        }
    }
    fn dispatch_mut(&mut self, op: Self::WriteOperation) -> Self::Response {
        match op {
            WriteOp::Put(key, value) => self.storage.insert(key, value),
        }
    }
}
```

Code Example: Replicate a HashMap

```
struct NrHashMap { storage: HashMap<u64, u64> }

enum WriteOp { Put(u64, u64) }
enum ReadOp { Get(u64) }

impl Dispatch for NrHashMap {
    type ReadOperation = ReadOp;
    type WriteOperation = WriteOp;
    type Response = Option<u64>;

    fn dispatch(&self, op: Self::ReadOperation) -> Self::Response {
        match op {
            ReadOp::Get(key) => self.storage.get(&key).map(|v| *v),
        }
    }
    fn dispatch_mut(&mut self, op: Self::WriteOperation) -> Self::Response {
        match op {
            WriteOp::Put(key, value) => self.storage.insert(key, value),
        }
    }
}
```

Code Example: Replicate a HashMap

```
struct NrHashMap { storage: HashMap<u64, u64> }

enum WriteOp { Put(u64, u64) }
enum ReadOp { Get(u64) }

impl Dispatch for NrHashMap {
    type ReadOperation = ReadOp;
    type WriteOperation = WriteOp;
    type Response = Option<u64>;

    fn dispatch(&self, op: Self::ReadOperation) -> Self::Response {
        match op {
            ReadOp::Get(key) => self.storage.get(&key).map(|v| *v),
        }
    }
    fn dispatch_mut(&mut self, op: Self::WriteOperation) -> Self::Response {
        match op {
            WriteOp::Put(key, value) => self.storage.insert(key, value),
        }
    }
}
```

Code Example: Replicate a HashMap

```
struct NrHashMap { storage: HashMap<u64, u64> }

enum WriteOp { Put(u64, u64) }
enum ReadOp { Get(u64) }

impl Dispatch for NrHashMap {
    type ReadOperation = ReadOp;
    type WriteOperation = WriteOp;
    type Response = Option<u64>;

    fn dispatch(&self, op: Self::ReadOperation) -> Self::Response {
        match op {
            ReadOp::Get(key) => self.storage.get(&key).map(|v| *v),
        }
    }
    fn dispatch_mut(&mut self, op: Self::WriteOperation) -> Self::Response {
        match op {
            WriteOp::Put(key, value) => self.storage.insert(key, value),
        }
    }
}
```

Code Example: Replicate a HashMap

```
struct NrHashMap { storage: HashMap<u64, u64> }

enum WriteOp { Put(u64, u64) }
enum ReadOp { Get(u64) }

impl Dispatch for NrHashMap {
    type ReadOperation = ReadOp;
    type WriteOperation = WriteOp;
    type Response = Option<u64>;

    fn dispatch(&self, op: Self::ReadOperation) -> Self::Response {
        match op {
            ReadOp::Get(key) => self.storage.get(&key).map(|v| *v),
        }
    }
    fn dispatch_mut(&mut self, op: Self::WriteOperation) -> Self::Response {
        match op {
            WriteOp::Put(key, value) => self.storage.insert(key, value),
        }
    }
}
```

Code Example: Replicate a HashMap

```
struct NrHashMap { storage: HashMap<u64, u64> }

enum WriteOp { Put(u64, u64) }
enum ReadOp { Get(u64) }

impl Dispatch for NrHashMap {
    type ReadOperation = ReadOp;
    type WriteOperation = WriteOp;
    type Response = Option<u64>;

    fn dispatch(&self, op: Self::ReadOperation) -> Self::Response {
        match op {
            ReadOp::Get(key) => self.storage.get(&key).map(|v| *v),
        }
    }
    fn dispatch_mut(&mut self, op: Self::WriteOperation) -> Self::Response {
        match op {
            WriteOp::Put(key, value) => self.storage.insert(key, value),
        }
    }
}
```

Code Example: Replicate a HashMap

```
let logsize = 2 * 1024 * 1024;
let log = Log::<<NrHashMap as Dispatch>::WriteOperation>::new(logsize);

let replica1 = Replica::<NrHashMap>::new(&log);
let replica2 = Replica::<NrHashMap>::new(&log);

let tid1 = replica1.register();

let r = replica1.execute(ReadOp::Get(1), tid1);
let r = replica1.execute_mut(WriteOp::Put(1, 1), tid1);
```

Code Example: Replicate a HashMap

```
let logsize = 2 * 1024 * 1024;
let log = Log::<<NrHashMap as Dispatch>::WriteOperation>::new(logsize);

let replica1 = Replica::<NrHashMap>::new(&log);
let replica2 = Replica::<NrHashMap>::new(&log);

let tid1 = replica1.register();

let r = replica1.execute(ReadOp::Get(1), tid1);
let r = replica1.execute_mut(WriteOp::Put(1, 1), tid1);
```

Code Example: Replicate a HashMap

```
let logsize = 2 * 1024 * 1024;
let log = Log::<<NrHashMap as Dispatch>::WriteOperation>::new(logsize);

let replica1 = Replica::<NrHashMap>::new(&log);
let replica2 = Replica::<NrHashMap>::new(&log);

let tid1 = replica1.register();

let r = replica1.execute(ReadOp::Get(1), tid1);
let r = replica1.execute_mut(WriteOp::Put(1, 1), tid1);
```

Code Example: Replicate a HashMap

```
let logsize = 2 * 1024 * 1024;
let log = Log::<<NrHashMap as Dispatch>::WriteOperation>::new(logsize);

let replica1 = Replica::<NrHashMap>::new(&log);
let replica2 = Replica::<NrHashMap>::new(&log);

let tid1 = replica1.register();

let r = replica1.execute(ReadOp::Get(1), tid1);
let r = replica1.execute_mut(WriteOp::Put(1, 1), tid1);
```

Code Example: Replicate a HashMap

```
let logsize = 2 * 1024 * 1024;
let log = Log::<<NrHashMap as Dispatch>::WriteOperation>::new(logsize);

let replica1 = Replica::<NrHashMap>::new(&log);
let replica2 = Replica::<NrHashMap>::new(&log);

let tid1 = replica1.register();

let r = replica1.execute(ReadOp::Get(1), tid1);
let r = replica1.execute_mut(WriteOp::Put(1, 1), tid1);
```

Code Example: Replicate a HashMap

```
let logsize = 2 * 1024 * 1024;
let log = Log::<<NrHashMap as Dispatch>::WriteOperation>::new(logsize);

let replica1 = Replica::<NrHashMap>::new(&log);
let replica2 = Replica::<NrHashMap>::new(&log);

let tid1 = replica1.register();

let r = replica1.execute(ReadOp::Get(1), tid1);
let r = replica1.execute_mut(WriteOp::Put(1, 1), tid1);
```

Problems with NR

- **Frequent mutating operations (e.g. FS)**
  - mostly independent (commutative)
- **Limit scalability**

Concurrent Node Replication (CNR)

Turn an *already concurrent data structure* into a NUMA-aware concurrent data structure.

- **Multiple logs**
  - assign commutative operations to different logs
  - assign conflicting operations to the same log
- **Multiple combiners per NUMA node**
  - concurrently append and apply operation logs

Outline

- Background & Overview
- Node Replication
- NrOS Design
- Evaluation
- Conclusion

NrOS

- **Designed around per-NUMA node kernel replicas**
- **Major subsystems**
  - NR-vMem: virtual memory management
  - NR-FS: in-memory file system
  - NR-Scheduler: process management

NR-vMem

- **Per-process mapping B-Tree & hardware PT are replicated**
- **Mutating Ops:** `Map`, `Unmap`, `Adjust`
- **Non-mutating Ops:** `Resolve`
- **Problem: Out-of-band read by hardware**
  - `Map`: page fault → `Resolve`
  - `Unmap` & `Adjust`: IPI → update replica → TLB flush

NR-FS

- **Entire in-memory FS data structure is replicated**
- **Problem: POSIX read operations mutate kernel state (e.g. fd offset)**
  - only implement `pread`/`pwrite` in kernel
  - fd offset in userspace lib
- **Problem: Large amound data in operation log**
  - allocate kernel buffer, put only references in log

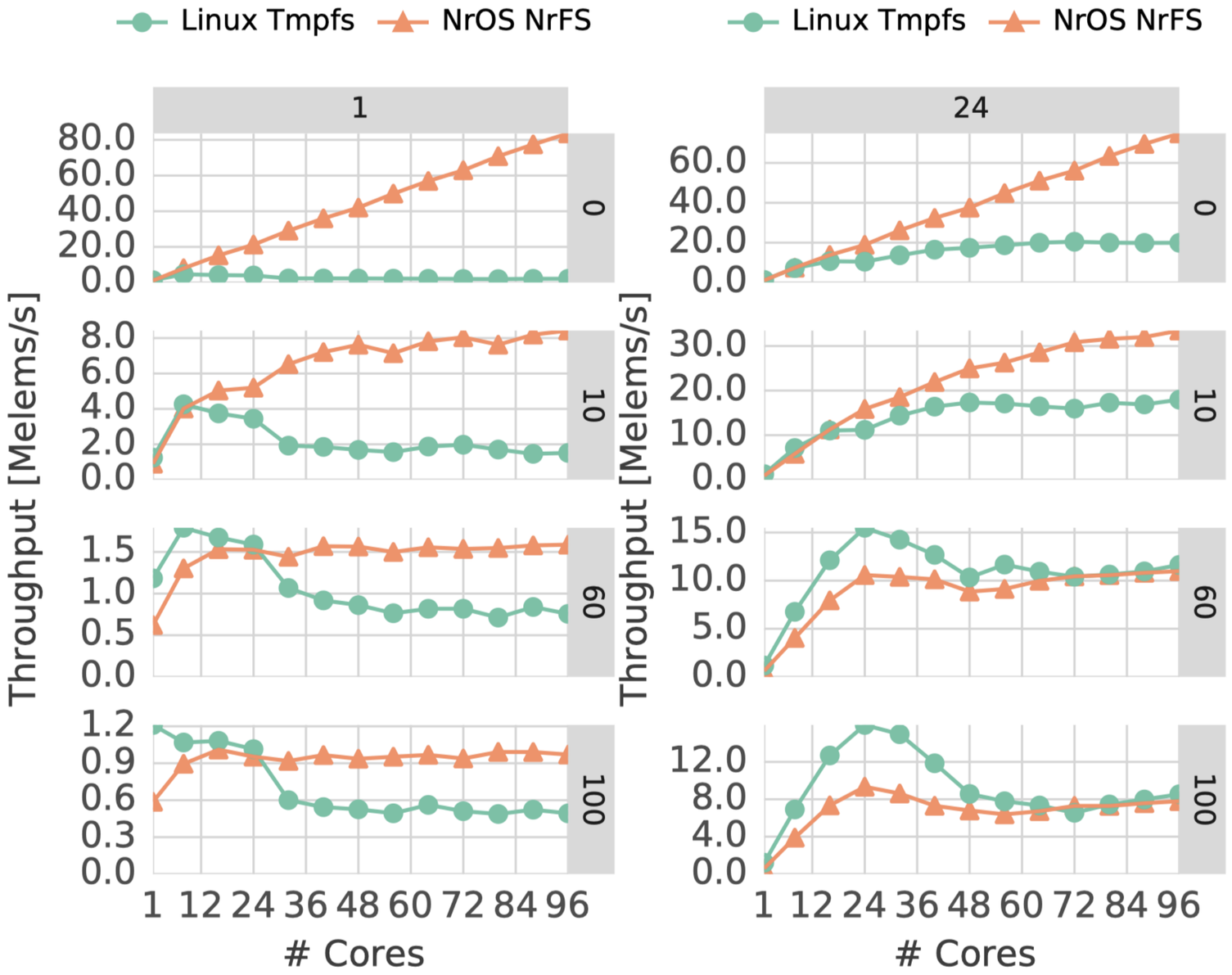
Outline

- Background & Overview
- Node Replication
- NrOS Design
- Evaluation
- Conclusion

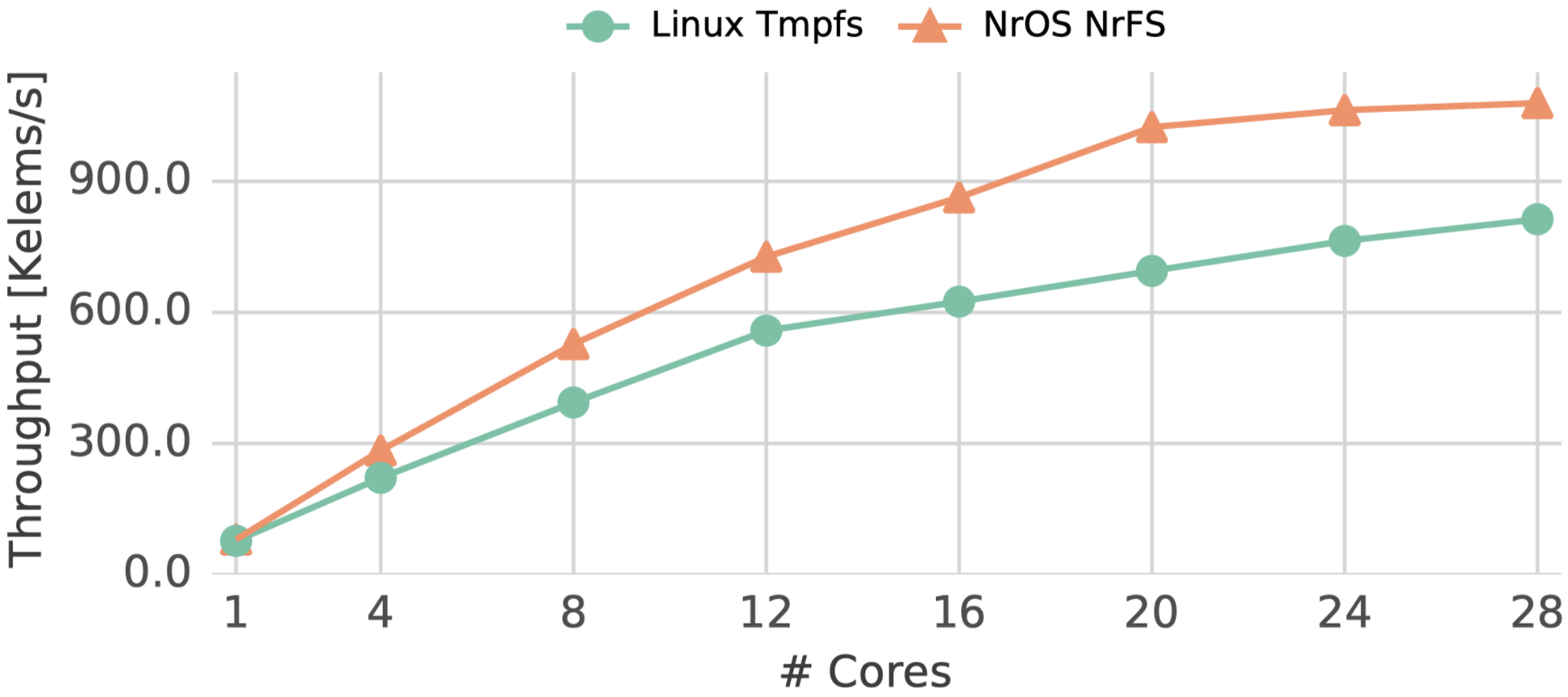
Evaluation Platforms

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

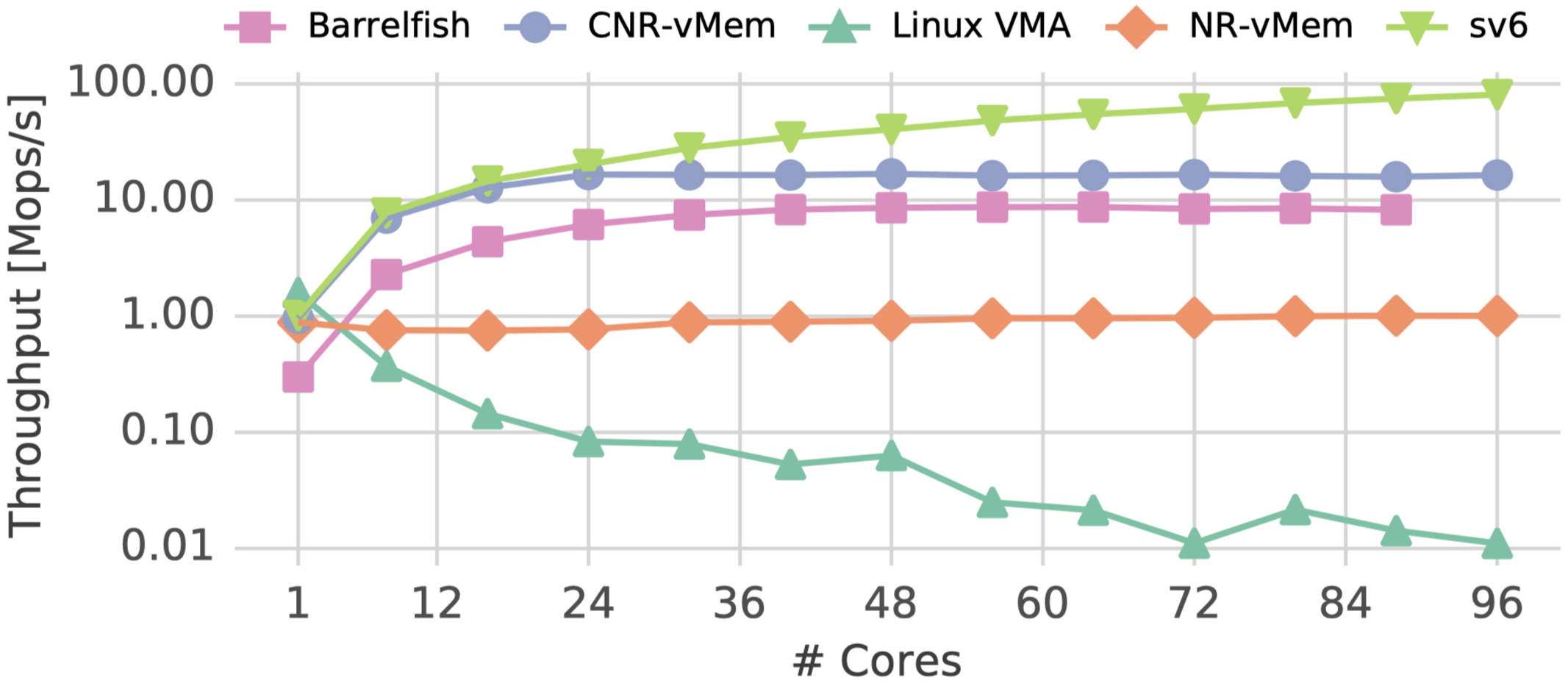
NR-FS Microbenchmark



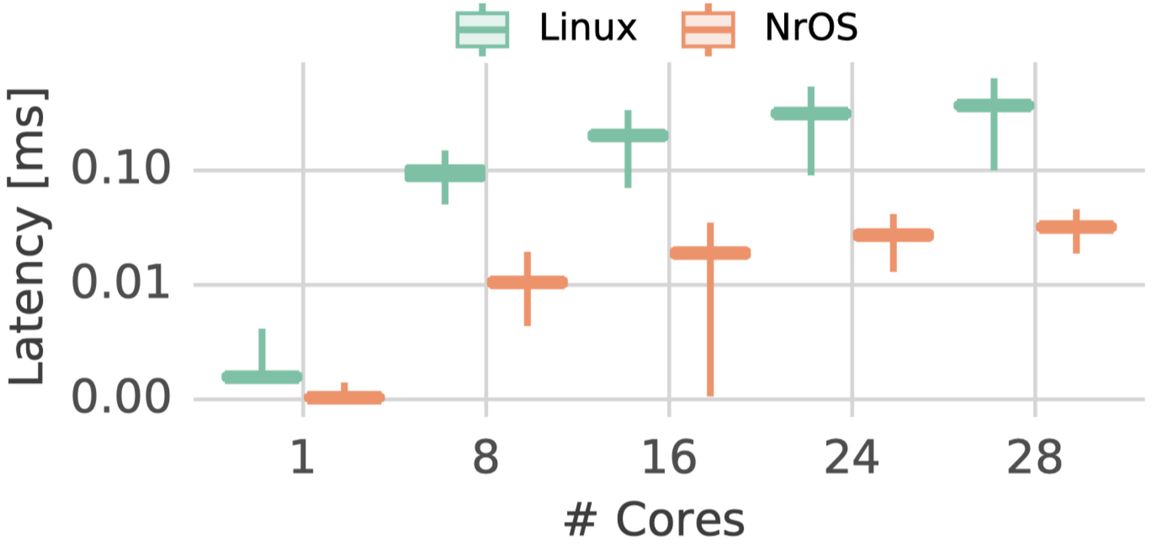
NR-FS LevelDB



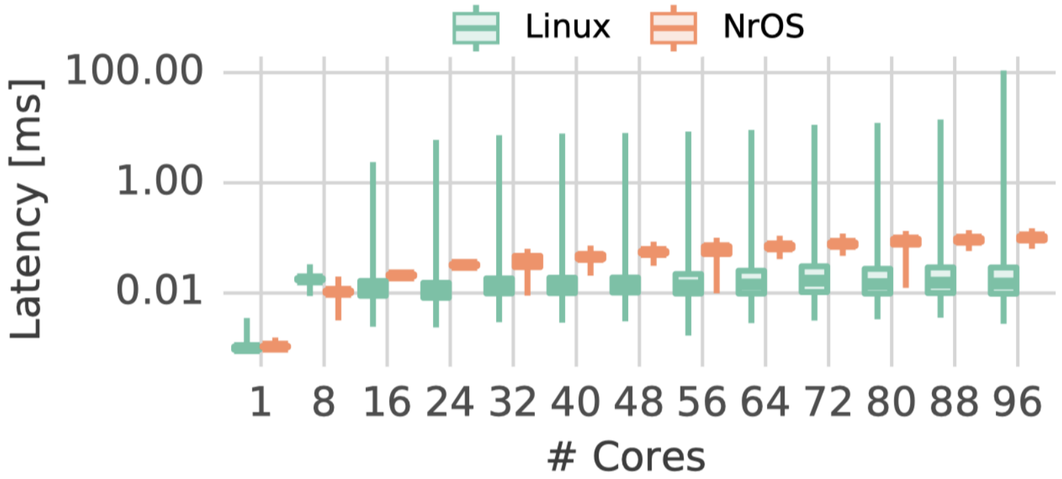
NR-vMem Map Throughput



NR-vMem Map Latency

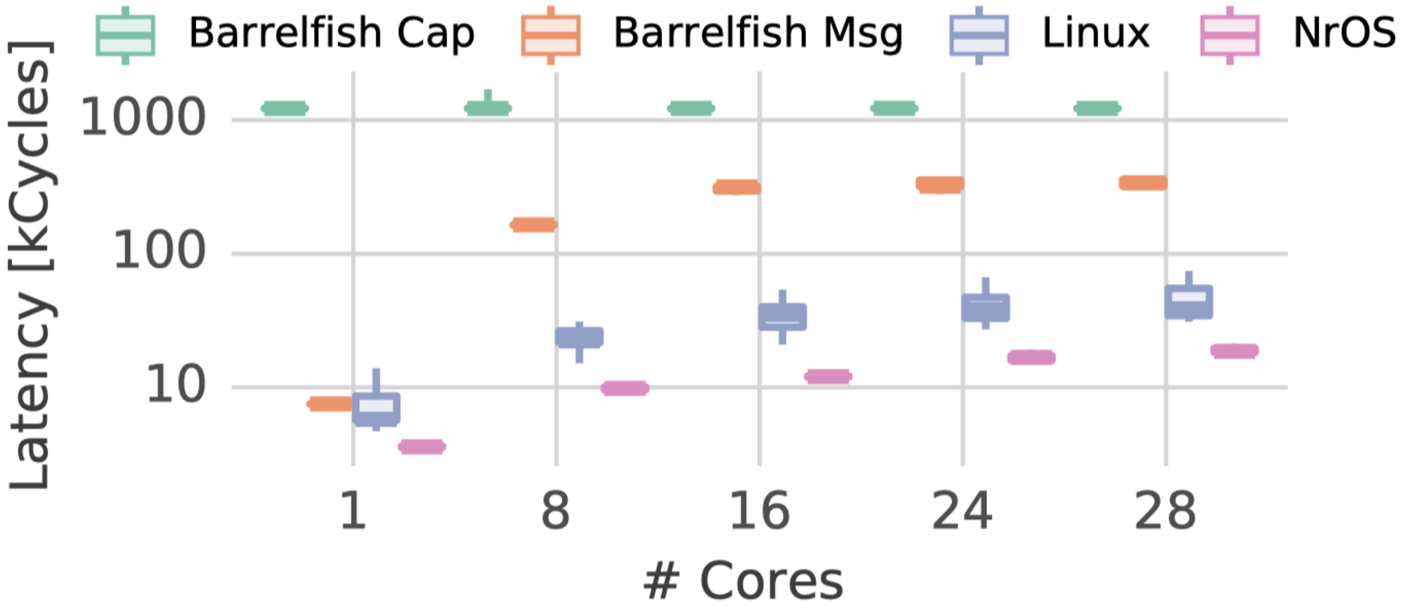


(a) Map latency for 2×14 Skylake.



(b) Map latency for 4×24 Cascade.

NR-vMem Unmap Latency



(c) Unmap latency on 2×14 Skylake.

Outline

- Background & Overview
- Node Replication
- NrOS Design
- Evaluation
- Conclusion

NRkernel Principles

- Combining replicated and shared state
- Replica consistency via operation log
- Compiler-enforced memory and concurrency safety

Future Directions

- **Relaxing consistency**
- **Verifying correctness**
- **Extending NrOS for compute clusters**

Thank you!