Integrating Unikernel Optimizations in a General Purpose OS

Ali Raza
Boston University
aliraza@bu.edu

Thomas Unger
Boston University
tommyu@bu.edu

Matthew Boyd
Boston University
mcboyd@bu.edu

Eric Munson
Boston University
munsoner@bu.edu

Parul Sohal
Boston University
psohal@bu.edu

Ulrich Drepper
Red Hat
drepper@redhat.com

Richard Jones
Red Hat
rjones@redhat.com

Daniel Bristot de Oliveira
Red Hat
bristot@redhat.com

Larry Woodman
Red Hat
lwoodman@redhat.com

Renato Mancuso
Boston University
rmancuso@bu.edu

Jonathan Appavoo
Boston University
jappavoo@bu.edu

Orran Krieger
Boston University
okrieg@bu.edu

Abstract

We explore if unikernel techniques can be integrated into a general-purpose OS while preserving its battle-tested code, development community, and ecosystem of tools, applications, and hardware support. Our prototype demonstrates both a path to integrate unikernel techniques in Linux and that such techniques can result in significant performance advantages. With a re-compilation and link to our modified kernel, applications show modest performance gains. Expert developers can optimize the application to call internal kernel functionality and optimize across the application/kernel boundary for more significant gains. While only one process can be optimized, standard scripts can be used to launch it, and other processes can run alongside it, enabling the use of standard user-level tools (prof, bash,...) and support for both virtual and physical servers. The changes to the Linux kernel are modest (1250 LOC) and largely part of a configuration build target.

1 Introduction

There is growing evidence that the structure of today's general purpose operating systems is problematic for a number of key use cases. For example, applications that require high-performance I/O use frameworks like DPDK [1] and SPDK [2] to bypass the kernel and gain unimpeded access to hardware devices [13, 38]. In the cloud, client workloads are typically run inside dedicated virtual machines, and a kernel designed to multiplex the resources of many users and processes is instead being replicated across many single-user, often single-process, environments [41].

In response, there has been a resurgence of research systems exploring the idea of a libraryOS, or *a unikernel*, where an application is linked with a specialized kernel and deployed directly on virtual hardware [11]. Compared with Linux, unikernels have demonstrated significant advantages in boot time [17, 24], security [42], resource utilization [8, 26], and I/O performance [39].

As with any operating system, widespread adoption of a unikernel will require enormous and ongoing investment by a large community. Justifying this investment is difficult since unikernels target only niche portions of the broad use-cases of general-purpose OSes. In addition to their intrinsic limitation as single application environments, with few exceptions, existing unikernels support only virtualized environments and, in many cases, only run on a single processor core. Moreover, they do not support accelerators (e.g., GPUs and FPGAs) that are increasingly critical to achieving high performance in a post Dennard scaling world.

Some systems have demonstrated that it is possible to create a unikernel that re-uses much of the battle-tested code of a general-purpose OS and supports a wide range of applications. Examples include NetBSD based Rump Kernel [15], Windows based Drawbridge [34] and Linux based Linux Kernel Library (LKL)[35]. These systems, however, require significant changes to the general-purpose OS, resulting in a fork of the codebase and community. As a result, ongoing investments in the base operating system are not necessarily applicable to the forked unikernel.

To avoid the investment required for a different OS, the recent Lupine [20] and X-Containers [40] projects explore exploiting Linux's innate configurability to enable application-specific customizations. These projects avoid the hardware overhead of system calls between user and kernel mode, but to avoid code changes to Linux, they do not explore deeper optimizations. Essentially these systems preserve the boundary between the application and the underlying kernel, giving

up on any unikernel performance advantages that depend on linking the application and kernel code together.

The Unikernel Linux (UKL) project started as an effort to exploit Linux's configurability to try to create a new unikernel in a fashion that would avoid forking the kernel. If this is possible, we hypothesized that we could create a unikernel that would support a wide range of Linux's applications and hardware, while becoming a standard part of the ongoing investment by the Linux community. Our experience has led us to a different, more powerful goal; enabling a kernel that can be configured to span a spectrum between a general-purpose operating system and a pure unikernel.

At the general-purpose end of the spectrum, if all UKL configurations are disabled, a standard Linux kernel is generated. The simplest *base model* configuration of UKL supports many applications, albeit with only modest performance advantages. Like unikernels, a single application is statically linked into the kernel and executed in privileged mode. However, the base model of UKL preserves most of the invariants and design of Linux, including a separate page-able application portion of the address space and a pinned kernel portion, distinct execution modes for application and kernel code, and the ability to run multiple processes. The changes to Linux to support the UKL base model are modest (~550 LoC), and the resulting kernel support all hardware and applications of the original kernel as well as the entire Linux ecosystem of tools for deployment and performance tuning. UKL base model shows a modest 5% improvement in syscall latency.

Once an application is running in the UKL base model, a developer can move along the spectrum towards a unikernel by 1) adapting additional configuration options that may improve performance but will not work for all applications, and/or 2) modifying the applications to directly invoke kernel functionality. Example configuration options we have explored avoid costly transition checks between application and kernel code, use simple return (rather than `iret`) from page faults and interrupts, and use shared stacks for application and kernel execution etc. Application modifications can, for example, avoid scheduling and exploit application knowledge to reduce the overhead of synchronization and polymorphism. Experiments show up to 83% improvement in syscall latency and substantial performance advantages for real workloads, e.g., 26% improvement in Redis throughput while improving tail latency by 22%. A latency sensitive workloads show 100 times improvement. The full UKL patch to Linux, including the base model and all configurations, is 1250 LoC.

Contributions of this work include:

1. An existence proof that unikernel techniques can be integrated into a general-purpose OS in a fashion that does not need to fragment/fork it.
2. A demonstration that a single kernel can be adopted across a spectrum between a unikernel and a general purpose OS.
3. A demonstration that performance advantages are possible; applications achieve modest gains with no changes, and incremental effort can achieve more significant gains.

We discuss our motivations and goals for this project in Section 2, and ur overall approach to bring unikernel techniques to Linux in Section 3. Section 4 describes key implementation details. In Section 5, we evaluate and discuss the implications of the current design and implementation. Finally, Section 6 and 7 contrast UKL to previous work and describe research directions that this work enables.

2 Motivation & Goals

UKL seeks to explore a spectrum between a general-purpose operating system and a unikernel in order to: (1) enable unikernel optimizations demonstrated by earlier systems while preserving a general-purpose operating system's (2) broad application support, (3) broad hardware support, and (4) the ecosystem of developers, tools and operators. We motivate and describe each of these four goals.

2.1 Unikernel optimizations

Unikernels fundamentally enable optimizations that rely on linking the application and kernel together in the same address space. Example optimizations that previous systems have adopted include 1) avoiding ring transition overheads [20]; 2) exploiting the shared address space to pass pointers rather than copying data [39]; 3) exploiting fine-grained control over scheduling decisions, e.g., deferring preemption in latency-sensitive routines; 4) enabling interrupts to be efficiently dispatched to application code [39]; 5) exploiting knowledge of the application to remove code that is never used [24]; 6) employing kernel-level mechanisms to optimize locking and memory management [16], for instance, by using Read-Copy-Update (RCU) [28], per-processor memory, and DMA-aided data movement; and 7) enabling compiler, link-time, and profile-driven optimizations between the application and kernel code.

Ultimately our goal with UKL is to explore the full spectrum between general purpose and highly specialized unikernels. For this paper, our goal is to enable applications to be linked into the Linux kernel, and explore what, if any, improvements can be achieved by modest changes to the application and general-purpose system.

2.2 Application support

One of the fundamental problems with unikernels is the limited set of applications that they support. By their nature, unikernels only enable a single process, excluding any application that requires helper processes, scripts, etc. Moreover, the limited set of interfaces typically requires substantial porting effort for any application, and library that the application uses.

UKL seeks to enable unikernel optimizations to be broadly applicable. Our goal is to enable any unmodified Linux application and library to use UKL, with a re-compilation, as long as only one application needs to be linked into the kernel. Once the application is functional, the developer can incrementally enable unikernel optimizations/configurations. A large set of applications should be able to achieve some gain on the general-purpose end of the spectrum, while a much smaller set of applications will be able to achieve more substantial gains as we move toward the unikernel end.

2.3 Hardware support

Another fundamental problem with unikernels is the lack of support for physical machines and devices. While recent unikernel research has mostly focused on virtual systems, some recent [14, 39] and previous [4, 9, 11, 21, 31] systems have demonstrated the value of per-application specialized operating systems on physical machines. Unfortunately, even these systems were limited to very specific hardware platforms with a restricted set of device drivers. This precludes a wide range of infrastructure applications (e.g., storage systems, schedulers, networking toolkits) that are typically deployed bare-metal. Moreover, the lack of hardware support is an increasing problem in a post-Dennard scaling world, where performance depends on taking advantage of the revolution of heterogeneous computing.

Our goal with UKL is to provide a unikernel environment capable of supporting the complete HCL of Linux, allowing applications to exploit any hardware (e.g. GPUs, TPUs, FPGAs) enabled in Linux. Our near term goal, while supporting all Linux devices, is to focus on x86-64 systems. Much like KVM became a feature of Linux on x86 and was then ported to other platforms; we expect that, if UKL is accepted upstream, communities interested in non-x86 architectures will take on the task of porting and optimizing UKL for their platforms.

2.4 Ecosystem

While application and hardware support are normally thought of as the fundamental barriers for unikernel adoption, the problem is much larger. Linux has a huge developer community, operators that know how to configure and administer it, a massive body of battle-tested code, and a rich set of tools to support functional and performance debugging and configuration.

Our goal with UKL is, while enabling developers to adopt extreme optimizations that are inconsistent with the broader ecosystem, the entire ecosystem should be preserved on the general-purpose end of the spectrum. This means operational as well as functional and performance debugging tools should just work. Standard application and library testing systems should, similarly, just work.[1] Most of all, the base changes needed to enable UKL need to be of a nature that they don't break assumptions of the battle tested Linux code, can be accepted by the community, and can be tested and maintained as development on the system progresses.

[1] In fact, we used a great deal of `glibc` and libpthread's internal unit tests to identify and fix problems within UKL.

3 Design

UKL's `base model` enables an application to be linked into the kernel while preserving the (known and unknown) invariants and assumptions of applications and Linux. Once an application runs on UKL, an expert programmer can then adopt specific unikernel optimizations by choosing (additional) configuration options and/or modifying the application to invoke kernel functionality directly. We first describe the base model and then some of the unikernel optimizations we have explored.

3.1 Base Model

UKL is similar to many unikernels in that it involves modifications to a base library and a kernel, and has a build process that enables a single application to be statically linked with kernel code to create a bootable kernel. In the case of UKL, the modifications are to glibc and the Linux kernel. As a result of the wide variety of architectures supported by `glibc` and Linux, it was possible to introduce the majority of changes we required in a new UKL target architecture; most of the hooks we require to override code already exist in common code.

The base model of UKL differs from unikernels in 1) support for multiple processes, 2) address space layout, and 3) in maintaining distinct execution models for applications and kernel.

Support for multiple processes: One key area where UKL differs from unikernels is that, while only one application can be linked into the kernel, UKL enables other applications to run unmodified on top of the kernel. Support for multiple processes is critical to support many applications that are logically composed of multiple processes (§2.2), standard configuration and initialization scripts for device bring-up (§2.3), and the tooling used for operations, debugging and testing (§2.4).

Address space layout: UKL preserves the standard Linux virtual address space split between application and kernel. The application heap, stacks, and mmapped memory regions are all created in the user portion of the address space. Kernel data structures (e.g., task structs, file tables, buffer cache) and kernel memory management services (e.g., vmalloc and kmalloc) all use the kernel portion of the address space. Since the kernel and application are compiled and linked together, the application (and kernel) code and data are all allocated in the kernel portion of the virtual address space.

We found it necessary to adapt this address space layout because Linux performs a simple address check to see if an address being accessed is pinned or not; modifying this layout would have resulted in changes that would be difficult to get accepted (2.4). Unfortunately, this layout has two negative

implications for application compatibility. First, (see 4) applications have to be compiled with different flags to use the higher portion of the address space. Second, it may be problematic for applications with large initialized data sections that in UKL are pinned.

Execution models: Even though the application and kernel are linked together, UKL differs from unikernels in providing fundamentally different execution models for application and kernel code. Application code uses large stacks (allocated from the application portion of the address space), is fully preemptable, and uses application-specific libraries. This model is critical to enabling a large set of applications to be supported without modification (2.2).

Kernel code, on the other hand, runs on pinned stacks, accesses pinned data structures, and uses kernel implementation of common routines. This model was required to avoid substantial modifications to Linux that would not have been accepted by the community (2.4).

On transition between the execution models, UKL performs the same entry and exit code of the Linux kernel, with the difference that: 1) transitions to kernel code are done with a procedure call rather than a `syscall`, and 2) transitions from the kernel to application code are done via a `ret` rather than a `sysret` or `iret`. This transition code includes changing between application and kernel stacks, RCU handling, checking if the scheduler needs to be invoked, and checking for signals. In addition, it includes setting a per-thread `ukl_mode` to identify the current mode of the thread so that subsequent interrupts, faults and exceptions will go through normal transition code when resuming interrupted application code.

3.2 Unikernel Optimizations

While preserving existing execution modes enables most applications to run with no modifications on UKL, the performance advantages of just avoiding `syscall`, `sysret`, and `iret` operations are, as expected, modest. However, once an application is linked into the kernel, different unikernel optimizations are possible. First, a developer can apply a number of configuration options that may improve performance. Second, a knowledgeable developer[2] can improve performance by modifying the application to call internal kernel routines and violating, in a controlled fashion, the normal assumptions of kernel versus application code.

3.2.1 Configuration Options. Here we discuss the configuration options that have the biggest impact.

Bypassing entry/exit code: On Linux, whenever control transitions between application and kernel through system calls, interrupts, and exceptions, some entry and exit code is executed, and it is expensive. We introduced a configuration (UKL_BYP) that allows the application, on a per-thread basis, to tell UKL to bypass entry and exit code for some number of transitions between application and kernel code. As we will see, this model results in significant performance gains for applications that make many small kernel requests.

Config	Feature
UKL_BYP	Bypass entry exit code
UKL_NSS	Avoid stack switches
UKL_NSS_PS	Avoid stack switches with pinned user stacks
UKL_RET	Replace iret with ret
UKL_PF_DF	Use dedicated stack on double faults
UKL_PF_SS	Use dedicated stack on all faults

Table 1. UKL Configuration options

A developer can invoke an internal kernel routine directly, where no automatic transition paths exist, e.g., invoking `vmalloc` to allocate pinned pre-allocated kernel memory rather than normal application routines. The use of such memory not only avoids subsequent faults but also results in less overhead when kernel interfaces have to copy data to and from that memory.

Avoiding stack switches: Linux runs applications on dynamically sized user stacks, and kernel code on fixed-sized, pinned kernel stacks. This stack switch, every time kernel functionality is invoked, breaks the compiler's view and limits cross-layer optimizations, e.g., link-time optimizations, etc. The developer can select between two UKL configurations that avoid stack switching(UKL_NSS and UKL_NSS_PS); where (see implementation) each is appropriate for a different class of application. Currently, LTO in Linux is only possible with CLANG and glibc, and some other libraries can only be compiled with gcc. There are efforts underway in the community to both enable `glibc` to be compiled with CLANG and to enable Linux LTO with gcc. We hope by the time this paper is published, we will be able to demonstrate the results of LTO with one or the other of these.

`ret` *versus* `iret`*:* Linux uses `iret` when returning from interrupts, faults and exceptions. `iret` can be an expensive instruction when compared to a simple `ret` instruction, but it makes sense when control has to be returned to user mode because it guarantees atomicity while changing the privilege level, updating instruction and stack pointers, etc. UKL_RET configuration option uses `ret` and ensures atomicity by enabling interrupts only after returning to the application stack.

3.2.2 Application Modifications. Along with the above mentioned configurations, applications can be modified to gain further performance benefits. Developers can, by taking advantage of application knowledge, explore deeper optimizations. For example, they may be able to assert that only one

[2]Expertise is needed to perform these customizations. For example, if an application calls an internal kernel routine passing a pointer to an application data structure that resides on a page that has not yet been accessed/allocated, the kernel will fail. We are just starting to develop a body of use cases and examples that should inform developers on the care they should take for different optimizations.

thread is accessing a file descriptor and avoid costly locking operations. As another example, they may know *a priori* that an application is using TCP and not UDP and that a particular write operation in the application will always be to a TCP socket, avoiding the substantial overhead of polymorphism in the kernel's VFS implementation. As we optimize specific operations, we are building up a library of helper functions that cache and simplify common operations.

UKL base model ensures that the application and kernel execution models stay separate, with proper transitions between the two. But applications may find it beneficial to run under the kernel execution model, even for short times. Applications can toggle a per-thread flag which switches them to the kernel-mode execution, allowing application threads to be treated as kernel threads, so they won't be preempted. This can be used as a 'run-to-completion' mode where performance-critical paths of the application can be accelerated.

4 Implementation

The size of the UKL base model patch to Linux kernel 5.14 is approximately 550 lines and the full UKL patch (base model plus all the configuration options mentioned in Table 1) is 1250 lines. The vast majority of these changes are target-specific, i.e., in the x86 architecture directory.

UKL takes advantage of the existing kernel Kconfig and `glibc` build systems. These allow target-specific functionality to be introduced that doesn't affect generic code or code for other targets. All code changes made in UKL base model and subsequent versions are wrapped in macros which can be turned on or off through kernel and `glibc` build time config options. All the changes required are compiled out when Linux and `glibc` are configured for a different target.

We found that UKL patch can be so small due to many favorable design decisions by the Linux community. For instance, Linux's low level transition code has recently undergone massive rewritings to reduce assembly code and move functionality to C language. This has allowed UKL transition code changes to be localized to that assembly code. Further, the ABI for application threads dedicates a register (`fs` to point to thread-local storage, while kernel threads have no such concept but instead dedicate a register (`gs` to point to processor-specific memory. If a register was used by both Linux and `glibc`, UKL would have had to add code to save and restore it on transitions; instead, both registers can be preserved.

In addition to the kernel changes, about 4,700 lines of code are added or changed in `glibc`. These number is inflated because according to the `glibc` development approach, any file that needs to be modified has to be first copied to a new subdirectory and then modified. All the UKL changes are well contained in a separate directory. At build time, this directory is searched first for a target file before searching the default location.

Building UKL:. UKL code in Linux (protected by `#ifdefs`) is enabled by building with specific Kconfig options. UKL requires the application's and the needed user libraries' code to be compiled and statically linked with the kernel, so dynamically loadable system libraries cannot be used. All code must be built with two special flags. The first flag disables the red zone (`-mno-red-zone`). Hermitux [33] takes the design approach of forcing all faults, interrupts, and exceptions to use dedicated stacks through the Intel interrupt stack table (IST) mechanism. This allows the red zone to remain safe while all interrupts etc., are serviced on dedicated kernel stacks. While we could have adopted this technique into UKL, it would have required drastic code changes. The second flag (`-mcmodel=kernel`) generates the code for kernel memory model. This is needed because application code has to link with kernel code and be loaded in the highest 2GB of address space instead of the lower 2GB that is the default for user code.

The modified kernel build system combines the application object files, libraries, and the kernel into a final `vmlinux` binary which can be booted bare-metal or virtual. To avoid name collisions, before linking the application and kernel together, all application symbols (including library ones) are prefixed with `ukl_`. Kernel code typically has no notion of thread-local storage or C++ constructors, so the kernel's linker script is modified to link with user-space code and ensure that thread-local storage and C++ constructors work. Appropriate changes to kernel loader are also made to load the new ELF sections along with the kernel.

Changes to* `execve`*: We modified `execve` to skip certain steps (like loading the application binary, which does not exist in UKL), but most steps run unmodified. Of note, `execve` will jump straight to the `glibc` entry point when running the UKL thread instead of trying to read the application binary for an entry point. `glibc` initialization happens almost as normal, but when initializing thread-local storage, changes had to be made to allow `glibc` to read symbols set by the kernel linker script instead of trying to read them from the (non-existent) ELF binary. C++ constructors run in the same way as in a normal process. Command-line parameters to `main` are extracted from a part of the Linux kernel command line, allowing these to be changed without recompilation.

Transition between application and kernel code: On transitions between application and kernel code, the normal entry and exit code of the Linux kernel is executed, with the only change being that transitions code use `call/ret` instead of `syscall/sysret`.

The different configurations of UKL, mentioned in Table 1 involve changes to the transitions between application and kernel code. All changes were made through Linux (`SYSCALL_DEFINE`) macros and `glibc` (`INLINE_SYSCALL`) macros. For example, to enable UKL_BYP mode, we generate a stub in the kernel `SYSCALL_DEFINE` macro that is invoked by the corresponding `glibc` macro. We use a per-thread flag ((ukl_byp)

to identify if the bypass optimization is turned on or off for that thread.

Linux tracks whether a process is running in user mode or kernel mode through the value in the `CS` register, but UKL is always in kernel mode (except for the normal user-space, which runs in user mode). So the UKL thread tracks this in a flag (`ukl_mode`) in the kernel's thread control block i.e., `task_struct`.

The UKL_RET configuration option replaces `iret` after application code is interrupted by a page fault or interrupt with a `ret`. The challenge is that we cannot enable interrupts until we have switched from the kernel stack to the application stack, or the system might land in an undefined state. To do so, we first copy the return address and user flags from the current stack to the user stack. Then we switch to the user stack, and this ensures that even if interrupts are enabled now, we are on the correct stack where we can return to again. We then pop the flags, and then do a simple `ret` because return address is already on stack. We make sure to restore user flags at the very end, because restoring user flags would turn interrupts on. This has allowed us performance improvement while also ensuring correct functionality.

Enabling shared stacks: In the UKL base model, a stack switch occurs between user and kernel stack when transition between user and kernel code happens. To enable link-time optimizations, it is important to avoid those transitions. The UKL_NSS configuration option involves changes to the low level transition code to avoid stack switch. While this works, it limits how UKL can be deployed because it breaks the expectation that different processes can run alongside the UKL application. To illustrate the problem, consider the case of an inter-processor interrupt to another processor for TLB invalidation. In this case, Linux stores information on the current process stack, which is a user stack if stack switching is turned off. On the other processor, some non-UKL thread might be running, which is interrupted by the IPI. Kernel will inherit that other process's page tables and then try to access the information stored on the UKL thread's user stack, essentially trying to access user pages that might not be mapped in the current page tables, resulting in a kernel panic. When this configuration option is used, required tools and setup scripts need to run before the UKL application runs, and clean up scripts, etc., run after the UKL application finishes execution.

The inability to run concurrent processes alongside the UKL application precluded a class of applications. So the UKL_NSS_PS configuration option allocates large, fixed-sized stacks in the kernel part of the address range. This allowed multiple processes to run concurrently. This, however precluded a different class of applications, i.e., those which create a large number of threads or forks, etc., which might exhaust the kernel part of the address space.

Page-faults: If UKL_NSS configuration option is on, deadlocks can occur. Imagine if some kernel memory management code was being executed, e.g., `mmap`, the current thread must have taken a lock on the memory control struct (`mm_struct`). During the execution of that code, if a stack page fault occurs (which is normal for user stacks), control moves to the page fault handler, which then tries to take the lock of `mm_struct` to read which virtual memory area (VMA) the faulting address belongs to and how to handle it. Since the lock was already taken, the page fault handler waits. But the lock will never be given up because that same thread is in the page fault handler. We solved this by saving a reference to user stack VMA when a UKL thread or process is created. In case of page faults, while user stacks are used throughout, we first check if the faulting address is a stack address by comparing it against the address range given in the saved VMA. If so, we know it's a stack address, and the code knows how to handle it without taking any further lock. If not, we first take a lock to retrieve the correct VMA and move forward normally.

In kernel mode, on a page fault, the hardware does not switch to a fresh stack. It tries to push some state on its current (user) stack. Since there is no stack left to push this state, UKL gets a double fault. We fix this through the UKL_PF_DF configuration option by causing the hardware to raise a double fault and then checking, in the double fault handler, if the fault is to a stack page, and if so, branch to the regular page fault handler (double fault always gets a dedicated stack, so it does not *triple fault*).

We also came up with the UKL_PF_SS configuration option to solve this problem, i.e., by updating the IDT to ensure that the page fault handler always switches to a dedicated stack through the Interrupt Stack Table (IST) mechanism.

Clone and Fork: To create UKL threads, the user-space pthread library runs `pthread_create` which further calls `clone`. We modified this library to pass a new flag `CLONE_UKL` to ensure the correct initial register state is copied into the new task either from the user stack or kernel stack, depending on whether the parent is configured to switch to kernel stack or not.

5 Evaluation

After our experimental environment (§5.1), §5.2 discusses our experience with UKL supporting the fundamental non-performance goals of enabling Linux's application support, HCL, and ecosystem.

In Section 5.3 microbenchmarks are used to evaluate the performance of UKL on simple system calls (§5.3.1), more complex system calls (§5.3.2) and page faults (§5.3.3). We find that, while the advantage of just avoiding the hardware overhead of system calls is small, the advantage of adopting unikernel optimizations is large for simple kernel calls (e.g., 83%) and significant for page faults (e.g., 12.5%). Moreover, the improvement is significant even for expensive kernel calls that transfer 8KB of data (e.g., 24%).

In Section 5.4 we evaluate applying unikernel optimizations to both throughput (Redis §5.4.1, Memcached §5.4.2) and latency bound (Secrecy §5.4.3) applications. We find that configuration options provided by UKL can enable significant throughput improvements (e.g., 12%) and a simple 10 line change in Redis code results in more significant gains (e.g., 26%). The results are even more dramatic for latency-sensitive applications where configuration changes result in 15% improvement and a trivial application change enables a 100x improvement in performance.

5.1 Experimental Setup

Experiments are run on Dell R620 servers configured with 128G of ram arranged as a single NUMA node. The servers have two sockets, each containing an Intel Xeon CPU E5-2660 0 @ 2.20GHz, with 8 real cores per socket. The processors are configured to disable Turbo Boost, hyper-threads, sleep states, and dynamic frequency scaling. The servers are connected through a 10Gb link and use Broadcom NetXtreme II BCM57800 1/10 Gigabit Ethernet NICs. Experiments run on multiple computers use identically configured machines attached to the same top of rack switch to reduce external noise. On the software side, we use Linux 5.14 kernel and `glibc` version 2.31. Linux and different configurations of UKL were built with same compile-time config options. We ran experiments on virtual and physical hardware and got consistent and repeatable results. In the interest of space, we only report bare-metal numbers unless stated otherwise.

5.2 Linux application hardware & ecosystem

The fundamental goals of the UKL project are to integrate unikernel optimizations without losing Linux's broad support for applications, hardware, and ecosystem. We discuss each of these three goals in turn.

Application support: As expected, we have had no difficulty running any Linux application as normal user-level processes on our modified kernel. We have used hundreds of unmodified binaries running as normal user-level processes without effort. That includes all the standard UNIX utilities, bash, different profilers, `perf`, and eBPF tools.

Dozens of unmodified applications have been tested as optimization targets for UKL. These include Memcached [29], Redis [36], Secrecy [22], a small TCP echo server, simple test programs for C++ constructors and the STL, a complex C++ graph based benchmark suite [7], a performance benchmark called LEBench [37], and a large number of standard `glibc` and pthread unit test programs.

There are some challenges in getting some applications running on UKL. First, as expected, one needs to be able to re-compile and statically link both the application and all its dependencies. Second, we have hit a number of programs that by default invoke `fork` followed by `exec` e.g., Postgress, and many that are dependent on the dynamic loader through calls to `dlopen` and others.[3] Third, we have run into issues of proprietary applications available in only binary form, e.g., user-level libraries for GPUs.

Project	LoC	Files	SubSys	Outcome
Popcorn	7763	64	14	Out of tree
NetGPU	3827	45	14	Rejected
DAMON	3805	24	3	Rejected
KML	3177	70	16	Out of tree
BPFStruct	2639	32	10	Accepted
BPFDump	2343	32	8	Accepted
ArmMTE	1764	63	14	Accepted
NFTOffload	1579	56	24	Accepted
UKL	1250	33	10	-
KRSI	1085	29	11	Accepted
LoopFS	891	27	5	Rejected
FSGSBASE	562	16	9	Accepted
BPFDisp	501	11	9	Accepted
ArmAsym	370	13	9	Rejected
BPFSleep	315	23	9	Accepted
IOURestrictions	194	2	2	Accepted
CapPerfMon	98	18	14	Accepted

Table 2. Comparison of UKL patch to Kernel-Mode Linux (KML) and a selection of Linux features described in Linux Weekly News (LWN) articles in 2020. We show patch size, files touched (how complex it is to reason about), subsystems impacted (number of upstream kernel maintainers who need to review and approve it), and the current status of the change.

Hardware support: For hardware, we have not run into any compatibility issues and have booted or kexeced to UKL on five different x86-64 servers and virtualization platforms. The scripts and tools used to deploy and manage normal Linux machines were used for UKL deployments as well.

Ecosystem: Due to having a full-fledged userspace, we have been able to run all the different applications, utilities, and tools that can run on unmodified Linux. This has been extremely critical in building UKL, i.e., we use all the debugging tools and techniques available in Linux. We have been able to profile UKL workloads with `perf` and able to identify code paths that could be squashed for performance benefits (see fig. 5).

The UKL patch size for the base model is around 550 lines, and the full UKL patch with all the configurations is 1250 lines. Since the patch is small and non-invasive, we are hopeful that we can work with the Linux community towards upstream acceptance.

Table 2 compares the UKL patch to Kernel-Mode Linux (KML) and a selection of Linux features described in Linux

[3] UKL supports a modified `fork`, where initialized data (kernel page tables) are not copy-on-write, but are shared across forks. Any applications depending on `exec`, correct semantics of `fork` or the dynamic loader are not supported.

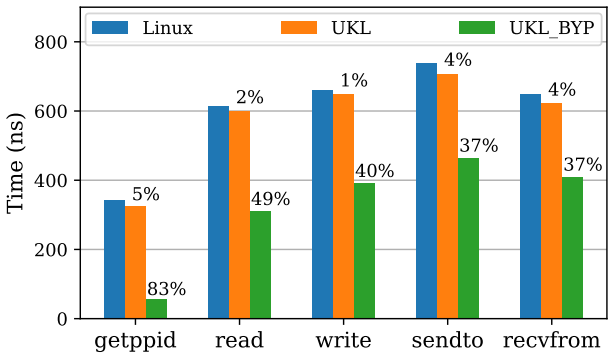

Figure 1. Comparison of Linux, UKL base model, and UKL with bypass configuration for simple system calls. With modern hardware, the UKL advantage of avoiding the system call overhead is modest (<5%). However, there appears to be significant advantage for simple calls with BYP to avoid transition checks between application and kernel code.

Weekly News (LWN) [23] articles in 2020. For comparison, the KML[25] patch, used in the recent Lupine work, that runs applications in kernel mode is 3177 LOC, a complexity that has resulted in the patch not being accepted upstream. In contrast, UKL both provides richer functionality than KML, and is much simpler. This simplicity is due to three fortuitous changes since KML was introduced. First, UKL, takes advantage of recent changes to the Linux kernel that make the changes to assembly much less intrusive. Second, UKL supports only x64-64, while KML was introduced at a time when it was necessary to support i386 to be relevant. Third, UKL does not deal with older hardware, like the i8259 PIC, that had to be supported by KML.

5.3 Microbenchmarks

Unikernels offer the opportunity to dramatically reduce the overhead of interactions between application and kernel code. We evaluate how UKL optimizations impact the overhead of simple system calls (§5.3.1), more expensive system calls (§5.3.2), and page faults (§5.3.3). Our results contradict recent work that suggests that the advantages are modest; we see that the reduction in overhead is larger (e.g., 90%) than previously reported and has a significant impact even for requests with large payloads (e.g., 24% with 8KByte `recvfrom()`).

5.3.1 System call base performance. Figure 1, compares the overhead of simple system calls between Linux, UKL's base model, and UKL_BYP. Results were gathered using the (slightly modified[4] LEBench [37]) microbenchmark to measure the base latency of getppid read, write, sendto, and recvfrom (all with 1 byte payloads).

[4]We modified LEBench to use `mmap` instead of `malloc` to give repeatable results, and to output raw numbers instead of just averages

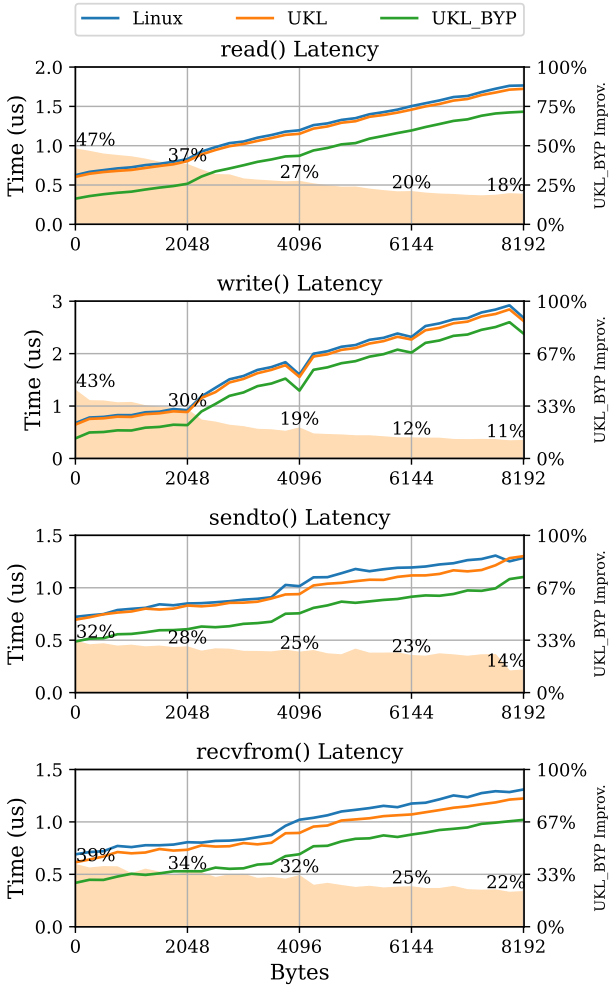

Figure 2. Comparison of Linux, UKL base model, and UKL with bypass configuration for four simple system calls. With increasing payload for each system call, UKL shows modest improvement over Linux. But there is a significant advantage for UKL, which bypasses the entry and exit code (UKL_BYP). Orange area shows percentage improvement of UKL_BYP over Linux, which decreases as payload increases but is still significant for 8KB payload.

We find that the advantage of the base model of UKL that essentially replaces `syscall`/`sysret` instructions with `call`/`ret` is modest, i.e., less than 5%. However, the UKL BYP configuration that avoids expensive checks on transitions between application and kernel code can be up to 83% for a `getppid`; suggesting that optimizing the transition between application code may have a significant performance impact.

5.3.2 Large requests. Figure 2 contrasts the performance of Linux to UKL and UKL_BYP for read, write, sendto and recvfrom as we use LEBench [37] microbenchmark to vary the payload up to 8KB of data. Again, baseline UKL shows

very little improvement over Linux, but UKL_BYP shows a significant constant improvement. The right vertical axis also shows the downward trend of percentage improvement of UKL_BYP compared to Linux. As the time spent in the kernel increases, the percentage gain decreases. But even for payloads of up to 8KB, the percentage improvement is still significant, i.e., between 11% and 22%.

It is interesting to contrast our results with those from the recent Lupine [20] work.[5] Surprisingly they observed that just eliminating the system call overhead is significant (40%) for a null system call, but since they found that (like us) the improvement dropped to below 5% in most cases, they concluded that the benefit of co-locating the application and kernel is minimal. Our results suggest that the major performance gain comes not from eliminating the hardware cost but from eliminating all the checks on the transition between the application and kernel code and that reducing this overhead has a significant impact on even expensive system calls.

5.3.3 Page Fault handling. Figure 3 compares three different schemes we have for handing page faults, i.e., UKL_PF_DF, UKL_PF_SS and (UKL_RET_PF_DF). For UKL_PF_DF, we see close to 5% improvement in page fault latency compared to Linux. UKL_PF_SS is also comparable to the previous case, which means that stack switch on every page fault is not too costly, and most of the benefit over Linux in both these cases is due to handling page faults in kernel mode and avoiding ring transition. (UKL_RET_PF_DF) gives us more than 12.5% improvement over normal Linux. In all these cases, since the time taken to service more page faults increases, the improvement over normal Linux also increases, which is why we see a constant percentage improvement. Unmodified applications can choose anyone of these options through build time Linux config options.

We repeated this experiment for non-stack page faults, i.e., on mapped memory and got the same results.

5.4 Application performance

We want to see how real world applications perform on UKL. We chose three different types of applications: a simple application (Redis [36]) used by previous works as well [18, 20], a more complex application (Memcached [29]) that many unikernels don't support unmodified, and a latency-sensitive application (Secrecy [22]). Our results show significant advantages in Redis (26%), Memcached (8%), and Secrecy (100x).

5.4.1 Simple Application: Redis. We use Redis, a widely used in-memory database, to measure the performance of UKL and its different configurations in real world applications. For this experiment, we ran Redis server on UKL on bare metal and ran the client on another physical node in the network. We use the Memtier benchmark [30] to test Redis. Through

[5]We have not been able to reproduce their result, but are still working to generate their kernel on our system.

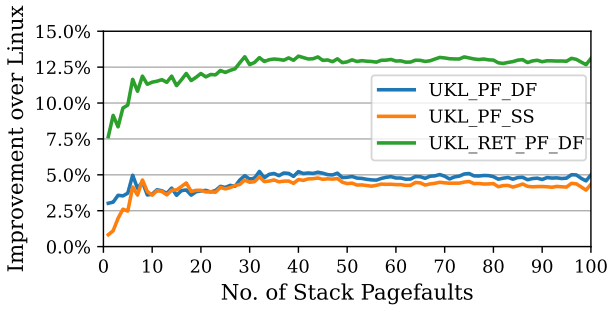

Figure 3. Percentage improvement in stack page faults over Linux. UKL which handles problematic page faults on double fault stack (UKL_PF_DF) and UKL which handles all page faults on a dedicated stack (UKL_PF_SS) show around 5% improvement over Linux. UKL_RET_PF_DF is further configured to use `ret` instead of `iret` when returning from page faults, and shows moe than 12.5% improvement over Linux.

Memtier benchmark, we create 300 clients, each sending 100 thousand requests to the server. The ratio of `get` to `set` operations in 1 to 10. We ran Redis on Linux, UKL_RET_BYP and UKL_RET_BYP with deeper shortcuts. Figure 4 helps us visualize the latency distribution for these requests.

To better understand where the time was being spent, we profiled Redis UKL with `perf`. Figure 5, which is part of the flame graph [12] we generated, shows two clear opportunities for performance improvement. Blue arrows show how we could shorten the execution path by bypassing the entry and exit code for `read` and `write` system calls and invoke the underlying functionality directly. Figure 4 shows how Redis on UKL_RET shows improvement in average and 99th percentile tail latency when it bypasses the entry and exit code (UKL_RET_BYP). Table 3 shows that UKL_RET_BYP has 11% better tail latency and 12% better throughput.

Looking at Figure 5 again, the green arrows show that `read` and `write` calls, after all the polymorphism, eventually translate into `tcp_recvmsg` and `tcp_sendmsg` respectively. To investigate any potential benefit of shortcutting deep into the kernel, we wrote some code in the kernel to interface `read` and `write` with `tcp_recvmsg` and `tcp_sendmsg` respectively. We then modified Redis (10 lines modified) to call our interface functions instead of `read` and `write`. Our results show (Figure 4) further improvement in average and 99th percentile tail latency i.e., UKL_RET_BYP (shortcut). Table 3 shows that UKL_RET_BYP (shortcut) has 22% better tail latency and 26% better throughput.

Figure 4 provides us some nice insights into future possibilities. There is almost a 0.5ms difference in the shortest latencies for Linux versus UKL_RET_BYP (shortcut) case. This means that there is an opportunity to further reduce the average and tail latencies to sit closer to the smallest latency case.

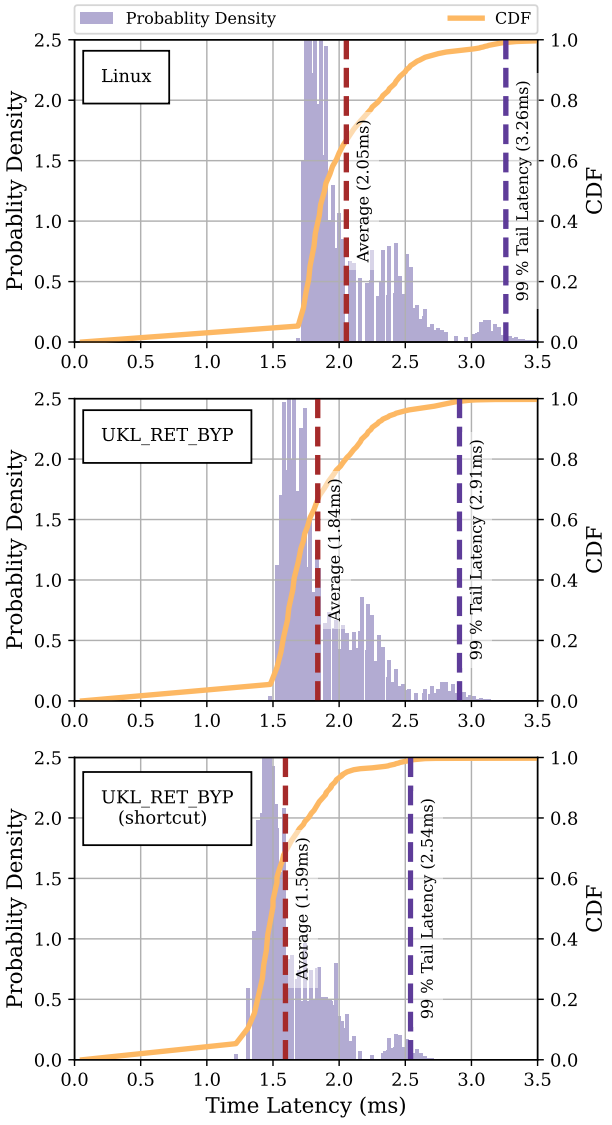

Figure 4. Latency plots for Redis running on Linux (top), UKL_RET_BYP (middle) and UKL_RET_BYP with deeper shortcuts (bottom). The horizontal axis shows latency in ms. The purple bars show the probability density of all the requests made to Redis, and the average and 99th percentile tail latencies are also marked. Right hand vertical axis is for CDF, and the orange line shows the CDF of the latencies of all the requests.

Lupine shows slightly better results than baseline Linux for Redis, but it does so in virtualization on a lightweight hypervisor. It would be interesting to see how UKL performs in that setting, even though there is a huge difference in kernel versions used by Lupine and UKL.

5.4.2 Complex Application: Memcached. Memcached is a multithreaded workload that relies heavily on pthreads

System	99% tail(ms)	tail % improv.	t-put (Kb/s)	t-put % improv.
Linux	3.26	-	6375.20	-
UKL_RET_BYP	2.91	11%	7154.68	12%
UKL_RET_BYP (shortcut)	2.54	22%	8022.54	26%

Table 3. Redis Throughput and Latency:

library and `glibc` 's internal synchronization mechanisms. It is an interesting application because unikernels generally don't support complex applications, and systems like EbbRT [39] first have to port Memcached. To evaluate Memcached, we use the Mutilate benchmark [5]. This benchmark uses multiple clients to generate a fixed queries-per-second load on the server and then measures the latency. We ran the clients in userspace on the same node as Memcached UKL to remove any network delays, and we pinned the Memcached server and clients to separate cores. We used Mutilate to generate queries based on Facebook's workloads [5]. For different configurations of UKL, we measured how many queries per second Memecached can serve while keeping the 99% tail latency under the 500 us service level agreement. Figure 6 shows Memcached with UKL_RET performs similar to Memcached on Linux, i.e., both serve around 73 thousand queries before exceeding the 500 us threshold. Memcached on UKL_RET_BYP can serve around 77 thousand queries (around 5% improvement), and Memcached on UKL_RET_BYP (shortcut) can serve up to 79 thousand queries (around 8% improvement) before going over the 500 us threshold.

This experiment also serves as a functionality and compatibility result; a comparatively large application with multiple threads etc. can run on UKL.

5.4.3 Latency Sensitive Application: Secrecy. Secrecy [?] is a multi-party computation framework for secure analytics on private data. While Redis and Memcached are throughput sensitive, Secrecy is latency-sensitive. This represents an important class of applications, e.g., highspeed financial trading, etc. Secrecy is a three node protocol with each node sending data to its successor and receiving from its predecessor with the third node sending to the first. Computation is done row by row with a round of messages that act as a barrier between each row.

We used a test in the Secrecy implementation for a GROUP-BY operator which groups rows in a table by key attributes and counts the number of rows per group. Messages used in a round of communication are each very small, between 8 and 24 bytes each, so we configured each TCP socket to use TCP_NODELAY to avoid stalls caused by congestion control. Using this test executable, we ran experiments with 100, 1000, and 10,000 input rows and measured the time required to

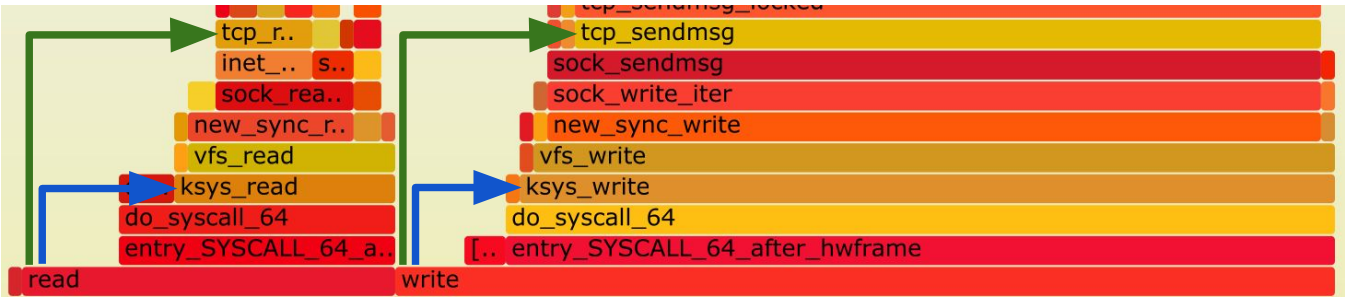

Figure 5. Part of a flame graph generated after profiling Redis-UKL base model with `perf`. The `read` and `write` functions at the bottom reside in Redis code. Blue arrows show the code bypassed in UKL_BYP, and green arrows show deeper shortcuts i.e., calling deep inside the kernel directly from Redis code.

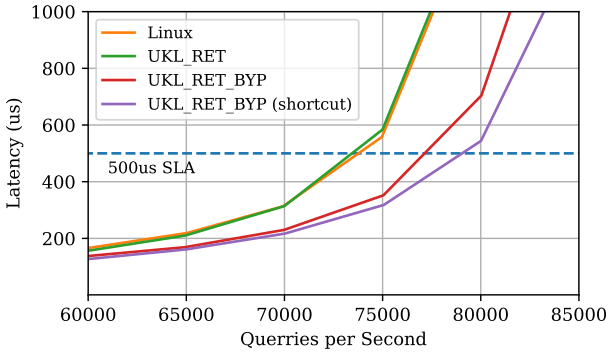

Figure 6. 99% tail latency for Memcached against increasing queries per second. A 500 us SLA is also shown to marked for reference. Memcached built with Linux and UKL_RET can handle similar rate of queries while staying under the 500 us tail latency. UKL_RET_BYP and UKL_RET_BYP (shortcut) can handle higher rate of queries at the same tail latency.

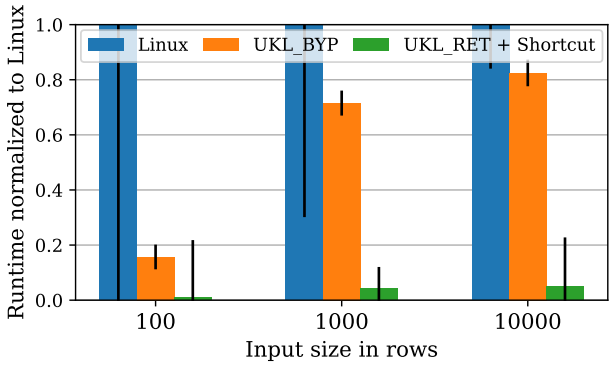

Figure 7. Normalized means of the run time of a secrecy group-by operator with increasing row counts. The run time improvements of the UKL_BYP system were larger than in other systems, however, the shortcut system shows a dramatic reduction in run time. Also of note is the reduction of the variation between runs, both UKL_BYP and UKL_RET + shortcut showed significantly smaller standard deviations with respect to their means.

complete the GROUP-BY. Each system and row size combination was run 20 times, and the worst two runs for each combination were discarded.

Figure 7 shows the run times of the three systems normalized to the run time of Linux and the error bars show the coefficient of variation for each configuration. As with other experiments, the UKL_BYP configuration shows a modest improvement in run time. However, when we use the deeper shortcut to the TCP send and receive functions, we see significant (100x) runtime improvements.

The improvement of the shortcut system over the others was larger than anticipated, so we reran the experiments and achieved the same level of performance. To verify that the work was still happening, we collected a capture of all the inter-node traffic using Wireshark and verified that the same number of TCP packets traveled between nodes in all three system setups for a 100 row experiment. We also instrumented the send and receive paths in Secrecy to collect individual times for send and receive calls in each system for a 100 row run. The mean and standard deviation of send times for Linux were 2.23us and 1.14us, respectively, and the values for receive times on Linux were 1,100us and 3,300us, respectively. The shortcut showed send mean and standard deviation of 896ns and 1,755ns, which is a significant speed up, but the receive numbers were 638ns and 3,888ns.

It appears that, with the shortcut, the system is never having to wait on packet delivery on top of bypassing system call entry and exit paths, so the shortcut system is never put to sleep waiting on incoming messages. We believe that because Secrecy is latency-sensitive and because we accelerate the send path, we ensure that no node ever has to wait for data and can move to the next round of processing immediately. Moreover, the shortcut implicitly disables scheduling on transitions, ensuring that the application is always run to completion. This is critical for an application with frequent barriers.

6 Related Work

There has been a huge body of research on unikernels that we categorize as clean slate designs, forks of existing operating systems, and incremental systems.

Clean Slate Unikernels: Many unikernel projects are written from scratch or use a minimal kernel like MiniOS [6] for bootstrapping. These projects have complete control over the language and methodology used to construct the kernel. MirageOS [24] uses OCaml to implement the unikernel and uses the language and compiler level features to ensure robustness against vulnerabilities and small attack surface. Similarly, OSv [16] uses lock-free scheduling algorithms to gain performance benefits for unmodified applications. Implementations in clean-slate unikernels can also be fine-tuned for performance of specific applications, e.g., Minicache optimizes Xen and MiniOS for CDN based use case [19]. Further, from scratch implementations can easily expose efficient, low-level interfaces to applications e.g., EbbRT [39]. Different clean slate unikernels can often be polar opposites in some regards, exposing the wide range of choices available to them. For instance, some might target custom APIs for performance [24, 39] while like HermiTux [33] target full Linux ABI compatibility. Recently, efforts like Unikraft [18] provide strong POSIX support while also allowing custom APIs for further performance gains.

These unikernel offer compelling trade-offs to general-purpose operating systems. These include improved security and smaller attack surfaces e.g., Xax [10] and MirageOS [24], shorter boot times e.g., ClickOS [27] and LightVM [26], efficient memory use through single address space e.g., OSv [16] and many others, and better run-time performance e.g., EbbRT [39], Unikraft [18] and SUESS [8]. Some approaches target direct access to virtual or physical hardware [39]. A number of researchers have directly confronted the problem of compatibility, e.g., OSv [16] is almost Linux ABI compatible and HermiTux is fully ABI compatible with Linux binaries [33]. Other projects aim to make building unikernels easier e.g., EbbRT [39], Libra [3] and Unikraft [18].

The UKL effort was inspired by the tremendous results demonstrated by clean slate unikernels. Our research targets trying to find ways to integrate some of the advantages these systems have shown into a general-purpose OS.

Forks of General Purpose OS. A number of projects either fork an existing general-purpose OS code base or reuse a significant portion of one. Examples include Drawbridge [34] which harvests code from Windows, Rump kernel [15] which uses NetBSD drivers and Linux Kernel Library (LKL) [35] which borrows code from Linux. These systems, although constrained by the design and structure of the original OS, generally have better compatibility with existing applications [34]. The codebase these systems fork are well tested [35] and can serve as building blocks for other research projects, e.g., Rump [15] has been used in other projects [32].

Our goal in UKL is to try to find a way to integrate unikernel optimizations without having the fork the original OS.

Incremental Systems. There are systems, e.g., Kernel Mode Linux (KML) [25], Lupine [20] and X-Containers [40] which use an existing general-purpose operating system (Linux) but make comparatively fewer changes. This way, a lot of working knowledge of users of Linux can easily transfer over to these systems, but in doing so, these systems only expose the system call entry points to applications and don't make any further specializations. Unlike UKL, they don't co-optimize the application and kernel together. Lupine [20] and X-Containers [40] demonstrate opportunities in customizing Linux through build time configurations, and that is orthogonal and complementary to UKL. UKL can also benefit from a customized Linux and then add unikernel optimizations on top of that.

7 Concluding remarks

UKL creates a unikernel target of `glibc` and the Linux kernel. The changes are modest, and we have shown even with these, it is possible to achieve substantial performance advantages for real workloads, e.g., 26% improvement in Redis throughput while improving tail latency by 22%. UKL supports both virtualized platforms and bare-metal platforms. While we have not tested a wide range of devices, we have so far experienced no issues using any device that Linux supports. Operators can configure and control UKL using the same tools they are familiar with, and developers have the ability to use standard Linux kernel tools like `BPF` and `perf` to analyze their programs.

UKL differs in a number of interesting ways from unikernels. First, while application and kernel code are statically linked together, UKL provides very different execution environments for each; enabling applications to run in UKL with no modifications while minimizing changes to the invariants (whatever they are) that the kernel code expects. Second, UKL enables a knowledgable developer to incrementally optimize performance by modifying the application to directly take advantage of kernel capabilities, violating the normal assumptions of kernel versus application code. Third, processes can run on top of UKL, enabling the entire ecosystem of Linux tools and scripting to just work.

We have repeatedly thought that we were only a few weeks away from a stable system, and it has only been recently that we had a design and a set of changes that met our fundamental goals. While the set of changes to create UKL ended up being very small, it has taken us several years of work to get to this point. The unique design decisions are a result of multiple, typically much more pervasive, changes to Linux as we changed directions and gained experience with how the capability we wanted could be integrated into Linux. It is in some sense an interesting experience that the very modularity of Linux that enables a broad community to participate both: 1) makes it very difficult to understand how to integrate a change like

UKL and, 2) can be harnessed to enable the change in a very small number of lines of code.

The focus of our work so far has been on functionality and just a proof of concept of a performance advantage in order to justify integrating the code into Linux. Now that we have achieved that, we plan to start working on getting UKL upstreamed as a standard target of Linux so that the community will continue to enhance it.

We have only started performance optimizing UKL. As our knowledge of Linux has increased, a whole series of simple optimizations that can be readily adopted have become apparent beyond the current efforts. How hard will it be to introduce and/or exploit zero-copy interfaces to the application? How hard will it be to reduce some of the privacy assumptions implicit in the BSD socket interface when only one application consumes incoming data?

These kernel-centric optimizations are just the start. From an application perspective, we believe that UKL will provide a natural path for improving performance and reducing the complexity of complex concurrent workloads. Concurrent operations on shared resources must be regulated. Often the burden falls onto the user code. From the user-level, it is hard to determine whether synchronization is needed, and the controlling operations and controlled entities usually live in the kernel. If the user code moves into the kernel and has the same privileges, some operations might become faster or possible in the first place. For instance, in a garbage collector, it might be necessary to prevent or at least detect whether concurrent accesses happen. With easy and fast access to the memory infrastructure (e.g., page tables) and the scheduler, many situations in which explicit, slow synchronization is needed might get away with detecting and cleaning up violations of the assumptions.

If the Linux community accepts UKL, we believe it will not only impact Linux but may become a very important platform for future research. While the benefits to researchers of broad applications on HCL support are obvious. Perhaps less obvious, as unikernel researchers, is the ability to use tools like `ktest` to deploy and manage experiments, `BPF` and `perf` to be able to understand performance, have been incredibly valuable.

References

[1] Dpdk - data plane development kit. https://www.dpdk.org/. Accessed on 2021-10-7.

[2] Storage Performance Development Kit. https://spdk.io/, 2018. (Accessed on 01/16/2019).

[3] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert W Wisniewski. Libra: a library operating system for a jvm in a virtualized execution environment. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 44–54, 2007.

[4] Thomas E Anderson. *The case for application-specific operating systems*. University of California, Berkeley, Computer Science Division, 1993.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[7] Scott Beamer, Krste Asanovic, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[8] James Cadden, Thomas Unger, Yara Awad, Han Dong, Orran Krieger, and Jonathan Appavoo. Seuss: skip redundant paths to make serverless fast. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[9] David R Cheriton and Kenneth J Duda. A caching model of operating system kernel functionality. *ACM SIGOPS Operating Systems Review*, 29(1):83–86, 1995.

[10] John R Douceur, Jeremy Elson, Jon Howell, and Jacob R Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, volume 8, pages 339–354, 2008.

[11] Dawson R Engler, M Frans Kaashoek, and James O'Toole Jr. Exokernel: An operating system architecture for application-level resource management. *ACM SIGOPS Operating Systems Review*, 29(5):251–266, 1995.

[12] Brendan Gregg. The flame graph. *Commun. ACM*, 59(6):48–57, may 2016.

[13] Intel. https://www.dpdk.org/, 2010. [Online; accessed 17-January-2019].

[14] Antti Kantee. *The design and implementation of the anykernel and rump kernels*. 2nd edition, 2016.

[15] Antti Kantee et al. Flexible operating system internals: the design and implementation of the anykernel and rump kernels. 2012.

[16] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. Osv—optimizing the operating system for virtual machines. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 61–72, 2014.

[17] Ricardo Koller and Dan Williams. Will Serverless End the Dominance of Linux in the Cloud? In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, pages 169–173. ACM, 2017.

[18] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 376–394, 2021.

[19] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. Unikernels everywhere: The case for elastic cdns. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 15–29, 2017.

[20] Hsuan-Chi Kuo, Dan Williams, Ricardo Koller, and Sibin Mohan. A linux in unikernel clothing. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–15, 2020.

[21] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE journal on selected areas in communications*, 14(7):1280–1297, 1996.

[22] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. Secrecy: Secure collaborative analytics on secret-shared data. *CoRR*, abs/2102.01048, 2021.

[23] Linux Weekly News. https://lwn.net/, note = "(Accessed on 05/30/2022)".

[24] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1):461–472, 2013.

[25] Toshiyuki Maeda and Akinori Yonezawa. Kernel mode linux: Toward an operating system protected by a type theory. In *Annual Asian Computing Science Conference*, pages 3–17. Springer, 2003.

[26] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233, 2017.

[27] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *11th {USENIX} symposium on networked systems design and implementation ({NSDI} 14)*, pages 459–473, 2014.

[28] Paul E McKenney and John D Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, volume 509518, 1998.

[29] Memcached. https://memcached.org/. (Accessed on 05/30/2022).

[30] Memtier Benchmark.

[31] José Moreira, Michael Brutman, José Castanos, Thomas Engelsiepen, Mark Giampapa, Tom Gooding, Roger Haskin, Todd Inglett, Derek Lieber, Pat McCarthy, et al. Designing a highly-scalable operating system: The blue gene/l story. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, pages 118–es, 2006.

[32] Ruslan Nikolaev, Mincheol Sung, and Binoy Ravindran. Librettos: a dynamically adaptable multiserver-library os. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 114–128, 2020.

[33] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 59–73, 2019.

[34] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the library os from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, pages 291–304, 2011.

[35] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. Lkl: The linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333. IEEE, 2010.

[36] Redis. https://redis.io/, note = "(Accessed on 05/30/2022)".

[37] Xiang Ren, Kirk Rodrigues, Luyuan Chen, Camilo Vega, Michael Stumm, and Ding Yuan. An analysis of performance evolution of linux's core operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 554–569, 2019.

[38] Luigi Rizzo. netmap: A novel framework for fast packet i/o. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112, Boston, MA, June 2012. USENIX Association.

[39] Dan Schatzberg, James Cadden, Han Dong, Orran Krieger, and Jonathan Appavoo. Ebbrt: A framework for building per-application library operating systems. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 671–688, 2016.

[40] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. X-containers: Breaking down barriers to improve performance and isolation of cloud-native containers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 121–135, 2019.

[41] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking Behind the Curtains of Serverless Platforms. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 133–145, Berkeley, CA, USA, 2018. USENIX Association.

[42] Dan Williams, Ricardo Koller, Martin Lucina, and Nikhil Prakash. Unikernels as processes. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 199–211, 2018.