Enoki: High Velocity Linux Kernel Scheduler Development

Samantha Miller
University of Washington

Anirudh Kumar
University of Washington

Tanay Vakharia
University of Washington

Ang Chen
University of Michigan

Danyang Zhuo
Duke University

Thomas Anderson
University of Washington

Abstract

Kernel task scheduling is important for application performance, adaptability to new hardware, and complex user requirements. However, developing, testing, and debugging new scheduling algorithms in Linux, the most widely used cloud operating system, is slow and difficult. We developed Enoki, a framework for high velocity development of Linux kernel schedulers. Enoki schedulers are written in safe Rust, and the system supports live upgrade of new scheduling policies into the kernel, userspace debugging, and bidirectional communication with applications. A scheduler implemented with Enoki achieved near identical performance (within 1% on average) to the default Linux scheduler CFS on a wide range of benchmarks. Enoki is also able to support a range of research schedulers, specifically the Shinjuku scheduler, a locality aware scheduler, and the Arachne core arbiter, with good performance.

CCS Concepts: • **Software and its engineering → Scheduling**; **Agile software development**.

Keywords: scheduling, kernel development, development velocity

ACM Reference Format:
Samantha Miller, Anirudh Kumar, Tanay Vakharia, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2024. Enoki: High Velocity Linux Kernel Scheduler Development . In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3627703.3629569

1 Introduction

Kernel scheduler behavior is central to application performance, adaptability to new hardware, and complex user requirements. Many applications have short, latency sensitive tasks and bursty workloads, where suboptimal kernel decisions can lead to high tail latency and poor overall job performance [23, 34, 58]. Heterogeneous hardware, such as non-uniform and tiered memory or accelerators, increases the complexity of scheduling decisions [8, 56]. Energy use is also increasingly important; neither underloading nor fully loading a server provides peak server energy efficiency [9, 36]. Additionally, applications may have information that can help improve scheduling decisions, such as workload characteristics or hardware preferences, but in most kernels the scheduler is oblivious to user preferences beyond simple priorities and hand coded placement [55, 63].

Efficiently supporting these changes will require new scheduler designs or new features in existing schedulers. Although kernel schedulers could theoretically be adapted to handle these new demands, developing and testing new kernel schedulers can be difficult and time consuming. Kernel code is difficult to write correctly and debug and slow to deploy. There are only three mainline schedulers implemented in the Linux kernel, the most widely used cloud operating system.

To address this, some researchers have used kernel bypass to implement new schedulers [23, 34, 58, 64]. This approach increases development velocity by removing the need to recompile the kernel and providing access to userspace debugging tools. However, it interferes with resource sharing between the scheduler and the rest of the system and complicates deployment and maintenance, limiting the potential reach of the research [68].

GhOSt [32] aims to provide a general purpose deployable solution for userspace schedulers in Linux. GhOSt uses an upcall approach where scheduling policy decisions are made by a userspace scheduler while the mechanism remains in the kernel. GhOSt schedulers can be implemented in small amounts of userspace code and redeployed easily, but each scheduling decision requires scheduling the userspace scheduler, adding significant overhead to scheduling decisions. To mitigate some of the latency overhead, ghOSt uses an asynchronous model where the kernel can continue to take interrupts and make scheduling decisions while the userspace scheduler runs. This means the scheduling decisions may be out of date.

Other kernel subsystems, particularly networking, can use eBPF [26] for high velocity kernel development, and ghOSt implemented support for scheduler eBPF hooks. Using eBPF,

ACM ISBN 979-8-4007-0437-6/24/04.
https://doi.org/10.1145/3627703.3629569

users can load programs to customize kernel scheduler code, provided that the structure of the scheduler does not change. It is difficult to implement large or complicated code, such as an entire scheduler, using eBPF. For example, the default Linux scheduler, the Completely Fair Scheduler, is over 6,000 lines of code. Additionally, the eBPF trust model is not a good match for scheduling. eBPF considers loaded programs potentially malicious and verifies that they will not disrupt kernel execution. It is not clear how to implement this for scheduling since bad scheduling policy decisions, particularly choosing to run a task on a CPU where it is not queued, can cause the kernel to crash, violating eBPF's safety requirements.

Previously, our prior work Bento [49] has shown that we can have high development velocity and high performance for Linux kernel file systems by writing file systems in safe Rust and supporting live upgrade and userspace debugging. In our prior work, we found that Linux kernel development velocity is affected by the prevalence of bugs, the difficulty of debugging, and the disruption caused by redeployment. New code means new bugs. Bugs in the kernel can have catastrophic consequences on the kernel's functionality, making developers cautious to introduce new code. Once code has been introduced, it is difficult to debug. Kernel code is difficult to debug due to the lack of debugging tools, further increasing the cost of kernel development. And even after kernel code is stable, deploying it typically involves rebooting the machine or killing processes. Public cloud providers limit the rate of rebooting to meet reliability SLAs. Bento enables file systems written entirely in safe Rust, eliminating whole classes of bugs at compile time, particularly those that are most likely to crash the kernel. Bento file systems can be run in both the kernel and userspace using a synchronous kernel to userspace trampoline method, allowing developers to debug their code using userspace tools. Bento also enables live redeployment of file systems with only a short 15 ms pause in service availability. A file system written with Bento performed competitively to ext4, the default Linux file system.

However, file systems and schedulers have different environments and performance tolerances, and the techniques from Bento cannot be applied directly to schedulers. Bento sits behind the file page cache, reducing the cost of interposition. It is not clear if Bento's design is low enough latency for scheduling. Schedulers cannot be run in userspace using the same type of synchronous trampoline as Bento because the scheduler code cannot block waiting for a response from userspace. While a 15 ms pause during redeployment is short for file systems, it would be too long to pause scheduling. It is also not obvious if Bento supports code that is general enough to meet the demands of scheduler designs. Schedulers are more performance critical than file systems, and may need to rely on specific data structure or algorithm designs to ensure performance. Additionally, Bento does not address logical correctness errors. In the file system, non-memory safety correctness errors will generally affect only the applications using the file system, while correctness errors in the scheduler code can crash the kernel.

Inspired by Bento, we built a framework called Enoki. The goal of Enoki is to enable high velocity development and deployment of high performance schedulers in the Linux kernel. We envision that Enoki schedulers will be used for both research prototyping and production deployments. Enoki schedulers are fast to write and debug and easy to test with seamless resource sharing with the rest of the kernel. Enoki supports schedulers running in the Linux kernel for wide deployability, but our approach is not restricted to the Linux kernel. A new Enoki scheduler is written in safe Rust against a clean interface, making it less likely to introduce bugs. Enoki enables dynamic update of scheduling code in a live kernel without rebooting, with a pause of only $10\mu s$. Using a record and replay system, scheduling policies can be debugged using userspace tools. Since Enoki schedulers are implemented in the kernel, they can coordinate easily with other kernel schedulers, such as passing cores between schedulers or applications.

We used Enoki to implement several different scheduling algorithms to demonstrate its flexibility in supporting a variety of scheduler designs and to understand the overhead and performance of Enoki compared to native implementations of the same schedulers. We implemented a weighted fair queuing scheduler and evaluated it against Linux's default scheduler CFS. Despite our scheduler being much simpler than CFS, and including the Enoki framework overhead, it achieves an average of only 0.74% slowdown across 36 application benchmarks, with a maximum slowdown of 8.57%. We also implemented the Shinjuku [34] scheduler, a locality aware scheduler that co-locates tasks of the same user-defined class, and the core arbiter from the Arachne[64] multilevel thread scheduler. The Enoki Shinjuku and Arachne schedulers performed competitively with native implementations, and the locality aware scheduler showed the potential to provide significant performance benefit. Two of the schedulers were implemented by undergraduates with no prior Linux kernel programming experience. The code is publicly available at https://github.com/smiller123/enoki.

2 Motivation and Approach

Linux includes three schedulers: a real time scheduler, an earliest deadline first scheduler, and the Completely Fair Scheduler (CFS). CFS is the default and implements a version of weighted fair queuing. These schedulers are all quite large and complex; CFS is over 6000 lines of code, and even the simpler real time and deadline schedulers are over 1500 lines of code. This complexity has led to a number of bugs, particularly performance bugs due to a complicated load balancing mechanism [43].

Although CFS's weighted fair queuing algorithm works well for many of the tasks run on desktop or server machines. Other schedulers can have advantages in some cases. With more application knowledge, more optimal decisions are possible. For example, for workloads composed of many very small tasks, shorter time slices can lower average job completion time [34]. Multilevel scheduling can give better performance isolation and flexibility by allowing applications to define their own policies based on their workloads [64]. Nest [36] improves energy efficiency for jobs with fewer tasks than cores by reusing warm cores rather than spreading tasks across many cold cores. Because these schedulers do not need to work well in all circumstances, they can potentially be much smaller and simpler than CFS. Nevertheless, uptake into Linux has been slow.

We propose Enoki, a framework for high velocity development of Linux kernel schedulers. Enoki's trust model assumes that scheduler developers are trusted but clumsy; they are not trying to break the kernel but may accidentally introduce bugs. Our overall goal is to allow non-expert programmers to be able to successfully design, implement, debug, and deploy new schedulers.

There are several challenges to achieving high development velocity for Linux kernel schedulers:

- Buggy code: The Linux kernel is a monolith of complicated C code, causing bugs to be common. The lack of modularity in the Linux kernel and the potentially large consequences of kernel bugs force developers to code slowly and carefully. In Enoki, we rely on safe language features. By enabling schedulers implemented entirely in safe Rust and by introducing a novel application of the type system to check for scheduling correctness errors, we eliminate whole classes of bugs, such as memory errors and race conditions, without introducing significant performance overhead or overly limiting scheduler designs [50].
- Limited interaction: Some scheduler designs, such as two level schedulers that require coordination with user threads, are difficult to implement in Linux because they require interaction with the userspace tasks that Linux does not support. By allowing generic, scheduler-defined interactions between the scheduler and userspace tasks, Enoki can support scheduler designs that cannot be implemented in Linux today, such as those that require application specific hints or two level schedulers.
- Disruptive upgrade: Current Linux schedulers are compiled into the kernel source, so deploying a new scheduler requires recompiling the kernel and rebooting the machine, decreasing the availability of running applications. Enoki schedulers can be upgraded quickly, without rebooting the machine and with only a short period of service downtime.
- Slow debugging: The kernel does not have access to debugging tools, such as those commonly used in userspace. We support record and replay debugging in Enoki. To diagnose bugs not caught by the Enoki framework, developers can record the calls between the kernel and the scheduler and replay later entirely at userspace.
- Resource sharing: Schedulers should be able to share resources with the rest of the system. In Linux, different applications can use different schedulers, sharing cores and cycles between the schedulers. Enoki schedulers are implemented in the kernel to enable fine grained core sharing across applications and schedulers.

3 Enoki

The high level overview of Enoki is shown in Figure 1. Enoki is composed of two major components, Enoki-C and libEnoki.

Part of Enoki, Enoki-C is implemented in C and compiled into the Linux kernel. It interfaces directly with the core scheduling code and the kernel scheduling data structures. Enoki-C handles registering, deregistering, and upgrading schedulers and sets up and manages infrastructure for communication channels between userspace and the kernel scheduler and the record and replay system. It handles the unsafe work that is required for scheduling on behalf of Enoki schedulers, such as performing state updates to kernel `task_struct` data structure, managing interactions with the kernel run-queues when adding or moving tasks, and manipulating raw pointers to read and pass data to the scheduler module. Enoki-C also translates the calls from the core scheduling code into calls that the scheduler can implement safely by ensuring that all data passed to the scheduler can be safely accessed. Enoki schedulers do not directly manipulate kernel state or run-queues.

The other component, libEnoki, is a Rust library that is compiled with the scheduler code into a module that is dynamically loaded into the kernel. This library provides safe interfaces so the scheduler code can access the kernel and implements functions for loading and managing the scheduler. It contains some unsafe Rust because it must handle interactions with the C code in Enoki-C, which is inherently unsafe. Each scheduler is written entirely in safe Rust and only needs to provide the logic for the scheduling algorithm. The scheduler module is not sandboxed further; once it is loaded into the kernel, it runs like any other kernel code.

When the scheduler module is loaded, libEnoki calls Enoki-C to register the newly available scheduler. This registers the ID of the scheduler being loaded and a processing function in libEnoki for parsing calls from Enoki. User tasks can switch to using the new scheduler using its defined ID value. During regular operation, Enoki-C processes calls from the core scheduler code for these tasks, forwarding the calls to the processing function in libEnoki and managing updates to kernel data structures, such as the CPU's run queue. When the module is unloaded, libEnoki similarly unregisters the scheduler with Enoki-C, and no new tasks can be attached to the scheduler.

3.1 Safe Interfaces

Enoki provides schedulers with safe interfaces, both the interface that schedulers are required to implement and the interfaces for schedulers to access kernel functionality, such as locks and timers. With safe interfaces, Enoki schedulers can be implemented entirely in safe Rust, preventing whole

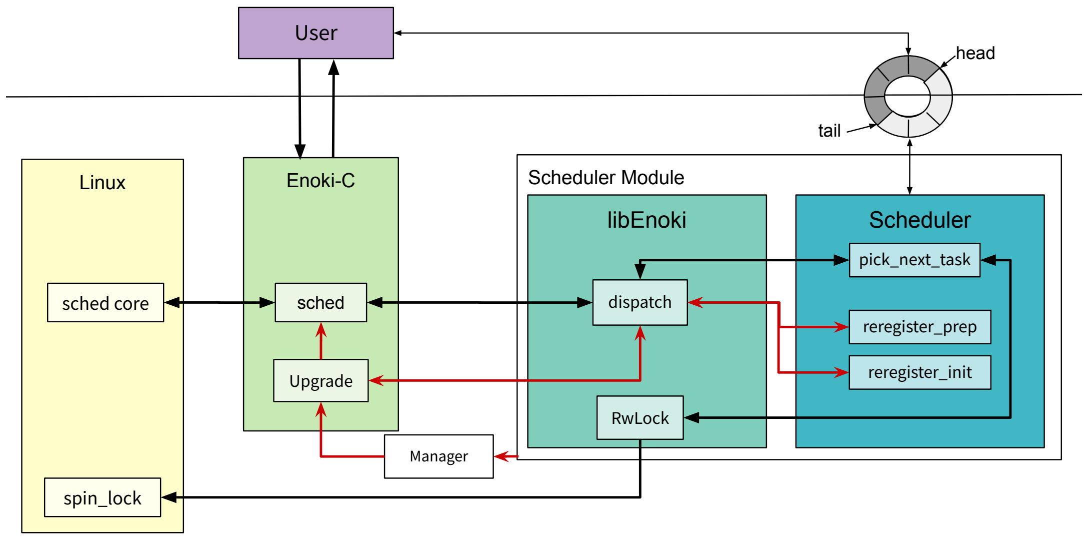


Figure 1. A high level diagram Enoki. Enoki-C is written in C and compiled into the kernel. It interacts with libEnoki, a Rust library that is compiled with the scheduler code into the dynamically loaded scheduler module. The black lines represent code pathways during normal execution. Red lines represent module insertion and upgrade.

classes of bugs and reducing time spent debugging new schedulers. Enoki-C and libEnoki work together to provide safe interfaces for the scheduler code.

Enoki's threat model assumes that the developer is well meaning and knowledgeable but makes occasional mistakes. Enoki aims to help this developer prevent these mistakes by catching low-level errors that do not depend on the scheduler behavior, such as NULL pointer dereferences and data races, at compile time. Enoki does not aim to prevent all bugs, and bugs that depend on the scheduler's semantic behavior can remain uncaught. For example, schedulers implemented with Enoki can deadlock, lose tasks, and violate work conservation. We attempt to catch as many of these bugs as we can at runtime, but cannot guarantee that all instances are caught. We will discuss our mechanism for catching some of the semantic bugs later when discussing the API.

Enoki-C takes the interface defined by the core scheduler code and translates it into an interface based on message passing. Like the core scheduler code, this is a synchronous interface; it consists of function calls where the caller waits for the the callee to return before progressing. It is not necessary for safety that the interface be based on message passing, but it helps enforce certain safety properties, such as no shared pointers, preventing memory bugs across the interface. Enoki-C handles direct interactions with kernel data structures, such as pulling information like the runtime of the task or the current CPU of the task. This information is placed into per-function type "message" data structures that are passed to the registered processing function in libEnoki.

The processing function in libEnoki parses each "message" to determine which scheduler function is being invoked and handles the unsafety of interacting directly with C code through the Rust FFI (Foreign Function Interface) layer. The processing function pulls the fields from the "message" and passes them to the scheduler function being called. If the function returns a value, libEnoki writes that value back into the "message" data structure to return the value to Enoki-C.

To enforce that the scheduler code implements the required behavior, libEnoki provides a Rust trait for scheduler types. This trait defines the functions that a scheduler module must implement to be loadable as an Enoki scheduler. A trait defines a set of functionality that a type must have to implement the trait and can provide default implementations for shared behavior on those types. Traits can be used as bounds on function arguments, and any type that implements a trait can be used where the trait is called for.

The EnokiScheduler trait (shown in Table 1) specifies the functions that a scheduler should provide. Most of these functions are very similar to the functions defined by the core scheduling code to implement a Linux scheduler, with some notable differences.

The core function for a scheduler is `pick_next_task` which tells the core kernel scheduler code which task should be run next. Other important functions are `task_new`, `task_wakeup` and similar functions for tracking task state, `migrate_task_rq` for moving tasks between cores, `balance` for telling the core scheduler to move tasks to rebalance load, and error functions to return error values from the kernel.

For example, consider a simple scheduler that keeps a queue of tasks assigned to each core and schedules these tasks first come, first serve on each core. When a task is created on this scheduler, the kernel calls `select_task_rq`. The scheduler returns the core the task should be assigned to. The kernel

API Function	Description
`get_policy(&self) → i32`	Get the policy number.
`pick_next_task(&self, ..., sched: Option⟨Schedulable⟩) → Option⟨Schedulable⟩`	Pick the next task for the CPU.
`pnt_err(&self, ..., sched: Option⟨Schedulable⟩)`	Could not schedule chosen task.
`task_dead(&self, pid)`	A task died.
`task_blocked(&self, ...)`	A task blocked.
`task_wakeup(&self, ..., sched: Schedulable)`	A task woke up.
`task_new(&self, ..., sched: Schedulable)`	There is a new task.
`task_preempt(&self, ..., sched: Schedulable)`	A task was preempted.
`task_yield(&self, ..., sched: Schedulable)`	A task yielded.
`task_departed(&self, ...) → Schedulable`	A task left the scheduler.
`task_affinity_changed(&self,..)`	Change task's allowed CPUs.
`task_prio_changed(&self,..)`	Change task's priority.
`task_tick(&self, ...)`	A timer has triggered.
`select_task_rq(&self, ...) → i32`	Choose the CPU for a task.
`migrate_task_rq(&self, ..., sched: Schedulable) → Schedulable`	A task is moving CPUs.
`balance(&self, ...) → Option ⟨u64⟩`	Rebalance tasks onto CPU.
`balance_err(&self, ..., sched: Option⟨Schedulable⟩)`	Could not move the chosen task.
`reregister_prepare(&mut self) → Option⟨TransferOut⟩`	Prepare for an upgrade.
`reregister_init(&mut self, Option⟨TransferIn⟩)`	Initialize during an upgrade.
`register_queue(&self, RingBuffer⟨UserMessage⟩) -> i32`	Register a user to kernel hint queue.
`register_reverse_queue(&self, RingBuffer⟨RevMessage⟩) -> i32`	Register a kernel to user queue.
`enter_queue(&self, id: i32, ...)`	Check the user hint queue.
`unregister_queue(&self, id: i32) → RingBuffer⟨UserMessage⟩)`	Unregister the user to kernel hint queue.
`unregister_rev_queue(&self, id: i32) → RingBuffer⟨RevMessage⟩)`	Unregister the kernel to user queue.
`parse_hint(&self, hint: UserMessage)`	Synchronously parse hint.

Table 1. The API of the EnokiScheduler Trait. This is the API that a scheduler module must implement to be loadable as an Enoki scheduler. Most of the functions are used for managing the state of tasks in the scheduler. The `reregister` functions handle live upgrade. The queue functions and `parse_hint` are for user-to-kernel communication, and the reverse queue functions are for kernel-to-user communication.

attaches the task to the core and calls `task_new`. The scheduler places this task at the back of the queue for the specified core, storing the `Schedulable` (discussed more later). When a core becomes idle, the kernel calls `pick_next_task` for that core. The scheduler pops the first task off the core's queue and returns the `Schedulable` for the task. The kernel switches to the task and run it. If there is an error, the kernel calls `pnt_err`. If the task blocks, the kernel calls `task_blocked`. When the task wakes up, the kernel calls `task_wakeup`, and the scheduler adds the task to the back of the wakeup core's queue. Periodically, the kernel calls the scheduler's `balance` function on each core so the scheduler can rebalance tasks across cores. The scheduler returns the ID of any tasks it wants moved to the balancing core. The kernel then tries to move the task to the specified core. If it succeeds, the kernel calls `migrate_task_rq`, and the scheduler moves the task to the new core's queue. If it fails, the kernel calls `balance_err`.

While these functions are modeled after the Linux scheduler interface they could be adapted to a different operating system. An Enoki scheduler is only expected to manage its own state in response to these calls; the kernel's core scheduling code decides when to call each function and Enoki-C manages kernel state.

In Linux, schedulers are expected to track their tasks' runtimes using kernel timing functions. In our system, Enoki-C tracks the runtime of tasks on behalf of the scheduler and passes this to the scheduler when the task state changes, such as blocking, waking, and yielding, and to `pick_next_task`. Enoki records this information for deterministic replay.

The `pick_next_task` function in Linux expects the scheduler to choose a task on the CPU's run-queue, and if this expectation is violated, the kernel can crash. While this is a semantic bug, we attempt to catch it to limit kernel crashes due to buggy schedulers. To prevent this bug, we introduce a new type called `Schedulable` that represents a task and what core it can safely be scheduled on. When tasks are created, blocked or unblocked, or moved between run-queues, libEnoki creates a `Schedulable` data structure to indicate which core's run-queue the task is on and passes ownership of it to the scheduler at the corresponding call. The scheduler returns the data structure back to Enoki as the return value for `pick_next_task` as proof that the task can be safely scheduled on the core. In libEnoki, the core in the `Schedulable` is checked against the core being assigned, and if the check fails, the data structure's ownership is returned to the scheduler using the `pnt_err` call.

A `Schedulable` also cannot be copied or cloned, so the scheduler cannot hold onto an old `Schedulable` to act as validation after it has returned it to Enoki when moving or running the task. The scheduler must instead receive a new `Schedulable`, such as through the `task_wakeup` call or as the current task in `pick_next_task`.

Enoki cannot always prevent the scheduler from holding onto an invalid `Schedulable.` The `migrate_task_rq` function, which moves tasks between cores, passes in a new `Schedulable` for the new core and requires the scheduler to return the old one so the scheduler will only have validation to run the task on one core. We cannot check at compile time that the scheduler returns the correct `Schedulable` for the old core, so it is possible for a scheduler to keep the wrong data structure. Additionally, we cannot require that the scheduler return a `Schedulable` in `task_blocked` or `task_dead` because these functions are sometimes called when the task is not schedulable and the scheduler has nothing to return.

In libEnoki, we also provide a safe interface for receiving hints from userspace (subsection 3.3).

3.2 Live Upgrade

We support live upgrade of schedulers in Enoki. Live upgrade allows an upgraded version of a scheduler to replace an older version of the same scheduler without rebooting the machine or killing any of the tasks using the scheduler. Because the scheduling code is running throughout the upgrade, we must ensure that any state maintained by the scheduler, such as task runtimes and which task is runnable on which core, remains consistent across the upgrade and is available to the new scheduler after the upgrade.

We ensure consistency of the state by quiescing the scheduler module during the upgrade. Since the scheduler module is only invoked from Enoki-C, we know that the state of the scheduler will remain consistent as long as all calls from Enoki-C are completed before the upgrade begins and any new calls wait until the upgrade is completed. We implement this using a simple per-scheduler read-write lock. Non-upgrade calls into the scheduler module acquire the lock in read mode, allowing multiple concurrent calls into the scheduler module. When an upgrade begins, the lock is acquired in write mode, preventing any of the non-upgrade calls from entering the scheduler module. When the upgrade is finished, the lock is released, and the non-upgrade calls can proceed, now directed to the new version of the scheduler.

After the scheduler module is quiesced, Enoki-C calls into `reregister_prep`, informing the scheduler that an upgrade is occurring. The module then performs any necessary maintenance and returns a data structure with any state it wishes to pass to the new version of the scheduler. Enoki-C then calls `reregister_init` in the new version of the scheduler, passing the data structure returned from the old scheduler. The new scheduler can then initialize itself based on the provided state, claiming ownership of some or all of the state if it wishes. Because Enoki-C acquired the read-write lock described above in write mode, we know that no other calls can enter either scheduler module during the upgrade, and it is safe for the scheduler to manipulate its internal state.

After ensuring that state remains consistent across the scheduler during an upgrade, the rest of the upgrade is fairly straightforward. Enoki-C swaps pointers so that its internal data structure now points to the new module, and the upgrade completes. Calls to the scheduler will use the new pointer, and so will call into the new scheduler module. The old scheduler module can then be safely unloaded.

Limitations. Because we quiesce the scheduler during an upgrade, there is a period when the scheduler is blocked and cannot accept non-upgrade calls, leading to a short service blackout. Enoki trusts the scheduler to have short, well-defined code paths so the read locks will be given up quickly and the upgrade can progress. We also trust the scheduler to upgrade and release the write lock quickly. Another limitation is that the state passing data structure that the new scheduler expects must be the same as the state passing data structure exported by the old scheduler because the memory is passed directly. This data structure is otherwise completely custom, and the new scheduler can export a different data structure to the next scheduler upgrade. This does not mean the upgraded scheduler must use the same state layout as the old scheduler. Rather the new scheduler needs to initialize itself using the state passing data structure exported from the old scheduler, but can define a new state passing data structure for the next version.

3.3 Custom Scheduler Hints

Correct scheduling decisions often depend on the behavior of the workload being scheduled. Tasks that communicate with each other or operate on the same data benefit from being scheduled on the same NUMA node, or with more recent split last level cache architectures, on cores with the same last level cache [55]. On a heterogeneous CPU, some tasks might prefer to be scheduled on certain cores or devices or co-located with other tasks [56].

In addition to sending scheduling hints from userspace to the kernel, it can also be useful for information to flow from the kernel to userspace. For example, in scheduler activations [4], the scheduler provides information about scheduling events to the user-level thread scheduler, such as when cores become available or when user tasks block in the kernel. Another example could be to utilize machine learning to improve scheduling decisions [29].

Enoki supports custom scheduler-defined hints, both from userspace to the kernel and vice versa. Each scheduler that supports hints defines data structures indicating the type of hints that it expects to receive and the type of hints it will send to the user. We enforce that these types can be read-shared across the user/kernel boundary without violating memory safety, but otherwise put no restrictions on them. For example, for our locality aware scheduler, we pass locality hints in the form of the task ID and the hint value. For our two-level scheduler, we pass core requests in the form of the process ID and the number of cores per priority level, and core reclamation requests as a single boolean value. Other applications can define the hint data structure as needed based on the use case.

Queues can be shared across a live upgrade as long as both versions of the scheduler use the same hint data structures. The scheduler passes the queues as part of the shared state during the upgrade. If the next scheduler version will use different hints, old queues must be closed before the upgrade, and new ones can be created after.

3.4 Record and Replay

Kernel debugging is notoriously difficult. Many debugging tools are difficult or even impossible to use on the kernel. While Enoki prevents many bugs that would crash the kernel, logic bugs can still exist in the schedulers. We want to provide a mechanism to simplify debugging by allowing the scheduler code to be run and debugged at userspace.

To this end, Enoki implements a record and replay system for scheduling events. When running in record mode, libEnoki records each call and hint sent to the scheduler. The replay system implements a replacement version of libEnoki to replay these records to the scheduler, now running in userspace. The exact same scheduler code is run during both record and replay. The replay is primarily aimed at understanding the behavior of the recorded scheduler. The trace could be used as input to a modified scheduler, but the policy changes may affect the order of scheduler actions and the scheduler behavior. Consulting a userspace scheduler synchronously on every operation, like Bento's approach to file system debugging, is infeasible because the scheduler subsystem cannot block waiting for a response from a userspace program that itself needs to be scheduled to run. Likewise, consulting a userspace scheduler asynchronously, as in ghOSt, can result in different behavior between the kernel and user versions.

Record. LibEnoki must record the sequence of information provided to the scheduler, both procedure calls into the scheduler and hints from the queue, so they can be replayed exactly as they were originally played in the kernel. We also record responses returned by the scheduler so we can alert the user if the scheduler returns a different result during replay. We do not record or attempt to replay the exact timing of the messages. Where timing is relevant to scheduler decisions, such as the runtime of a task, that information is provided in the message from the kernel and so will be recorded. We assume that the scheduler does not attempt to validate this information or track its own timing and that the scheduler does not contain other sources of nondeterminism.

Enoki's record and replay system must also handle concurrency. The Linux kernel is multi-threaded, and multiple kernel threads can call into an Enoki scheduler at once, e.g. on different cores. Due to Rust's properties and Enoki's structure, we know that potential race conditions are protected by locks, and we can identify and record the order of lock acquisitions. As long as locks are acquired in the same order during record and replay and the behavior of the scheduler is deterministic, the results should be the same [30]. In libEnoki, we include recording functionality in the shim wrappers around the kernel lock functions to record lock creation, acquisition and release, along with the address of the lock and the ID of the accessing kernel thread. All other message records are also tagged by the thread ID of the kernel thread calling into the scheduler.

Recording messages in the scheduler stack is non-trivial. In many cases, Enoki is called while the kernel has interrupts disabled. Writing to a file has the potential to sleep, so we cannot write the messages to a log file while in the scheduler context. Even printing to the kernel log must be delayed until out of the scheduler context.

Similarly to userspace hints, we use a ring buffer to solve this. We run a separate userspace task that creates a ring buffer queue shared with Enoki-C. When libEnoki wants to record a message, it sends it to Enoki-C, which adds the message to the queue. The userspace task consumes messages on the queue and writes them to a file. If the buffer overruns, events may be dropped.

In order to record messages, the userspace record task must be running and the scheduler must have been compiled in record mode. By default, libEnoki does not record messages.

Replay. Replay consumes the file created during recording. The replay utility sends the recorded messages directly to the scheduler code in the same order they were called and validates the responses against the recorded ones.

To handle concurrent replay, we ensure that all locks are acquired in the same order in replay as they were during record. First, the replay system analyzes the log and parses out the lists of operations on each lock, using the lock's memory address to differentiate the locks. The locks are then created, passing in the list of acquisitions for each lock. We assume that locks are created in the same order during replay as they were during record and are not deallocated.

To allow for concurrent operations, the replay system starts a thread per recorded message in the record log as it replays the log. When each replay thread is created, the replay system names it with the ID of the associated kernel thread. When the replay thread attempts to acquire a lock, the lock checks whether it is the next to acquire the lock. If not, the thread is blocked until its turn.

Enoki does not support upgrading the scheduler during the record and replay process.

4 Implementation

4.1 Enoki

The lines of code for Enoki are shown in Table 2. Enoki is implemented in Linux 5.11 and is based on the kernel component from ghOSt [32]. The scheduler libEnoki includes the EnokiScheduler trait, hint queues, and support for record and live upgrade. Other libEnoki provides safe access to kernel

Component	Lang.	LOC	Unsafe LOC
Enoki-C	C	2411	N/A
Scheduler libEnoki	Rust	962	94
Other libEnoki	Rust	5870	2858
Userspace Record	Rust	95	10
Replay	Rust	646	0

Table 2. Lines of code for the Enoki components. The scheduler libEnoki includes the EnokiScheduler trait, hint queues, and support for record and live upgrade. Other libEnoki provides safe access to kernel data structures and functions. Part of Other libEnoki is shared with the analogous library from Bento (libBentoKS).

data structures and functions and general support for Rust kernel programming and is shared with libBentoKS from Bento. Enoki could be ported to newer versions of the kernel with relative ease, particularly because the scheduler interface appears to be quite stable. Note that the ideas behind Enoki are not unique to Linux; however their application to other operating systems is left for future work.

4.2 Schedulers

To demonstrate Enoki's flexibility, we implemented a selection of schedulers: a weighted fair queuing scheduler based on Linux CFS, a specialized research scheduler based on Shinjuku, and schedulers that utilize the userspace hinting and bidirectional communication features of Enoki for locality aware scheduling and two level scheduling. As baselines, we use the default Linux CFS and the ghOSt [32]

4.2.1 CFS and Weighted Fair Queuing. CFS uses per-core run-queues, meaning it first assigns tasks to cores, and then chooses the next task for the core from among the assigned tasks. On each core, CFS implements a version of weighted fair queuing, dividing CPU time proportionally between groups of tasks, and then within each group, while respecting priority. It uses a calculation called *vruntime* to track which group/task to choose next, choosing the group/task with the lowest weighted accumulated runtime. The *vruntime* is calculated based on the task's runtime, modified by its priority; tasks with higher priority accrue *vruntime* slower. To prevent sleeping tasks from accruing a large *vruntime* debt and therefore running for too long after they wake, newly woken tasks receive the maximum of their old *vruntime* and the *vruntime* of the task with the lowest *vruntime* in the run-queue minus a several millisecond threshold. If a newly woken task has a smaller *vruntime* than the current task, it preempts the current task when a system timer ticks. Otherwise, task switches occur only after a task has run for its allocated time slice. In order to prevent starvation, CFS attempts to run every task at least once per time period, where the period depends on the number of tasks, with a minimum of 6ms. When tasks are created or woken, the length of the period adjusts to include the new task. New tasks are run at the end of the period, but recently woken tasks can be run earlier.

CFS places running tasks onto cores and moves tasks between cores to achieve better performance. By default, CFS will attempt to even out the amount of work per core, based on information such as the amount of time the core is idle or overloaded, the priority of the tasks on the core, the capacity of the cores on the machine, and the preferences of the tasks. In certain cases, CFS will co-locate tasks on specific cores, such as if it is required to by the user. CFS attempts to place tasks so that newly woken tasks can be scheduled promptly. When a task is woken or a core becomes idle, CFS will move tasks so the newly idle cores are used. CFS rebalances task placement every 1-10ms, depending on the configuration, or when cores become newly idle. CFS first tries to move tasks to cores within the same NUMA node, and will not balance tasks across NUMA nodes unless there are more than a threshold more tasks running on the busier NUMA node.

Enoki Weighted Fair Queuing. To evaluate the overhead of Enoki, we implemented our own scheduler based on CFS. Our version does not provide the full complexity of the algorithm; instead, we are interested in showing the overhead of our approach on benchmarks relative to CFS. We compute *vruntime* for per-core time slices but use a much simpler method for determining task placement. If a core is about to become idle and another core had a waiting task, our scheduler steals waiting work from the core with the longest queue of tasks. Otherwise, our scheduler does not rebalance tasks. We found this compromise allowed our scheduler to achieve good performance on a wide array of benchmarks with much less complexity - 646 lines of code versus 6247 for CFS.

4.2.2 GhOSt and the Shinjuku Scheduler. GhOSt [32] is a research framework for userspace schedulers. GhOSt implements a trampoline approach; calls from the core scheduler code are forwarded to the userspace scheduler, which sends its scheduling decisions to the kernel. GhOSt uses an asynchronous message passing model, i.e. the core scheduler code does not wait for the userspace scheduler to respond before choosing what to run. We evaluate microbenchmarks against two ghOSt schedulers, the per-CPU FIFO scheduler, which runs per-core schedulers that manage tasks assigned to that core, and the SOL scheduler, a latency-optimized FIFO scheduler that manages all cores, running the scheduler on a separate core. We also implement a version of the Shinjuku [34] scheduler and evaluate it against ghOSt's version of the same scheduler.

The Shinjuku scheduler [34] achieves low latency for workloads with short, high priority tasks and longer, low priority tasks with an efficient version of shortest task first. Shinjuku uses centralized first-come-first-serve scheduling. After each task has run for 5 to 15μs Shinjuku preempts it, placing it at the back of the queue. However, Shinjuku does not run in Linux, instead depending on the Dune [12] operating system. Unlike Linux, Dune applies a single scheduling algorithm for all tasks on the machine.

To evaluate Enoki's ability to support a specialized scheduler for Shinjuku-style workloads, we implemented a scheduler based on the Shinjuku scheduler using Enoki. Our scheduler implements an approximation of a first-come-first-serve queue of tasks with fast preemption across the multiple kernel run-queues. Our preemption slice is 10μs instead of 5μs to prevent overloading the scheduler. This scheduler was implemented in 285 lines of code.

4.2.3 Locality Aware Scheduler.

We also implemented a locality aware scheduler using Enoki that co-locates tasks that communicate heavily with each other or benefit from cache sharing. This scheduler uses Enoki's userspace hinting mechanism to inform the scheduler about which tasks to colocate. The application sends the ID of each newly created thread and a locality value to indicate which tasks should be co-located. Unlike Linux's *taskset* cgroup, these hints do not need to specify the core for each task, only its colocation, which the scheduler can ignore if non-optimal, such as when there are too many tasks on a given core. This scheduler was implemented in 203 lines.

4.2.4 Two-level Scheduler.

The Arachne user-level scheduler provides two-level thread management: applications request cores and manage user-level threads on the assigned cores [4, 64].

In Arachne, both the core arbiter and the runtime are implemented in userspace. The core arbiter relies on Linux's `cpuset` mechanism to manage core assignments. The runtime sends messages to the core arbiter over a socket, and the core arbiter either responds on the socket or uses a shared memory page. This socket allows the runtime to manually block if a core is not available.

We reimplemented the Arachne core arbiter as a kernel scheduler using Enoki. This scheduler uses Enoki's bidirectional userspace hints. We use the user-to-kernel queue to send core requests to the Enoki core arbiter; we use the kernel-to-userspace queue for core reclamation requests. The Enoki core arbiter executes the same decisions as the Arachne core arbiter, but uses standard kernel scheduling mechanisms for assigning, moving, and blocking user scheduler activations rather than relying on `cpuset` and sockets. The Enoki version of the core arbiter is implemented in 579 lines of code. We compare this scheduler against both CFS and unmodified Arachne.

5 Evaluation

In our evaluation, we test Enoki's ability to meet our goal of high velocity development for a wide variety of high performance schedulers with minimal runtime overhead. In section 6, we discuss the development experience of Enoki. In this section, we evaluate the performance of the baseline and Enoki schedulers. We evaluate whether these schedulers can achieve equivalent performance to native implementations on microbenchmarks and application workloads. For research schedulers, we evaluate the scheduler against using benchmarks from the original paper.

5.1 Setting

Benchmarks were performed on either an 8 core, one-socket machine with an Intel i7-9700 CPU running at 3.00GHz or (for scalability tests) an 80 core, two-socket machine with Intel Xeon Gold 6138 CPUs running at 2.00GHz.

5.2 Microbenchmarks

We used microbenchmarks to evaluate the latency and scalability imposed by the Enoki framework. We ran the same microbenchmarks on all the schedulers we implemented.

Latency. Table 3 shows the results of the `perf bench sched pipe` benchmark, averaged over three runs of the benchmark. This benchmark starts two tasks that send 1 million messages back and forth using the `pipe` system call. After each message, the sending task sleeps until the other task responds. By default, all schedulers put the two tasks on different cores. We also ran the benchmarks forcing both tasks to be on the same core.

The CFS and Enoki WFQ schedulers implement similar behavior. The ghOSt SOL and FIFO schedulers implement different algorithms. The ghOSt SOL algorithm attempts to schedule tasks as quickly as possible, and so should have minimum scheduling latency. The Arachne scheduler uses userspace threads while the others use processes because the Arachne userspace scheduler is built to manage userspace threads.

Compared to CFS, our Enoki WFQ scheduler adds 0.4 μs (0.6μs) of latency per message, for the two (one) core case. This represents a 12% to 20% overhead on this benchmark. We found that this overhead was due to 100-150 ns of overhead per invocation of the Enoki scheduler. The scheduler is invoked four times per schedule operation: once due to the current task blocking, once due to the next task waking, once to allow the scheduler to rebalance tasks, and once for the scheduler to choose the next task. Together this results in 400-600 ns, or 0.4-0.6 μs, of overhead per schedule operation. This overhead could possibly be reduced with further optimization. Our version of the Shinjuku scheduler has slightly higher overhead because it starts a reschedule timer on every operation, while CFS and our WFQ scheduler only start a reschedule timer when multiple tasks are present. The locality aware scheduler is slightly faster because it is simpler. The Enoki version of Arachne is much faster than the others because it uses userspace threads instead of processes for blocking and waking threads.

The GhOSt schedulers perform significantly worse than both CFS and Enoki. The per-CPU FIFO scheduler performs worse when both tasks are placed on the same core because they are sharing the core with the ghOSt userspace scheduler. On every schedule operation, the scheduler first must be scheduled and run on the core.

The more often an application triggers scheduling actions, the more scheduling latency will impact the application. Workloads with many small tasks or many sleeps and wakes, and therefore many invocations of the scheduler, will be most affected by this overhead.

Message Latency (μs)	CFS	GhOSt SOL	GhOSt FIFO	WFQ	Shinjuku	Locality	Arachne
One Core	3.0	6.0	9.1	3.6	4.0	3.5	0.1
Two Cores	3.6	5.8	7.0	4.0	4.4	3.9	0.2

Table 3. Scheduler latency for the `perf bench sched pipe` benchmark in μs per wakeup. Enoki adds around 0.6μs of latency over CFS in the worst case and outperforms the ghOSt schedulers due to the synchronous nature of the benchmark.

Worker Threads		CFS	GhOSt SOL	GhOSt FIFO	WFQ	Shinjuku	Locality	Arachne
2 Tasks (μs)	50th	74	66	101	78	79	80	1
	99th	101	132	170	104	109	105	1
40 Tasks (μs)	50th	139	192	152	170	168	175	1
	99th	320	1354	1806	323	307	324	1

Table 4. Schbench benchmark with two message threads and 2 and 40 worker threads per message thread. Thread wakeup latencies are measured in μs.

Scalability. To measure the scalability of Enoki, we evaluate the tail latency of task schedules when there are a large number of tasks using the schbench benchmark. This benchmark starts a number of message threads and worker threads. Each message thread and its worker threads send messages back and forth. Schbench reports the median and 99% tail latency of task schedules throughout the benchmark. The tested configurations use 2 message threads and 2 or 40 worker threads, resulting in a maximum of 80 worker threads, the same as the number of cores on the machine. When the number of worker threads is larger than the number of cores, the scheduling latency is influenced by waiting time more than scheduler performance. We use a 5s warmup time for each run.

The results are shown in Table 4. CFS and the Enoki WFQ scheduler showed similar results except for the median with 40 threads. The Enoki version of Arachne has lower latency than the other schedulers because of its userspace mechanisms for blocking and waking threads.

5.3 WFQ Scheduler Applications

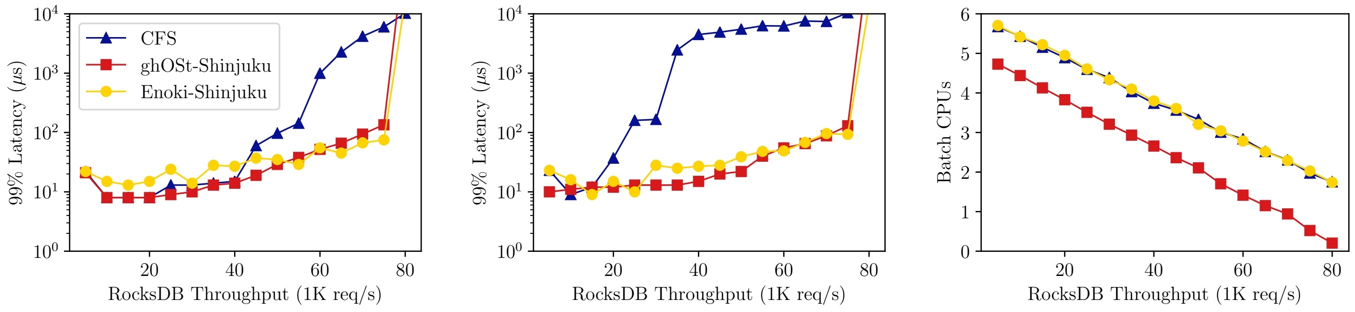
To evaluate whether our WFQ scheduler performs competitively with CFS, we ran multithreaded application benchmarks using benchmark suites that covered a wide range of use case patterns. We ran benchmarks from the NAS Parallel Benchmark suite [7] and the Phoronix Multi-core Test Suite [61]. The NAS Parallel Benchmark suite from NASA is a set of benchmarks to evaluate parallel performance on supercomputers. We ran nine of the ten NAS Benchmarks, excluding the DC benchmark that targets computational grids. We used the "C" benchmark size, the largest standard benchmark size. The Phoronix Multi-Core Test Suite contains a very large collection of multithreaded application benchmarks. There are over 90 applications, and many have multiple workloads. We report the same collection of 27 benchmarks as reported in Nest [36]. Phoronix runs each benchmark three times unless the standard deviation is greater than 5%, in which case it will rerun the benchmark until the standard deviation is low enough, up to a maximum of 15 times. The results for both benchmark suites are in Table 5.

The NAS benchmarks show very little performance difference between the CFS and Enoki WFQ schedulers, with a maximum of 2.16% difference on the LU benchmark. The NAS benchmarks all start one task per core, so this is expected behavior. Performance on the Phoronix benchmarks is also quite similar for both schedulers across the benchmarks. The largest slowdowns were 8.22% and 8.57% on the Cassandra Writes benchmark, which issued writes to an Apache Cassandra database, and the Zstandard compression benchmark using compression level 3, long mode, respectively. This was likely due to the WFQ scheduler's simpler mechanism for rebalacing tasks making less optimal decisions for these benchmarks. We found that the balancing mechanism most affected the Arrayfire, Cassandra, and Zstandard compression benchmarks. Interestingly, we also saw speedup for Enoki on some benchmarks. The largest speedup was on the Zstandard compression benchmark using level 8, long mode. This was likely due to the simplified balancing mechanism. Overall, the geometric mean of the performance differences between the schedulers across the benchmarks was 0.74%.

We also ran these benchmarks on ghOSt. The SOL minimal FIFO scheduler described in the ghOSt paper is much slower and does not run some of the benchmarks. A new ghOSt-based weighted fair queueing scheduler is in progress, and runs correctly on all but one benchmark. It is slower on most benchmarks than Enoki. However, since it is not finished, a quantitative comparison would not be fair to ghOSt at this time.

5.4 Shinjuku Scheduler

To evaluate our version of the Shinjuku scheduler, we ran the RocksDB benchmarks used in the Shinjuku and ghOSt papers. These benchmarks send queries to an in-memory RocksDB database, with 99.5% GET requests and 0.5% range queries. Replicating how this benchmark was run in ghOSt, each GET is assigned to take 4 μs and each range query to take 10 ms. If RocksDB responds too quickly, the benchmark spins until the assigned time has elapsed. Three cores were reserved, one for background tasks, one for the load generator, and one for the scheduler if required. The load generator passes tasks to a total of 50 workers running on the other five cores. We evaluate against the ghOSt version of the Shinjuku scheduler. We could not compare against the original Shinjuku scheduler because it requires a specific NIC that we do not have. Both the ghOSt



(a) Tail latency for a dispersive load on RocksDB.
(b) Tail latency for a dispersive load on RocksDB co-located with a batch app.
(c) CPU share for a batch application co-located with RocksDB.

Figure 2. Comparison of the Enoki Shinjuku (μs scale preemption) scheduler compared to CFS and the ghOSt Shinjuku scheduler using RocksDB with and without a batch application. Latency graphs are in log scale.

Shinjuku and Enoki Shinjuku schedulers use a preemption timer of 10 μs. The results are shown in Figure 2.

In the first benchmark, shown in Figure 2a, only the RocksDB application is run. Note that the y-axis is in log scale. Both the Enoki and ghOSt Shinjuku schedulers achieve low tail latency at high loads due to the short preemption timer; long running range query tasks are preempted quickly, allowing the short GET queries to run. On CFS, tasks run for much longer before being preempted, by default 750 μs, so the GET queries spent more time waiting to be run.

In the second benchmark, a batch application was co-located with RocksDB. For the CFS and Enoki tests, the batch application was run using CFS. RocksDB is given a priority of -20 while the batch application is 19 (lower is higher priority). The batch application is run on ghOSt using a lower priority than RocksDB. Figure 2b shows the tail latency of RocksDB, and Figure 2c shows the CPU share of the batch application.

Both the Enoki and ghOSt Shinjuku schedulers achieve similar tail latency as with no background task because they give priority to the RocksDB workers, with a small time slice. The tail latency on CFS worsens with addition of the batch task. In the first experiment with no batch application, the Enoki scheduler achieved 30% lower latency than the ghOSt scheduler at high load (above 65K req/s). High load results in the most stress on the scheduler because it must repeatedly preempt long running tasks to schedule the short tasks.

The batch application receives a similar share of the CPU on CFS and the Enoki Shinjuku scheduler. CFS itself shares cycles between the RocksDB workers and the batch application according to their niceness values. When there are no RocksDB requests the Enoki scheduler seamlessly cedes cycles to CFS to run the batch application. The ghOSt Shinjuku scheduler provides substantially less CPU time to the batch application because it must pay the overhead of the userspace scheduler. In Enoki and CFS, the scheduler is run on the same core as the application as part of regular kernel calls. All of the schedulers give the batch task a much higher CPU share than the original Shinjuku scheduler would [32].

5.5 Locality Aware Scheduler

To evaluate the effectiveness of using hints in our locality aware scheduler, we used a modified version of the schbench benchmark. This benchmark starts a specified number of message threads and worker threads. Each message thread and its worker threads send messages back and forth. The benchmark records the wakeup latency of the worker threads to evaluate scheduler overhead. When a message thread and its worker threads are on the same core, wakeup latency can be very low. However, the benchmark uses a futex to wait which does not set the WF_SYNC flag when waking the workers, so Linux can not detect this pattern [55]. In our modified version of the benchmark, we send hints to an Enoki locality aware scheduler to co-locate the message thread with its workers, but place each set of message and worker threads on a different core. We compare this approach to CFS and to the locality aware scheduler with random placement (no hints) as baselines. We also compare to CFS using cgroups to test if the flexibility provided by the hints is necessary for a performance benefit. Cgroups enable specifying a set of cores that a process should run on, but do not support different sets of cores for different threads within the same process. We use cgroups to place all the threads on one core.

The results are shown in Table 6. We use two message threads and two worker threads per message thread. CFS and the locality aware scheduler with random placement perform similarly because both spread tasks across cores. The locality aware scheduler with hints achieves significantly lower 99% latency. Using cgroups to put all threads on one core improves median latency at a cost of much worse tail latency due to the added competition between threads.

5.6 Arachne Scheduler

We evaluate the Enoki version of the Arachne scheduling using a version of the Realistic memcached workload from

Benchmark	CFS	WFQ	
NAS Benchmarks			
BT (total Mops/s)	26669.1	26682.3	−0.05 %
CG (total Mops/s)	4535.8	4475.7	1.32 %
EP (total Mops/s)	487.9	491.9	−0.83 %
FT (total Mops/s)	14886.8	14716.5	1.14 %
IS (total Mops/s)	1297.4	1284.9	0.96 %
LU (total Mops/s)	30469.4	29811.4	2.16 %
MG (total Mops/s)	8601.4	8535.9	0.76 %
SP (total Mops/s)	11797.0	11705.6	0.78 %
UA (total Mops/s)	73.8	73.1	0.87 %
Phoronix Multicore			
Arrayfire, 1 (GFLOPS)	812.98	820.29	−0.90 %
Arrayfire, 2 (ms)	26.72	26.71	−0.04 %
Cassandra, 1 (Op/s)	55100	50573	8.22 %
ASKAP, 4 (Iter/s)	161.46	161.12	0.22 %
Cpuminer, 2 (kH/s)	51363	51390	−0.05 %
Cpuminer, 3 (kH/s)	35667	35390	0.78 %
Cpuminer, 4 (kH/s)	9499.87	9494.90	0.05 %
Cpuminer, 6 (kH/s)	258100	261667	−1.38 %
Cpuminer, 11 (kH/s)	29400	29323	0.26 %
Ffmpeg, 1, 1 (s)	23.98	24.73	3.13 %
Graphics-Magick, 4 (Iter/m)	781	779	0.26 %
OIDN, 1 (Images/s)	0.31	0.30	3.23 %
OIDN, 2 (Images/s)	0.31	0.30	3.23 %
OIDN, 3 (Images/s)	0.15	0.15	0 %
Rodina, 3 (s)	159.32	160.00	0.43 %
Zstd, 2 (MB/s)	856.1	782.7	8.57 %
Zstd, 4 (MB/s)	153.1	165.4	−8.03 %
AVIFEnc, 4 (s)	14.94	15.33	2.62 %
Libgav1, 1 (FPS)	262.95	261.21	0.66 %
Libgav1, 2 (FPS)	67.28	66.58	1.04 %
Libgav1, 3 (FPS)	222.70	216.51	2.78 %
Libgav1, 4 (FPS)	64.10	63.54	0.87 %
OneDNN, 4, 1 (ms)	4.26	4.18	−1.85 %
OneDNN, 5, 1 (ms)	9.71	9.10	−6.31 %
OneDNN, 7, 1 (ms)	4166.31	4164.74	−0.04 %
OneDNN, 7, 2 (ms)	4166.40	4161.15	−0.13 %
OneDNN, 7, 3 (ms)	4164.25	4163.34	−0.02 %

Table 5. Performance comparison of Linux CFS and Enoki WFQ on the NAS Parallel Benchmarks and a selection of the Phoronix Multicore benchmarks (version 10.8). Full names are provided in the appendix. The maximum slowdown is 8.57%. The geometric mean over all benchmarks is 0.74%.

the Arachne paper [64]. We use the Mutilate benchmark utility [51] to generate load for the memcached server, using the key size and distribution, value size and distribution, and inter-arrival distribution of the Facebook ETC workload [6], 1 million records, and 3% updates. Four clients are used to generate load, enough to make the benchmark server-bound in our tests. Each client creates 16 threads and four connections per thread. An extra client is used to evaluate latency. We compare the Enoki version of Arachne against the original Arachne code and a baseline version of memcached running on CFS. Both of the Arachne versions automatically scale between two and seven cores, reserving one for background tasks. The baseline version uses all eight cores on the machine.

The Enoki version of Arachne achieves similar performance to the original Arachne scheduler, better than CFS at high load.

Latency	CFS	CFS One Core	Random	Hints
50th (μs)	33	17	46	2
99th (μs)	50	32032	49	4

Table 6. Wakeup latency for the schbench benchmark with two message threads and two worker threads per message thread. Thread wakeup latencies were in microseconds. All runs used a 5s warmup time and ran the benchmark for 30s.

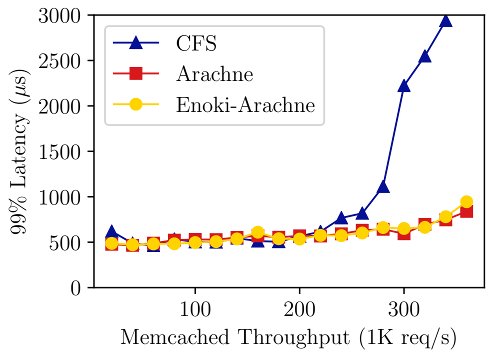


Figure 3. Tail latency of requests to a Memcached server using Mutilate comparing baseline Memcached using CFS to a version using Arachne and a version using Arachne modified to use an Enoki core arbiter.

5.7 Live Upgrade

We evaluate the performance impact of live upgrade using the schbench benchmark and timing instrumentation in the kernel. We track the tail latency of schedule operations before, during, and after the upgrade. The upgrade interruption was too short to affect the tail latency of the schbench operations, so we repeated the evaluation with timing calls inserted in the kernel. We evaluate this benchmark on both the one socket machine using 2 message threads and 2 workers per message thread and the two socket machine with 2 message threads and both 2 and 40 workers per message thread. On the one socket machine, the upgrade takes 1.5μs. On the two socket machine, the upgrade takes 9.9μs and 10.1μs for 2 and 40 workers, respectively. All were averaged over three runs.

5.8 Record and Replay

We evaluate the performance of record and replay using the `perf bench sched pipe` benchmark on the WFQ scheduler. This benchmark completes in around 4 seconds during regular operation. During record, the benchmark completes in around 30 seconds, and the replay takes around 3 minutes. Record is slower than regular operation because all relevant operations must be sent to the record program, which writes a file asynchronously on a different core.

During replay, the first 30 seconds are spent reading the file and parsing lock operations. The rest is due to the mechanism for ensuring concurrent operations occur in the same order as the live execution. This benchmark has many calls to the scheduler. In the kernel, these operations are very fast. During replay, if threads arrives in a different order than what was recorded, we need to block the thread and wake it up later to try again. These constant sleeps and wakes add latency and account for most of the execution time during replay.

6 Discussion

In this section, we report on our experience using Enoki to develop kernel schedulers. Enoki's design made scheduler development much more straightforward than it would be in Linux. In fact, two of the schedulers (the Shinjuku and Locality Aware schedulers) were written by undergraduates with no prior experience in Linux kernel development. They were able to start coding with relatively little ramp up and were able to build, test, and debug their code on their own. They primarily benefited from modularity, because they were using Enoki when record and replay and live upgrade had not been implemented.

Being able to use the `Schedulable` data structure as proof that a task is schedulable on a core allowed us to ensure that we were picking runnable tasks. This helped us identify and address bugs faster because we would see a compile time error from `pick_next_task` returning an incorrect `Schedulable` thread rather than putting the kernel into a bad state by trying to run a task on the wrong core.

We ran into relatively few runtime bugs, and those bugs were fairly easy to address. Most have been deadlocks, which are painful to encounter because they force a reboot, but proved surprisingly easy to debug, in line with prior OS development experience [16]. Often, relatively few lock operations were touched in between two runs of the code, so finding the relevant changes to lock operations was easy. With more complicated deadlocks, it was easy to look through the scheduler code and check the order of lock acquires because the largest scheduler (WFQ) was only around 600 lines of code. Of the non-deadlock bugs, most were conceptual, such as a benchmark having tasks yield when we had not yet implemented `task_yield` or miscalculating the fair time slice to assign tasks in WFQ. While fixing these bugs sometimes required rebooting the machine, we never had to recompile the kernel to fix them, so our iteration times were quite short.

The general design of an interposable, message-passing based interface that uses memory sharing under the hood had a number of benefits. We gained many of the benefits of modularity for development velocity because of clean interfaces and isolation from the rest of the kernel, allowing the developer to focus their energy on the algorithm being implemented. In its implementation, Enoki uses function calls and memory sharing, limiting the overhead that can come with modular designs. Because Enoki schedulers run in the kernel, the core scheduling code can quickly and easily make synchronous calls to the scheduler, allowing it to quickly respond to changes in state.

Enoki's design also made it easy to implement additional features. Because all the functionality was contained in the module and Enoki-C contacts the scheduler through a single function pointer, live upgrade is as simple as quiescing the state and replacing the function pointer. Due to Rust's support for generic data types and traits, custom hint data structures could be defined as type parameters on the scheduler and any requirements on the data types can be expressed as trait bounds. Enoki's design supported record and replay debugging very smoothly. The interposable, message passing based interface and made it simple to record relevant state that was passed into the scheduler or returned from calls into the kernel. Recording nondeterministic behavior is one of the main challenges for record and replay on parallel systems, but using safe Rust made this much simpler. Due to Rust's safety guarantees, we knew that the scheduler could not contain race conditions or any other undefined behavior, so the only sources of nondeterminism were timing and the order of lock acquisitions. All timing state was handled by the kernel and passed into the scheduler, and so was automatically handled by recording the messages. To correctly handle concurrency, we only needed to record and replay the order in which locks were acquired.

7 Related Work

Scheduler Frameworks. GhOSt [32] is the closest analog to Enoki. It allows the user to replace the kernel scheduler with a userspace scheduler. Calls to the kernel scheduler are forwarded to the userspace scheduler agent, which responds with decisions that are then applied in the kernel. The kernel does not wait for decisions from the userspace scheduler to schedule a task, instead applying decisions asynchronously at a later call into the kernel scheduler. GhOSt provides good development velocity but can add significant overhead and scheduling latency, and the asynchronous model makes it difficult to precisely mirror kernel scheduler decisions. Mvondo et.al. [52] also proposed a framework for implementing new schedulers in Linux, inspired by how eBPF enables developers to implement new networking functionality for the kernel. As far as we can tell, this system has not been built yet. AMD engineers have proposed a method for providing userspace hints to a kernel scheduler [55], but only a small number of predefined types of hints are supported.

There are other scheduler frameworks and design paradigms that are orthogonal to this work. Scheduler activations [4] proposes a mechanism for two-level scheduling that assumes the kernel allocates processors to applications, and applications schedule their own threads. Arachne implements a similar mechanism. As shown by our implementation of Arachne, Enoki can be used to implement the kernel component of a two-level scheduling system. Ipanema [38] is a framework for formal verification of scheduler properties, focusing particularly on work conservation. Both Ipanema and Enoki build Linux kernel schedulers as modules and target bug prevention, but the goals and mechanisms are different. Ipanema is able to capture high level safety properties like work conservation, but limits schedulers by requring them to be written in the Ipanema DSL.

Safe Languages in OS Development. It is common for research operating systems to be written entirely in a type-safe high-level language. Examples include Pilot [65], SPIN [14], Singularity [33], Biscuit [22], Redox [66], Tock [39], Theseus [17], and RedLeaf [53]. None of these target improving development velocity for a widely used commercial operating system. These approaches show that benefits can be gained by using safe languages, but all involve rewriting the entire operating system. We are also not the first to integrate Rust into the Linux kernel [40, 42, 49], but these projects do not provide safe interfaces for schedulers. Most similar to our work is Bento [49]. Bento takes a similar approach to Enoki but applied to file systems. Because it targets file systems and sits behind the kernel page cache, Bento can tolerate much higher latency than Enoki.

The Berkeley Packet Filter (eBPF) [48] enables safe extensibility in Linux using a restricted, type safe language and programming environment. eBPF programs can only be inserted at specific points in the kernel and are verified by the kernel before being run. GhOSt [32] extends eBPF to enable implementing part of the scheduler using eBPF. eBPF programs can be updated at runtime, but the locations where code is inserted cannot be. To ensure safety of user code running in the kernel, eBPF is quite restrictive. Programs are limited to only running at pre-defined points in the kernel, only calling whitelisted functions, and not having unbounded loops. For example, it would be difficult to express a red-black tree or B-tree in eBPF, as used in CFS and WFQ.

Live Upgrade. Live upgrade in Enoki is most similar to the techniques used in Bento [49], but applied to schedulers. Plugsched [44] does live upgrade of the entire scheduler subsystem of the Linux kernel without modifying any kernel code using code analysis to detect boundaries of the scheduler subsystem and stack analysis to maintain consistent state across the upgrade. Other tools for live upgrade of Linux systems include ksplice [5, 57], kpatch [62], or kGraft [60]. These replace individual functions with new implementations, and are primarily used for security patches that do not modify kernel data structures. Other research systems, such as K42 [11], PROTEOS [28], LUCOS [21], and DynAMOS [45], support upgrading complex, modular components in either new operating systems (K42 and PROTEOS) or in Linux using shadow data structures or virtualization (LUCOS and DynAMOS). Our work enables live upgrade of large modules in a commodity operating system with state transfer by introducing a framework layer that handles quiescing state.

Running in Userspace. Running operating system services in userspace can also increase development velocity of operating system components [1, 41], though this can sacrifice performance. Most similar to Enoki is GhOSt [32]. Another approach is to run the component in userspace on dedicated cores with communication through shared memory queues [10, 13, 35, 46]. This approach has been used for implementing new network stacks [35, 46] and scheduler algorithms [34, 64]. With direct access to hardware devices, performance can often be competitive with an equivalent kernel implementation. However, interaction with the rest of the kernel can be restricted using this approach, possibly limiting the ability for any part of the system to make global decisions [68].

Record and Replay. Record and replay systems enable debugging code by first recording a trace of operations and then later replaying those operations against the target code, enabling debugging of the recorded trace. These systems require instrumentation around the target code in order to record the necessary data. There are a variety of approaches to record and replay, from recording the whole system [25, 70] to using kernel instrumentation to record user programs [24, 47, 67] to running both the target code and the instrumentation in userspace [15, 27, 59, 69] to instrumenting language and application runtimes [2, 19]. One project [20] specifically records and replays kernel modules using mechanisms for whole binary analysis and instrumentation of kernel module interfaces. As far as we know, we are the first project to selectively record a kernel module, as opposed to the whole kernel, and to replay the behavior at userspace on the same code as ran in the kernel. There is a long history of work on deterministic record and replay on multicore systems to ensure that data accesses are performed in the same order during record and replay [3, 18, 30, 31, 37, 54, 67]. We combine content-based message passing recording, where the content of all messages across a message passing interface are recorded, with a software only shared memory recording scheme, where synchronization and access to shared state are recorded. Unlike other shared memory recording schemes, we do not have to detect and record race conditions due to Rust's safety guarantees, so we record only synchronization accesses.

8 Conclusion

This paper presents Enoki, a framework for rapid development of high performance Linux kernel schedulers. Enoki enables safe, high performance kernel schedulers with seamless live upgrade, bidirectional user communication channels, and record and replay debugging. A scheduler implemented with Enoki is able to achieve similar performance to CFS, the default Linux scheduler, on a wide range of benchmarks. Other Enoki schedulers mimic recent research schedulers, but integrated with Linux. Enoki's schedulers can be upgraded with only 10.1us of service interruption, and the record and replay debugging allows for slow but functional userspace debugging of the kernel scheduler code.

Acknowledgements

We are grateful to our shepherd Baptiste Lepers and the anonymous reviewers for their feedback and useful comments. This work was supported in part by NSF grants CNS-1856636, CNS-2105868, CNS-2106388, CNS-2214272, CNS-2238665, CNS-2345339, as well as grants from Alibaba, Amazon, Cisco, Google, Microsoft, and VMware.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. 1986. Mach: A New Kernel Foundation For UNIX Development. In *Summer USENIX*.

[2] Bowen Alpern, Jong-Deok Choi, Ton Ngo, Manu Sridharan, and John M. Vlissides. 2001. A Perturbation-Free Replay Platform for Cross-Optimized Multithreaded Applications. In *Proceedings of the 15th International Parallel &; Distributed Processing Symposium (IPDPS '01)*. IEEE Computer Society, USA, 23.

[3] Gautam Altekar and Ion Stoica. 2009. ODR: Output-Deterministic Replay for Multicore Debugging. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 193–206. https://doi.org/10.1145/1629575.1629594

[4] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1991. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91)*. Association for Computing Machinery, New York, NY, USA, 95–109. https://doi.org/10.1145/121132.121151

[5] Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: Automatic Rebootless Kernel Updates. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys '09)*. 12 pages. https://doi.org/10.1145/1519065.1519085

[6] Berk Atikoglu, Yuehai Xu, E. Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *Measurement and Modeling of Computer Systems*.

[7] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. 1991. The NAS Parallel Benchmarks—Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing '91)*. Association for Computing Machinery, New York, NY, USA, 158–165. https://doi.org/10.1145/125826.125925

[8] Antonio Barbalace, Marina Sadini, Saif Ansary, Christopher Jelesnianski, Akshay Ravichandran, Cagil Kendir, Alastair Murray, and Binoy Ravindran. 2015. Popcorn: Bridging the Programmability Gap in Heterogeneous-ISA Platforms. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2741948.2741962

[9] Luiz Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Vol. 13. https://doi.org/10.2200/S00874ED3V01Y201809CAC046

[10] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. 29–44.

[11] Andrew Baumann, Gernot Heiser, Jonathan Appavoo, Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, and Jeremy Kerr. 2005. Providing Dynamic Update in an Operating System. In *USENIX Annual Technical Conference (ATC '05)*. 1 pages.

[12] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-Level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, USA, 335–348.

[13] Brian Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1991. User-level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems* 9, 2 (May 1991).

[14] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. 1995. Extensibility Safety and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. 267–283.

[15] Sanjay Bhansali, Wen-Ke Chen, Stuart de Jong, Andrew Edwards, Ron Murray, Milenko Drinić, Darek Mihočka, and Joe Chau. 2006. Framework for Instruction-Level Tracing and Analysis of Program Executions. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*. Association for Computing Machinery, New York, NY, USA, 154–163. https://doi.org/10.1145/1134760.1220164

[16] Andrew Birrell. 1989. *An Introduction to Programming with Threads*. Technical Report 35. Digital Systems Research Center. https://www.microsoft.com/en-us/research/publication/an-introduction-to-programming-with-threads/ A revised version appeared in Systems Programming with Modula-3, Prentice Hall, 1991.

[17] Kevin Boos, Namitha Liyanage, Ramla Ijaz, and Lin Zhong. 2020. Theseus: an Experiment in Operating System Structure and State Management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 1–19. https://www.usenix.org/conference/osdi20/presentation/boos

[18] Aurelien Bouteiller, George Bosilca, and Jack Dongarra. 2007. Retrospect: Deterministic Replay of MPI Applications for Interactive Distributed Debugging. In *Proceedings of the 14th European Conference on Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI'07)*. Springer-Verlag, Berlin, Heidelberg, 297–306.

[19] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive Record/Replay for Web Application Debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology (UIST '13)*. Association for Computing Machinery, New York, NY, USA, 473–484. https://doi.org/10.1145/2501988.2502050

[20] Bo Chen, Zhenkun Yang, Li Lei, Kai Cong, and Fei Xie. 2020. Automated Bug Detection and Replay for COTS Linux Kernel Modules with Concolic Execution. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 172–183. https://doi.org/10.1109/SANER48275.2020.9054797

[21] Haibo Chen, Rong Chen, Fengzhe Zhang, Binyu Zang, and Pen-Chung Yew. 2006. Live Updating Operating Systems Using Virtualization. In *Proceedings of the 2nd International Conference on Virtual Execution Environments (VEE '06)*. 35–44. https://doi.org/10.1145/1134760.1134767

[22] Cody Cutler, M. Frans Kaashoek, and Robert T. Morris. 2018. The Benefits and Costs of Writing a POSIX Kernel in a High-Level Language. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI'18)*. USENIX Association, USA, 89–105.

[23] Max Demoulin, Josh Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Highly-Dispersed Datacenter Workloads with Perséphone. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*.

[24] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 525–540. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/devecsery

[25] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. 2003. ReVirt: Enabling Intrusion Analysis through

Virtual-Machine Logging and Replay. *SIGOPS Oper. Syst. Rev.* 36, SI (dec 2003), 211–224. https://doi.org/10.1145/844128.844148

[26] eBPF.io. 2023. eBPF. https://ebpf.io/

[27] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. Replay Debugging for Distributed Applications. In *2006 USENIX Annual Technical Conference (ATC '06)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/2006-usenix-annual-technical-conference/replay-debugging-distributed-applications

[28] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2013. Safe and Automatic Live Update for Operating Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. 279–292. https://doi.org/10.1145/2451116.2451147

[29] Robert Glaubius, Terry Tidwell, Christopher Gill, and William D. Smart. 2010. Real-Time Scheduling via Reinforcement Learning. In *Proceedings of the Twenty-Sixth Conference on Uncertainty in Artificial Intelligence (UAI '10)*. AUAI Press, Arlington, Virginia, USA, 201–209.

[30] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. 2008. R2: An Application-Level Kernel for Record and Replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*. USENIX Association, USA, 193–208.

[31] Jeff Huang, Peng Liu, and Charles Zhang. 2010. LEAP: Lightweight Deterministic Multi-Processor Replay of Concurrent Java Programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10)*. Association for Computing Machinery, New York, NY, USA, 385–386. https://doi.org/10.1145/1882291.1882361

[32] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. GhOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3477132.3483542

[33] Galen C. Hunt and James R. Larus. 2007. Singularity: Rethinking the Software Stack. *SIGOPS Oper. Syst. Rev.* 41, 2 (apr 2007), 37–49. https://doi.org/10.1145/1243418.1243424

[34] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. 2019. Shinjuku: Preemptive Scheduling for μsecond-scale Tail Latency. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI '19)*. USENIX Association, Boston, MA, 345–360. https://www.usenix.org/conference/nsdi19/presentation/kaffes

[35] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas E. Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. Article 24.

[36] Julia Lawall, Himadri Chhaya-Shailesh, Jean-Pierre Lozi, Baptiste Lepers, Willy Zwaenepoel, and Gilles Muller. 2022. OS Scheduling with Nest: Keeping Tasks Close Together on Warm Cores. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 368–383. https://doi.org/10.1145/3492321.3519585

[37] Thomas LeBlanc and John Mellor-Crummey. 1987. Debugging Parallel Programs with Instant Replay. *IEEE Trans. Comput.* 36, 4 (apr 1987), 471–482. https://doi.org/10.1109/TC.1987.1676929

[38] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. 2020. Provable Multicore Schedulers with Ipanema: Application to Work Conservation. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 3, 16 pages. https://doi.org/10.1145/3342195.3387544

[39] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. 234–251.

[40] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2019. Securing the Device Drivers of Your Embedded Systems: Framework and Prototype. In *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES '19)*. Article 71. https://doi.org/10.1145/3339252.3340506

[41] Jochen Liedtke. 1995. On Microkernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. 237–250.

[42] Linux-kernel-module-rust 2021. Linux-kernel-module-rust. Retrieved Janurary 18, 2023 from https://github.com/fishinabarrel/linux-kernel-module-rust

[43] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. 2016. The Linux Scheduler: A Decade of Wasted Cores. In *Proceedings of the Eleventh European Conference on Computer Systems (EuroSys '16)*. Association for Computing Machinery, New York, NY, USA, Article 1, 16 pages. https://doi.org/10.1145/2901318.2901326

[44] Teng Ma, Shanpei Chen, Yihao Wu, Erwei Deng, Zhuo Song, Quan Chen, and Minyi Guo. 2023. Efficient Scheduler Live Update for Linux Kernel with Modularization. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 194–207. https://doi.org/10.1145/3582016.3582054

[45] Kristis Makris and Kyung Dong Ryu. 2007. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys '07)*. 327–340. https://doi.org/10.1145/1272996.1273031

[46] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, and et al. 2019. Snap: A Microkernel Approach to Host Networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*. 399–413. https://doi.org/10.1145/3341301.3359657

[47] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazières, and Mendel Rosenblum. 2017. Towards Practical Default-On Multi-Core Record/Replay. *SIGPLAN Not.* 52, 4 (apr 2017), 693–708. https://doi.org/10.1145/3093336.3037751

[48] Steven McCanne and Jacobson Van. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Winter USENIX*.

[49] Samantha Miller, Kaiyuan Zhang, Mengqi Chen, Ryan Jennings, Ang Chen, Danyang Zhuo, and Thomas Anderson. 2021. High Velocity Kernel File Systems with Bento. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 65–79. https://www.usenix.org/conference/fast21/presentation/miller

[50] Samantha Miller, Kaiyuan Zhang, Danyang Zhuo, Shibin Xu, Arvind Krishnamurthy, and Thomas Anderson. 2019. Practical Safe Linux Kernel Extensibility. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Bertinoro, Italy) *(HotOS '19)*. Association for Computing Machinery, New York, NY, USA, 170–176. https://doi.org/10.1145/3317550.3321429

[51] Mutilate 2015. Mutilate. Retrieved May 1, 2023 from https://github.com/leverich/mutilate

[52] Djob Mvondo, Antonio Barbalace, Jean-Pierre Lozi, and Gilles Muller. 2022. Towards User-Programmable Schedulers in the Operating System Kernel. https://sites.google.com/view/spma22eurosys/home The 11th Workshop on Systems for Post-Moore Architectures, SPMA 2022 ; Conference date: 05-04-2022 Through 05-04-2022.

[53] Vikram Narayanan, Tianjiao Huang, David Detweiler, Dan Appel, Zhaofeng Li, Gerd Zellweger, and Anton Burtsev. 2020. RedLeaf: Isolation and Communication in a Safe Operating System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 21–39. https://www.usenix.org/conference/osdi20/presentation/narayanan-vikram

[54] S. Narayanasamy, G. Pokam, and B. Calder. 2005. BugNet: continuously recording program execution for deterministic replay debugging. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 284–295. https://doi.org/10.1109/ISCA.2005.16

[55] K Prateek Nayak. 2022. sched: Userspace Hinting for Task Placement. https://lwn.net/Articles/907680/

[56] Edmund B. Nightingale, Orion Hodson, Ross McIlroy, Chris Hawblitzel, and Galen Hunt. 2009. Helios: Heterogeneous Multiprocessing with Satellite Kernels. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 221–234. https://doi.org/10.1145/1629575.1629597

[57] Oracle Ksplice 2009. Oracle Ksplice. Retrieved January 18, 2023 from https://ksplice.oracle.com/

[58] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[59] Harish Patil, Cristiano Pereira, Mack Stallcup, Gregory Lueck, and James Cownie. 2010. PinPlay: A Framework for Deterministic Replay and Reproducible Analysis of Parallel Programs. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10)*. 2–11. https://doi.org/10.1145/1772954.1772958

[60] Vojtech Pavlik. 2021. kGraft: Live Kernel Patching. Retrieved January 18, 2023 from https://www.suse.com/c/kgraft-live-kernel-patching/

[61] Phoronix 2023. Phoronix Multicore. Retrieved Janurary 18, 2023 from https://openbenchmarking.org/suite/pts/multicore

[62] Josh Poimboeuf. 2022. Introducing kpatch: Dynamic Kernel Patching. Retrieved January 18, 2023 from https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching

[63] Arash Pourhabibi, Mark Sutherland, Alexandros Daglis, and Babak Falsafi. 2021. Cerebros: Evading the RPC Tax in Datacenters. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '21)*. Association for Computing Machinery, New York, NY, USA, 407–420. https://doi.org/10.1145/3466752.3480055

[64] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: Core-Aware Thread Management. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. USENIX Association, Carlsbad, CA. https://www.usenix.org/conference/osdi18/presentation/qin

[65] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray, and Stephen C. Purcell. 1980. Pilot: An Operating System for a Personal Computer. *Commun. ACM* 23, 2 (Feb. 1980), 81–92. https://doi.org/10.1145/358818.358822

[66] Redox 2022. Redox. Retrieved Janurary 18, 2023 from https://www.redox-os.org/

[67] Michiel Ronsse and Koen De Bosschere. 1999. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Trans. Comput. Syst.* 17, 2 (may 1999), 133–152. https://doi.org/10.1145/312203.312214

[68] Hugo Sadok, Zhipeng Zhao, Valerie Choung, Nirav Atre, Daniel S. Berger, James C. Hoe, Aurojit Panda, and Justine Sherry. 2021. We Need Kernel Interposition over the Network Dataplane. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 152–158. https://doi.org/10.1145/3458336.3465281

[69] Yasushi Saito. 2005. Jockey: A User-Space Library for Record-Replay Debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging (AADEBUG'05)*. Association for Computing Machinery, New York, NY, USA, 69–76. https://doi.org/10.1145/1085130.1085139

[70] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, and Boris Weissman. 2007. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS.*

A Appendix

Arrayfire, 1	Arrayfire, BLAS CPU
Arrayfire, 2	Arrayfire, Conjugate Gradient CPU
Cassandra, 1	Cassandra, Writes
ASKAP, 4	ASKAP, Hogbom Clean OpenMP
Cpuminer, 2	Cpuminer, Triple SH-256 Onecoin
Cpuminer, 3	Cpuminer, Quad SHA-256 Pyrite
Cpuminer, 4	Cpuminer, Myriad-Groestl
Cpuminer, 6	Cpuminer, Blake-2 S
Cpuminer, 11	Cpuminer, Skeincoin
Ffmpeg, 1, 1	Ffmpeg, libx264, Live
Graphics-Magick, 4	Graphics-Magick, Resizing
OIDN, 1	OIDN, RT.hdr_alb_nrm.3840x2160
OIDN, 2	OIDN, RT.ldr_alb_nrm.3840x2160
OIDN, 3	OIDN, RTLightmap.hdr.4096x4096
Rodina, 3	Rodina, OpenMP Leukocyte
Zstd, 2	Zstd, 3 Long Mode Compression
Zstd, 4	Zstd, 8 Long Mode Compression
AVIFEnc, 4	AVIFEnc, 6 Lossless
Libgav1, 1	Libgav1, Summer Nature 1080p
Libgav1, 2	Libgav1, Summer Nature 4k
Libgav1, 3	Libgav1, Chimera 1080p
Libgav1, 4	Libgav1, Chimera 1080p 10-bit
OneDNN, 4, 1	OneDNN, IP Shapes 1D, f32
OneDNN, 5, 1	OneDNN, IP Shapes 3D, f32
OneDNN, 7, 1	OneDNN, RNN Training, f32
OneDNN, 7, 2	OneDNN, RNN Training, u8s8f32
OneDNN, 7, 3	OneDNN, RNN Training, bf16bf16bf16

Table 7. Full names of the Phoronix Multicore benchmarks presented in the paper.

A.1 WFQ Functional Equivalence

We run several benchmarks to evaluate whether our Enoki WFQ scheduler correctly implements the expected behavior of a WFQ scheduler. We compare the Enoki WFQ scheduler to CFS, also a WFQ scheduler, to ensure that our scheduler implements equivalent behavior.

One benchmark evaluates whether the scheduler fairly shares CPU time among ready tasks. This benchmark started five tasks that each perform CPU-intensive work, specifically repeatedly adding to a value. By default these tasks are placed on different cores. We expect these tasks to complete in roughly the same amount of time due to the lack of competition. We then force the schedulers to place all of the tasks on the same core. We saw the expected behavior. On both schedulers, the tasks took around 4.6 seconds to complete when the tasks are not co-located. When the tasks were co-located, the tasks took around 22.2 seconds.

We also evaluate how the schedulers handle weighting by reducing one of the five tasks to minimum priority. We would expect the lowest priority task to complete slower than the others; the other tasks should share time fairly and complete in the same amount of time. Again, we saw the expected behavior on both schedulers. When one task was reduced to the minimum priority, all four other tasks complete in around 17.6 seconds and the lowest priority task completes in 4.4 seconds later.

A third benchmark tests task placements. We start one task per core with each task performing CPU-intensive work, specifically repeatedly adding to a value. We expect both schedulers to default to placing each task on a separate core. We then force one task to change cores. With no movement, each task completes in around 9 seconds on both schedulers and there is very low variation in task runtime. When a task is moved, all tasks still complete in around 9 seconds on both schedulers. CFS shows roughly the same variation in task runtime even when a task was moved. The Enoki WFQ scheduler shows higher standard deviation in task runtimes when a task is moved, from 0.001s to 0.018s, because it takes longer to move a task due to its less sophisticated methods of rebalancing tasks.

B Artifact Appendix

B.1 Abstract

Our artifact consists of a custom kernel and framework for safe, modular schedulers. We document how to run our primary scheduler on the main set of benchmarks.

B.2 Description & Requirements

We have a main repository at https://github.com/smiller123/enoki. It points to the three repositories that make up the project. The documented scheduler and benchmark are included in one of these repositories. More information is provided in the README files. We tested this code on a QEMU VM image with 6 cores and a 90 GiB image. If you would like access to our VM image, send an email to sm237@cs.washington.edu.

B.2.1 How to access

The code can be accessed at https://github.com/smiller123/enoki. It points to three subrepositories located at https://gitlab.cs.washington.edu/sm237/enoki-kernel, https://gitlab.cs.washington.edu/sm237/enoki-schedulers, https://github.com/smiller123/bento/tree/enoki-support. All repositories are public.

B.2.2 Software dependencies

Instructions for installing all software dependencies are provided in the repository. First install the kernel. Instructions are found in the enoki-kernel repository. You will also need to install Rust. Instructions to install the correct version are in enoki-schedulers.

B.2.3 Benchmarks

We document the perf pipe scheduling latency benchmark and the Phoronix application benchmarks. The perf pipe benchmark is a simple benchmark that starts two processes that repeatedly sleep and wake each other up. It was previously known as Hackbench. The Phoronix benchmark suite includes a variety of application workloads. Some of the other benchmarks are included in the repository, but are not documented or automated yet.

B.3 Set-up

Detailed instructions are included in the repositories. At a high level, first build and install the custom kernel. Then build the scheduler and load it into the kernel. Run the benchmarks. Then unload the scheduler.

B.4 Evaluation workflow

I will only discuss the claims relevant to the benchmarks we have documented at this point.

B.4.1 Experiments

Experiment (E1): [perf pipe] [A few minutes to compile, a few seconds to run the benchmark]: A simple scheduling latency benchmark. It creates two processes that sleep and wake each other up.

[How to]

[Preparation] Compile and load the scheduler in the enoki-schedulers subrepository. Compile and install perf from the custom kernel source (instructions provided in the enoki-schedulers subrepository README). Then build the benchmark file enoki_pipe_test.c, described in the enoki-schedulers subrepository and found in the perf_test directory.

[Execution] Execute `sudo ./enoki_test`. For the baseline, execute `./cfs_test`. Do not use the machine while the tests are running.

[Results] The benchmark will output the time per operation in μs/op.

Experiment (E2): [Phoronix] [A few minutes to compile, 1.5-2 hours to run the benchmark for each configuration]: A set of application benchmarks.

[How to]

[Preparation] Compile and load the scheduler in the enoki-schedulers subrepository. Then build the enoki_test.c benchmark file, also described in the enoki-schedulers subrepository and found in the phoronix_test directory.

[Execution] Execute `./enoki_test`. For the baseline, execute `./cfs_test`. Do not use the machine while the tests are running.

[Results] This will run the set of Phoronix benchmarks described in the paper three times each. The results can be compared using
`../phoronix-test-suite/phoronix-test-suite compare-results-to-baseline cfseval enokieval`.

B.5 Notes on Reusability

Other schedulers can be implemented using Enoki by writing schedulers that follow the same pattern of the schedulers in the enoki-schedulers directory. Specifically, the scheduler's main code should be included in the `src/sched.rs` file. Some other files should be copied, just replacing the scheduler name and/or data structure: `Makefile`, `kernel/Cargo.tomp`, `kernel/Kbuild`, `kernel/src/lib.rs`, `kernel-record/Cargo.toml`, `kernel-record/Kbuild`, `kernel-record/src/lib.rs`, `replay/Cargo.toml`, `replay/src/main.rs`, and `src/module.c`. The userspace part of the record can be copied unchanged: `record/Cargo.toml`, `record/src/main.rs`