Node Replicated Kernel?

Node Replicated Kernel (NRK) is a research prototype OS
Research, but is now being developed collaboratively by a
and academia. It is intended as a basis to explore ideas al
systems for hardware of the future. NRK is written from s
assembly), and it runs on x86 platforms.

How is NRK different from Linux, W

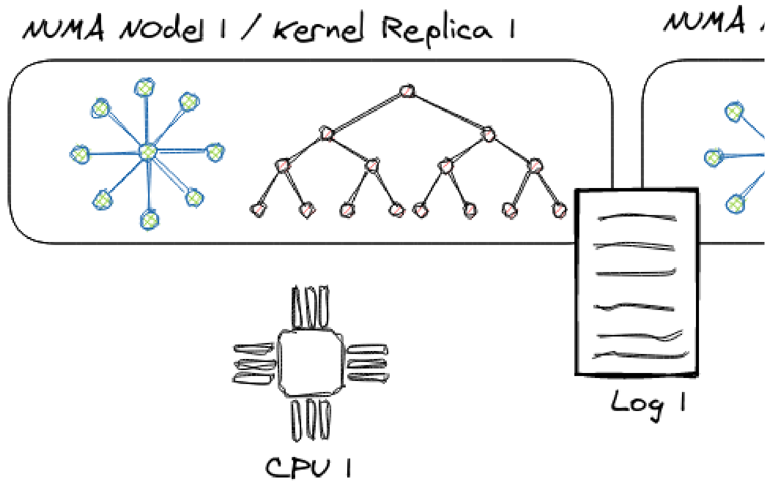
At present, NRK lacks many of the features of an operatin
anyone other than systems researchers taking measurem
example, there is currently no GUI or Shell, and only very
it's probably easier to compare NRK in it's current form to
a fully-featured OS.

From an architectural point of view, the NRK kernel is a ne
multi-kernel OS:

> A multikernel operating system treats a multi-core mac
> cores, as if it were a distributed system. It does not ass
> implements inter-process communications as message

Unfortunately, such a model also brings unnecessary com
multi-kernel (Barrelfish) relied on per-core communicatio
(2PC, 1PC) to achieve agreement, replication and sharing

We overcome this complexity in NRK by using logs: The ke
threaded data structures which are automatically replicat
logs make sure the replicas are always synchronized. Our
bears resemblance to state machine replication in distrib
threaded data-structures into linearizable, concurrent str
details on scenarios where this approach can heavily outp
lock-based or lock-free data-structures.



Schematic view of the NRK kernel. Core kernel data-
system with a log.

In user-space, NRK runs applications in ring 3: Each applic
isolated process/container/lightweight VM with little oppo
processes. To run existing, well-known applications like m
process can link against rumpkernel, which instantiates a
user-space libraries (libc, libpthread etc.) within the proce
decent support for POSIX.

Finally, NRK is written from scratch in Rust: We take advan
with a rich type-system for OS implementation, to gain be
guarantees at compile time, while not impacting perform

Kernel Architecture

The NRK kernel is a small, light-weight (multi-)kernel that
virtual memory, a coarse-grained scheduler, as well as an

One key feature of the kernel is how it scales to many cor
data-structure replication with operation logging. We exp
for this in the Node Replication and Concurrent Node Rep

Node Replication (NR)

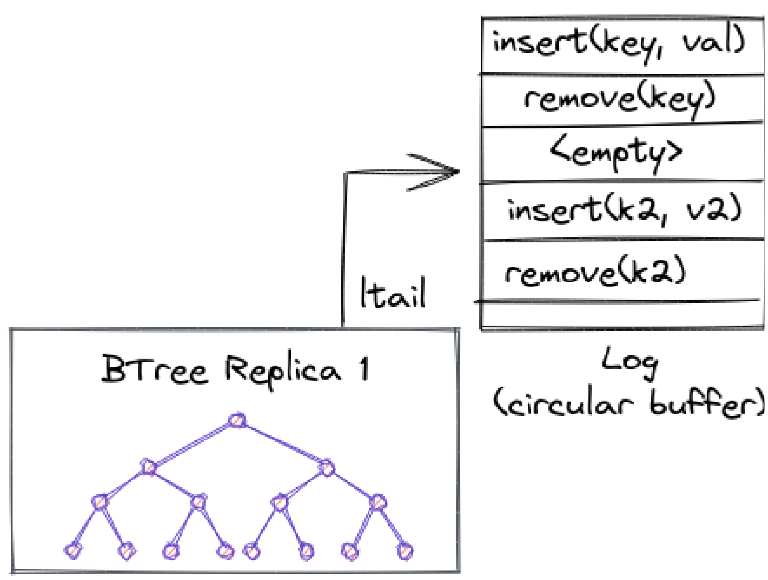
NR creates linearizable NUMA-aware concurrent data stru
data structures. NR replicates the sequential data structu
operation log to maintain consistency between the replica
concurrency using a readers-writer lock and from write co
flat combining. In a nutshell, flat combining batches opera
executed by a single thread (*the combiner*). This thread als
the log; other replicas read the log to update their interna

Next, we explain in more detail the three main technique
logs, scalable reader-writer locks and flat combining.

The operation log

The log uses a circular buffer to represent the abstract sta
Each entry in the log represents a mutating operation, an
log tail gives the index to the last operation added to the l

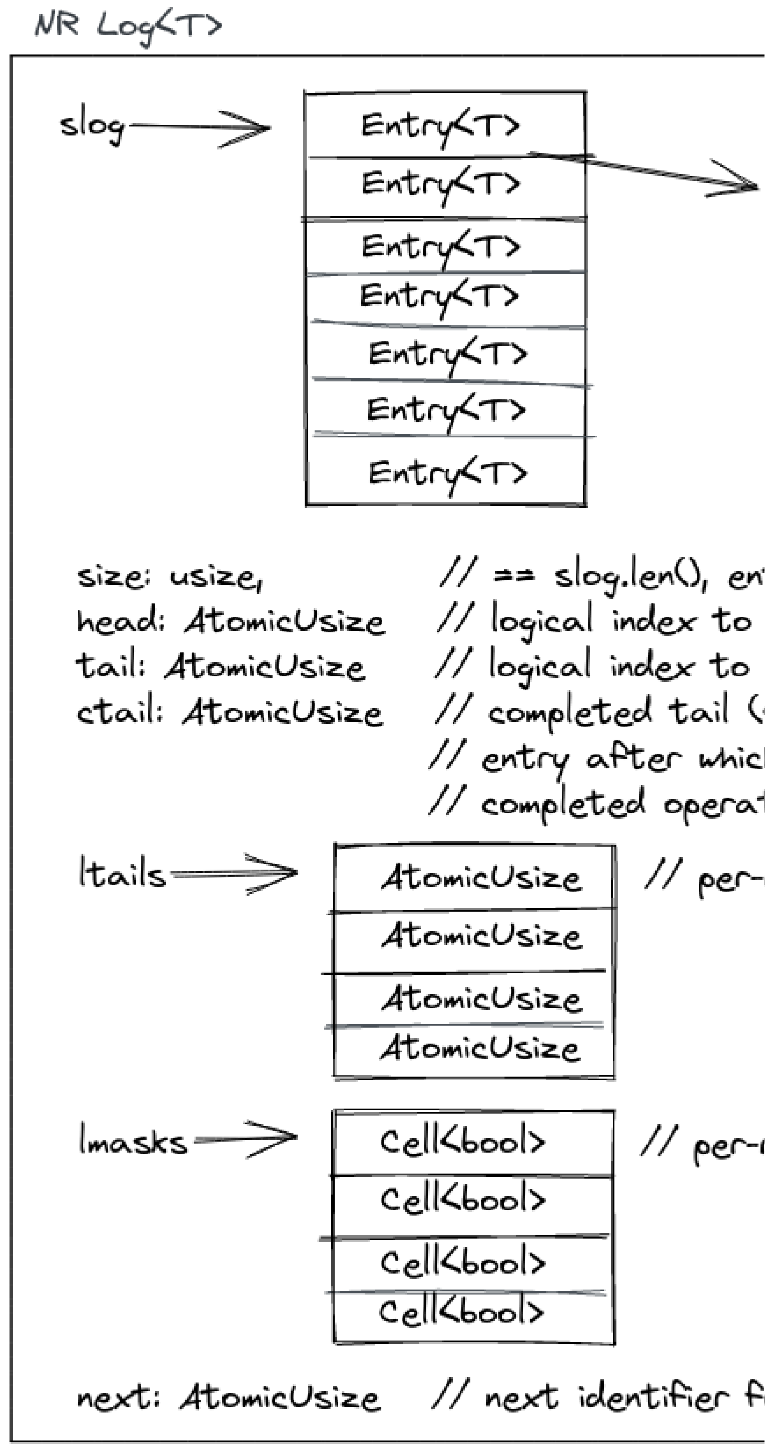
A replica can lazily consume the log by maintaining a per-
how far from the log the replica has been updated. The lo
so that entries can be garbage collected and reused. NR c
have not been applied on each replica. This means at leas
must be executing operations on the data structure, othe
replicas would not be able to make any more progress.



Shows a log containing mutable operations for a 2x contains a local tail (ltail1 and ltail2) pointer into the oldest entry that is still outstanding (i.e., needs to be replica before it can be overwritten).

Next, we're going to explain the log in more detail for the directly working on this, likely these subsections are prett

Log memory layout



The log structure and it's members as they appear i

The log contains an array or slice of entries, `slog`, where the replicas. It also maintains a global `head` and `tail` w indicate the current operations in the log that still need to or have not been garbage collected by the log. The `head` still needs to be processed, and the `tail` to the newest. which tracks per-replica progress. At any point in time, an subregion given by `head..tail`. `lmasks` are generation replica wraps around in the circular buffer.

`next` is used internally by the library to hand out a regist
with the log (this is done only in the beginning, during set
consuming from/appending to the log.

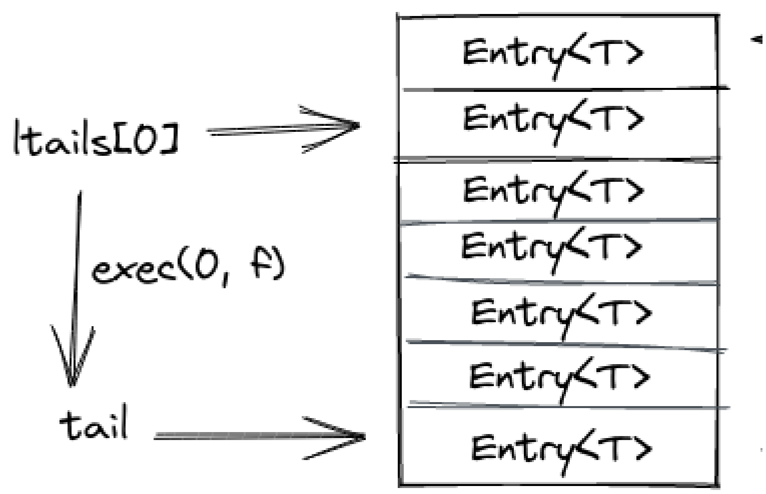
The `ctail` is an optimization for reads and tracks the ma
log entry (`< tail`) after which there are no completed op

Consuming from the log

The `exec` method is used by replicas to execute any outs
takes a closure that is executed for each outstanding entr
that calls `exec` .

Approximately, the method will do two things:

1. Invoke the closure for every entry in the log which is
   afterwards sets `ltail = tail` .
2. Finally, it may update the `ctail` if `ltail > ctail` t



Indicates what happens to ltail[0] and ctail when rep
be updated to be equal to tail and the ctail (currentl
is ltail[1] will point to a new max which is the same a
also applies all the entries between ltail[0] and tail a
replica 0.

Appending to the log

The append operation is invoked by replicas to insert mu
slice of new operations coming from the replica and the s
apply some outstanding operations. This is necessary as t

insert more operations. Then, we'd first have to update o
replicas to make progress).

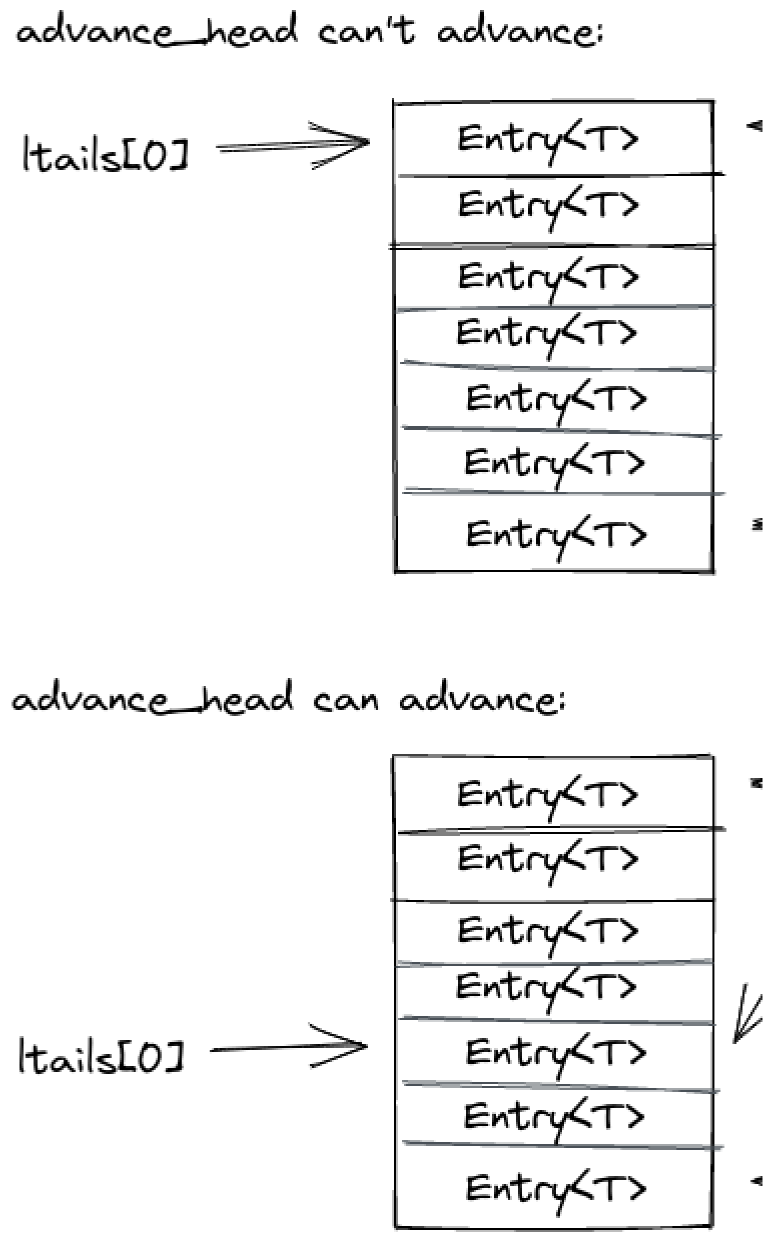
Approximately, the method will end up doing these steps

1. If there isn't enough space to insert our new batch c
   entries from the log until we have enough space ava
2. Do a compare exchange on the log `tail` to reserve
3. Insert entries into the log
4. See if we can collect some garbage aka old log entri

Garbage collecting the log

`advance_head` has the unfortunate job to collect the log g
little as periodically advancing the head pointer.

For that it computes the minium of all `ltails`. If this min
head, it waits (and calls `exec` to try and advance its local
the head, it will update the head to that new minimum lta



Shows the two possible scenarios in advance head: replica 0 points to the current head, so we can't adv to wait. In the bottom diagram, both ltail[0] and ltail head, so we don't need the entries between head..lt advanced.

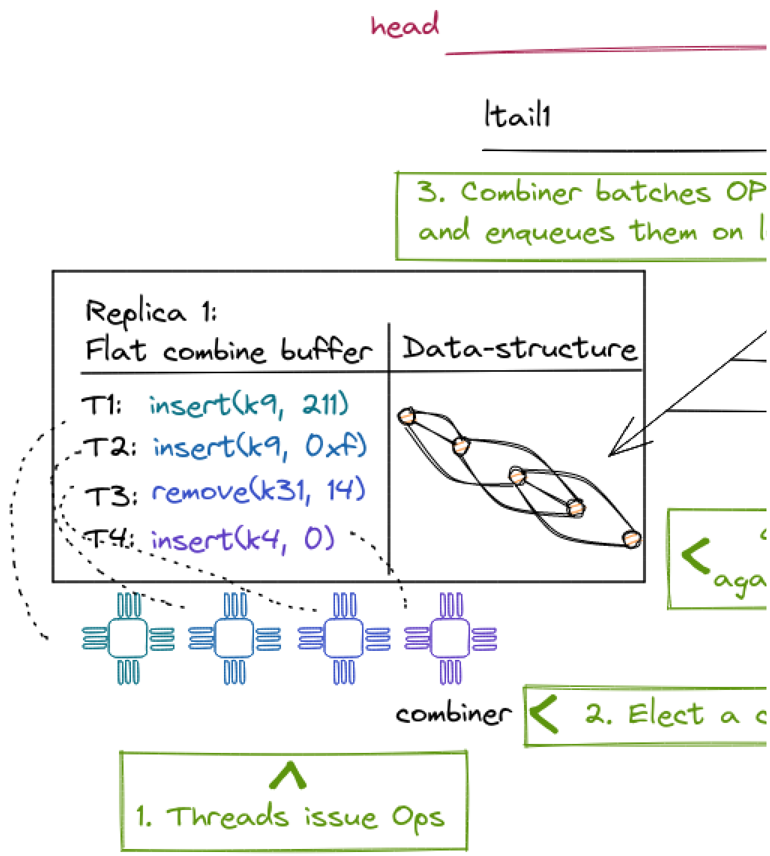
Flat combining

NR uses [flat combining](#) to allow threads running on the sa resulting in better cache locality both from flat combining local to the node's last-level cache.

1. The combiner can batch and combine multiple oper
cost than executing each operation individually. For
priority queue can be done with a single atomic inst
one atomic instruction for each operation.

2. A single thread accesses the data structure for multi
cache locality for the whole batch of operations.

NR also benefits from combining operations placed on th
single atomic operation for the whole batch instead of on



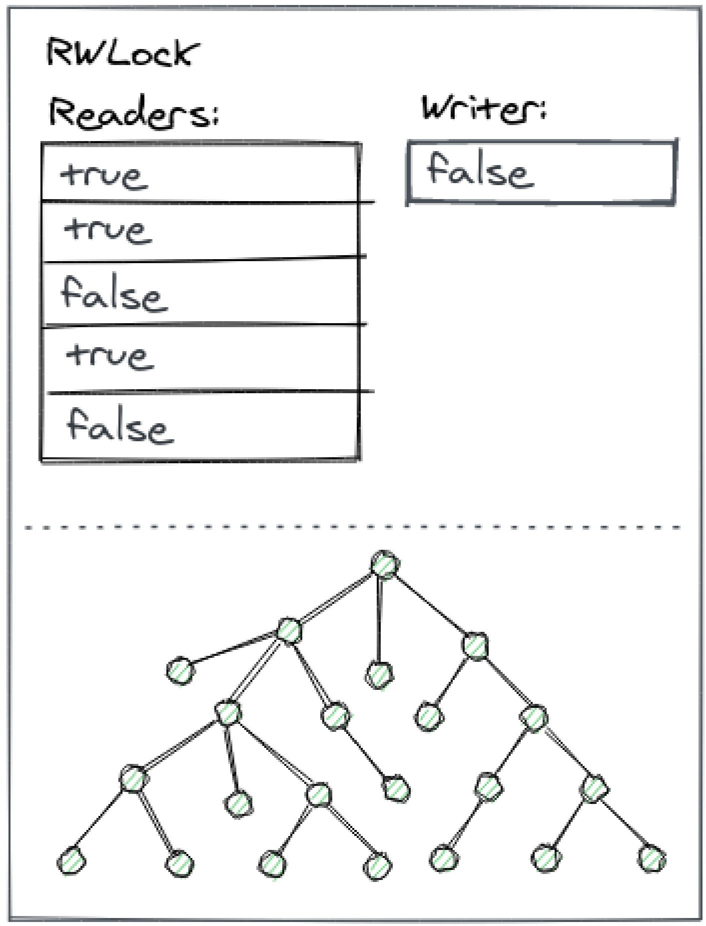
Flat combining multiple operations from threads on
thread is elected who will aggregate outstanding op
put them on the log in one batch (3) and then execu
(4), after executing any previously outstanding opera

The optimized readers-writer lock

NR uses a writer-preference variant of the distributed RW
synchronization of the combiner and reader threads whe

Each reader acquires a local per-thread lock; the writer ha
readers its intent to acquire the writer lock. A reader first
acquires its local lock if there is no writer; then it checks tl
lock) and releases the reader lock if it notices the writer d
(writer preference). The combiner also acquires the writer
Thus, we give highest priority to the writer, because we do
This lock allows readers to read a local replica while the c
operations to the log, increasing parallelism.

If there is no combiner, a reader might have to acquire th
from the log to avoid a stale read, but this situation is rare



The RWLock used to protect the data-structure in ev
cache-line sized atomic booleans for reader threads
thread. It allows concurrent readers but only a single
Readers also have to make sure to advance the repl
the log.

A concurrent non-mutating operati

A non-mutation operation (read) can execute on its threa
entry, but it needs to ensure that the replica is not stale. T
tail when the operation begins, and waits until a combine
observed tail, or acquires the lock and updates the replica

If there is no combiner, the read thread will acquire the w

A concurrent mutating operation (u

A thread T executing a mutating operation (update) needs
thread T executing this operation fails to acquire it, anoth
spin-waits to receive its operation's result from the existin

However, T must still periodically attempt to become the
missed T's operation.

If T acquires the lock, it becomes *the combiner* and execut
concurrent threads on the same replica. To do so, T appe
with a single atomic operation and ensures that the replic
operations and returning their results to the waiting threa

Source and code example

Node-replication (NR) is released as a stand-alone library.

To give an idea, here is an example that transforms an sin
Rust standard library) into a concurrent, replicated hash-t

```
//! A minimal example that implements a replicatec
use std::collections::HashMap;
use node_replication::Dispatch;
use node_replication::Log;
use node_replication::Replica;

/// The node-replicated hashmap uses a std hashmap
#[derive(Default)]
struct NrHashMap {
    storage: HashMap<u64, u64>,
}

/// We support mutable put operation on the hashma
#[derive(Clone, Debug, PartialEq)]
enum Modify {
    Put(u64, u64),
}

/// We support an immutable read operation to look
#[derive(Clone, Debug, PartialEq)]
enum Access {
    Get(u64),
}

/// The Dispatch traits executes `ReadOperation` (
/// and `WriteOperation` (our `Modify` enum) again
/// data-structure.
impl Dispatch for NrHashMap {
    type ReadOperation = Access;
    type WriteOperation = Modify;
    type Response = Option<u64>;

    /// The `dispatch` function applies the immuta
    fn dispatch(&self, op: Self::ReadOperation) ->
        match op {
            Access::Get(key) => self.storage.get(&
        }
    }

    /// The `dispatch_mut` function applies the mu
    fn dispatch_mut(&mut self, op: Self::WriteOper
        match op {
            Modify::Put(key, value) => self.storag
        }
    }
}
```

As we can see we need to define two operation enums (`M` end up on the log, and `Access` for immutable/read opera implement the `Dispatch` trait from NR for our newly defi responsible to route the operations defined in `Access` ar

structure. The full example, including how to create replic
repository.

Node Replication (NR)

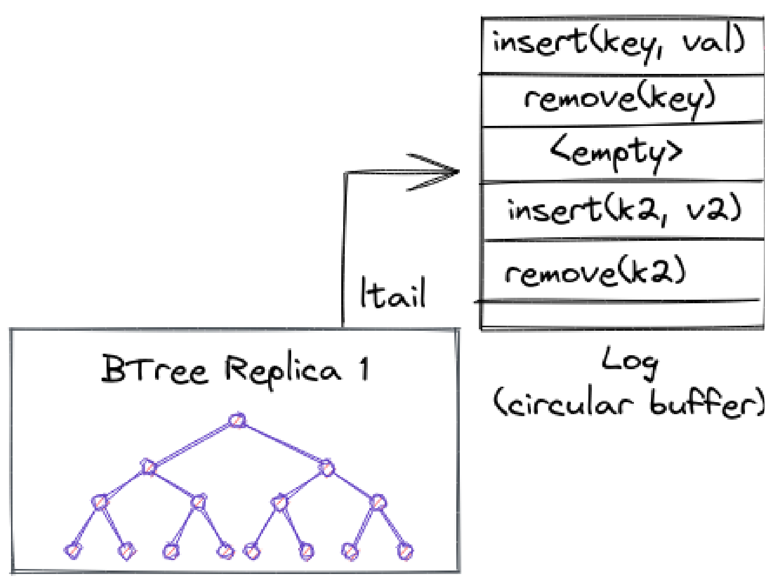
NR creates linearizable NUMA-aware concurrent data stru data structures. NR replicates the sequential data structu operation log to maintain consistency between the replica concurrency using a readers-writer lock and from write co *flat combining*. In a nutshell, flat combining batches opera executed by a single thread (*the combiner*). This thread als the log; other replicas read the log to update their interna

Next, we explain in more detail the three main technique logs, scalable reader-writer locks and flat combining.

The operation log

The log uses a circular buffer to represent the abstract sta Each entry in the log represents a mutating operation, an *log tail* gives the index to the last operation added to the l

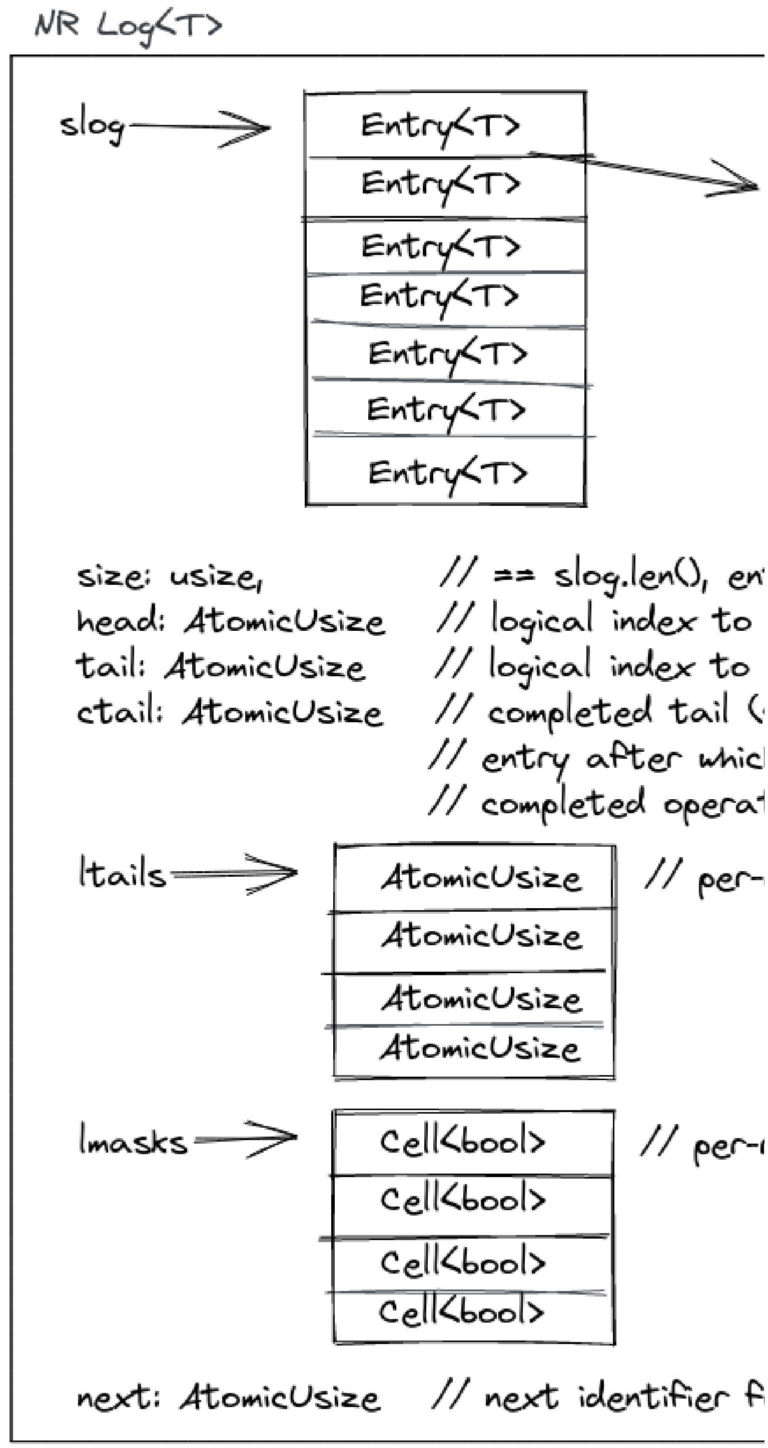
A replica can lazily consume the log by maintaining a per- how far from the log the replica has been updated. The lo so that entries can be garbage collected and reused. NR c have not been applied on each replica. This means at leas must be executing operations on the data structure, othe replicas would not be able to make any more progress.



Shows a log containing mutable operations for a 2x
contains a local tail (ltail1 and ltail2) pointer into the
oldest entry that is still outstanding (i.e., needs to be
replica before it can be overwritten).

Next, we're going to explain the log in more detail for the
directly working on this, likely these subsections are prett

Log memory layout



The log structure and it's members as they appear i

The log contains an array or slice of entries, `slog`, where the replicas. It also maintains a global `head` and `tail` w indicate the current operations in the log that still need to or have not been garbage collected by the log. The `head` still needs to be processed, and the `tail` to the newest. which tracks per-replica progress. At any point in time, an subregion given by `head..tail`. `lmasks` are generation replica wraps around in the circular buffer.

`next` is used internally by the library to hand out a regist
with the log (this is done only in the beginning, during set
consuming from/appending to the log.

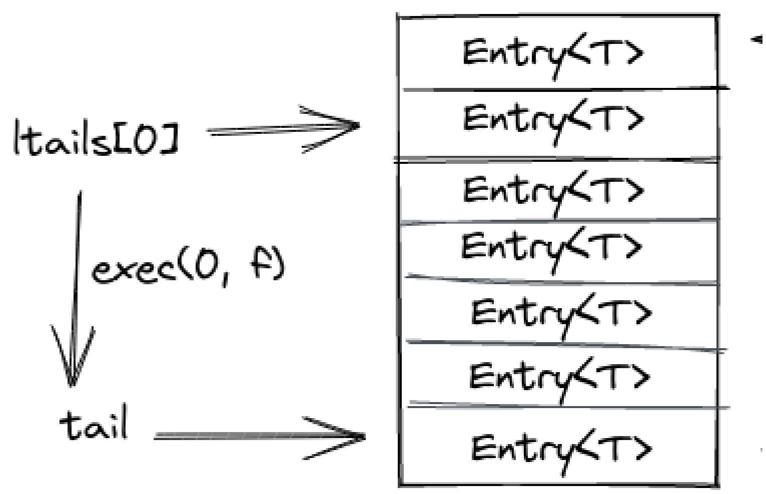
The `ctail` is an optimization for reads and tracks the ma
log entry (`< tail`) after which there are no completed op

Consuming from the log

The `exec` method is used by replicas to execute any outs
takes a closure that is executed for each outstanding entr
that calls `exec` .

Approximately, the method will do two things:

1. Invoke the closure for every entry in the log which is
   afterwards sets `ltail = tail` .
2. Finally, it may update the `ctail` if `ltail > ctail` t



Indicates what happens to ltail[0] and ctail when rep
be updated to be equal to tail and the ctail (currentl
is ltail[1] will point to a new max which is the same a
also applies all the entries between ltail[0] and tail a
replica 0.

Appending to the log

The append operation is invoked by replicas to insert mu
slice of new operations coming from the replica and the s
apply some outstanding operations. This is necessary as t

insert more operations. Then, we'd first have to update o
replicas to make progress).

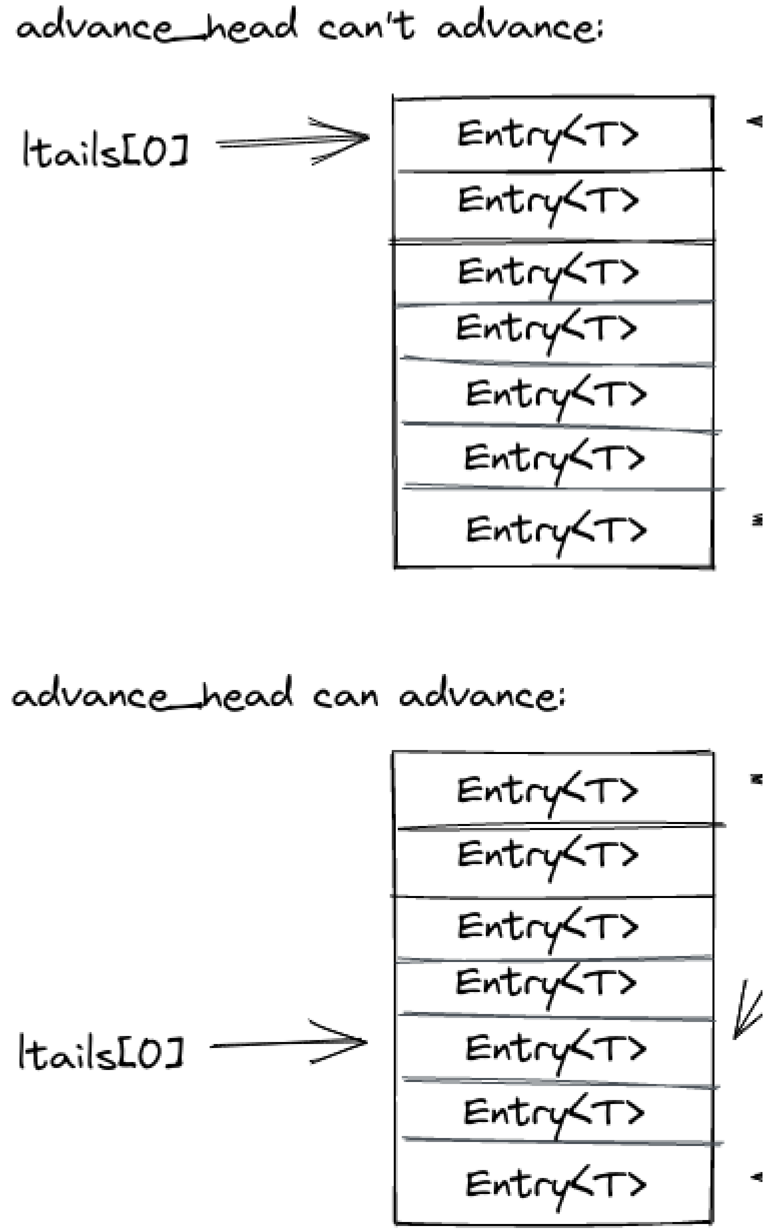
Approximately, the method will end up doing these steps

1. If there isn't enough space to insert our new batch c
   entries from the log until we have enough space ava
2. Do a compare exchange on the log `tail` to reserve
3. Insert entries into the log
4. See if we can collect some garbage aka old log entrie

Garbage collecting the log

`advance_head` has the unfortunate job to collect the log g
little as periodically advancing the head pointer.

For that it computes the minium of all `ltails`. If this min
head, it waits (and calls `exec` to try and advance its local
the head, it will update the head to that new minimum lta



Shows the two possible scenarios in advance head: replica 0 points to the current head, so we can't adv to wait. In the bottom diagram, both ltail[0] and ltail head, so we don't need the entries between head..lt advanced.

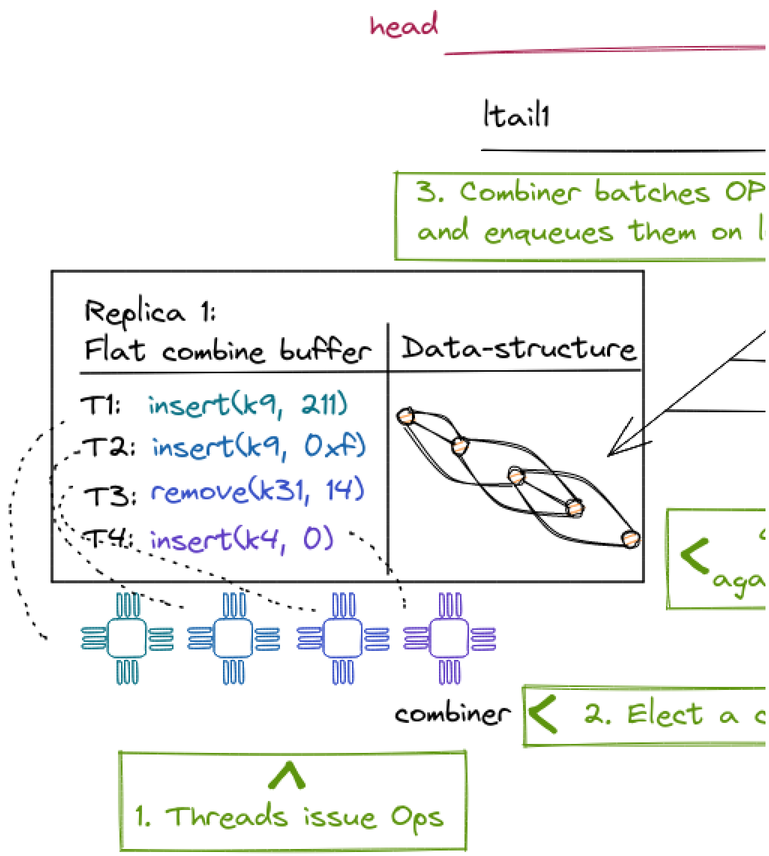
Flat combining

NR uses flat combining to allow threads running on the sa resulting in better cache locality both from flat combining local to the node's last-level cache.

1. The combiner can batch and combine multiple oper
   cost than executing each operation individually. For
   priority queue can be done with a single atomic inst
   one atomic instruction for each operation.

2. A single thread accesses the data structure for multi
   cache locality for the whole batch of operations.

NR also benefits from combining operations placed on th
single atomic operation for the whole batch instead of on



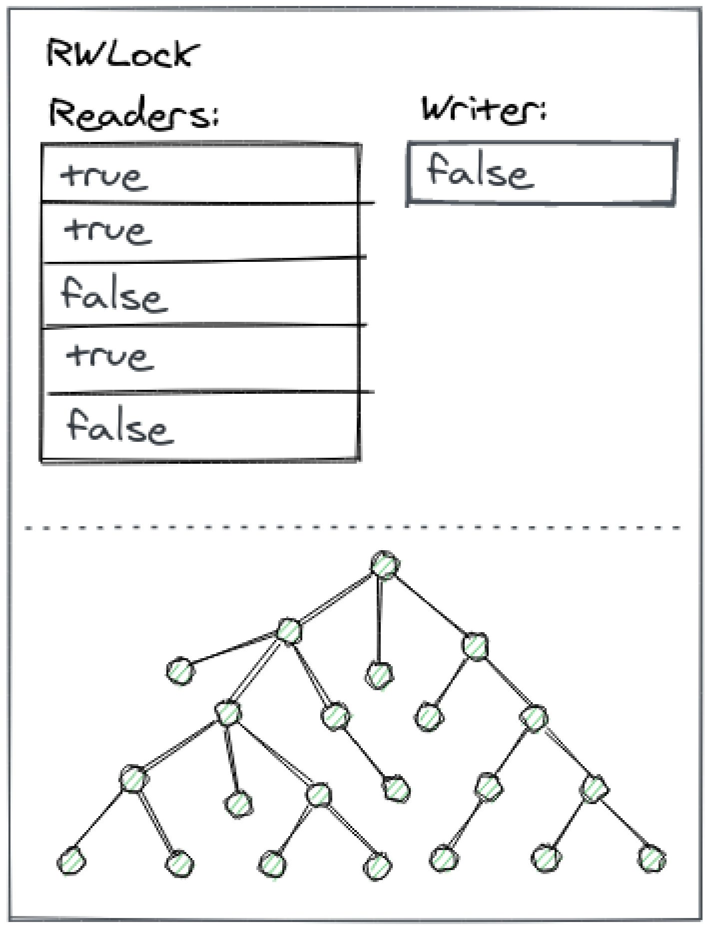
Flat combining multiple operations from threads on
thread is elected who will aggregate outstanding op
put them on the log in one batch (3) and then execu
(4), after executing any previously outstanding opera

The optimized readers-writer lock

NR uses a writer-preference variant of the distributed RW
synchronization of the combiner and reader threads whe

Each reader acquires a local per-thread lock; the writer ha
readers its intent to acquire the writer lock. A reader first
acquires its local lock if there is no writer; then it checks tl
lock) and releases the reader lock if it notices the writer d
(writer preference). The combiner also acquires the writer
Thus, we give highest priority to the writer, because we do
This lock allows readers to read a local replica while the co
operations to the log, increasing parallelism.

If there is no combiner, a reader might have to acquire th
from the log to avoid a stale read, but this situation is rare



The RWLock used to protect the data-structure in ev
cache-line sized atomic booleans for reader threads
thread. It allows concurrent readers but only a singl
Readers also have to make sure to advance the repl
the log.

A concurrent non-mutating operati

A non-mutation operation (read) can execute on its threa
entry, but it needs to ensure that the replica is not stale. T
tail when the operation begins, and waits until a combine
observed tail, or acquires the lock and updates the replica

If there is no combiner, the read thread will acquire the w

A concurrent mutating operation (u

A thread T executing a mutating operation (update) needs
thread T executing this operation fails to acquire it, anoth
spin-waits to receive its operation's result from the existin

However, T must still periodically attempt to become the
missed T's operation.

If T acquires the lock, it becomes *the combiner* and execut
concurrent threads on the same replica. To do so, T appe
with a single atomic operation and ensures that the replic
operations and returning their results to the waiting threa

Source and code example

Node-replication (NR) is released as a stand-alone library.

To give an idea, here is an example that transforms an sin
Rust standard library) into a concurrent, replicated hash-t

```
//! A minimal example that implements a replicated
use std::collections::HashMap;
use node_replication::Dispatch;
use node_replication::Log;
use node_replication::Replica;

/// The node-replicated hashmap uses a std hashmap
#[derive(Default)]
struct NrHashMap {
    storage: HashMap<u64, u64>,
}

/// We support mutable put operation on the hashma
#[derive(Clone, Debug, PartialEq)]
enum Modify {
    Put(u64, u64),
}

/// We support an immutable read operation to look
#[derive(Clone, Debug, PartialEq)]
enum Access {
    Get(u64),
}

/// The Dispatch traits executes `ReadOperation` (
/// and `WriteOperation` (our `Modify` enum) again
/// data-structure.
impl Dispatch for NrHashMap {
    type ReadOperation = Access;
    type WriteOperation = Modify;
    type Response = Option<u64>;

    /// The `dispatch` function applies the immuta
    fn dispatch(&self, op: Self::ReadOperation) ->
        match op {
            Access::Get(key) => self.storage.get(&
        }
    }

    /// The `dispatch_mut` function applies the mu
    fn dispatch_mut(&mut self, op: Self::WriteOper
        match op {
            Modify::Put(key, value) => self.storag
        }
    }
}
```

As we can see we need to define two operation enums (`M` end up on the log, and `Access` for immutable/read opera implement the `Dispatch` trait from NR for our newly defi responsible to route the operations defined in `Access` ar

structure. The full example, including how to create replic
repository.

Concurrent Node Replicati

Some OS subsystems can become limited by node-replica
frequently mutating but would otherwise naturally comm

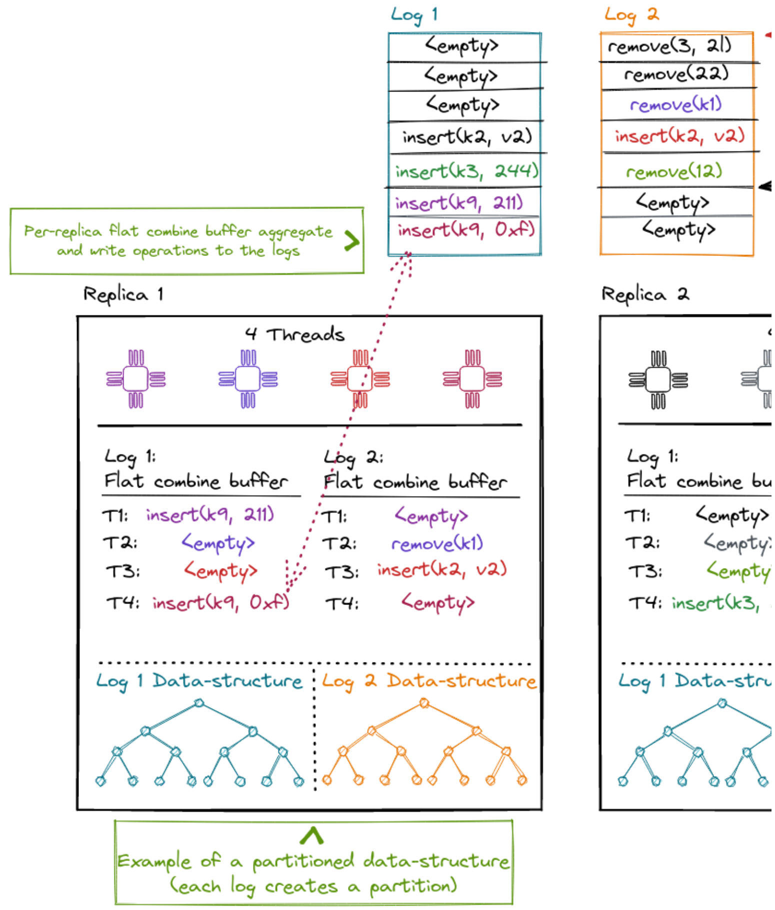
NR allows multiple combiners from different replicas to m
scalability is limited because

1. all combiner threads are operating on a single, shar
2. each replica is a sequential data structure, which rec
   writer lock.

To address these problems, we can use CNR, a technique
approach by leveraging operation commutativity present
operations are said to be *commutative* if executing them i
structure in the same abstract state. Otherwise, we say op

As NR, CNR replicates a data structure across NUMA node
the replicas. However, CNR uses commutativity to scale th
logs, by assigning commutative operations to different log
operations always use the same log and thus have a total
concurrent or partitioned data structures for replicas, wh
combiners on each replica -- one for each shared log. This
writer lock and scales access to the data structure.

CNR lifts *an already concurrent data structure* to a NUMA-a
original data structure can be a concurrent (or partitioned
small number of threads (4-8 threads) within a single NUM
lock-free or lock-based and may result in poor performan
concurrent data structure that works well for a large num
across NUMA nodes, and that is resilient to contention.



In CNR a replica can distribute commuting operatio
replica maintains one flat-combining buffer per log
aggregated. One elected combiner thread commits
batch to the log and then applies the ops against th

Compared to NR, the replicated data-structure is no
reader-writer lock by default. Instead, the data-struc
as in this diagram, use a lock-free approach, or rely

CNR Operations and Linearizability

CNR is useful for *search data structures*, with operations `i`
and `range-scan(x, y)`. These operations often benefit f
both on the abstract operation type and on its input argu
boosting, CNR considers the abstract data type for establi
data structure implementation.

Consider, for example, the `insert(x)` operation. Two op
operate on distinct arguments: e.g., `insert(x)` and `inse`
concrete implementation of the data structure could be a

that `insert(x)` and `insert(x+1)` are not commutative b
memory locations. However, the original data structure a
shared memory locations, due to its concurrent nature. H
CNR and can be safely executed concurrently.

Interface

CNR's interface is similar to NR, but it adds *operation class* mutable and immutable operations. CNR relies on the pro conflicting operations by assigning them to the same ope information to allocate conflicting operations to the same the same NUMA node, to the same combiner too. In cont executed by different combiners and can use different sh executed concurrently.

As with NR, CNR executes different steps for mutating (up operations. Each of these operations uses only one of the according to its log's order. Different logs are not totally c different logs are commutative.

In addition, CNR special-cases another type of operation, operation class. These are operations that conflict with m operation has to read the entire data structure to determ operation to a single class, all other operations need to be commutativity benefit.

Scan-type operations span multiple logs and need a *consi* logs involved in the operation, obtained *during the lifetime* consistent state, the thread performing the scan collects a inserting the operation in these logs. This atomic snapsho point.

We show the CNR API, with the additional traits implemen using our earlier example in the NR section:

```
use chashmap::CHashMap;
use cnr::{Dispatch, LogMapper};

/// The replicated hashmap uses a concurrent hashm
pub struct CNRHashMap {
    storage: CHashMap<usize, usize>,
}

/// We support a mutable put operation on the hash
#[derive(Debug, PartialEq, Clone)]
pub enum Modify {
    Put(usize, usize),
}

/// This `LogMapper` implementation distributes th
/// in a round-robin fashion. One can change the
/// data locality based on the data sturucture lay
impl LogMapper for Modify {
    fn hash(&self, nlogs: usize, logs: &mut Vec<us
        debug_assert!(logs.capacity() >= nlogs, "gua
        debug_assert!(logs.is_empty(), "guarantee on

        match self {
            Modify::Put(key, _val) => logs.push(*key
        }
    }
}

/// We support an immutable read operation to look
#[derive(Debug, PartialEq, Clone)]
pub enum Access {
    Get(usize),
}

/// `Access` follows the same operation to log map
/// ensures that the read and write operations for
/// the same log.
impl LogMapper for Access {
    fn hash(&self, nlogs: usize, logs: &mut Vec<us
        debug_assert!(logs.capacity() >= nlogs, "gua
        debug_assert!(logs.is_empty(), "guarantee on

        match self {
            Access::Get(key) => logs.push(*key % nlog
        }
    }
}

/// The Dispatch traits executes `ReadOperation`
/// and `WriteOperation` (our Modify enum) against
/// data-structure.
impl Dispatch for CNRHashMap {
    type ReadOperation = Access;
```

```
    type WriteOperation = Modify;
    type Response = Option<usize>;

    /// The `dispatch` function applies the immutat
    fn dispatch(&self, op: Self::ReadOperation) ->
        match op {
            Access::Get(key) => self.storage.get(&k
        }
    }

    /// The `dispatch_mut` function applies the mut
    fn dispatch_mut(&self, op: Self::WriteOperatior
        match op {
            Modify::Put(key, value) => self.storage
        }
    }
}
```

CNR is available as a stand-alone rust library together wit

Comparison to NR and Notation

CNR benefits from the NR techniques, such as flat combir operation description, which reduce contention and inter

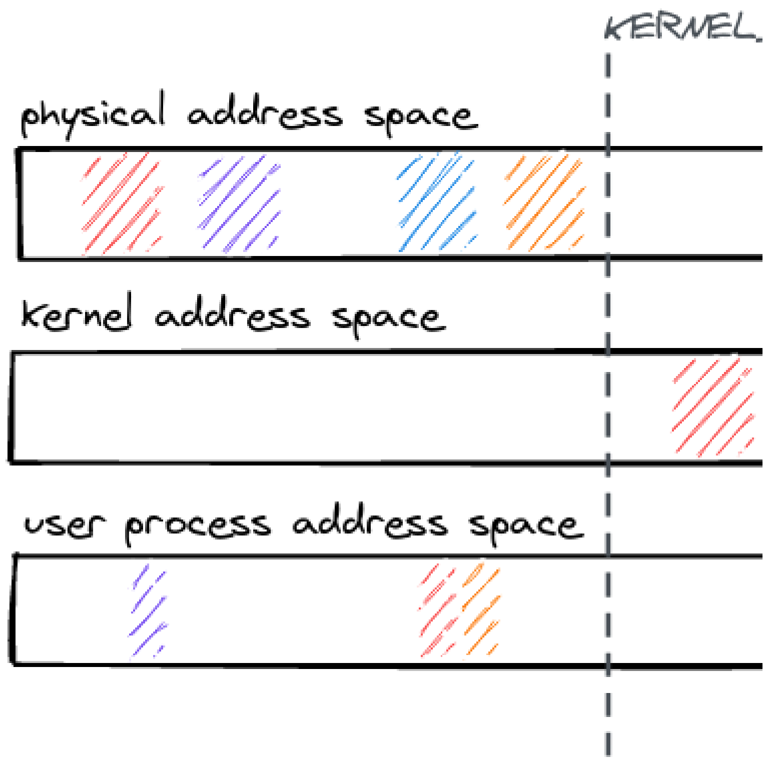
As NR, CNR has two drawbacks: increased memory footpr from re-executing each operation on each replica. CNR ha single shared log can be split into multiple, smaller sharec parallelism within each NUMA node by using a concurren increases parallelism across NUMA nodes by using multip

Memory

Kernel address space

The kernel address space layout follows a simple scheme
mapped with a constant offset `KERNEL_BASE` in the kerne
physical address can be accessed in the kernel by adding

All physical memory is always accessible in the kernel anc
mapped/unmapped at runtime. The kernel binary is linke
loaded into physical memory and then relocated to run a
(`KERNEL_BASE` + physical address).



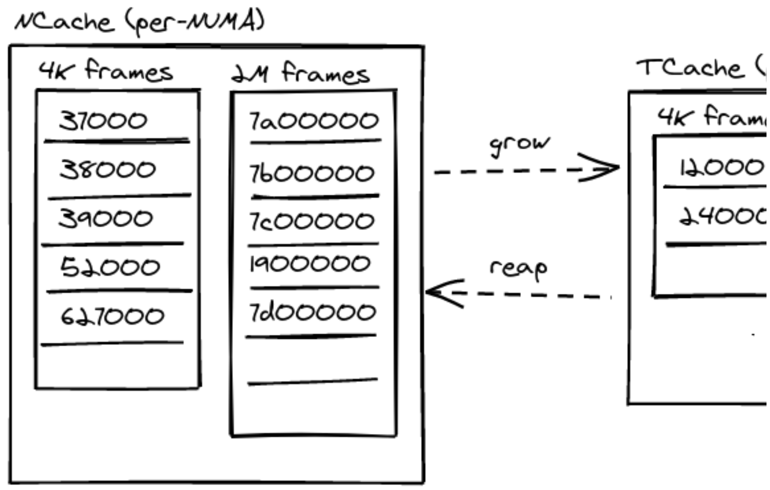
A view of different address spaces in nrk (physical, k

Physical memory

Physical memory allocation and dynamic memory allocati
basic subsystems that do not rely on NR. Replicated subs
but that allocation operation itself should not be replicate

mapping in a page table, each page table entry should re on all replicas (though, each replica should have its own p replicated, each allocation operation would be repeated c

At boot time, the affinity for memory regions is identified, NUMA node caches (FrameCacheLarge). The FrameCache further into two classes of 4 KiB and 2 MiB frames. Every KiB and 2 MiB frames for fast, no-contention allocation w size. If it is empty, it refills from its local FrameCacheLarge FrameCacheSmall and FrameCacheLarge implement a ca controls the flow between TCaches and NCaches.



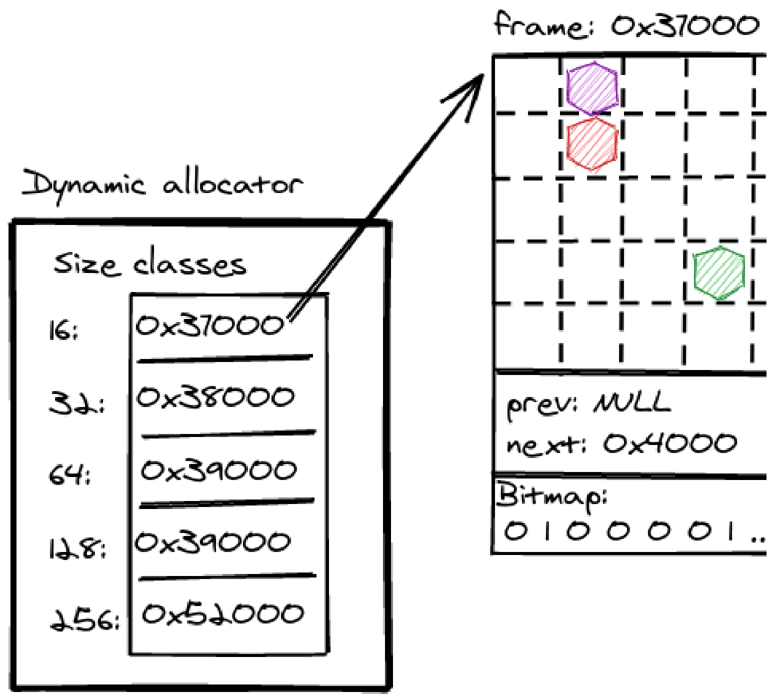
Shows a global per-NUMA FrameCacheLarge and a p Cores allocate 4K or 2M pages directly from the Fran from the FrameCacheLarge when empty (grow). TCa frames in stacks to allows for quick allocation and d

Dynamic memory

Since NRK is implemented in Rust, memory management compiler to track the lifetime of allocated objects. This eli after-free, uninitialized memory *etc.*), but the kernel still h of memory. NRK uses fallible allocations and intrusive dat errors gracefully.

The dynamic memory allocator in nrk provides an implem interface. It uses size classes and different allocators per allocator), while incorporating some of the simple and eff each size class, 2MiB or 4 KiB frames are used which are s

given class. A bitfield at the end of every frame tracks the
frame (*e.g.,* to determine if its allocated or not).



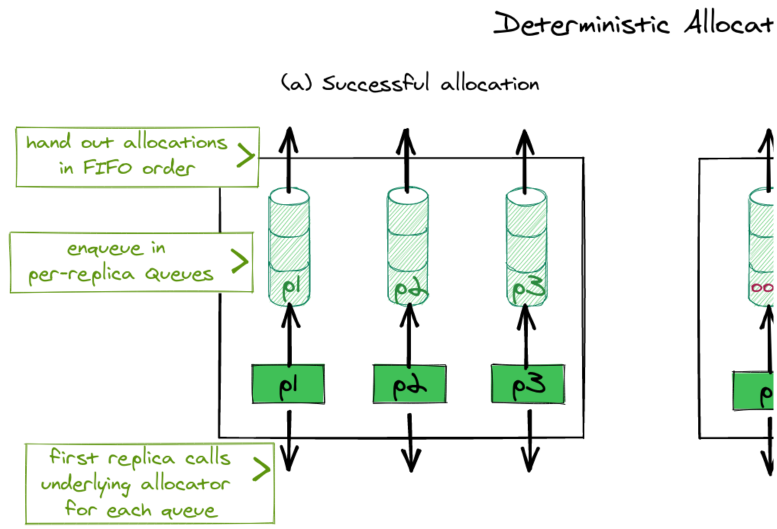
The dynamic memory allocator for kernel objects in
containing two frames for less than 16 byte objects.
allocated slots along with per-frame meta-data (prev
indicate allocated blocks. Typically, one dynamic me
instantiated.

Deterministic memory

The kernel has to explicitly handle running out of memory
handle out-of-memory errors gracefully by returning an e
almost all cases in the kernel (with some exceptions durin
3rd party dependencies (e.g., to parse ELF binaries) are no
allocations yet.

Another issue is that handling out-of-memory errors in pr
becomes a little more challenging: Allocations which happ
deterministic (e.g. they should either succeed on all replic
would end up in an inconsistent state if after executing ar
successful and some had unsuccesful allocations. Making
equal amounts of memory available is infeasible because
times and meanwhile allocations can happen on other co
this problem in nrk by requiring that all memory allocatio

through a deterministic allocator. In the deterministic allc
allocation request allocates memory on behalf of all othe
allocator remembers the results temporarily, until they ar
which are running behind. If an allocation for any of the r
enqueue the error for all replicas, to ensure that all replic
Allocators in nrk are chainable and it is sufficient for the c
in the chain so it doesn't necessarily have to be invoked fo
request. Our implementation leverages custom allocators
heap allocator is used for individual data-structures.



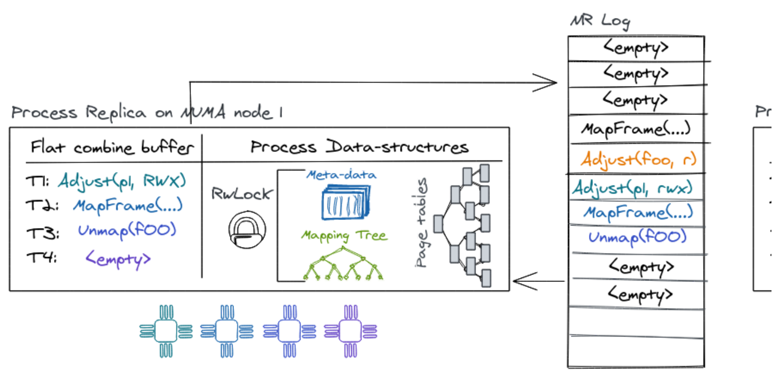
The deterministic memory allocator generally forwa
underlying memory allocator. However, the first req
on behalf of all other replicas and store the results t
(one per replica). The deterministic allocator ensure
have a successful allocations for a given request or (
a result back.

Process structure

Virtual memory

NRK relies on the MMU for isolation. Like most conventio
NRK uses a per-process mapping database (as a B-Tree) t
to construct the process's hardware page tables. NRK cur
paging. Both the B-Tree and the hardware page tables are
that are wrapped behind the NR interface for concurrenc

Therefore, the mapping database and page tables are rep
the biggest part of the process abstraction. A process exp
operations to modify its address space: `MapFrame` (to inse
mappings); and `Adjust` (to change permissions of a map
supports a non-mutating `Resolve` operation (that advan
address space state).



Schematic diagram of a node-replicated process in N

Virtual memory and NR

There are several aspects of the virtual-memory design th
with NR:

For example, the virtual-memory has to consider out-of-b
table walkers. Normally a read operation would go throug
all outstanding operations from the log first. However, a h
have this capability. A race arises if a process maps a page
replica B accesses that mapping in userspace before repli
this can be handled since it generates a page fault. In ord
handler advances the replica by issuing a `Resolve` opera

corresponding mapping of the virtual address generating
process can be resumed since the `Resolve` -operation wil
mapping is found, the access was an invalid read or write

`Unmap` or `Adjust` (*e.g.,* removing or modifying page-table
entries on cores where the process is active to ensure TLI
software, by the OS, and commonly referred to as perforr
core will start by enqueuing the operation for the local re
the unmap (or adjust) operation has been performed at le
replica and is enqueued as a future operation on the log f
processor interrupts (IPIs) to trigger TLB flushes on all cor
process. As part of the IPI handler the cores will first ackn
they must make sure to advance their local replica with o
forces the unmap/adjust if not already applied), then poll
information about the regions that need to be flushed, ar
Meanwhile the initiator will invalidate its own TLB entries
acknowledgments from other cores before it can return t

Scheduler

In NRK, the kernel-level scheduler is a coarse-grained sch
processes. Processes make system calls to request for mo
kernel notifies processes core allocations and deallocatio
process allocates executor objects (*i.e.,* the equivalent of a
dispatch a given process on a CPU. An executor mainly co
for the upcall handler and one for the initial stack) and a r
metadata. Executors are allocated lazily but a process kee
them over time.

In the process, a userspace scheduler reacts to upcalls ind
core, and it makes fine-grained scheduling decisions by d
design means that the kernel is only responsible for coars
implements a global policy of core allocation to processes

The scheduler uses a sequential hash table wrapped with
process structure and to map process executors to cores.
a process; to allocate and deallocate executors for a proc
given core.

File System

The NrFS is a simple, in-memory file system in nrk that su
(`open` , `pread` , `pwrite` , `close` , *etc.*).

NrFS tracks files and directories by mapping each path to
each inode number to an in-memory inode. Each inode h
and a list of file pages. The entire data structure is wrappe
replication.

User Space

The main user-space components are explained in more

KPI: Kernel Public Interface

The Kernel Public Interface (KPI) is the lowest level user-sp
with. As the name suggests it is the common interface de
space programs. It is special because it is the only library
kernel code.

The KPI contains the syscall interface and various struct d
between the kernel and user-space. If in the future, we ca
not try to keep the syscall ABI compatible but would rathe
boundary.

Typically, the KPI functionality will rarely be accessed dire
parts of it are re-exported or wrapped by the vibrio librar
`lib/kpi`.

Lineup

Lineup is a user-space, cooperative thread scheduler that threads). It supports many synchronization primitives (mu barriers etc.), thread-local storage, and has some basic su fringe for compiler-assisted context-switching. The sched

Upcalls

The kernel can notify the scheduler about events through to notify about more available cores (or removal of cores) page-faults from kernel to user-space.

The mechanism for this is inspired by scheduler activatio program agree on a common save area to store a CPU co arrives at the kernel, it will save the current CPU context (save-area and resume the process with a new (mostly em registered upcall handler instead. The upcall handler gets triggered the interruption through function arguments sc measures to react. After the event is handled, the upcall h (from before the interruption) from the common save are computation left off before the upcall (or decide not to co

Vibrio

Virbio is the user-space library that provides most of the f
applications in user-space. It is found in `lib/vibrio`.

Memory

The user-space memory manager provides a `malloc` anc
`GlobalAlloc` implementation for rust programs. We rely
uses for small--medium sized blocks (between 0 and 2 Mi
by allocating memory with the map syscall.

RumpRT

A rumpkernels is a componentized NetBSD kernel that ca
environments. It contains file systems, a POSIX system ca
SCSI protocol stack, virtio, a TCP/IP stack, libc and libpthre

Vibrio has a `rumprt` module which provides the necessar
rumpkernel inside a user-space process (e.g., the rumpus
advantage that it's possible to run many POSIX compatibl
building a fully-fledged POSIX compatibility layer into NrC

- Bare-metal and Xen implementations for rumprun
- Some supported applications
- PhD thesis about rumpkernels

Vibrio dependency graph

Vibrio uses the following crates / dependencies:

```
vibrio
├── arrayvec
├── bitflags
├── crossbeam-utils
├── cstr_core
├── hashbrown
├── kpi
├── lazy_static
├── lineup
├── log
├── rawtime
├── rumpkernel
├── serde_cbor
├── slabmalloc
├── spin
└── x86
```

rkapps

The rkapps directory (in `usr/rkapps`) is an empty project build file contains the steps to clone and build a few diffe (memcached, LevelDB, Redis etc.) that we use to test user

The checked-out program sources and binaries are place your build directory:

```
target/x86_64-nrk-none/<debug | release>/build/rka
```

The build for these programs can be a bit hard to underst

1. Clone the packages repo which has build instructior running on rumpkernels.
2. For each application that we want to build (enabled/ in the respective directory. This will compile the app rumpkernel toolchain. The toolchain be found in a s `target/x86_64-nrk-none/<debug | release>/buil`
3. Linking binaries with vibrio which provides the low-l

> For more information on how to run rkapps applicatio section.

Development

This chapter should teach you how to Build, Run, Debug,

Configuration

Some tips and pointers for setting up and configuring the

VSCode

VSCode generally works well for developing nrk. The `rus`
`rls` which often has build issues due to the project not h

Git

For first time git users or new accounts, you'll have to con

```
git config --global user.name "Gerd Zellweger"
git config --global user.email "mail@gerdzellweger
```

To have better usability when working with submodules,
submodules automatically when doing a `git pull` etc.

```
git config --global submodule.recurse true
```

Fetch multiple submodules in parallel:

```
git config --global submodule.fetchJobs 20
```

We don't allow merge requests on master, to always keep
can be helpful:

```
[alias]
    purr = pull --rebase
```

Adding a new submodule to the repository

1. `cd lib`
2. `git submodule add <path-to-repo> <foldername>`

Removing a submodule in the repository

1. Delete the relevant section from the .gitmodules file
2. Stage the .gitmodules changes: `git add .gitmodule`
3. Delete the relevant section from .git/config.
4. Run `git rm --cached path_to_submodule` (no traili
5. Run `rm -rf .git/modules/path_to_submodule` (no
6. Commit changes

Styleguide

Code format

We rely on rustfmt to automatically format our code.

Code organization

We organize/separate imports into three blocks (all separ

- 1st block for core language things: `core`, `alloc`, `st`
- 2nd block for libraries: `vibrio`, `x86`, `lazy_static`
- 3rd block for internal imports: `crate::*`, `super::*`
- 4th block for re-exports: `pub(crate) use::*` etc.
- 5th block for modules: `mod foo;` etc.

Afterwards a `.rs` file should (roughly) have the following

- 1st `type` declarations
- 2nd `const` declarations
- 3rd `static` declarations
- 4th `struct`, `fn`, `impl` etc. declarations

Visibility

Avoid the use of `pub` in the kernel. Use `pub(crate)`, `pub`
code elimination.

Assembly

We use AT&T syntax for assembly code (`options(att_syr`

Cargo features

Libraries and binaries only have non-additive / non-confli
compilation problems quickly (e.g. with `cargo build --a`

Errors

The `KError` type is used to represent errors in the kernel
should only be used once/in a single location (to be easy t
descriptive name.

Formatting Commit Messages

We follow the conventions on How to Write a Git Commit

Be sure to include any related GitHub issue references in
for referencing issues and commits.

Github pull requests & history

Since github doesn't do fast-forward merges through the
the command line to keep the same commit hashes of th

```
git checkout master
git merge --ff-only feature-branch-name
```

Building

There are two sets of dependencies required for the deve
dependencies. We typically build, develop and test using t
nrk in QEMU. Other Linux systems will probably work but
of all dependencies. Other operating systems likely won't
adjustments for code and the build-process.

Check-out the source tree

Check out the nrk sources first:

```
git clone <repo-url>
cd nrk
```

The repository is structured using git submodules. You'll h
submodules separately:

> In case you don't have the SSH key of your machine reg
> need to convert all submodule URLs to use the https p
> this sed script before proceeding:
>
> ```
> sed -i'' -e 's/git@github.com:/https:\/\/github.
> ```

```
git submodule update --init
```

Dependencies

If you want to build without Docker, you can install both b
executing `setup.sh` in the root of the repository directly
latest Ubuntu LTS). The script will install all required OS pa
some additional rust programs and dependencies. To run
also have to install the DCM-based scheduler dependenci

The build dependencies can be divided into these categor

- Rust (nightly) and the `rust-src` component for com
- `python3` (and some python libraries) to execute the
- Test dependencies (qemu, corealloc, dhcpd, redis-be
- Rumpkernel dependencies (gcc, zlib1g etc.)
- Build for documentation (mdbook)

See `scripts/generic-setup.sh` function `install_build_`

Use Docker

We provide scripts to create a docker image which contai

> To use Docker, it needs to be installed in your system.
> steps:
>
> ```
> sudo apt install docker.io
> sudo service docker restart
> sudo addgroup $USER docker
> newgrp docker
> ```

To create the image execute the following command in th

```
bash ./docker-run.sh
```

This will create the docker image and start the container. running inside the Docker container. You can build the OS dependencies natively.

> The script will create a user inside the docker containe the host system (same username and user ID).

You can rebuild the image with:

```
bash ./docker-run.sh force-build
```

To exit the container, just type `exit` to terminate the she

Build without running

To just build the OS invoke the `run.py` script (in the kern
(no-run flag).

```
python3 kernel/run.py -n
```

If you want to run the build in a docker container, run `ba`
beforehand. The source directory tree will be mounted in

Using `run.py`

The `kernel/run.py` script provides a simple way to build settings and configuration. For a complete set of paramet `run.py --help` instructions.

As an example, the following invocation

```
python3 run.py --kfeatures test-userspace --cmd='l
rkapps init --ufeatures rkapps:redis --machine qem
qemu-cores 2
```

will

- compile the kernel with Cargo feature `test-userspa`
- pass the kernel the command-line arguments `log=i` (sets logging to info and starts redis.bin for testing)
- Compile two user-space modules `rkapps` (with carg features)
- Deploy and run the compiled system on `qemu` with allocated to the VM

If Docker is used as build environment, it is necessary to f required features inside the Docker container:

```
python3 run.py --kfeatures test-userspace --mods r
rkapps:redis -n
```

Afterwards, the aforementioned command can be used t container with the given configuration. The `run.py` script already been build and will directly start `qemu`.

Sometimes it's helpful to know what commands are actua to figure out what the exact qemu command line invocati be supplied.

Depending on the underlying system configuration NRK r local network can not be established. In this case, the foll issue:

1. Disable AppArmor. Detailed instructions can be foun
2. Manually start the DHCP server immediately after N

```
sudo dhcpd -f -d tap0 --no-pid -cf ./kernel/t
```

Baremetal execution

The `kernel/run.py` script supports execution on bareme
argument:

```
python3 run.py --machine b1542 --verbose --cmd "lo
```

This invocation will try to run nrk on the machine describ

A TOML file for a machine has the following format:

```
[server]
# A name for the server we're trying to boot
name = "b1542"
# The hostname, where to reach the server
hostname = "b1542.test.com"
# The type of the machine
type = "skylake2x"
# An arbitrary command to set-up the PXE boot env
# This often involves creating a hardlink of a fil
# of the machine and pointing it to some pxe boot
pre-boot-cmd = "./pxeboot-configure.sh -m E4-43-4B

# run.py support only booting machines that have a
[idrac]
# How to reach the ilo/iDRAC interface of the mach
hostname = "b1542-ilo.test.com"
# Login information for iDRAC
username = "user"
password = "pass"
# Serial console which we'll read from
console = "com2"
# Which iDRAC version we're dealing with (current
idrac-version = "3"
# Typical time until machine is booted
boot-timeout = 320

[deploy]
# Server where binaries are deployed for booting w
hostname = "ipxe-server.test.com"
username = "user"
ssh-pubkey = "~/.ssh/id_rsa"
# Where to deploy kernel and user binaries
ipxe-deploy = "/home/gz/public_html/"
```

An iPXE environment that the machine will boot from nee
should be compiled with UEFI and ELF support for runnin

> Note that the current support for bare-metal executior
> machines with an iDRAC management console (needec
> redfish or SNMP support will be added in the future.

Compiling the iPXE bootloader

TBD.

Debugging

Currently the debugging facilities are not as good as on a However, there are some options available: gdb, `printf`- at code. We will discuss the options in this chapter.

GDB support in the kernel

> tldr: To use gdb, add `--kgdb` to `run.py`.

NRK provides an implementation for the gdb remote prot communication. This means you can use gdb to connect t

To use it, start `run.py` with the `--kgdb` argument. Once

```
Waiting for a GDB connection on I/O port 0x2f8...
Use `target remote localhost:1234` in gdb session
```

Next, connect with GDB to the kernel, using:

```
$ cd kernel
$ gdb ../target/x86_64-uefi/<debug | release>/esp,
[...]
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
[...]
```

> If you execute gdb in the kernel directory, the `.gdbini` `target remote localhost:1234` for you. But you have "trusted" path by adding this line to `$HOME/.gdbinit`:
>
> ```
> add-auto-load-safe-path <REPO-BASE>/kernel/.gdb
> ```

The GDB dashboard works as well, just insert `target rem` `.gdbinit` file.

Breakpoints

> tldr: use `break` or `hbreak` in gdb.

Currently the maximum limit of supported breakpoints (a

Why? Because we use the x86-64 debug registers for brea registers. Our gdb stub implements both software and ha registers.

An alternative technique would be to either insert `int3` i let gdb do it automatically if software interrupts are mark However, this is a bit more complicated because we need (e.g., the debug interrupt handler should ideally not hit a debug registers, this is fairly easy to achieve, as we just ha enable them when we resume, whereas the `int3` approa bunch of `.text` offsets. On the plus side it would enable this ever becomes necessary.

Watchpoints

Again the maximum limit is four watchpoints (and breakp

Use `watch -l <variable>` to set a watchpoint. The `-l` o the memory location of the variable/expression rather th not supported as gdb may try to overwrite `.text` locatio execute) in the kernel.

`printf` debugging with the log crate

Here are a few tips:

- Change the log-level of the kernel to info, debug, or `cmd='log=info'`
- Logging can also be enabled per-module basis. For e just the `gdbstub` library and the `gdb` module in the `cmd` invocation for `run.py` would look like: `--cmd "log='gdbstub=trace,nrk::arch::gdb=trace'"`

- Change the log-level of the user-space libOS in vibric
- Make sure the Tests run (to see if something broke).

Figuring out why things failed

Maybe you'll encounter failures, for example like this one

```
[IRQ] GENERAL PROTECTION FAULT: From Any memory re
checks.
No error!
Instruction Pointer: 0x534a39
ExceptionArguments { vec = 0xd exception = 0x0 rip
0x13206 rsp = 0x5210400928 ss = 0x1b }
Register State:
Some(SaveArea

rax =                0x0 rbx =               0x0
0x0
rsi =                0x0 rdi =      0x5210400a50
0x5210400928
r8  =                0x2 r9  =      0x5202044c00
0x28927a
r12 =       0x520e266810 r13 =          0x7d8ac0
0x686680
rip =           0x534a39 rflags = FLAGS_RF | FLAGS
FLAGS_IOPL3 | FLAGS_IF | FLAGS_PF | FLAGS_A1)
stack[0] = 0x5210400958
stack[1] = 0x53c7fd
stack[2] = 0x0
stack[3] = 0x0
stack[4] = 0x0
stack[5] = 0x0
stack[6] = 0x52104009b8
stack[7] = 0x534829
stack[8] = 0x5210400a50
stack[9] = 0x5210400a50
stack[10] = 0x0
stack[11] = 0x268
```

The typical workflow to figure out what went wrong:

1. Generally, look for the instruction pointer (`rip` whic
2. If the instruction pointer (and `rsp` and `rbp`) is belo
user-space when the failure happened (you can also
it's easier to tell from the other registers).
3. Determine exactly where the error happened. To do
which was running. Those are usually located in `tar`

uefi/<release|debug>/esp/<binary> .

4. Use `addr2line -e <path to binary> <rip>` to see
5. If the failure was in kernel space, make sure you adj PIE offset where the kernel binary was executing in t following line `INFO: Kernel loaded at address: 0` bootloader early during the boot process. Substract offset in the ELF file.
6. Sometimes `addr2line` doesn't find anything, it's go gives more context: `objdump -S --disassemble --c uefi/<release|debug>/esp/<binary> | less`
7. The function that gets reported might not be useful case, look for addresses that could be return addres them too (e.g., `0x534829` looks suspiciously like a re
8. If all this fails, something went wrong in a bad way, r debugging.

Always find the first occurrence of a failure in the seria is not very robust, it still quite often triggers cascading relevant.

Debugging rumpkernel/NetBSD com

nrk user-space links with a rather large (NetBSD) code-ba in there, it's sometimes helpful to temporarily change or g C code.

You can edit that code-base directly since it gets checked For example, to edit the `rump_init` function, open the fil NetBSD source here: `target/x86_64-nrk-none/release/l netbsd/sys/rump/librump/rumpkern/rump.c`

Make sure to identify the correct $HASH that is used for t multiple `rumpkernel-*` directories in the build dir, otherv

After you're done with edits, you can manually invoke the

As a simple example you can search for `rump_init(void)` `none/release/build/rumpkernel-$HASH/out` and add a `p` following steps should ensure the print also appears on t

```
cd target/x86_64-nrk-none/release/build/rumpkernel
./build-rr.sh -j24 nrk -- -F "CFLAGS=-w"
# Delete ../target/x86_64-nrk-none/debug/build/in
# Invoke run.py again...
```

> If you change the compiler/rustc version, do a clean bu
> your changes might be overridden as the sources exist
> (`target`). It's a good idea to save changes somewhere
> important.

Debugging in QEMU/KVM

If the system ends up in a dead-lock, you might be able to
south by asking qemu. Deadlocks with our kernel design
locking APIs) it can definitely happen.

The following steps should help:

1. Add `--qemu-monitor` to the run.py invocation to sta
2. Connect to the monitor in a new terminal with `teln`
3. You can use `info registers -a` to get a dump of th
   or any other command to query the hypervisor state
4. If you're stuck in some loop, getting a couple registe
   invoking `info registers` just once.

When developing drivers that are emulated in qemu, it ca
the interface in QEMU to see what state the device is in. F
for `vmxnet3` in the sources, you can change the `#undef s`
`hw/net/vmxnet_debug.h` to `#define` and recompile the c
look similar to this snippet below):

```
#define VMXNET_DEBUG_CB
#define VMXNET_DEBUG_INTERRUPTS
#define VMXNET_DEBUG_CONFIG
#define VMXNET_DEBUG_RINGS
#define VMXNET_DEBUG_PACKETS
#define VMXNET_DEBUG_SHMEM_ACCESS
```

Testing

If you've found and fixed a bug, we better write a test for
and methodologies to ensure everything works as expect

- Regular unit tests: Those can be executed running `c`
  Sometimes adding `RUST_TEST_THREADS=1` is necess
  runner/frameworks used. This should be indicated i
- A slightly more exhaustive variant of unit tests is pro
  to make sure that the implementation of kernel sub
  model implementation.
- Integration tests are found in the kernel, they typica
  rexpect to interact with the guest.
- Fuzz testing: TBD.

Running tests

To run the unit tests of the kernel:

1. `cd kernel`
2. `RUST_BACKTRACE=1 RUST_TEST_THREADS=1 cargo te`

To run the integration tests of the kernel:

1. `cd kernel`
2. `RUST_TEST_THREADS=1 cargo test --test '*'`

If you would like to run a specific integration test you can

1. `RUST_TEST_THREADS=1 cargo test --test '*' -- u`

If you would like to run a specific set of integration tests, y
`test`:

1. `RUST_TEST_THREADS=1 cargo test --test s00_core`

In case an integration test fails, adding `--nocapture` at th
will make sure that the underlying `run.py` invocations ar
helpful to figure out the exact `run.py` invocation that a te
yourself manually for debugging.

> Parallel testing for he kernel is not possible at the mom
> for testing.

The commitable.sh script automatically runs the unit and

```
cd kernel
bash commitable.sh
```

Writing a unit-test for the kernel

Typically these can just be declared in the code using `#[t` run under the `unix` platform. A small hack is necessary t and run under unix too: When run on a x86-64 unix platfo kernel in `arch/x86_64/` will be included as a module nam would be `arch`. This is a double-edged sword: we can no metal code (great), but we can also easily crash the test p MSR for example (e.g, things that would require ring 0 pri

Writing an integration test for the k

Integration tests typically spawns a QEMU instance and b space with a custom set of Cargo feature flags. Then it pa the expected output. Part of those custom compile flags v function than the one you're seeing (which will go off to lo programs for example).

There is two parts to the integration test.

- The host side (that will go off and spawn a qemu ins tests. It is found in `kernel/tests`.
- The corresponding main functions in the kernel that example are located at `kernel/src/integration_ma`

To add a new integration test the following tests may be r

1. Modify `kernel/Cargo.toml` to add a feature (under

2. Optional: Add a new `xmain` function and test impler `kernel/src/integration_main.rs` with the used fe also be possible to re-use an existing xmain functior name used to include it.
3. Add a runner function to one of the files in `kernel/` cargo feature runs it and checks the output.

Integration tests are divided into categories and named a tests run in a sensible order):

- `s00_*` : Core kernel functionality like boot-up and fa
- `s01_*` : Low level kernel services: SSE, memory alloc
- `s02_*` : High level kernel services: ACPI, core booting
- `s03_*` : High level kernel functionality: Spawn cores,
- `s04_*` : User-space runtimes
- `s05_*` : User-space applications
- `s06_*` : Rackscale (distributed) tests

Benchmarks are named as such:

- `s10_*` : User-space applications benchmarks
- `s11_*` : Rackscale (distributed) benchmarks

The `s11_*` benchmarks may be configured with two featu

- `baseline` : Runs NrOS configured similarly to racksc
- `affinity-shmem` : Runs the `ivshmem-server` using s option requires preconfiguring hugetlbfs with `sudo` having a kernel with 2MB huge pages enabled, and t node, with a command like: `echo <page-num> | suc` `/proc/sys/vm/nr_hugepages_mempolicy` The numbe verified with `numastat -m`.

Network

nrk has support for three network interfaces at the mome and e1000 are available by using the respective rumpkerr vmxnet3 is a standalone implementation that uses `smolt` capable of running in ring 0.

Network Setup

The integration tests that run multiple instances of nrk re
those integration tests, the test framework calls run.py wi
destroy existing conflicting tap interfaces and create new
the number of hosts in the test. Then, to run the nrk insta
`no-network-setup` flag.

To setup the network for a single client and server (`--wor`
following command:

```
python3 run.py --kfeatures integration-test --cmd
workers 2 --network-only
```

Ping

A simple check is to use ping (on the host) to test the netw
Adaptive `ping -A`, flooding `ping -f` are good modes to
stack work and can handle an "infinite" amount of packet

Some expected output if it's working:

```
$ ping 172.31.0.10
64 bytes from 172.31.0.10: icmp_seq=1 ttl=64 time=
64 bytes from 172.31.0.10: icmp_seq=2 ttl=64 time=
64 bytes from 172.31.0.10: icmp_seq=3 ttl=64 time=
64 bytes from 172.31.0.10: icmp_seq=4 ttl=64 time=
```

For network tests, it's easiest to start a DHCP server for th
IP by communicating with the server:

```
# Stop apparmor from blocking a custom dhcp instar
service apparmor stop
# Terminate any (old) existing dhcp instance
sudo  killall dhcpd
# Spawn a dhcp server, in the kernel/ directory do
sudo dhcpd -f -d tap0 --no-pid -cf ./tests/dhcpd.c
```

A fully automated CI test that checks the network using pi
invoked with the following command:

```
RUST_TEST_THREADS=1 cargo test --test '*' -- s04_u
```

socat and netcat

`socat` is a helpful utility on the host to interface with the
port and print on incoming packets on the command line

```
socat UDP-LISTEN:8889,fork stdout
```

Similarly we can use `netcat` to connect to a port and sen

```
nc 172.31.0.10 6337
```

The integration tests `s05_redis_smoke` and `s04_userspa`
tool to verify that networking is working as expected.

tcpdump

tcpdump is another handy tool to see all packets that are
For debugging nrk network issues, this command is usefu

```
tcpdump -i tap0 -vvv -XX
```

Tracing

Use Intel PT (processor-trace). TBD.

Benchmarking

This chapter provides notes and pointers on how to set-u
benchmarking and run various OS micro-benchmarks.

Microbenchmarks

File-system

The code contains an implementation of the fxmark benc
located at `usr/init/src/fxmark`.

To run the fxmark benchmarks invoke the following comr

```
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
```

fxmark supports several different file benchmarks:

- *drbh*: Read a shared block in a shared file
- *drbl*: Read a block in a private file.
- *dwol*: Overwrite a block in a private file.
- *dwom*: Overwrite a private block in a shared file.
- *mwrl*: Rename a private file in a private directory.
- *mwrm*: Move a private file to a shared directory.
- *mix*: Access/overwrite a random block (with fixed pe

> By default the integration test might not run all benchr
> to change what benchmarks are run or study it to dete
> arguments to `run.py`.

Address-space

The following integration tests benchmark the address-sp

- `s10_vmops_benchmark`: This benchmark repeatedly i
in the process' address space, while varying the nun
core works in its own partition of the address space
throughput (operations per second).

- `s10_vmops_latency_benchmark`: Same as `s10_vmops`
instead of throughput.

- `s10_vmops_unmaplat_latency_benchmark` : The benc
  space, then spawns a series of threads on other cor
  unmaps the frame and measures the latency of the
  dominated by completing the TLB shootdown proto
- `s10_shootdown_simple` : The benchmark measures t
  programming the APIC and sending IPIs to initiate ar

The benchmark code is located at `usr/init/src/vmops/` .

```
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
nocapture
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
--nocapture
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
```

Network

TBD.

Benchmarking Redis

Redis is a simple, single-threaded key--value store written
single-threaded performance of the system.

Automated integration tests

The easiest way to run redis on nrk, is to invoke the redis

- `s05_redis_smoke` will spawn nrk with a redis instan few commands to test basic functionality.
- `s10_redis_benchmark_virtio` and `s10_redis_benc` redis instance and launch the `redis-benchmark` CLI The results obtained by redis-benchmark are parsec `redis_benchmark.csv`. The `virtio` and `e1000` suff used.

```
cd kernel
# Runs both _virtio and _e1000 redis benchmark tes
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
```

Launch redis manually

You can also do the steps that the integration test does m
The `apparmor teardown` is necessary if you don't have a s
`dhcpd.conf` at this location.

```
cd kernel
sudo service apparmor teardown
sudo dhcpd -f -d tap0 --no-pid -cf ./tests/dhcpd.c
```

Next run the redis server in nrk (exchange the nic parame
NIC):

```
python3 run.py \
  --kfeatures test-userspace \
  --nic e1000 \
  --cmd "log=info init=redis.bin" \
  --mods rkapps \
  --ufeatures "rkapps:redis" \
  --qemu-settings="-m 1024M"
```

Finally, execute the redis-benchmark on the host.

```
redis-benchmark -h 172.31.0.10 -n 10000000 -p 6379
```

You should see an output similar to this:

```
====== SET ======
  10000000 requests completed in 10.29 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

0.31% <= 1 milliseconds
98.53% <= 2 milliseconds
99.89% <= 3 milliseconds
100.00% <= 4 milliseconds
100.00% <= 4 milliseconds
972100.75 requests per second

====== GET ======
  10000000 requests completed in 19.97 seconds
  50 parallel clients
  3 bytes payload
  keep alive: 1

0.14% <= 1 milliseconds
6.35% <= 2 milliseconds
77.66% <= 3 milliseconds
94.62% <= 4 milliseconds
97.04% <= 5 milliseconds
99.35% <= 6 milliseconds
99.76% <= 7 milliseconds
99.94% <= 8 milliseconds
99.99% <= 9 milliseconds
99.99% <= 10 milliseconds
100.00% <= 11 milliseconds
100.00% <= 11 milliseconds
500726.03 requests per second
```

redis-benchmark

`redis-benchmark` is a closed-loop benchmarking tool tha
Ubuntu by installing the `redis-tools` package:

```
sudo apt-get install redis-tools
```

Example invocation:

```
redis-benchmark -h 172.31.0.10 -t set,ping
```

For maximal throughput, use pipelining (`-P`), and the virt

```
redis-benchmark -h 172.31.0.10 -n 10000000 -p 637
```

Redis on Linux

> You'll need a Linux VM image, see the Compare agains
> create one.

Before starting the VM we can re-use the DHCP server co
DHCP server on the host that configures the network of t

```
# Launch a DHCP server (can reuse nrk config)
cd nrk/kernel
sudo dhcpd -f -d tap0 --no-pid -cf ./tests/dhcpd.
```

Next, start the VM. To have the same benchmarks conditi
launch the VM like this (select either e1000 or virtio, gene
emulated e1000):

e1000 NIC:

```
qemu-system-x86_64 \
  --enable-kvm -m 2048 -k en-us --smp 2 \
  -boot d ubuntu-testing.img -nographic \
  -net nic,model=e1000,netdev=n0 \
  -netdev tap,id=n0,script=no,ifname=tap0
```

virtio NIC:

```
qemu-system-x86_64 \
  --enable-kvm -m 2048 -k en-us --smp 2 \
  -boot d ubuntu-testing.img -nographic \
  -net nic,model=virtio,netdev=n0 \
  -netdev tap,id=n0,script=no,ifname=tap0
```

Inside the Linux VM use the following steps to install redis

```
sudo apt install vim git build-essential libjemall
git clone https://github.com/antirez/redis.git
cd redis
git checkout 3.0.6
make
```

Finally, start redis:

```
cd redis/src
rm dump.rdb && ./redis-server
```

Some approximate numbers to expect on a Linux VM anc

e1000, no pipeline:

- SET 50k req/s
- GET 50k req/s

virtio, `-P 29`:

- SET 1.8M req/s
- GET 2.0M req/s

Redis on the rumprun unikernel

Install the toolchain:

```
git clone https://github.com/rumpkernel/rumprun.g
cd rumprun
# Rumprun install
git submodule update --init
./build-rr.sh hw -- -F CFLAGS='-w'
. "/root/rumprun/./obj-amd64-hw/config-PATH.sh"
```

Build the redis unikernel:

```
# Packages install
git clone https://github.com/gz/rumprun-packages.g
cd rumprun-packages

cp config.mk.dist config.mk
vim config.mk

cd redis
make -j8
rumprun-bake hw_generic redis.bin bin/redis-server
```

Run the unikernel

```
# Run using virtio
rumprun kvm -i -M 256 -I if,vioif,'-net tap,script
-W if,inet,dhcp  -b images/data.iso,/data -- redis
# Run using e1000
rumprun kvm -i -M 256 -I if,wm,'-net tap,ifname=ta
nic,model=e1000'  -W if,inet,dhcp  -b images/data.
```

Run the benchmark

```
redis-benchmark -h 172.31.0.10
```

Approximate numbers to expect:

- virtio: PING ~100k req/s
- e1000 PING ~30k req/s

Benchmarking Memcachec

Yet another key--value store written in C, but compared t

Automated integration test

The easiest way to run memcached on nrk, is to invoke th

```
cd kernel
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
```

This test will spawn memcached on one, two and four thr latency with memaslap.

Launch memcached manually

Start the server binary on the VM instance:

```
cd kernel
python3 run.py  \
    --kfeatures test-userspace-smp \
    --cmd 'log=info init=memcached.bin' \
    --nic virtio \
    --mods rkapps \
    --qemu-settings='-m 1024M' \
    --ufeatures 'rkapps:memcached' \
    --release \
    --qemu-cores 4 \
    --verbose
```

As usual, make sure `dhcpd` is running on the host:

```
cd kernel
sudo service apparmor teardown
sudo dhcpd -f -d tap0 --no-pid -cf ./tests/dhcpd.c
```

Start the load-generater on the host:

```
memaslap -s 172.31.0.10 -t 10s -S 10s
```

memaslap: Load generator

memaslap measures throughput and latency of a memca
this:

```
memaslap -s 172.31.0.10:11211 -B -S 1s
```

Explanation of arguments:

- `-B` : Use the binary protocol (faster than the ASCII va
- `-S 1s` : Dump statistics every X seconds

The other defaults arguments the tool assumes are:

- 8 client threads with concurrency of 128 sockets
- 1000000 requests
- SET proportion: 10%
- GET proportion: 90%

> Unfortunately, the memaslap binary does not come wi
> Follow the steps in the CI guide to install it from source

LevelDB

And, yet another key--value store written in C, but this on
(unlike and memcached or Redis).

Automated integration test

The easiest way to run LevelDB on nrk, is to invoke the int

```
cd kernel
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
```

This test will run the db-bench binary for LevelDB which r
the LevelDB process. Our test is configured to create a da
size of 64 KiB, and then perform 100k random lookups. Th
increasing the amount of cores/threads.

Launch dbbench manually

An example invocation to launch the db-bench binary dire

```
python3 run.py --kfeatures test-userspace-smp \
    --cmd "log=info init=dbbench.bin initargs=28 a
benchmarks=fillseq,readrandom --reads=100000 --nur
    --nic virtio --mods rkapps --ufeatures rkapps:
    --release --qemu-cores 28 --qemu-nodes 2 --qem
    --qemu-affinity --qemu-prealloc
```

dbbench example output

Running dbbench should ideally print an output similar to

```
LevelDB:    version 1.18
Keys:       16 bytes each
Values:     100 bytes each (50 bytes after compres
Entries:    1000000
RawSize:    110.6 MB (estimated)
FileSize:   62.9 MB (estimated)
WARNING: Snappy compression is not enabled
------------------------------------------------
fillseq      :       1.810 micros/op;   61.1 MB/s
fillsync     :       0.000 micros/op;    inf MB/s
fillrandom   :       1.350 micros/op;   81.9 MB/s
overwrite    :       2.620 micros/op;   42.2 MB/s
readrandom   :       0.000 micros/op; (1000000 of
readrandom   :      10.440 micros/op; (1000000 of
readseq      :       0.206 micros/op;  537.4 MB/s
readreverse  :       0.364 micros/op;  303.7 MB/s
compact      :   60000.000 micros/op;
readrandom   :       2.100 micros/op; (1000000 of
readseq      :       0.190 micros/op;  582.1 MB/s
readreverse  :       0.301 micros/op;  367.7 MB/s
fill100K     :     390.000 micros/op;  244.6 MB/s
crc32c       :       5.234 micros/op;  746.3 MB/s
snappycomp   :       0.000 micros/op; (snappy fail
snappyuncomp :       0.000 micros/op; (snappy fail
acquireload  :       0.000 micros/op; (each op is
```

Build steps

Some special handling is currently encoded in the build-p dbbench is a C++ program and C++ uses libunwind. Howe Rust also uses libunwind and this leads to duplicate symb C++ toolchain provide it (the non-hacky solution would pr provided unwind symbols).

Implications:

- We have a `-L${RUMPRUN_SYSROOT}/../../obj-amd64` LevelDB Makefile (`$CXX` variable)
- We pass `-Wl,-allow-multiple-definition` to `rump` now defined twice (vibrio and NetBSD unwind lib)

See code in `usr/rkapps/build.rs` which adds flag for thi

If you'll ever find yourself in a situation where you need t not necessary except when debugging build), you can use

```
cd nrk/target/x86_64-nrk-none/<release | debug>/bu
export PATH=`realpath ../../../rumpkernel-$HASH/ou

RUMPRUN_TOOLCHAIN_TUPLE=x86_64-rumprun-netbsd make
RUMPRUN_TOOLCHAIN_TUPLE=x86_64-rumprun-netbsd make
RUMPBAKE_ENV="-Wl,-allow-multiple-definition" RUMI
rumprun-netbsd rumprun-bake nrk_generic ../../../
```

> You might also want to delete the `rm -rf build` and target of the Makefile if you want to call clean to recom

Run LevelDB on the rumprun uniker

Build unikernel:

```
git clone https://github.com/rumpkernel/rumprun.g
cd rumprun
./build-rr.sh hw -- -F CFLAGS='-w'
. "/PATH/TO/config-PATH.sh"
```

Build LevelDB:

```
# Packages install
git clone https://github.com/gz/librettos-packages
cd librettos-packages/leveldb

RUMPRUN_TOOLCHAIN_TUPLE=x86_64-rumprun-netbsd make
RUMPRUN_TOOLCHAIN_TUPLE=x86_64-rumprun-netbsd make
RUMPRUN_TOOLCHAIN_TUPLE=x86_64-rumprun-netbsd rump
bin/db_bench
```

Run it in a VM:

```
rm data.img
mkfs.ext2 data.img 512M
rumprun kvm -i -M 1024 -g '-nographic -display cu
TEST_TMPDIR=/data dbbench.bin
```

Artifact Evaluation

Thank you for your time and picking our paper for the art

This file contains the steps to run experiments used in ou
Replication and Sharing in an Operating System.

> All the experiments run smoothly on the c6420 CloudL
> issues if one tries to run the experiments on some oth
> configuration.

Reserve a cloudlab machine

Please follow the given steps to reserve a machine to run

1. Setup an account on CloudLab, if not already preser
2. Log in to CloudLab and setup a password-less ssh k
3. Start an experiment by clicking on `Experiments` on
4. Use the node type c6420 (by entrying `Optional phy`
   the node with the Ubuntu 20.04 disk image.

Download the code and setup the e

Download and checkout the sources:

```
cd $HOME
git clone https://github.com/vmware-labs/node-repl
cd nrk
git checkout osdi21-ae-v2
bash setup.sh
```

Configure the lab machine

Add password-less sudo capability for your user (scripts r

```
sudo visudo
# Add the following line at the bottom of the file
$YOUR_USERNAME_HERE  ALL=(ALL) NOPASSWD: ALL
```

Add yourself to the KVM group:

```
sudo adduser $USER kvm
```

Disable apparmor, an annoying security feature that bloc
during testing. You can also set-up a rule to allowing this
the test machine:

> Most likely apparmor is not installed if you're using clo
> will fail and you can ignore that.

```
sudo systemctl stop apparmor
sudo systemctl disable apparmor
sudo apt remove --assume-yes --purge apparmor
```

Unfortunately, for apparmor and kvm group changes to t

```
sudo reboot
```

Do a test run

> Note: Most of our benchmarks takes a while to finish, s
> tmux session, or increase the session timeout to avoid

To check if the environment is setup properly, run

```
source $HOME/.cargo/env
cd $HOME/nrk/kernel
python3 ./run.py --release
```

The script downloads needed crates, compiles the OS anc
can take a few minutes).

If everything worked, you should see the following last lin

```
[...]
[DEBUG] - init: Initialized logging
[DEBUG] - init: Done with init tests, if we came h
[SUCCESS]
```

Figure 3: NR-FS vs. tmpfs

Please follow the given steps to reproduce Figure 3 in the

NrFS results

To execute the benchmark, run:

```
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
```

The command runs all NR-FS microbenchmarks and store `fxmark_benchmark.csv`. This step can take a while (~30-6

If everything worked, you should see an output like this o

```
[...]
Invoke QEMU: "python3" "run.py" "--kfeatures" "tes
"log=info initargs=32X8XmixX100" "--nic" "e1000" "
"fxmark" "--release" "--qemu-cores" "32" "--qemu-n
"49152" "--qemu-affinity"
Invoke QEMU: "python3" "run.py" "--kfeatures" "tes
"log=info initargs=32X12XmixX100" "--nic" "e1000"
"fxmark" "--release" "--qemu-cores" "32" "--qemu-n
"49152" "--qemu-affinity"
Invoke QEMU: "python3" "run.py" "--kfeatures" "tes
"log=info initargs=32X16XmixX100" "--nic" "e1000"
"fxmark" "--release" "--qemu-cores" "32" "--qemu-n
"49152" "--qemu-affinity"
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0
finished in 2769.78s
```

Linux tmpfs results

You can also generate the `tmpfs` result on Linux:

```
cd $HOME
git clone https://github.com/gz/vmopsbench
cd vmopsbench
git checkout c011854
bash scripts/run.sh
```

The above command runs the benchmark and writes the
`fsops_benchmark.csv`.

Plot Figure 3

All the plot scripts are in a github repository, execute the

```
cd $HOME
git clone https://github.com/ankit-iitb/plot-scrip
```

To install the required dependencies, run:

```
cd $HOME/plot-scripts
sudo apt install python3-pip
pip3 install -r requirements.txt
```

Plot the Figure 3 by running:

```
# python3 fsops_plot.py <Linux fsops csv> <NrOS fs
python3 fsops_plot.py $HOME/vmopsbench/fsops_bench
$HOME/nrk/kernel/fxmark_benchmark.csv
```

> Arguments given in the plot scripts assume that the re
> run. Please use the argument order given in the comm
> some reason.

Figure 4: LevelDB

Figure 4 in the paper compares LevelDB workload perforr

LevelDB on NrOS

To run the LevelDB benchmark on NrOS execute:

```
cd $HOME/nrk/kernel
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
```

This step will take ~15-20min. If everything worked, you s
the end:

```
[...]
Invoke QEMU: "python3" "run.py" "--kfeatures" "tes
"log=info init=dbbench.bin initargs=32 appcmd=\'--
benchmarks=fillseq,readrandom --reads=100000 --nur
nic" "virtio" "--mods" "rkapps" "--ufeatures" "rka
"--qemu-cores" "32" "--qemu-nodes" "2" "--qemu-mer
"--qemu-prealloc"
readrandom   : done: 3200000,  949492.348 ops/sec;
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0
finished in 738.67s
```

The command runs benchmarks and stores the results in

LevelDB on Linux

To run the LevelDB benchmark on Linux follow the steps
repository in a different path than NrOS.

```
cd $HOME
git clone https://github.com/amytai/leveldb.git
cd leveldb
git checkout 8af5ca6
bash run.sh
```

The above commands run the benchmarks and writes the
`linux_leveldb.csv`.

Plot the LevelDB figure

Make sure that steps to download the plot scripts and ins
already been performed as explained in Plot Figure 3 bef

Run the following commands to plot the Figure 4.

```
cd $HOME/plot-scripts
# python3 leveldb_plot.py <Linux leveldb csv> <Nr(
python3 leveldb_plot.py $HOME/leveldb/linux_levelc
$HOME/nrk/kernel/leveldb_benchmark.csv
```

Figure 5 / 6a / 6c

Figure 5 in the paper compares address-space insertion t
with Linux.

NR-VMem

To run the throughput benchmark (Figure 5) on NrOS exe

```
cd $HOME/nrk/kernel
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
```

This step will take ~3min. If everything worked, you shoul
end:

```
Invoke QEMU: "python3" "run.py" "--kfeatures" "tes
"log=info initargs=32" "--nic" "e1000" "--mods" "-
"--release" "--qemu-cores" "32" "--qemu-nodes" "2'
qemu-affinity"
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0
finished in 118.94s
```

The results will be stored in `vmops_benchmark.csv`.

To run the latency benchmark (Figure 6a) on NrOS execut

```
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
nocapture
```

This step will take ~2min. If everything worked, you shoul
end:

```
Invoke QEMU: "python3" "run.py" "--kfeatures" "tes
"log=info initargs=32" "--nic" "e1000" "--mods" "-
vmops,latency" "--release" "--qemu-cores" "32" "--
"32768" "--qemu-affinity"
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0
finished in 106.67s
```

The results will be stored in `vmops_benchmark_latency.c`

To run the unmap latency benchmark (Figure 6c) on NrOS

```
cd $HOME/nrk/kernel
RUST_TEST_THREADS=1 cargo test --test s10* -- s10_
--nocapture
```

This step will take ~2min. If everything worked, you shoul
end:

> Be aware unmap latency numbers might be impacted

```
Invoke QEMU: "python3" "run.py" "--kfeatures" "tes
"log=info initargs=32" "--nic" "e1000" "--mods" "-
unmaplat,latency" "--release" "--qemu-cores" "32"
memory" "32768" "--qemu-affinity"
ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0
finished in 97.38s
```

The results will be stored in `vmops_unmaplat_benchmark_`

Linux VMem

To run the benchmark on Linux follow the steps below.

```
cd $HOME/vmopsbench
git checkout master
bash scripts/linux.bash throughput
bash scripts/linux.bash latency
bash scripts/linux-tlb.bash latency
```

The results for Figure 5, 6a, and 6c will be store in:

- Figure 5 in `vmops_linux_maponly-isolated-shared_threads_all_throughput_results.csv`
- Figure 6a in `Linux-Map_latency_percentiles.csv`
- Figure 6c in `Linux-Unmap_latency_percentiles.csv`

Plot Figure 5 and 6a and 6c

Go to the plot-scripts repository:

```
cd $HOME/plot-scripts
```

Plot Figure 5:

```
# python3 vmops_throughput_plot.py  <linux vmops c
python3  vmops_throughput_plot.py  $HOME/vmopsbenc
shared_threads_all_throughput_results.csv $HOME/nr
```

Plot Figure 6a:

```
# python3 map_latency_plot.py <linux map-latency c
python3 map_latency_plot.py $HOME/vmopsbench/Linux
$HOME/nrk/kernel/vmops_benchmark_latency.csv
```

Plot Figure 6c:

```
# python3 mapunmap_latency_plot.py <linux unmap-la
csv>
python3 mapunmap_latency_plot.py $HOME/vmopsbench/
Unmap_latency_percentiles.csv
$HOME/nrk/kernel/vmops_unmaplat_benchmark_latency.
```

Baseline Operating System

Contains steps to get other operating systems compiled a

Compare against Linux

To get an idea if nrk is competitive with Linux performanc creating an image. The following steps create an `ubuntu-` ubuntu-minimal installer:

```
wget http://archive.ubuntu.com/ubuntu/dists/bionic
amd64/current/images/netboot/mini.iso
qemu-img create -f vmdk -o size=20G ubuntu-testing
kvm -m 2048 -k en-us --smp 2 --cpu host -cdrom mir
# Follow installer instructions
```

Afterwards the image can be booted using `kvm`:

```
kvm -m 2048 -k en-us --smp 2 -boot d ubuntu-testir
```

Switch to serial output

One step that makes life easier is to enable to serial input graphical QEMU interface. To enable serial, edit the grub follows in the VM:

```
GRUB_CMDLINE_LINUX_DEFAULT=""
GRUB_TERMINAL='serial console'
GRUB_CMDLINE_LINUX="console=tty0 console=ttyS0,115
GRUB_SERIAL_COMMAND="serial --speed=115200 --unit=
```

Then you must run `update-grub` to update the menu ent VM using (not the `-nographic` option):

```
qemu-system-x86_64 --enable-kvm -m 2048 -k en-us -
testing.img -nographic
```

Compare against Barrelfish

TBD.

Compare against sv6

To clone & build the code (needs an older compiler versic

```
git clone https://github.com/aclements/sv6.git
sudo apt-get install gcc-4.8 g++-4.8
CXX=g++-4.8 CC=gcc-4.8 make
```

Update `param.h`:

```
QEMU        ?= qemu-system-x86_64 -enable-kvm
QEMUSMP     ?= 56
QEMUMEM     ?= 24000
```

Run:

```
CXX=g++-4.8 CC=gcc-4.8 make qemu`
```

Rackscale

One of the baselines for rackscale is NrOS. To run the rac
corresponding NrOS baslines, run them with `--feature`

Environment

This chapter contains various notes on configuration and the hypervisor (QEMU) to use either pass-through or emu nrkernel and develop for it.

Install QEMU from sources

Make sure the QEMU version for the account is is >= 6 . Tl build it from scratch, if it the Ubuntu release has a lesser

First, make sure to uncomment all #deb-src lines in /etc/a uncommented. Then, run the following commands:

For any build:

```
sudo apt update
sudo apt install build-essential libpmem-dev libda
apt source qemu
sudo apt build-dep qemu
```

For non-rackscale:

```
wget https://download.qemu.org/qemu-6.0.0.tar.xz
tar xvJf qemu-6.0.0.tar.xz
cd qemu-6.0.0
```

For non-rackscale OR rackscale:

```
git clone https://github.com/hunhoffe/qemu.git qem
cd qemu
git checkout --track origin/dev/ivshmem-numa
```

For any build:

```
./configure --enable-rdma --enable-libpmem
make -j 28
sudo make -j28 install
sudo make rdmacm-mux

# Check version (should be >=6.0.0)
qemu-system-x86_64 --version
```

You can also add `--enable-debug` to the configure script
(useful for source information when stepping through qe

Note that sometimes `make install` doesn't actually repl
`ivshmem-server` to find the current location and then ove
`qemu/build/contrib/ivshmem-server/ivshmem-server`.

Use NVDIMM in QEMU

> **tldr:** The `--qemu-pmem` option in `run.py` will add persi
> want to customize further, read on.

Qemu has support for NVDIMM that is provided by a mer
backend-ram. A simple way to create a vNVDIMM device a
following command-line options:

```
-machine pc,nvdimm
-m $RAM_SIZE,slots=$N,maxmem=$MAX_SIZE
-object memory-backend-file,id=mem1,share=on,mem-
-device nvdimm,id=nvdimm1,memdev=mem1
```

Where,

- the `nvdimm` machine option enables vNVDIMM featu
- `slots=$N` should be equal to or larger than the tota vNVDIMM devices, e.g. $N should be >= 2 here.
- `maxmem=$MAX_SIZE` should be equal to or larger thar and vNVDIMM devices.
- `object memory-backend-file,id=mem1,share=on,me` creates a backend storage of size `$NVDIMM_SIZE` .
- `share=on/off` controls the visibility of guest writes. multiple guests will be visible to each other.
- `device nvdimm,id=nvdimm1,memdev=mem1` creates a whose storage is provided by above memory backer

Guest Data Persistence

Though QEMU supports multiple types of vNVDIMM back can guarantee the guest write persistence is:

- DAX device (e.g., `/dev/dax0.0` ,) or
- DAX file(mounted with dax option)

When using DAX file (A file supporting direct mapping of p
write persistence is guaranteed if the host kernel has sup
mmap system call and additionally, both 'pmem' and 'sha
backend.

NVDIMM Persistence

Users can provide a persistence value to a guest via the o
command line option:

```
-machine pc,accel=kvm,nvdimm,nvdimm-persistence=cp
```

There are currently two valid values for this option:

`mem-ctrl` - The platform supports flushing dirty data fror
NVDIMMs in the event of power loss.

`cpu` - The platform supports flushing dirty data from the
event of power loss.

Emulate PMEM using DRAM

Linux systems allow emulating DRAM as PMEM. These de
Memory Region by the OS. Usually, these devices are fast
not provide any persistence. So, such devices are used or

On Linux, to find the DRAM region that can be used as PM

```
dmesg | grep BIOS-e820
```

The viable region will have "usable" word at the end.

```
[    0.000000] BIOS-e820: [mem 0x0000000100000000-
```

This means that the memory region between 4 GiB (0x00(
(0x000000053fffffff) is usable. Say we want to reserve a 1(
need to add this information to the grub configuration file

```
sudo vi /etc/default/grub
GRUB_CMDLINE_LINUX="memmap=16G!4G"
sudo update-grub2
```

After rebooting with our new kernel parameter, the `dmes` persistent memory region like the following:

```
[    0.000000] user: [mem 0x0000000100000000-0x000
12)
```

We will see this reserved memory range as `/dev/pmem0` . I ready to use. Mount it with the dax option.

```
sudo mkdir /mnt/pmem0
sudo mkfs.ext4 /dev/pmem0
sudo mount -o dax /dev/pmem0 /mnt/pmem0
```

Use it as a `mem-path=/mnt/pmem0` as explained earlier.

Configure and Provision NVDIMMs

The NVDIMMs need to be configured and provisioned be Intel `ipmctl` tool can be used to discover and provision t

To show all the NVDIMMs attached to the machine, run:

```
sudo ipmctl show -dimm
```

To show all the NVDIMMs attached on a socket, run:

```
sudo ipmctl show -dimm -socket SocketID
```

Provisioning

NVDIMMs can be configured both in volatile (MemoryMo modes or a mix of two using `ipmctl` tool on Linux.

We are only interested in using the NVDIMMs in AppDirec NVDIMMs can be configured in two ways; AppDirect and

mode, the data is interleaved across multiple DIMMs, and
AppDirectNotInterleaved is used. To configure multiple D
run:

```
sudo ipmctl create -goal PersistentMemoryType=AppD
```

Reboot the machine to reflect the changes made using th
creates a region on each socket on the machine.

```
ndctl show --regions
```

To show the DIMMs included in each region, run:

```
ndctl show --regions --dimms
```

Each region can be divided in one or more namespaces in
the operating system. To create the namespace(s), run:

```
sudo ndctl create-namespace --mode=[raw/sector/fsd
```

The namespace can be created in different modes like raw
default mode is fsdax.

Reboot the machine after creating the namespaces, and t
/dev/*depending on the mode. For example, if the mode is fsd*
/dev/pmem.

Mount these devices:

```
sudo mkdir /mnt/pmem0
sudo mkfs.ext4 /dev/pmem0
sudo mount -o dax /dev/pmem0 /mnt/pmem0
```

These mount points can be used directly in the userspace
machine as explained earlier.

Use RDMA support in QEM

> **tldr:** The `-pvrdma` option in `run.py` will enable RDMA
> manually have to run `rdmacm-mux` and unload the Mell

QEMU has support for `pvrdma` (a para-virtual RDMA drive cards (like Mellanox). In order to use it (aside from the `--` `rdmacm-mux` during building), the following steps are nece

Install Mellanox drivers (or any other native drivers for yo

```
wget https://content.mellanox.com/ofed/MLNX_OFED-5
1.0.3.0-ubuntu20.04-x86_64.tgz
tar zxvf MLNX_OFED_LINUX-5.4-1.0.3.0-ubuntu20.04-x
cd MLNX_OFED_LINUX-5.4-1.0.3.0-ubuntu20.04-x86_64
./mlnxofedinstall --all
```

Before running the rdmacm-mux make sure that both ib_ aren't loaded, otherwise the rdmacm-mux service will fail

```
sudo rmmod ib_ipoib
sudo rmmod rdma_cm
sudo rmmod ib_cm
```

Start the QEMU `racadm-mux` utility (before launching a qe

```
./rdmacm-mux -d mlx5_0 -p 0
```

Inter-VM Communication u memory

> **tldr:** Use the `--qemu-ivshmem` and `--qemu-shmem-path` VM shared-memory support in QEMU.

This section describes how to use shared memory to com First, create a shared memory file (with hugepages):

```
echo 1024 > /sys/kernel/mm/hugepages/hugepages-204
sudo mkdir -p /mnt/hugepages
sudo mount -t hugetlbfs pagesize=2GB /mnt/huge
sudo chmod 777 /mnt/hugepages
```

Now, use a file on this mount point to create a shared me shared memory, Qemu allows two types of configuration

- Just the shared memory file: `ivshmem-plain`.
- Shared memory plus interrupts: `ivshmem-doorbell`

We use the plain shared memory configuration as the go machines. Add the following parameters to the Qemu co

```
-object memory-backend-file,size=2G,mem-path=/mnt,
file,share=on,id=HMB \
-device ivshmem-plain,memdev=HMB
```

Discover the shared memory file ins

Qemu exposes the shared memory file to the kernel by c run the following command to discover to check if the PC

```
lspci | grep "shared memory"
```

Running lspci should show something like:

```
00:04.0 RAM memory: Red Hat, Inc. Inter-VM shared
```

Use the `lspci` command to know more about the PCI de

```
lspci -s 00:04.0 -nvv
```

This should print the BAR registers related information. T
three BARs (depending on shared memory or interrupt de

- BAR0 holds device registers (256 Byte MMIO)
- BAR1 holds MSI-X table and PBA (only ivshmem-doo
- BAR2 maps the shared memory object

Since we are using the plain shared memory configuratio
the BAR0 and BAR2 as Region 0 and Region 1.

```
00:04.0 0500: 1af4:1110 (rev 01)
        Subsystem: 1af4:1100
        Physical Slot: 4
        Control: I/O+ Mem+ BusMaster- SpecCycle- M
Stepping- SERR+ FastB2B- DisINTx-
        Status: Cap- 66MHz- UDF- FastB2B- ParErr-
<MAbort- >SERR- <PERR- INTx-
        Region 0: Memory at febf1000 (32-bit, non-
        Region 2: Memory at 280000000 (64-bit, pre
```

Use the shared memory file inside C

If you only need the shared memory part, BAR2 suffices. T
shared memory in the guest and can use it as you see fit.
the shared memory is at `280000000` with the size of `2G`.

Here is a sample C program that writes to the shared mem

```
#include<stdio.h>
#include<stdint.h>
#include<unistd.h>
#include<fcntl.h>
#include<sys/mman.h>

int main() {
    void *baseaddr = (void *) 0x280000000; // BAR2
    uint64_t size = 2147483648; // BAR2 size

    int fd = open("/sys/bus/pci/devices/0000:00:04
    void *retaddr = mmap(baseaddr, size, PROT_READ
0);
    if (addr == MAP_FAILED) {
        printf("mmap failed");
        return 0;
    }

    uint8_t *addr = (uint8_t *)retaddr;
    addr[0] = 0xa;

    munmap(retaddr, size);
    close(fd);
}
```

Compile and run the program (use `sudo` to run).

Perform the similar steps to read the shared memory file

Discover CXL devices in Lin

This document aims to list out steps to discover CXL type
Since there is no hardware available, the only way to achi
Unfortunately, even the Qemu mainstream branch does r
tutorial uses a custom version of Qemu that supports CXL

Build custom Qemu version

First, download and build the custom Qemu version on yo

```
sudo apt install build-essential libpmem-dev libda
cd ~/cxl
git clone https://gitlab.com/bwidawsk/qemu.git
cd qemu
git checkout cxl-2.0v4
./configure --enable-libpmem
make -j 16
```

Check the version:

```
./build/qemu-system-x86_64 --version
```

```
QEMU emulator version 6.0.50 (v6.0.0-930-g1839565
Copyright (c) 2003-2021 Fabrice Bellard and the QE
```

Build custom Linux Kernel

Next, download the latest kernel version and build an ima

```
cd ~/cxl
git clone https://git.kernel.org/pub/scm/linux/ker
cd linux
make defconfig
```

`defconfig` generates default configuration values and st
kernel requires some special configuration changes to ha
these configuration flags are present in the `.config` file,
these flags.

```
CONFIG_ACPI_HMAT=y
CONFIG_ACPI_APEI_PCIEAER=y
CONFIG_ACPI_HOTPLUG_MEMORY=y
CONFIG_MEMORY_HOTPLUG=y
CONFIG_MEMORY_HOTPLUG_DEFAULT_ONLINE=y
CONFIG_MEMORY_HOTREMOVE=y
CONFIG_CXL_BUS=y
CONFIG_CXL_MEM=y
```

Once the configuration related changes are done, compil
image file.

```
make -j 16
cd ~/cxl
```

The image file should be in `linux/arch/x86_64/boot/bzI`

Run Qemu with CXL related parame

Qemu provides `-kernel` parameter to use the kernel ima

```
qemu/build/qemu-system-x86_64 -kernel linux/arch/x
 -nographic -append "console=ttyS0" -m 1024 --enab
```

The kernel runs until it tries to find the root fs; then it cras
to create a ramdisk.

```
mkinitramfs -o ramdisk.img
```

Press `Ctrl+a` and `c` to exit kernel and then press `q` t

Now, run qemu with the newly created ramdisk:

```
qemu/build/qemu-system-x86_64 -kernel linux/arch/x
    -append "console=ttyS0" -m 1024 -initrd ramdis
```

The kernel runs properly this time. Now, it is time to add
running Qemu:

```
qemu/build/qemu-system-x86_64 -kernel linux/arch/x
    -append "console=ttyS0" -initrd ramdisk.img -e
    -m 1024,slots=12,maxmem=16G -M q35,accel=kvm,c
    -object memory-backend-file,id=cxl-mem1,share=
window1,size=512M \
 -object memory-backend-file,id=cxl-label1,share=c
 -object memory-backend-file,id=cxl-label2,share=c
 -device pxb-cxl,id=cxl.0,bus=pcie.0,bus_nr=52,uid
base[0]=0x4c00000000,memdev[0]=cxl-mem1 \
 -device cxl-rp,id=rp0,bus=cxl.0,addr=0.0,chassis=
 -device cxl-rp,id=rp1,bus=cxl.0,addr=1.0,chassis=
 -device cxl-type3,bus=rp0,memdev=cxl-mem1,id=cxl-
 -device cxl-type3,bus=rp1,memdev=cxl-mem1,id=cxl-
```

Qemu exposes the CXL devices to the kernel and the kern
devices by running:

```
ls /sys/bus/cxl/devices/
```

or

```
dmesg | grep '3[45]:00'
```

References

- Booting a custom linux kernel in Qemu
- CXL 2.0 support in Linux
- CXL 2.0 + Linux + Qemu

Continuous integration (CI

We run tests using the github-actions infrastructure. The
up a new runner machine (and connect a github repo to i

Steps to add a new CI machine:

1. Install github-runner software on a new test machin
2. Give access to the benchmark repository
3. Configure software for the `github-runner` account
4. Disable AppArmor
5. Install a recent QEMU
6. Do a test-run
7. Start the new runner

Install github-runner software on a

Create a github-runner user first:

```
sudo useradd github-runner -m -s /bin/zsh
```

Add sudo capability for github-runner:

```
sudo visudo
# github-runner  ALL=(ALL) NOPASSWD: ALL
```

> For better security with self-hosted code exeuction, ma
> `Actions -> Runners -> Require approval for all ou`
> github repo settings!

Other than that, follow the steps listed under `Settings -`
`runner`:

```
sudo su github-runner
cd $HOME
<< steps from Web-UI >>
```

When asked for labels, make sure to give it a machine spe
use the following labels `skylake2x`, `skylake4x`, `cascade`

machine type and the number of sockets/NUMA nodes. M
should have the same tag to allow parallel test execution.

> If you add a new machine label, make sure to also add
> `_scripts` folder.

Don't launch the runner yet with `run.sh` (this happens fu

Give access to the benchmark repos

Benchmark results are uploaded automatically to git.

Generate a key for accessing the repository or use an exis
account. Also add the user to the KVM group. Adding you
logout/reboot which we do in later steps.

```
sudo adduser github-runner kvm
ssh-keygen
```

Then, add the pub key (`.ssh/id_rsa.pub`) to the github C

Configure software for the github ru

Install necessary software for use by the runner:

```
git clone git@github.com:vmware-labs/node-replicat
cd nrk/
bash setup.sh
source $HOME/.cargo/env
```

Install a recent qemu

Follow the steps in the Environment chapter.

Install memaslap

The memcached benchmark uses the `memaslap` binary th
not included in the Ubuntu libmemcached-tools deb pack
from the sources:

```
cd $HOME
sudo apt-get build-dep libmemcached-tools
wget https://launchpad.net/libmemcached/1.0/1.0.18
1.0.18.tar.gz
tar zxvf libmemcached-1.0.18.tar.gz

cd libmemcached-1.0.18/
LDFLAGS='-lpthread' CPPFLAGS='-fcommon -fpermissiv
fcommon' ./configure --enable-memaslap
CPPFLAGS='-fcommon' make -j12
sudo make install
sudo ldconfig

which memaslap
```

Disable AppArmor

An annoying security feature that blocks our DHCP server
up a rule for allowing this but it's easiest to just get rid of

```
sudo systemctl stop apparmor
sudo systemctl disable apparmor
sudo apt remove --assume-yes --purge apparmor
# Unfortunately for apparmor and kvm group changes
reboot:
sudo reboot
```

Do a test-run

After the reboot, verify that the nrk tests pass (this will tal
succeed too):

```
# Init submodules if not done so already:
cd nrk
git submodule update --init
source $HOME/.cargo/env

cd kernel
RUST_TEST_THREADS=1 cargo test --features smoke --
```

Start the runner

Finally, launch the runner:

```
cd $HOME/actions-runner
source $HOME/.cargo/env
./run.sh
```

Start runner as systemd service

```
cd $HOME/actions-runner
sudo ./svc.sh install
sudo ./svc.sh start
```

Check the runner status with:

```
sudo ./svc.sh status
```

Stop the runner with:

```
sudo ./svc.sh stop
```

Uninstall the service with:

```
sudo ./svc.sh uninstall
```

Repository settings

If the repo is migrated to a new location, the following set

1. Under Settings -> Secrets: Add secret `WEBSITE_DEPL` push the generated documentation to the correct w
2. Under Settings -> Options: Disable "Allow merge cor
3. Under Settings -> Branches: Add branch protection settings:
   - Require pull request reviews before merging
   - Dismiss stale pull request approvals when new
   - Require status checks to pass before merging
   - Require branches to be up to date before merg
   - Require linear history
4. Under Settings -> Actions -> Runners:
   - "Require approval for all outside collaborators'

Related Work

NRK takes inspiration from decades of academic researcl
exhaustive:

Operating Systems

- Barrelfish
- Tornado
- K42
- sv6
- Rumpkernel
- LibrettOS
- seL4
- Disco
- Mitosis

Scalable Data structures

- Flat combining
- Read-Log-Update and RLU with multi-versioning
- Predictive Log Synchronization
- OpLog

Log based designs

- ScaleFS
- Corfu
- Raft

Contributors

Here is a list of the contributors who helped to build NRK

- Amy Tai
- Ankit Bhardwaj
- Chinmay Kulkarni
- Christian Menges
- Erika Hunhoff
- Gerd Zellweger
- Irina Calciu
- Reto Achermann
- Ryan Stutsman
- Sanidhya Kashyap
- Stanko Novakovic
- Zack McKevitt

If you feel you're missing from this list, feel free to add yo