# A Linux in Unikernel Clothing

Hsuan-Chi Kuo
University of Illinois at Urbana-Champaign
hckuo2@illinois.edu

Dan Williams
IBM T.J. Watson Research Center
djwillia@us.ibm.com

Ricardo Koller
IBM T.J. Watson Research Center
kollerr@us.ibm.com

Sibin Mohan
University of Illinois at Urbana-Champaign
sibin@illinois.edu

## Abstract

Unikernels leverage library OS architectures to run isolated workloads on the cloud. They have garnered attention in part due to their promised performance characteristics such as small image size, fast boot time, low memory footprint and application performance. However, those that aimed at generality fall short of the application compatibility, robustness and, more importantly, community that is available for Linux. In this paper, we describe and evaluate Lupine Linux, a standard Linux system that—through *kernel configuration specialization* and *system call overhead elimination*—achieves unikernel-like performance, in fact outperforming at least one reference unikernel in all of the above dimensions. At the same time, Lupine can run any application (since it is Linux) when faced with more general workloads, whereas many unikernels simply crash. We demonstrate a *graceful degradation* of unikernel-like performance properties.

## 1 Introduction

Since the inception of cloud computing, the virtual machine (VM) abstraction has dominated infrastructure-as-a-service systems. However, it has recently been challenged, as both users and cloud providers seek more lightweight offerings. For example, alternatives such as OS-level containers have begun to attract attention due (in part) to their relatively lightweight characteristics in dimensions such as image size, boot time, memory footprint and overall performance.

In response, the virtualization stack has been evolving to become more lightweight in two main ways. First, modern virtual machine monitor designs, like lightVM [43] and AWS Firecracker [4] (or in a more extreme case unikernel monitors [63]), have reduced the complexity and improved the performance of the monitor. Second, alternatives to large, general-purpose guest operating systems have begun to emerge—whether it is a change in the userspace (e.g., from Ubuntu-based to Alpine-based), a change in the configuration of the guest kernel (e.g., TinyX [43]) or in the case of *unikernels* [41], a specialized library OS.

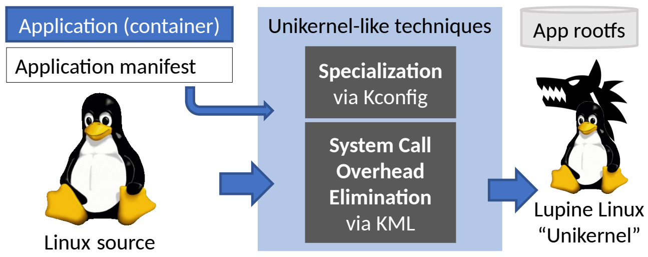
Application (container)
Application manifest
Linux source
Unikernel-like techniques
Specialization via Kconfig
System Call Overhead Elimination via KML
App rootfs
Lupine Linux "Unikernel"

**Figure 1.** Overview

Specializing the guest VM to the extent of running a library OS tailored to an application is a compelling prospect, especially when the application domain is limited. *Language-based* unikernels, such as MirageOS [41] (in which the library OS and application are entirely written in OCaml), have demonstrated a combination of security and lightweight properties. In an effort to expand the applicability of unikernel ideas, however, several unikernel/library OS projects, including HermiTux [45], Rumprun [15], Graphene [62] and OSv [32], have attempted to become more general and *POSIX-like*; some even going so far as claiming Linux binary compatibility [45]. Approaching some level of POSIX-like generality typically requires either a great implementation effort or a clever way to reuse existing POSIX-compatible kernel implementations (usually NetBSD [15, 32]). However, these approaches still fall short of true compatibility because of arbitrary restrictions (e.g., not supporting `fork`) and suffer by being unable to leverage the robustness, performance or, most importantly, the community of Linux.

In this paper, we make the observation that Linux is already highly configurable and seek to determine exactly how close it is to achieving the sought-after properties of unikernels. We describe *Lupine Linux* (Figure 1), in which we apply

*EuroSys '20, April 27–30, 2020, Heraklion, Greece*
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387526

two well-known unikernel-like techniques to Linux: *specialization* and the *elimination of system call overhead*. Though we do not propose a general solution for specialization, we specialize Lupine through the kernel's Kconfig mechanisms by (1) eliminating functionality from the kernel that is not necessary for the unikernel domain (e.g., support for hardware devices or multiprocessing) and (2) tailoring the kernel as much as possible to the particular application. Lupine eliminates system call overhead by running the application in the same privilege domain as the kernel via the existing (but not upstream) Kernel Mode Linux (KML) [42] patch.

When evaluating Lupine against a state-of-the-art lightweight VM (AWS Firecracker's *microVM*) and three POSIX-like unikernels, we find that Lupine outperforms microVM and at least one of the reference unikernels in all of the following dimensions: image size (4 MB), boot time (23 ms), memory footprint (21 MB), system call latency (20 $\mu$s), and application performance (up to 33% higher throughput than microVM). While both unikernel techniques played a role in improving performance, specialization had the largest effect: despite up to 40% reduction in system call latency due to KML on microbenchmarks, we found it improved application performance on macrobenchmarks by only 4%.

Regarding specialization via configuration, we attempted to determine the most practical degree of specialization for Lupine. To this end, we examined the effects of specialization in Lupine by *heuristically* creating specialized configurations for the top 20 cloud applications—that account for 83% of all downloads—as determined by popularity on Docker Hub. We categorize 550 configuration options from the microVM kernel and find only 19 of them are required to run all 20 applications, suggesting a tiny, application-agnostic kernel configuration that achieves unikernel-like performance without the complications of per-application specialization.

Lupine exploits Linux to eliminate the generality issues of other POSIX-like unikernels; it can run any application using Linux's highly-optimized implementation, including those that do not fit in the unikernel domain. In this context, we examine how unikernel properties degrade in the face of generality and find a *graceful degradation* property. Where other unikernels may crash on `fork`, Lupine continues to run. Moreover, we find virtually no overhead for supporting multiple address spaces and at worst an 8% overhead to support multiple processors, concluding that many unikernel restrictions are avoided unnecessarily for POSIX-like unikernels.

This paper makes the following contributions:

- Lupine Linux, an example of configuring and building a unikernel-like Linux through kernel configuration specialization and system call overhead elimination,
- an evaluation and comparison of the unikernel properties that can be achieved with Lupine Linux and
- an investigation into how unikernel performance characteristics degrade (or not) in the face of more general workloads and hardware in Lupine.

## 2 Unikernels

Unikernels [6, 13–15, 19, 32, 41, 44, 45, 57] have garnered widespread interest by providing a lightweight, simple, secure and high-performance alternative to the complex, conventional, general-purpose compute stacks that have evolved over many years. In this section, we give a brief background and classification of unikernel projects and their benefits, describing some of the techniques that they have used to achieve these benefits and identify some common limitations.

### 2.1 Background

Unikernels are the most recent incarnation of library OS [26, 46, 53, 62] designs. They are typically associated with cloud environments and consist of a single application linked with a library that provides enough functionality to run directly on a virtual-hardware-like interface. We organize unikernels into two categories: *language-based* and *POSIX-like*.

***Language-based.*** Language-based unikernels are library OS environments that are tied to a specific programming language runtime and libraries, for example, MirageOS [41] for OCaml, IncludeOS [19] for C++, Clive [13] for Go, HalVM [57] for Haskell, runtime.js [14] for JavaScript, Ling [6] for Erlang, and ClickOS [44] for Click router rules [33]. Language-based unikernels typically do not need to implement or adhere to POSIX functionality, which can lead to small images, suitability for compiler-based analyses and optimizations, and reliability or security from the use of language features. For example, 39 out of 40 of all bugs found in Linux drivers in 2017 were due to memory safety issues [25] that could have been avoided by a high-level language. However, the requirement for applications to adhere to a particular language and interface limits adoption.

***POSIX-like.*** POSIX-like unikernels are library OS environments that use a single address space and a single privilege level but attempt to provide some amount of compatibility with existing applications. OSv [32] and HermiTux [45] are two unikernels that boast binary compatibility with Linux applications but reimplement kernel functionality from scratch, losing the opportunity to benefit from the maturity, stability, performance and community of Linux. Unlike language-based unikernels and Rumprun [15], a POSIX-like unikernel that leverages NetBSD to avoid reimplementation, OSv and HermiTux do not require the application to be linked with the library OS which (mostly) eliminates the need to modify application builds[1] and eases deployment at the cost of losing specialization opportunities.

### 2.2 Benefits and Techniques

Unikernels achieve benefits like low boot times, security, isolation, small image sizes, low memory footprint and performance through a combination of optimizing the monitor [43, 63] and construction of the unikernel itself.[2]

***Lightweight monitors.*** Traditional virtual machine monitors like QEMU are general and complex, with 1.8 million lines of C code and the ability to emulate devices and even different CPU architectures. Recently, unikernel monitors [63] have shown that a unikernel's reduced requirement for faithful hardware emulation can result in a dramatically simpler, more isolated and higher performing monitor (which may not even require virtualization hardware [64]). As a result, unikernels have been shown to boot in as little as 5-10 ms, as opposed to hundreds of milliseconds for containers or minutes for VMs [34, 63], which is important for new compute models like serverless computing [1, 5]. At the same time, general-purpose monitors have also been reducing generality (such as forgoing some device emulation) for performance: for example, AWS Firecracker [4] and LightVM [43] optimize for boot time by eliminating PCI enumeration. Firecracker also improves the security posture of monitors by using a memory-safe language (Rust).

***Specialization.*** Unikernels embody a minimalist philosophy, where they only require those code modules that are needed for a particular application to run on virtual hardware. This leads to smaller image sizes, lower memory footprint, and smaller attack surfaces. In language-based unikernels, like MirageOS, the relatively tight integration between the language, package manager and build process implements this philosophy well. POSIX-like unikernels, like Rumprun, OSv or HermiTux, balance how much they specialize with compatibility and reuse of legacy code, but typically attempt to provide at least coarse-grained specialization.

***System Call Overhead Elimination.*** Unikernels contain, by definition, a single application. Therefore, logically, the library OS and the application exist in the same security and availability domain. As a result, unikernels typically run all code in the same CPU privilege domain and in the same address space in order to improve performance over traditional systems. There is no need to switch context between application and kernel functions.

[1]These systems typically maintain a curated application list. While modifications to the applications on the list are relatively minor, this approach severely limits what can run in practice, as we will see in Section 4.
[2]Some unikernels, especially language-based unikernels, use other techniques, discussed further in Section 6.

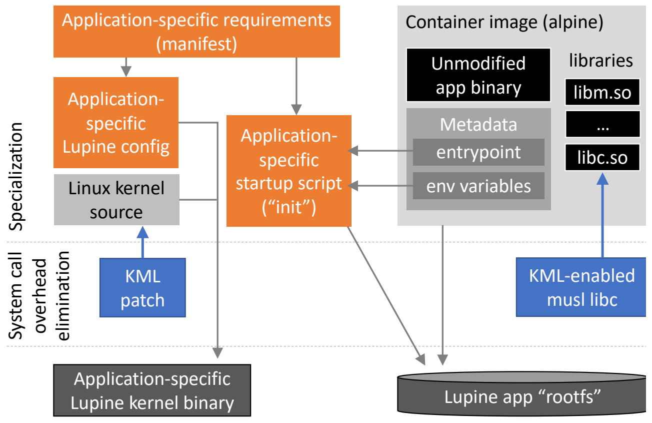
Application-specific requirements (manifest)
Container image (alpine)
Unmodified app binary
libraries
libm.so
...
libc.so
Application-specific Lupine config
Application-specific startup script ("init")
Metadata
entrypoint
env variables
Linux kernel source
Specialization
System call overhead elimination
KML patch
KML-enabled musl libc
Application-specific Lupine kernel binary
Lupine app "rootfs"

**Figure 2.** Specialization and system call overhead elimination in Lupine.

### 2.3 The Ideal Unikernel

Unfortunately, existing unikernels face challenges in terms of generality; applications may need to be written from scratch, potentially in an unfamiliar language (e.g., language-based unikernels). Those that do try to address generality (e.g., POSIX-like unikernels) find themselves in the unenviable position of trying to reimplement Linux, at least in part.

The ideal unikernel would enjoy all of the benefits that they are known for while also being able to support Linux applications and share its community.

## 3 Lupine Linux

We make Linux behave like a unikernel by *specialization* and *system call overhead elimination*. We specialize Lupine according to both applications and the unikernel constraints such as the single-process nature or expected deployment environments (Section 3.1). We eliminate the system call overheads by applying KML patches to Linux that allow applications to run in kernel mode (Section 3.2).

Like some POSIX-like unikernels (e.g., HermiTux [45] and OSv [32]), but unlike others (e.g., Rumprun [15]), Lupine is not a single, statically-linked binary. Instead, a Lupine unikernel consists of a kernel binary that dynamically loads the application code from a root filesystem (rootfs) as in other Linux-based systems. Figure 2 shows the specifics of the generation of Lupine kernel binary and root filesystem. Lupine kernel binary is configured to be a small, special-purpose Linux kernel image, obtained via the specialization highlighted in orange in Figure 2. Also, the kernel is enhanced with Kernel Mode Linux (KML) so that Lupine runs the target application as a *kernel-mode process*, thereby avoiding context switches. An *application manifest* informs the application-specific kernel configuration.

We leverage Docker container images to obtain minimal root filesystems with applications and all their dependencies

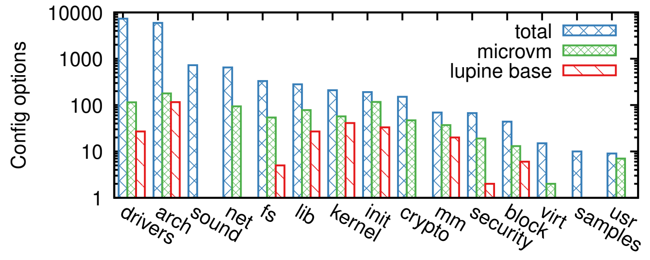

**Figure 3.** Linux kernel configuration options (log scale)

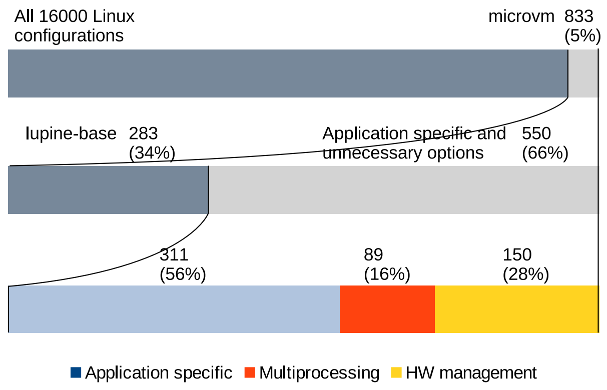

**Figure 4.** Breakdown of kernel configuration options down to unnecessary ones by unikernel property.

such as dynamically-linked libraries (e.g., `libc`, `libm`, etc.)[3]. As the root filesystem is specialized, Lupine does not employ a general-purpose `init` system. Instead, Lupine creates an application-specific startup script based on container metadata. For example, the *entrypoint* describes the parameters with which to invoke the application, and the *env variables* describe how to set up the environment. Like the kernel image, the script is informed by the application manifest; for example, it may initialize the network device, mount loopback devices or the proc filesystem, generate entropy, set ulimits, set environment variables, or create directories before executing the application. Finally, we convert the container images that include the application binary, a *KML-enabled* `libc` (described in Section 3.2) and the application-specific startup script into an `ext2` image that the specialized Lupine kernel will use as its root filesystem. At runtime, a standard virtual machine monitor (e.g., Firecracker) launches the Lupine kernel and rootfs.

The concrete details of the application manifest are out of scope for this paper. At its simplest, an application manifest could be a developer-supplied kernel configuration and startup script.

## 3.1 Specialization

The Linux kernel contains considerable facilities for specialization through its `Kconfig` configuration mechanism. In total, there are 15,953 configuration options for Linux 4.0. Configuration options determine whether features should be included in the kernel by compiling the source code for the feature and either linking it into the kernel image or into a module that can be dynamically loaded into a running kernel. Kernel configuration options empower users to select or enable (for example) support for a variety of hardware (e.g., device drivers), a variety of services to applications (e.g., filesystems, network protocols) and algorithms governing management of the most basic compute resources (e.g., memory management, scheduling).

Figure 3 shows the total number of available configuration options (by directory) in the Linux source tree. Unsurprisingly, almost half of the configuration options are found in drivers to support the wide range of devices that Linux runs on. Figure 3 also shows the breakdown of configuration options selected by AWS Firecracker's microVM configuration. This is a Linux configuration that allows a general-purpose workload to specifically run on the Firecracker monitor on the `x86_64` architecture. This configuration can safely omit a vast majority of configurable functionality because of the known constraints of Firecracker, as shown in Figure 4. For example, the vast majority of the driver and architecture-specific options are not necessary since the virtual I/O devices and architecture are pre-determined.

Even more configurable functionality can be safely omitted for Lupine because of the known constraints of the unikernel domain. As depicted in Figure 4, starting from Firecracker's microVM configuration, we manually removed approximately 550 (66%) of the selected options that we deemed potentially unnecessary for the unikernel domain as further described below. We refer to the remaining 283 (34%) configuration options as *lupine-base*.

We further manually classified the 550 removed options into categories based on features or design properties of unikernels and not based on the Linux kernel's structure as in Figure 3. *Application-specific* options are only necessary for certain applications and may be reintroduced on top of *lupine-base* to create an application-specific configuration. Others are not necessary for *any* unikernel application, either because of the single-process nature of unikernels or the predictable runtime environment of virtual machines in the cloud. We now describe these categories (Figure 4) and provide examples.

### 3.1.1 Application-specific options.

Unikernels are driven by a minimalist philosophy where they only contain functionality that the application needs. While compatibility with Linux often implies some compromises, an application-centric approach can be applied towards Linux kernel configuration. To this end, we categorize certain configuration options as application-specific,

[3]Tools such as Docker Slim [3] help ensure a minimal dependency set.

| Option | Enabled System Call(s) |
|---|---|
| ADVISE_SYSCALLS | madvise, fadvise64 |
| AIO | io_setup, io_destroy, io_submit, io_cancel, io_getevents |
| BPF_SYSCALL | bpf |
| EPOLL | epoll_ctl, epoll_create, epoll_wait, epoll_pwait |
| EVENTFD | eventfd, eventfd2 |
| FANOTIFY | fanotify_init, fanotify_mark |
| FHANDLE | open_by_handle_at, name_to_handle_at |
| FILE_LOCKING | flock |
| FUTEX | futex, set_robust_list, get_robust_list |
| INOTIFY_USER | inotify_init, inotify_add_watch, inotify_rm_watch |
| SIGNALFD | signalfd, signalfd4 |
| TIMERFD | timerfd_create, timerfd_gettime, timerfd_settime |

**Table 1.** Linux configuration options that enable/disable system calls.

which may or may not appear in any Lupine unikernel's kernel configuration. We also discuss various granularities at which an application manifest could inform kernel configuration, but leave the generation of such a manifest (which could utilize static or dynamic analysis [30, 31, 37]) to future work.

Unikernels embody *DevOps* industry trends, in which system configuration and runtime operations tasks are tightly integrated with application development. We identified approximately 100 network-related options, including a variety of less popular protocols and 35 filesystem-related configuration options that represent system configuration tradeoffs that depend on the (single) application. At a finer granularity, if we assume the application or container manifest details exactly which system calls an application will use,[4] then we can configure Linux to include some necessary system calls. For example, Table 1 lists configuration options that dictate whether one or more system calls (and their implementations) are compiled into the kernel binary. As an example of application-specific configuration, the `redis` key-value store requires EPOLL and FUTEX by default, whereas the `nginx` web server additionally requires AIO and EVENTFD. A Lupine kernel compiled for `redis` does not contain the AIO or EVENTFD-related system calls.

In addition to the above, some applications expect other services from the kernel, for instance, the `/proc` filesystem or `sysctl` functionality. Moreover, the Linux kernel maintains a substantial library that resides in the kernel because of its traditional position as a more privileged security domain. Unikernels do not maintain the traditional privilege separation but may make use of this functionality directly or indirectly by using a protocol or service that needs it (e.g., cryptographic routines for IPsec). We marked 20 compression-related and 55 crypto-related options from the microVM configuration as application-specific. Finally, Linux contains significant facilities for debugging; a Lupine unikernel can select up to 65 debugging and information-related kernel configuration options from microVM's configuration.

In total, we classified approximately 311 configuration options as application-specific as shown in Figure 4. In Section 4, we will evaluate the degree of application specialization via Linux kernel configuration (and its effects) achieved in Lupine for common cloud applications.

### 3.1.2 Unnecessary options.

Some options in microVM's configuration will, by definition, never be needed by any Lupine unikernel so they can be safely eliminated. We categorize these options into two groups: (1) those that stem from the single-process nature of unikernels and (2) those that stem from the expected virtual hardware environment in the cloud.

***Unikernels are not intended for multiple processes.*** The Linux kernel is intended to run multiple *processes*, thus requiring configurable functionality for synchronization, scheduling and resource accounting. For example, *cgroups* and *namespaces* are specific mechanisms that limit, account for and isolate resource utilization between processes or groups of processes. We classified about 20 configuration options related to cgroups and namespaces in Firecracker's microVM configuration.

Furthermore, the kernel is usually run in a separate, more privileged security domain than the application. As such, the kernel contains enhanced access control systems such as SELinux and functionality to guard the crossing from the application domain to the kernel domain, such as `seccomp` filters, all of which are all unnecessary for unikernels. More importantly, security options with a severe impact on performance are also unnecessary for this reason. For example, KPTI (kernel page table isolation [9]) forbids the mapping of kernel pages into processes' page table to mitigate the Meltdown [40] vulnerability. This dramatically affects system call performance; when testing with KPTI on Linux 5.0 we measured a 10x slowdown in system call latency. In total, we eliminated 12 configuration options due to the single security domain.

Linux is well equipped to run on multiple-*processor* systems. As a result, the kernel contains various options to include and tune SMP and NUMA functionality. On the other hand, since most unikernels do not support fork, the standard approach to take advantage of multiple processors is to run multiple unikernels.

Finally, Linux contains facilities for dynamically loading functionality through modules. A single application facilitates the creation of a kernel that contains all functionality it needs at build time.

[4]While generating the manifest is, in general, an open problem, several products and projects like DockerSlim[3] and Twistlock[11] rely on similar system-call information.

Overall, we attribute the removal of 89 configuration options to the single-process—*"uni"*—characteristics of unikernels as shown in Figure 4 (under "Multiple Processes"). In Section 5, we examine the relaxation of this property.

***Unikernels are not intended for general hardware.*** default configurations for Linux are intended to result in a general-purpose system. Such a system is intimately involved in managing hardware with configurable functionality to perform tasks, including power management, hotplug and driving and interfacing with devices. Unikernels, which are typically intended to run as virtual machines in the cloud, can leave many physical hardware management tasks to the underlying host or hypervisor. Firecracker's microVM kernel configuration demonstrates the first step by eliminating many unnecessary drivers and architecture-specific configuration options (as shown in Figure 3). Lupine's configuration goes further by classifying 150 configuration options—including 24 options for power management that can be left to the underlying host—as unnecessary for Lupine unikernels as shown in Figure 4.

### 3.2 Eliminating System Call Overhead

Kernel Mode Linux [42] is an existing patch to Linux that enables normal user processes to run in kernel mode, and call kernel routines directly without any expensive privilege transitions or context switches during system calls. Yet they are processes that, unlike kernel modules, do not require any change to the programming model and can take advantage of all system services for normal processes such as paging or scheduling.

While KML was designed for multiple applications to run, some as kernel-mode processes (identified by the path to the executable rooted at `/trusted`) and some as normal user-mode processes, the goal for Lupine is to mimic a unikernel that, by definition, only contains a single—privileged—application. As a result, we modify KML for Lupine so that all processes (of which there should be one) will execute in kernel mode. Note that, despite running the application with an elevated privilege level via KML, no kernel bypass occurs. Kernel execution paths enumerated due to system calls by an application remain identical regardless of whether KML is in use or not.

For the implementation of KML in Lupine, we applied the modified KML patch to the Linux kernel. We also patched *musl* `libc`, the `libc` implementation used by Alpine, for the distribution of Linux that we chose for the container images that form the basis of Lupine unikernel images. The patch is minimal: it replaces the `syscall` instruction used to issue a system call at each call site with a normal, same-privilege `call` instruction. The address of the called function is exported by the patched KML kernel using the `vsyscall`[5]. For

| Name | monitor | kernel ver | kernel conf | userspace |
|---|---|---|---|---|
| *MicroVM* | Firecracker | 4.0 | microVM | Alpine 3.10 |
| *Lupine* | Firecracker | 4.0 | lupine-base | Alpine 3.10 |

**Table 2.** Systems used to evaluate the Lupine unikernel.

most binaries that are dynamically linked, the patched `libc` can simply be loaded without requiring the recompilation of the binary. Statically linked binaries running on Lupine must be recompiled to link against the patched `libc`. Note that this is far less invasive than the changes required by many unikernels including not only recompilation but often a modified build.

## 4 Evaluation

The purpose of this evaluation is to show that Lupine can achieve most benefits of unikernels: small image size, fast boot time, small memory footprint, no system call overheads, and application performance. We compare several unikernel and non-unikernel systems as summarized in Table 2. *MicroVM* is a baseline, representing a state-of-the-art VM-based approach to running a Linux application on the cloud. *OSv*, *HermiTux* and *Rump* are unikernels that (partially) recreate Linux functionality inside their library OS. They provide comparison targets to define unikernel-like functionality for the purposes of the evaluation. All systems use the Firecracker monitor, except for HermiTux and Rump that use specialized unikernel monitors [63].[6]

We use the same Linux kernel version for all cases. We use Linux 4.0 with KML patches applied.[7] *MicroVM* uses AWS Firecracker's microVM configuration adapted to Linux 4.0. Lupine uses an application-specific configuration atop the microVM-derived *lupine-base* configuration, as described in Section 3.1, with 2 variants:

- *-nokml* is used to highlight the contribution of eliminating context switches versus specialization. *Lupine-nokml* differs from *lupine* in two ways: (1) it uses a kernel that does not have the KML patch applied, and (2) contains `CONFIG_PARAVIRT`, as also present in microVM, which unfortunately conflicts with KML despite being important for performance (as we will see).
- *-tiny* indicates Lupine is optimized for size over performance. *Lupine-tiny* differs from *lupine* in that it (1) is compiled to optimize for space with `-Os` rather than

[5]The original KML design took advantage of the 32-bit kernel and dynamic behavior in `glibc` to entirely avoid modifications to `glibc`. In 32-bit mode, some versions of `glibc` would dynamically select whether to do the newer, faster `sysenter` x86 instruction to enter the kernel on a system call or use the older, slower `int 0x80` mechanism. The decision was made based on information exported by the kernel via the `vsyscall` mechanism (a kernel page exported to user space). KML introduced a third option, `call`

[6]Both `uhyve` and `solo5-hvt` are descendants of `ukvm`.

[7]Linux 4.0 is the most recent available version for KML.

for performance with `-O2` and (2) has 9 modified configuration options that state clear space/performance tradeoffs in Kconfig (e.g., `CONFIG_BASE_FULL`).[8]

A third variant is not application-specific:

- *-general* is a Lupine kernel with a configuration derived from the union of all application-specific configurations from the most popular 20 applications as described in Table 3 in Section 4.1.

All experiments were run on a single server with an 8 core Intel Xeon CPU (E3-1270 v6) at 3.80GHz and 16 GB of memory. For a fair comparison, the unikernel (or guest) was limited to 1 VCPU (pinned to a physical core) as most unikernels are single-threaded and 512 MB of memory (except the experiment for memory footprint). This was done for all performance tests. The VM monitor could also make use of 3 additional CPUs and the benchmark client used the remaining 4 physical CPUs if a client was needed.

We present three main conclusions from the evaluation. First, we confirm that kernel specialization is important: Lupine achieves up to 73% smaller image size, 59% faster boot time, 28% lower memory footprint and 33% higher throughput than the state-of-the-art VM. However, we find that specialization on an application-level granularity may not be important: only 19 application-specific options cover the 20 most popular applications (83% of all downloads) and we find at most 4% reduction in performance by using a common configuration. Second, we find that, while running the application in the same privilege domain improves performance up to 40% on microbenchmarks, it only has a 4% improvement in macrobenchmarks, indicating that system call overhead should not be a primary concern for unikernel developers. Finally, we show that Lupine avoids major pitfalls of POSIX-like unikernels that stem from not being Linux-based, including both the lack of support for unmodified applications and performance from highly-optimized code.

## 4.1 Configuration Diversity

Lupine attempts to mimic the only-what-you-need approach of unikernels in order to achieve some of their performance and security characteristics. In this subsection, we evaluate how much specialization of the Linux kernel occurs in practice when considering the most popular cloud applications. Our primary finding is that many of the same configuration options are required by the most popular applications, and they are relatively few (19 options for the 20 most popular applications).

Unlike other unikernel approaches, Lupine poses no restrictions on applications and requires no application modifications, alternate build processes, or curated package lists.

[8]Determining exactly which options should be selected for a tiny kernel is difficult, but studies have shown that `tinyconfig` is a good starting point [16].

| Name | Downloads | Description | # Options atop *lupine-base* |
|---|---|---|---|
| nginx | 1.7 | Web server | 13 |
| postgres | 1.6 | Database | 10 |
| httpd | 1.4 | Web server | 13 |
| node | 1.2 | Language runtime | 5 |
| redis | 1.2 | Key-value store | 10 |
| mongo | 1.2 | NOSQL database | 11 |
| mysql | 1.2 | Database | 9 |
| traefik | 1.1 | Edge router | 8 |
| memcached | 0.9 | Key-value store | 10 |
| hello-world | 0.9 | C program “hello” | 0 |
| mariadb | 0.8 | Database | 13 |
| golang | 0.6 | Language runtime | 0 |
| python | 0.5 | Language runtime | 0 |
| openjdk | 0.5 | Language runtime | 0 |
| rabbitmq | 0.5 | Message broker | 12 |
| php | 0.4 | Language runtime | 0 |
| wordpress | 0.4 | PHP/mysql blog tool | 9 |
| haproxy | 0.4 | Load balancer | 8 |
| influxdb | 0.3 | Time series database | 11 |
| elasticsearch | 0.3 | Search engine | 12 |

**Table 3.** Top twenty most popular applications on Docker Hub (by billions of downloads) and the number of additional configuration options each requires beyond the *lupine-base* kernel configuration. [9]

As a result, we were able to directly run the most popular cloud applications on Lupine unikernels. To determine popularity, we used the 20 most downloaded container images from Docker Hub [2]. We find that popularity follows a power-law distribution: 20 applications account for 83% of all downloads. Table 3 lists the applications.

For each application, in place of an application manifest, we carried out the following process to determine the minimal viable configuration. First we ran the application as a standard container to determine *success criteria* for the application. While success criteria could include sophisticated test suites or achieving performance targets, we limited ourselves to the following tests. Language runtimes like `golang`, `openjdk` or `python` were tested by compiling (when applicable) a hello world application and testing that the message was correctly printed. Servers like `elasticsearch` or `nginx` were tested with simple queries or health status queries. `haproxy` and `traefik` were tested by checking the logs indicating that they were ready to accept traffic. We discuss the potential pitfalls of this approach in Section 6.

Once we had determined success criteria, we attempted to run the application on a Linux kernel built with the *lupine-base* configuration as described in Section 3.1. Recall that the base configuration is derived from microVM but lacks

[9]We exclude the Docker daemon in this table because Linux 4.0 does not support layered file systems, a prerequisite for Docker.

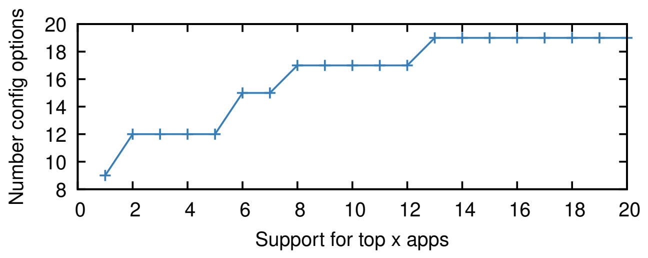

**Figure 5.** Growth of unique kernel configuration options to support more applications.

about 550 configuration options that we classified as hardware management, multiprocessing and application-specific. Some applications require no further configuration options to be enabled beyond *lupine-base*. For others, we added new options one by one while testing the application at each step. We expected all new options to be from the set classified as application-specific.

The process was manual: application output guided which configuration options to try. For example, an error message like *"the futex facility returned an unexpected error code"* indicated that we should add `CONFIG_FUTEX`, *"epoll_create1 failed: function not implemented"* suggested we try `CONFIG_EPOLL` and *"can't create UNIX socket"* indicated `CONFIG_UNIX`. Some error messages were less helpful and required some trial and error. Finally, some messages indicated that the application was likely not well-suited to be a unikernel. For example, `postgres` in Linux is made up of five processes (background writers, checkpointer, and replicator). It required `CONFIG_SYSVIPC`, an option we had classified as multi-process related and therefore not appropriate for a unikernel. Lupine can run such an application despite its obvious non-unikernel character, which is an advantage over other unikernel-based approaches. We will discuss the implications of relaxing unikernel restrictions in Section 5.

We conservatively estimate the time spent per application for a knowledgeable researcher or graduate student as 1 to 3 hours. However, we found knowledge of kernel options and experience accelerated the process. For example, we no longer need to perform trial and error for certain options, as we have learned that `CONFIG_FUTEX` is needed by glibc-based applications, and `CONFIG_EPOLL` is used by applications that utilize event polling.

Table 3 shows the number of configuration options (beyond *lupine-base*) deemed necessary to reach the success criteria for each application. Figure 5 depicts overlapping options thus showing how the union of the necessary configuration options grows as more applications are considered. The union of all configuration options is 19; in other words, a kernel (*lupine-general*) with only 19 configuration options added on top of the *lupine-base* configuration is sufficient to run all 20 of the most popular applications. The flattening of the growth curve provides evidence that a relatively small set of configuration options may be sufficient to support a large number of popular applications.

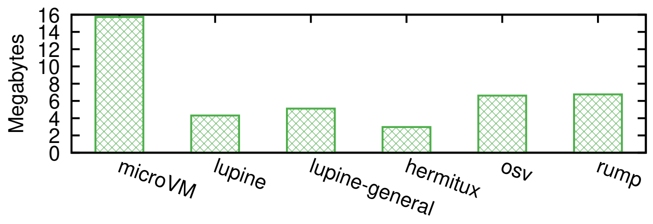

**Figure 6.** Image size for hello world.

As we show later in the evaluation, a kernel containing all of these common options, *lupine-general*, performs similarly to a manually configured kernel, an observation that matches recent results from building a systematic kernel debloating framework [35]. As a result, general users will likely not need to perform the manual process described in this section and can use *lupine-general* directly. It is an open question, however, to provide a guarantee that *lupine-general* is sufficient for a given workload.

## 4.2 Image Size

Most unikernels achieve small image sizes by eschewing generality. Similarly, Lupine uses the Linux kernel's configuration facilities for specialization. Figure 6 compares the kernel image size of Lupine to microVM and several unikernels—all configured to run a simple hello world application in order to measure the minimal possible kernel image size. The *lupine-base* image (sufficient for the hello world) is only 27% of the microVM image, which is already a slim kernel image for general cloud applications. When configuring Lupine to optimize for size over performance (*-tiny*), the Lupine image shrinks by a further 6%.

Figure 6 shows Lupine to be comparable to our reference unikernel images. All configurations except Rump utilize dynamic loading, so we report only the size of the kernel. To avoid unfairly penalizing unikernels like Rump, that statically link large libraries (like `libc`, which consists of 24M), we configure them to run a trivial hello world application without `libc`.

We also examined the effect on application-specific configuration on the Lupine kernel image size. We found that the image size of lupine kernels varied from 27 − 33% of microVM's baseline. Compared to *lupine-base*, this corresponds to an increase of up to 19 percent. Even with the largest Lupine kernel configuration (*lupine-general*, that is capable of running all of the top 20 applications) the resulting image size remains smaller than the corresponding OSv and Rump image sizes. We note that *lupine-general* is an upper bound for kernel image size for the kernels associated with any application in Table 3, including `redis`, `nginx`, etc.

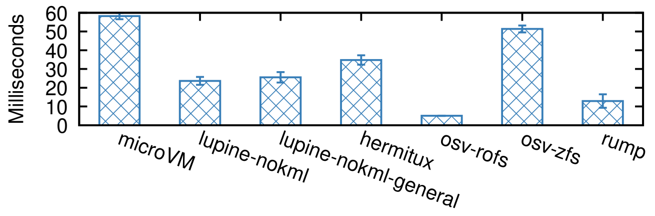

**Figure 7.** Boot time for hello world.

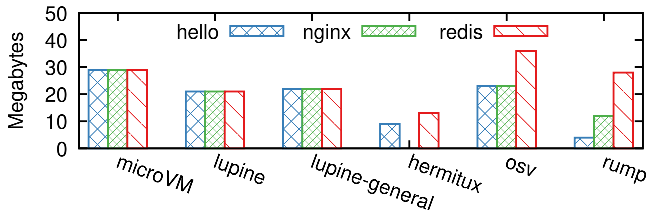

**Figure 8.** Memory footprint.

## 4.3 Boot Time

Figure 7 shows the boot time to run a hello-world application for each configuration. Firecracker logs the boot time of all Linux variants and OSv based on an I/O port write from the guest. We modified the unikernel monitors `solo5-hvt` and `uhyve` respectively to similarly measure boot time via an I/O port write from the guest.

As shown in Figure 7, use of a unikernel monitor does not guarantee fast boot time. Instead, unikernel implementation choices dominate the boot time. The OSv measurements show how dramatic the effects of unikernel implementation can be: when we first measured it using zfs (the standard r/w filesystem for OSv), boot time was 10x slower than the numbers we had seen reported elsewhere. After investigation, we found that a reduction in unikernel complexity to use a read-only filesystem resulted in the 10x improvement, thus underscoring the importance of implementation.

Lupine's configuration shows significant improvement over microVM and comparable boot time to the reference unikernels. In Figure 7, we present the boot time without KML (*lupine-nokml*). A primary enabler of fast boot time in Linux comes from the `CONFIG_PARAVIRT` configuration option which is active in microVM and *lupine-nokml*, but currently incompatible with KML. Without this option boot time jumps to 71 ms for Lupine. We believe that the incompatibilities with KML are not fundamental and could be overcome with engineering effort and would result in similar boot times to *lupine-nokml*. We do not find an improvement in Lupine's boot time when employing space-saving techniques (*-tiny*) with or without KML. In other words, the 6% reduction in image size described in Section 4.2 does not affect boot time thus implying that boot time is more about reducing the complexity of the boot process than the image size. For *lupine-general*, we measured an additional boot time of 2 ms. Note that this is still faster than HermiTux and OSv (with `zfs`). We note that, similar to image size, *lupine-general* conveys an upper bound in kernel boot time for the kernels associated with any application in Table 3, including `redis`, `nginx`, etc.

## 4.4 Memory Footprint

Unikernels achieve low memory footprint by using small runtime images that include only what is needed to run a particular application. We define the memory footprint for an application as the minimum amount of memory required by the unikernel to successfully run that application as defined by success criteria described in Section 4.1. We determine the memory footprint by repeatedly testing the unikernel with a decreasing memory parameter passed to the monitor. Our choice of applications was severely limited by what the (non-Lupine) unikernels could run without modification; we only present the memory footprint for three applications as shown in Figure 8. Unfortunately, HermiTux cannot run `nginx`, so we omit that bar.

Figure 8 shows the memory footprint for each application. In both application-specific and general cases, Lupine achieves a comparable memory footprint that is even smaller than unikernel approaches for `redis`. This is due in part to lazy allocation. While each of the unikernels shows variation in memory footprint, the Linux-based approaches (microVM and Lupine) do not.[10] There is no variation because the Linux kernel binary (the first binary to be loaded) is more or less the same size across applications. The binary size of the application is irrelevant if much of it is loaded lazily and even a large application-level allocation like the one made by `redis` may not be populated until later. However, an argument can be made in favor of eliminating laziness and upfront knowledge of whether sufficient resources will be available for an application. We further discuss this issue in the context of immutable infrastructure in Section 6.

## 4.5 System call latency microbenchmark

Unikernels claim low system call latency due to the fact that the application is directly linked with the library OS. Using Lupine, a Linux system, can achieve similar system call latency as other POSIX-like unikernel approaches. Figure 9 shows the `lmbench` system call latency benchmark for the various systems. [11] The results show that Lupine is competitive with unikernel approaches for the null (`getppid`), read

[10]OSv is similar to Linux in this case in that it loads the application dynamically, which is why `nginx` and `hello` exhibit the same memory footprint; we believe `redis` exhibits a larger memory footprint because of how the OSv memory allocator works.

[11]We only use the system call latency benchmark in `lmbench` due to lack of support in some unikernels for more complex benchmarks. However, we still report the full `lmbench` result for *lupine-general* and *microvm* in Appendix A.

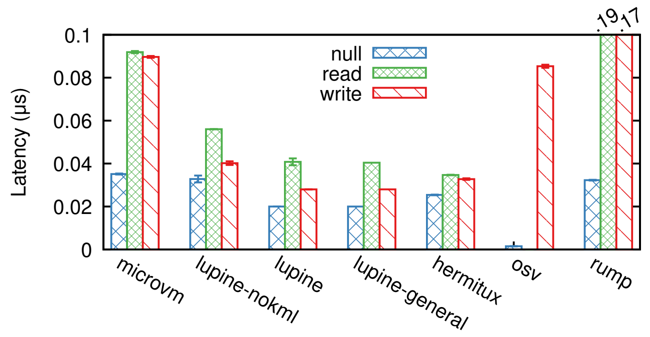

**Figure 9.** System call latency via `lmbench`.

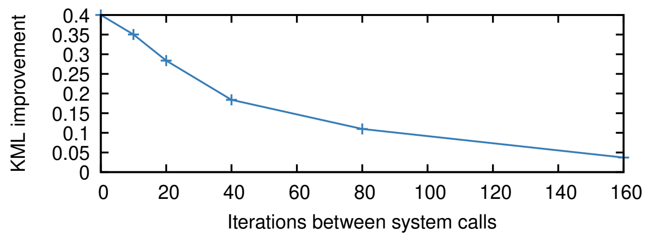

**Figure 10.** Relationship of KML syscall latency improvement to busying-waiting iterations (the more busy-waiting iterations the less frequent of user-kernel mode switching).

and write tests in `lmbench`. OSv shows the effects of implementation choices as `getppid` (issued by the *null* system call test) is hardcoded to always return 0 without any indirection. Read of `/dev/zero` is unsupported and write to `/dev/null` is almost as expensive as the microVM case.

Experimentation with *lupine-nokml* shows that both specialization and system call overhead elimination play a role. Specialization contributes up to 56% improvement (achieved during the write test) over microVM. However, we found no differences in system call latency between the application-specific and general variants (*lupine-general*) of Lupine. KML provides Lupine an additional 40% (achieved during the null test) improvement in system call latency over *lupine-nokml*.

To better understand the potential performance improvements of KML on Lupine, we designed a microbenchmark in which we issued the null (`getppid`) system call latency test in a loop, while inserting a configurable amount of CPU work via another tight loop to control the frequency of the switching between user and kernel mode: the frequency of switching decreases as the number of iterations increases.

In an extreme case where the application calls the system call without doing anything else (0 iterations) KML provides a 40% performance improvement. However, Figure 10 shows how quickly the KML benefits are amortized away: with only 160 iterations between the issued system calls the original 40% improvement in latency drops below 5%. We find

| Name | redis-get | redis-set | nginx-conn | nginx-sess |
|---|---|---|---|---|
| microVM | 1.00 | 1.00 | 1.00 | 1.00 |
| lupine-general | 1.19 | 1.20 | 1.29 | 1.15 |
| **lupine** | **1.21** | **1.22** | **1.33** | **1.14** |
| lupine-tiny | 1.15 | 1.16 | 1.23 | 1.11 |
| lupine-nokml | 1.20 | 1.21 | 1.29 | 1.16 |
| lupine-nokml-tiny | 1.13 | 1.13 | 1.21 | 1.12 |
| hermitux | .66 | .67 | ▒ | ▒ |
| osv | ▒ | ▒ | .87 | .53 |
| rump | .99 | .99 | 1.25 | .53 |

**Table 4.** Application performance normalized to MicroVM (Note: higher value is better).

similarly low KML benefits for real-world applications in Section 4.6.

## 4.6 Application performance

Unikernels boast good application performance due to lack of bloat and the elimination of system call latency. Table 4 shows the throughput of two popular Web applications: the `nginx` web server and the `redis` key-value store, normalized to microVM performance. As in the memory footprint experiment in Section 4.4, we were severely limited in the choice of applications by what the various unikernels could run without modification.

For clients, we used `redis-benchmark` to benchmark two common `redis` commands, `get` and `set`, measuring requests per second. For `nginx`, we used `ab` to measure requests per second. Under the connection-based scenario (*nginx-conn*), one connection sends only one HTTP request. Under the session-based scenario (*nginx-sess*), one connection sends one hundred HTTP requests.[12] We ran the clients on the same physical machine to avoid uncontrolled network effects.

As shown in Table 4, Lupine outperforms the baseline and all the unikernels. A general kernel (*lupine-general*) that supports 20 applications in Section 3 does not sacrifice application performance. We note that, as a unikernel-like system with a single trust domain, Lupine does not require the use of many recent security enhancements that have been shown to incur significant slowdowns, oftentimes more than 100% [52]. We attribute much of Lupine's 20% (or greater) application performance improvement (when compared to baseline) to disabling these enhancements. The poor performance of the unikernels is most likely due to the fact that the implementation of kernel functionality in Linux has been highly optimized over many years thanks to the large Linux community, beyond what other implementations can achieve. We would like to have more data points, but the inability to run applications on the unikernels is a significant challenge: even with these two extremely popular applications, OSv

[12]We use the `-keepalive` option in `ab`.

drops connections for `redis` and `nginx` has not been curated for HermiTux.

Within the Lupine variants, optimizing for space (e.g., -*tiny*) can cost up to 10 percentage points (for *nginx-conn*), while KML adds at most 4 percentage points (also for *nginx-conn*). As in the other experiments, KML and optimizing for size affects performance only a small amount relative to specialization via configuration.

## 5 Beyond Unikernels

Unikernel applications (and their developers) are typically restricted from using multiple processes, processors, security rings and users. These restrictions are often promoted as a feature (e.g., a single address space saves TLB flushes and improves context-switch performance [32, 45]) and justified or downplayed in certain contexts (e.g., many microservices do not utilize multi-processing [64]). Unfortunately, there is no room for bending the rules: as a unikernel, an application that issues `fork` will often crash or enter into an unexpected state by a stubbed-out `fork` implementation (e.g., continuing as a child where there is no parent). Such rigidity leads to serious issues for compatibility: as we encountered in our evaluation, it is unlikely that an existing application will run unmodified on a unikernel, even if the library OS is more-or-less binary compatible. Furthermore, there are situations where relaxing the unikernel restrictions is imperative. As a trivial example, building the Linux kernel with a single processor takes almost twice as long as with two processors.

Lupine is, at its core, a Linux system, and relaxing its unikernel properties is as simple as re-enabling the relevant configuration options. This results in a *graceful degradation* of unikernel-like performance properties. For example, rather than crashing on `fork`, Lupine can continue to execute correctly even if it begins to experience context switch overheads. Next, we investigate what the cost would be for Lupine to support applications that use multiprocessing features and whether including this support would adversely affect applications that do not.

We first consider the use of multiple address spaces and experiment with two different scenarios. First, we consider auxiliary processes that spend most of their time waiting either waking up or running in a frequency that does not interfere or create contention on resources with the application. We refer to such processes as *control processes*, i.e., processes that are responsible for monitoring the application for multiple purposes (e.g., shells, environment setup, recovery and analysis, etc.). In practice, it is extremely common, for example, to find a script that forks an application from a shell after setting up some environment variables. Lack of support for this case from existing POSIX-compliant unikernel implementations severely limits their generality. We design an experiment to show that such uses of multiple address spaces are not harmful to unikernel-like performance. Specifically, we measure the system call latencies after launching $2^i$ ($i = 0, 1, \ldots, 10$) control processes, using `sleep` as the control process. As shown in Figure 11, in all cases, there is no latency increase; all measurements (averaged over 30 runs) are within one standard deviation.

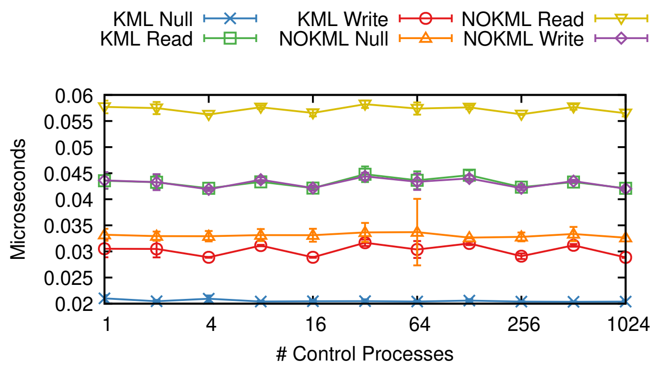

**Figure 11.** System call latency with different number of background control processes for KML and NOKML.

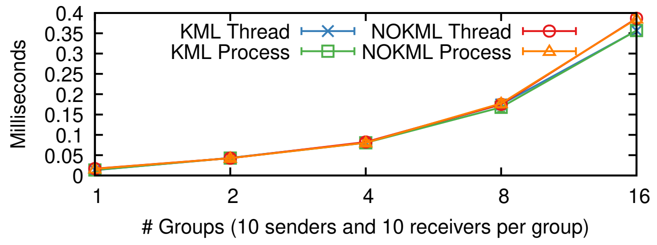

**Figure 12.** Perf context switch benchmark with threads and processes.

Second, we consider co-existing processes that contend resources with each other and may experience context switch overheads. To quantify these overheads, we compare the context switch overheads for threads that do not switch address spaces (to approximate unikernel behavior) versus processes. We use the messaging benchmark in `perf` [10] where $2^i$ ($i = 0, 1, 2, 3, 4$) groups (10 senders and 10 receivers per group) of threads or processes message each other via UNIX sockets. The benchmark implements threads with `pthread` and processes with `fork`. For each configuration, we average the results of 30 runs. As shown in Figure 12, surprisingly, in all numbers of groups, switching processes is not slower than switching threads as the maximum time increase is 3% (in the KML case) when there is 1 group. In some cases, we even see process switching outperforming thread switching by $0 - 4\%$. This finding matches prior work [8]. Since no performance is lost, we conclude that the adherence to a single address space is unfounded from a performance perspective

Next, we investigate the effects of Linux's symmetric multiprocessing (SMP) support as configured by the `CONFIG_SMP` kernel configuration option. We devised three experiments, *sem_posix*, *futex* and *make -j*, to show the worst-case scenario for supporting SMP: a system with one processor running applications that frequently context switch. We expect to see an overhead from a kernel that supports SMP versus a more unikernel-like kernel that does not. *sem_posix* and *futex* spawn up to 512 workers that rapidly exercise futex and POSIX semaphore wait and post operations. Each worker starts 4 processes sharing a futex or semaphore. *make -j* builds the Linux kernel using up to 512 concurrent processes. We found *sem_posix* incurs up to 3%, *futex* incurs up to 8%, and *make* incurs up to 3% overhead over a kernel without SMP support. In most cases, where there would be fewer context switches, we would expect even less overhead, so the choice to use SMP—rejected by unikernels—will almost always outweigh the alternative.

## 6 Discussion

The evaluation in Section 4 and the opportunity to gracefully degrade for non-unikernel workloads described in Section 5 make a compelling case for Lupine over other general-purpose unikernel approaches. Here we discuss the robustness of our analysis of Lupine and some benefits that unikernels achieve that Lupine does not.

### 6.1 Threats to validity

The conclusions drawn from our evaluation of Lupine rely on the correctness of our methodology, mainly in how the Linux kernel configuration—a notoriously messy part of Linux—is treated. The main risk is that Lupine has underestimated the necessary kernel configuration options for each application.

First, when determining the *lupine-base* configuration in Section 3, we may have misclassified certain kernel options as unnecessary rather than application specific. Moreover, the minimum set of configuration options that make up *lupine-base* may not be unique especially when considering options that provide similar functionality with different performance or space tradeoffs such as different compiler flags: `-O2` and `-Os`. Even if we were to find a different, more minimal *lupine-base*, the conclusions would hold.

Deriving an application-specific kernel configuration is more concerning. While not a focus of this paper—we assume its existence in the form of an application manifest—the evaluation of Lupine depends on an accurate application-specific kernel configuration. We determined configurations for the top 20 applications on Docker Hub based on a manual process based on simple success criteria and benchmarks that allowed us to quickly evaluate the configurations for many applications. When considering applications that do one thing and do it well (e.g., microservices), it may be more feasible to have a complete test suite to ensure that all configuration options are accounted for. In general, the problem is difficult: a large body of ongoing work attempts to derive kernel configuration from an application [30, 31, 37]. However, we believe the risk to be low: we noticed that many applications perform a series of checks when they start up, reducing the importance of complex success criteria. In our experience, in all cases, a weaker success criteria based on console output matched the configurations derived based on benchmark success.

Finally, we note that language-based unikernels, such as MirageOS [41], while unable to run existing POSIX applications, can use language-level analyses and package management techniques to determine application dependencies on OS functionality (e.g., networking), essentially removing the need for a manifest as needed by Lupine.

### 6.2 Unachieved unikernel benefits

Two of the unikernels we evaluated Lupine against (HermiTux [45] and Rump [23]) run on unikernel monitors [63] that are slim and optimized for unikernel workloads. Beyond boot times, unikernel monitors have been demonstrated to require such a small amount of host functionality that they can be implemented even as processes, thereby increasing performance while maintaining isolation [64, 65]. Linux does not currently run on a unikernel monitor, but it may possibly in the future given the fact that Linux can run in a variety of limited hardware environments (e.g., even with no MMU) or in environments that are not hardware like (e.g., User Mode Linux [12]).

The concept of *immutable infrastructure* has been associated with unikernels—especially language-based unikernels—in part because they push tasks that are traditionally done at deploy-time or later (like application configuration) to build time. As a result, the unikernel is specialized not just for an application but for a particular deployment of an application. However, general-purpose systems and applications often have dynamic or lazy properties—such as those seen when measuring the memory footprint in Section 4.4—that limit how immutable deployments can be. For example, interpreted code like JavaScript has become popular in the cloud but is dynamically loaded and interpreted. Dynamic behavior—which is pervasive in existing cloud applications and programming models—and immutability will continue to participate in a fundamental tussle as the cloud landscape evolves.

Another benefit of language-based unikernels that Lupine does not enjoy is the ability to perform language-based analyses or compiler optimizations that span both the application and kernel domain. For instance, MirageOS can employ whole-system optimization techniques on the OCaml code—from application to device drivers. POSIX-like unikernels tend to have less opportunity for this type of compiler optimization due to practical deployment and compatibility concerns when attempting to support legacy applications without completely changing their build processes. In the case of Linux, while link-time-optimization (LTO) exists for the kernel, it does not include the application in the analyses. Specializing the kernel via configuration, as shown for Lupine, may improve the results of LTO, but kernel routines or system calls cannot be inlined into the application without modifying both the kernel and application build processes. While some interesting new approaches are attempting this in the context of Linux [51], simultaneously maintaining full generality, or the ability to run any application (as Lupine can), remains a challenge.

## 7 Related Work

***Unikernel-like work that leverages Linux.*** The work that is most similar to ours includes efforts that attempt to leverage Linux as a unikernel. LightVM [43] demonstrates that VMs (and their associated tooling) can be as lightweight as containers and describes both mini-OS [7]-based unikernels and small Linux-based images using a tool called TinyX. While TinyX shares many design goals with Lupine, it does not focus on how to specialize through kernel configuration, nor does it examine the effects of running application code in the same privilege ring as the kernel.

X-container [55] uses Linux as a library OS by leveraging Xen [18] paravirtualization and eliminating guest kernel isolation. It uses binary translation to remove `syscall` instructions while avoiding application modifications (such as the musl `libc` modifications we make in Lupine). However, X-container requires modifications to both Linux and Xen. Unlike unikernel approaches, but similar to Lupine, X-container does support multi-processing and is targeted at container workloads. X-container does not investigate specializing the kernel through configuration as Lupine does.

UniLinux [20] and Unikernel Linux (UKL) [51] are recently proposed projects that share the main goal with Lupine: using Linux to achieve unikernel-like properties. For these, Lupine can provide a baseline; we evaluate how close Linux comes to these properties using existing techniques. UKL, on the other hand, involves relatively heavy modifications to the Linux source code, the build process, and the application, by building everything as a single binary and directly jumping to the application entry point from the kernel initialization code. Lupine uses existing mechanisms to achieve similar properties, thus making it immediately practical but unable to take advantage of techniques like cross-layer optimization that UKL may benefit from in the future.

Similarly, in other work, Linux has been used as a library but not a unikernel: the Linux Kernel Library [47] provides a way to compile Linux as a library for use by applications. It reuses Linux code (as does User Mode Linux [12]) to run in a deprivileged mode. LKL requires extensive modifications to Linux and has not been merged upstream. Also, LKL does not attempt unikernel-like isolation but provides a mechanism for user-space projects to take advantage of Linux-quality networking or filesystem implementations.

***The Linux configuration system*** The Linux configuration system is complex and various efforts have been undertaken to better understand it. The existing efforts that address kernel configuration include approaches for extracting the mapping between the Linux kernel configuration options and source files [24, 56], finding the correlation between configuration and the kernel size [16], and methods to select the minimum set of configurations to produce full coverage for testing [59, 61]. When applied to security, configuration-based kernel specialization has been shown to have significant benefits. For instance, Alharthi et al. [17] study 1530 Linux kernel vulnerabilities and show that 89% of these can be nullified via configuration specialization. Kurmus et al. [37] show that 50% to 85% of the attack surface can be reduced in either of these forms via configuration.

There are (semi-)automatic approaches to specialize the kernel via configuration [22, 29, 30, 35, 37–39, 43, 57, 60, 66]. These approaches use dynamic analysis techniques (e.g., tracing) to determine what parts of the kernel are executed and generate the configuration. They are currently limited by only considering code executed during the analysis phase. Similarly, debloating has been applied to user-space software with techniques such as program analysis, compiler optimization and machine learning [21, 27, 28, 48–50, 54, 58]. Though our results suggest it may not be necessary to specialize on a per-application basis, Lupine can benefit from such approaches [36, 37] as a way to generate application manifests.

## 8 Conclusion

While unikernels and library OS designs seem like a reasonable, lightweight alternative to more traditional virtual machines, the desire to increase the generality of unikernels along with an underestimation of the versatility of Linux has led us to stray too far from the potential benefits of unikernels. We show that Lupine, a pure Linux system, can outperform such unikernels in all categories including image size, boot time, memory footprint and application performance while maintaining vital properties, e.g., the community and engineering effort in the past three decades. Future research efforts should focus on making Linux specialization more effective and accessible.

## Acknowledgement

We thank the anonymous reviewers of EuroSys and our shepherd, Thomas Pasquier, for the valuable feedback that improved this paper. This work is supported in part by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizons Network. This work is also supported in part by the Office of Naval Research (ONR) grant N00014-17-S-B010.

Any opinions, findings, and conclusions made in this material are those of the authors and do not necessarily reflect the views of the funding agency.

## References

[1] AWS Lambda. https://aws.amazon.com/lambda/. (Accessed on 2016-03-04).

[2] Docker hub. https://hub.docker.com/.

[3] Dockerslim. https://dockersl.im/.

[4] Firecracker. https://firecracker-microvm.github.io/.

[5] IBM OpenWhisk. https://developer.ibm.com/open/openwhisk/. (Accessed on 2016-03-04).

[6] LING. http://erlangonxen.org.

[7] Mini-os. https://wiki.xenproject.org/wiki/Mini-OS.

[8] On threads, processes and co-processes. https://elinux.org/images/1/1c/Ben-Yossef-GoodBadUgly.pdf.

[9] Page table isolation (pti). https://www.kernel.org/doc/html/latest/x86/pti.html.

[10] perf - performance analysis tools for linux. http://man7.org/linux/man-pages/man1/perf.1.html.

[11] Twistlock | container security & cloud native security. https://www.twistlock.com/.

[12] The user-mode linux kernel home page. http://user-mode-linux.sourceforge.net/.

[13] Clive: Removing (most of) the software stack from the cloud. http://lsub.org/ls/clive.html, Apr. 2015.

[14] Javascript library operating system for the cloud. http://runtimejs.org/, Apr. 2015.

[15] The rumprun unikernel and toolchain for various platforms. https://github.com/rumpkernel/rumprun, Apr. 2015.

[16] M. Acher, H. Martin, J. A. Pereira, A. Blouin, J.-M. Jézéquel, D. E. Khelladi, L. Lesoil, and O. Barais. Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes. Research report, Inria Rennes - Bretagne Atlantique, Oct. 2019.

[17] M. Alharthi, H. Hu, H. Moon, and T. Kim. On the Effectiveness of Kernel Debloating via Compile-time Configuration. In *Proceedings of the 1st Workshop on SoftwAre debLoating And Delayering*, Amsterdam, Netherlands, July 2018.

[18] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. of ACM SOSP*, Bolton Landing, NY, Oct. 2003.

[19] A. Bratterud, A.-A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *Proc. of IEEE CloudCom*, Vancouver,Canada, Nov. 2015.

[20] T. Chen. Unikernelized Real Time Linux & IoT. In *Linux Foundation RT-Summit*, Prague, Czech Republic, Oct. 2017.

[21] Y. Chen, S. Sun, T. Lan, and G. Venkataramani. TOSS: Tailoring Online Server Systems through Binary Feature Customization. In *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST'18)*, Toronto, Canada, Oct. 2018.

[22] J. Corbet. A different approach to kernel configuration. https://lwn.net/Articles/733405/, Sept. 2016.

[23] J. Cormack. The rump kernel: A tool for driver development and a toolkit for applications.

[24] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. A Robust Approach for Variability Extraction from the Linux Build System. In *Proceedings of the 16th International Software Product Line Conference (SPLC'12)*, 2012.

[25] P. Emmerich, M. Pudelko, S. Bauer, and G. Carle. Writing User Space Network Drivers. *CoRR*, abs/1901.10664, 2019.

[26] D. R. Engler, M. F. Kaashoek, and J. W. O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proc. of ACM SOSP*, Copper Mountain, CO, Dec. 1995.

[27] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, 2018.

[28] Y. Jiang, D. Wu, and P. Liu. JRed: Program Customization and Bloatware Mitigation Based on Static Analysis. In *2016 IEEE 40th Annual Computer Software and Applications Conference*, 2016.

[29] J. Kang. A Practical Approach of Tailoring Linux Kernel. In *The Linux Foundation Open Source Summit North America*, Los Angeles, CA, Sept. 2017.

[30] J. Kang. An Empirical Study of an Advanced Kernel Tailoring Framework. In *The Linux Foundation Open Source Summit*, Vancouver, BC, Canada, Aug. 2018.

[31] J. Kang. Linux kernel tailoring framework. https://github.com/ultract/linux-kernel-tailoring-framework, Aug. 2018.

[32] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. OSv: optimizing the operating system for virtual machines. In *Proc. of USENIX Annual Technical Conf.*, Philadelphia, PA, June 2014.

[33] E. Kohler, R. Morris, B. Chen, J. Jannotti, and F. M. Kaashoek. The Click Modular Router. *ACM Transactions on Compuer Systems*, 18:263–297, August 2000.

[34] R. Koller and D. Williams. Will serverless end the dominance of linux in the cloud? In *Proc. of USENIX HotOS*, Whistler, BC, Canada, May 2017.

[35] H. Kuo, J. Chen, S. Mohan, and T. Xu. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(1), Mar. 2020.

[36] H. Kuo, A. Gunasekaran, Y. Jang, S. Mohan, R. B. Bobba, D. Lie, and J. Walker. MultiK: A Framework for Orchestrating Multiple Specialized Kernels. *CoRR*, abs/1903.06889, 2019.

[37] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*, San Diego, CA, USA, Feb. 2013.

[38] C.-T. Lee, Z.-W. Hong, and J.-M. Lin. Linux Kernel Customization for Embedded Systems By Using Call Graph Approach. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference (ASP-DAC'03)*, Jan. 2003.

[39] C.-T. Lee, J.-M. Lin, Z.-W. Hong, and W.-T. Lee. An Application-Oriented Linux Kernel Customization for Embedded Systems. *Journal of Information Science and Engineering*, 20(6):1093–1107, 2004.

[40] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[41] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proc. of ACM ASPLOS*, Houston, TX, Mar. 2013.

[42] T. Maeda and A. Yonezawa. Kernel mode linux: Toward an operating system protected by a type theory. In V. A. Saraswat, editor, *Advances in Computing Science – ASIAN 2003. Progamming Languages and Distributed Computation Programming Languages and Distributed Computation*, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[43] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My VM is lighter (and safer) than your container. In *Proc. of ACM SOSP*, Shanghai, China, Oct. 2017.

[44] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici. ClickOS and the art of network function virtualization. In *Proc. of USENIX NSDI*, Seattle, WA, Apr. 2014.

[45] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran. A binary-compatible unikernel. In *Proc. of ACM VEE*, Apr. 2019.

[46] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library os from the top down. *ACM SIGPLAN Notices*, 46(3):291–304, 2011.

[47] O. Purdila, L. A. Grijincu, and N. Tapus. LKL: The Linux kernel library. In *9th RoEduNet IEEE International Conference*, pages 328–333, June 2010.

[48] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee. RAZOR: A Framework for Post-deployment Software Debloating. In *Proceedings of the 28th USENIX Security Symposium*, Santa Clara, CA, USA, Aug. 2019.

[49] A. Quach, A. Prakash, and L. Yan. Debloating Software through Piece-Wise Compilation and Loading. In *Proceedings of the 27th USENIX Security Symposium*, Baltimore, MD, USA, Aug. 2018.

[50] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, 2017.

[51] A. Raza, P. Sohal, J. Cadden, J. Appavoo, U. Drepper, R. Jones, O. Krieger, R. Mancuso, and L. Woodman. Unikernels: The next stage of linux's dominance. In *Proc. of USENIX HotOS*, Bertinoro, Italy, May 2019.

[52] X. J. Ren, K. Rodrigues, L. Chen, C. Vega, M. Stumm, and D. Yuan. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 554–569, New York, NY, USA, 2019. Association for Computing Machinery.

[53] D. Schatzberg, J. Cadden, H. Dong, O. Krieger, and J. Appavoo. EbbRT: A framework for building per-application library operating systems. In *Proc. of USENIX OSDI*, Savannah, GA, Nov. 2016.

[54] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18)*, Sept. 2018.

[55] Z. Shen, Z. Sun, G.-E. Sela, E. Bagdasaryan, C. Delimitrou, R. Van Renesse, and H. 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 (ASPLOS'19)*, pages 121–135, New York, NY, USA, 2019. ACM.

[56] J. Sincero, R. Tartler, and D. Lohmann. Efficient Extraction and Analysis of Preprocessor-Based Variability. *ACM SIGPLAN Notices*, 46:33–42, 01 2011.

[57] K. Stengel, F. Schmaus, and R. Kapitza. Esseos: Haskell-based tailored services for the cloud. In *Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware*, ARM '13, pages 4:1–4:6, New York, NY, USA, 2013. ACM.

[58] C. Sun, Y. Li, Q. Zhang, T. Gu, and Z. Su. Perses: Syntax-guided Program Reduction. In *In Proceedings of the 40th International Conference on Software Engineering (ASE'18)*, 2018.

[59] R. Tartler, C. Dietrich, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Static analysis of variability in system software: The 90,000 #ifdefs issue. In *Proc. of USENIX Annual Technical Conf.*, Philadelphia, PA, June 2014.

[60] R. Tartler, A. Kurmus, B. Heinloth, V. Rothberg, A. Ruprecht, D. Dorneanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann. Automatic OS Kernel TCB Reduction by Leveraging Compile-time Configurability. In *Proceedings of the 8th USENIX Conference on Hot Topics in System Dependability (HotDep'12)*, 2012.

[61] R. Tartler, D. Lohmann, C. Dietrich, C. Egger, and J. Sincero. Configuration coverage in the analysis of large-scale system software. *SIGOPS OSR*, 45(3):10–14, 2012.

[62] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, page 9. ACM, 2014.

[63] D. Williams and R. Koller. Unikernel monitors: Extending minimalism outside of the box. In *Proc. of USENIX HotCloud*, Denver, CO, June 2016.

[64] D. Williams, R. Koller, M. Lucina, and N. Prakash. Unikernels as processes. In *Proc. of ACM SoCC*, Carlsbad, CA, Oct. 2018.

[65] D. Williams, R. Koller, and B. Lum. Say goodbye to virtualization for a safer cloud. In *Proc. of USENIX HotCloud*, Boston, MA, July 2018.

[66] L. M. Youseff, R. Wolski, and C. Krintz. Linux Kernel Specialization for Scientific Application Performance. Technical Report 2005-29, University of California Santa Barbara, 2005.

## A LMBench

We present the result of `lmbench` for *microvm* and *lupine-general* in Table 5.

| Op | MicroVM | Lupine-general |
|---|---|---|
| **Processor, Processes - times in microseconds - smaller is better** | | |
| Mhz | 4185 | 4176 |
| null call | 0.03 | 0.03 |
| null I/O | 0.08 | 0.05 |
| stat | 0.44 | 0.25 |
| open clos | 0.74 | 0.43 |
| slct TCP | 1.72 | 1.40 |
| sig inst | 0.09 | 0.07 |
| sig hndl | 0.60 | 0.40 |
| fork proc | 57.0 | 42.8 |
| exec proc | 202. | 156. |
| sh proc | 620. | 498. |
| **Context switching - times in microseconds - smaller is better** | | |
| 2p/0K ctxsw | 0.5800 | 0.4300 |
| 2p/16K ctxsw | 0.7000 | 0.5100 |
| 2p/64K ctxsw | 0.8500 | 0.6700 |
| 8p/16K ctxsw | 0.8900 | 0.7200 |
| 8p/64K ctxsw | 1.1800 | 1.0000 |
| 16p/16K ctxsw | 1.02000 | 0.81000 |
| 16p/64K ctxsw | 1.21000 | 1.02000 |
| ***Local* Communication latencies in microseconds - smaller is better** | | |
| 2p/0K ctxsw | 0.580 | 0.430 |
| Pipe | 1.837 | 1.181 |
| AF UNIX | 2.23 | 1.44 |
| UDP | 3.139 | 1.911 |
| TCP | 4.135 | 2.358 |
| TCP conn | 14. | 8.21 |
| **File & VM system latencies in microseconds - smaller is better** | | |
| 0K Create | 2.7555 | 1.2923 |
| File Delete | 1.7523 | 0.6989 |
| 10K Create | 8.2451 | 6.2253 |
| File Delete | 3.5362 | 1.2548 |
| Mmap Latency | 832.0 | 657.0 |
| Prot Fault | 0.274 | 0.279 |
| Page Fault | 0.10370 | 0.07770 |
| 100fd selct | 0.588 | 0.458 |
| ***Local* Communication bandwidths in MB/s - bigger is better** | | |
| Pipe | 9183 | 13.K |
| AF tNIX | 11.K | 14.K |
| TCP | 8563 | 9859 |
| File reread | 10.7K | 11.8K |
| Mmap reread | 16.0K | 15.9K |
| Bcopy (libc) | 12.6K | 12.5K |
| Bcopy (hand) | 9052.2 | 9060.0 |
| Mem read | 15.K | 15.K |
| Mem write | 12.1K | 12.1K |

**Table 5.** LMBench result for *microvm* and *lupine-general*