2023

Specializing a general-purpose operating system

BOSTON UNIVERSITY

GRADUATE SCHOOL OF ARTS AND SCIENCES

Dissertation

SPECIALIZING A GENERAL-PURPOSE OPERATING SYSTEM

by

ALI RAZA

B.S., LUMS, Pakistan, 2012
M.S., LUMS, Pakistan, 2016

Submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

2023

Approved by

First Reader

Orran Krieger, PhD
Professor of Electrical and Computer Engineering

Second Reader

Jonathan Appavoo, PhD
Associate Professor of Computer Science

Third Reader

Renato Mancuso, PhD
Assistant Professor of Computer Science

Fourth Reader

Abraham Matta, PhD
Professor of Computer Science

Fifth Reader

Richard West, PhD
Professor of Computer Science

"O my Lord! advance me in knowledge." Quran 20:114

To Ammi, Papa, Anza, Aapi, Maena, Bia, and most importantly, Ahad!

Acknowledgments

I want to express my deepest gratitude and appreciation to the following individuals who have been instrumental in shaping the successful completion of my Ph.D. thesis.

Prof. Orran Krieger, my advisor, for providing guidance, support, and patience throughout my journey. Your wisdom, knowledge, and expertise have helped shape me into the researcher I am today. I appreciate your dedication and commitment to my academic and personal growth. I am grateful for the opportunity to work with you and learn from you, and I will always cherish the memories of our collaboration. Thank you for being a great advisor, mentor, and friend.

Prof. Jonathan Appavoo and Prof. Renato Mancuso, my co-advisors, for your valuable contributions to my research. Your insights, feedback, and encouragement have been instrumental in shaping my work. I appreciate your support and the time and effort you have devoted to helping me achieve my goals.

Prof. Abraham Matta and Prof. Rich West for being part of my thesis committee and providing me with constructive feedback and guidance. Your expertise and insights have been invaluable in shaping my research, and I am grateful for your support throughout this process.

Ulrich Drepper, Richard Jones, Daniel Bristot de Oliveira, and Larry Woodman, for your mentorship. Despite your busy schedules, you always took the time to answer my questions, give me advice, and provide valuable feedback. Your contributions to my research and professional development have been invaluable, and I am grateful for the time and effort you have invested in my success. Your willingness to share your knowledge and expertise has helped me become a better researcher and inspired me to continue pursuing my passion for computer science. And Red Hat for supporting my research over the years.

To my Ammi and Papa, your unwavering love, support, and encouragement have been the foundation of my success. From the sacrifices you made to give me the best education possible, to the countless hours spent helping me with my studies, to the faith you had in me, your guidance and love have been instrumental in shaping me into the person I am today. I will always be grateful for your unwavering support and for instilling in me the values of hard work, determination, and perseverance.

To my sisters, Aapi, Maena, and Bia, thank you for your support throughout my life. Your love, encouragement, and belief in me have been a source of inspiration and motivation.

To my wife, Anza, thank you for your unconditional love, support, and patience throughout my Ph.D. journey. Your faith in me, your willingness to make sacrifices, and your constant encouragement have been the pillars of my success. From all the time spent at the lab to long hours of writing and editing, you have always been there, offering your support and encouragement every step of the way. Your unwavering love and commitment have provided the foundation on which I have built my academic and professional life. I am grateful for everything you have done for me.

And most importantly, to the light of my life, my son Ahad, thank you for inspiring me every day to be a better person. Your love, laughter, and joy have brought meaning and purpose to my life, and I am grateful for every moment we share together. I dedicate this thesis to you with the hope that it will inspire you to pursue your dreams and aspirations with passion and perseverance.

Ali Raza

SPECIALIZING A GENERAL-PURPOSE OPERATING SYSTEM

ALI RAZA

Boston University, Graduate School of Arts and Sciences, 2023

Major Professor: Orran Krieger, PhD
Professor of Electrical and Computer Engineering

ABSTRACT

This thesis aims to address the growing disconnect between the goals general-purpose operating systems were designed to achieve and the requirements of some of today's new workloads and use cases. General-purpose operating systems multiplex system resources between multiple non-trusting workloads and users. They have generalized code paths, designed to support diverse applications, potentially running concurrently. This generality comes at a performance cost. In contrast, many modern data center workloads are often deployed separately in single-user, and often single-workload, virtual machines and require specialized behavior from the operating system for high-speed I/O.

Unikernels, library operating systems, and systems that exploit kernel bypass mechanisms have been developed to provide high-speed I/O by being specialized to meet the needs of performance-critical workloads. These systems have demonstrated immense performance advantages over general-purpose operating systems but have yet to see widespread adoption. This is because, compared to general-purpose operating systems, these systems lack a battle-tested code base, a large developer community, wide application, and hardware support, and a vast ecosystem of tools, utilities,

etc.

This thesis explores a novel view of the design space; a generality-specialization spectrum. General-purpose operating systems like Linux lie at one end of this spectrum; they are willing to sacrifice performance to support a wide range of applications and a broad set of use cases. As we move towards the specialization end, different specializable systems like unikernels, library operating systems, and those that exploit kernel bypass mechanisms appear at different points based on how much specialization a system enables and how much application and hardware compatibility it gives up compared to general-purpose operating systems.

Is it possible, at compile/configure time, to enable a system to move to different points on the generality-specialization spectrum depending on the needs of the workload? Any application would just work at the generality end, where application and hardware compatibility and the ecosystem of the general-purpose operating system are preserved. Developers can then focus on optimizing performance-critical code paths only, based on application requirements, to improve performance. With each new optimization added, the set of target applications would shrink. In other words, the system would be specialized for a class of applications, offering high performance for a potentially narrow set of use cases.

If such a system could be designed, it would have the application and hardware compatibility and ecosystem of general-purpose operating systems as a starting point. Based on the target application, select code paths of this system can then be incrementally optimized to improve performance, moving the system to the specializable end of the spectrum. This would be different from previous specializable systems, which are designed to demonstrate huge performance advantages over general-purpose operating systems, but then try to retrofit application and hardware compatibility.

To explore the above question, this thesis proposes Unikernel Linux (UKL), which

integrates optimizations explored by specializable systems to Linux. It starts at the general-purpose end of the spectrum and, by linking an application with the kernel, kernel mode execution, and replacing system calls with function calls, offers a minimal performance advantage over Linux. This base model of UKL supports most Linux applications (after recompiling and relinking) and hardware. Further, this thesis explores common optimizations explored by specializable systems, e.g., faster transitions between application and kernel code, avoiding stack switches, run-to-completion modes, and bypassing the kernel TCP state machine to access low-level functions directly. These optimizations allow higher performance advantages over unmodified Linux but apply to a narrower set of workloads.

Contributions of this thesis include proposing a novel approach to specialization, i.e., adding optimizations to a general-purpose operating system to move it along the generality-specialization spectrum, an existence proof that optimizations explored by specializable systems can be integrated into a general-purpose operating system without major changes to the invariants, assumptions, and code of that general purpose operating system, a demonstration that the resulting system can be moved on the generality-specialization spectrum, and showing that performance gains are possible.

Contents

1 Introduction **1**

2 Background and Motivation **10**

2.1 Operating System Specialization 10

2.1.1 Opportunity for Specialization 11

2.1.2 Need for Specialization 11

2.2 Specialization Research and its Impact 12

2.3 Problems with Specializable Systems 14

2.3.1 Application support 14

2.3.2 Hardware support 15

2.3.3 Ecosystem 15

2.4 A Path Forward 16

3 Related Work **18**

3.1 Extensible Operating Systems 19

3.2 Unikernels and Library Operating Systems 27

3.2.1 Clean-Slate Systems 27

3.2.2 Split-Design Systems 36

3.2.3 Incremental Systems 42

3.3 Kernel Bypass 46

3.4 Generality-Specialization Spectrum 49

4 Architecture **54**

4.1 Goals 54

4.1.1 Optimizations 55
4.1.2 Application Support 56
4.1.3 Hardware Support 57
4.1.4 Ecosystem 57
4.2 Design 58
4.2.1 Base Model 59
4.2.2 Optimizations 61

5 Implementation **65**
5.1 Changes to Linux and `glibc` 65
5.2 Building UKL 66
5.3 Boot up and Initialization 68
5.4 Environment Tracking 68
5.5 Transition between execution environments 69
5.6 Permissions Checking 70
5.7 Changes to `execve` 71
5.8 Fork and Clone 71
5.9 Kernel to Application Transition with `UKL_RET` 73
5.10 Shared Stacks with `UKL_NSS` and `UKL_NSS_PS` 74
5.11 Page Faults with `UKL_PF_DF` and `UKL_PF_SS` 75
5.12 Bypassing Application-Kernel Transitions with `UKL_BYP` 77
5.13 Optimizations based on Application Modifications 78
5.14 Optimizations based on Application and Kernel Co-modifications . . 78

6 Evaluation **81**
6.1 Preserving Generality 82
6.1.1 Application support 82
6.1.2 Hardware support 83

6.1.3 Ecosystem 83
6.2 Experimental Setup 86
6.3 Microbenchmarks 86
6.3.1 System call base performance 88
6.3.2 Large requests 90
6.3.3 Page Fault handling 93
6.4 I/O Latency Benchmark 95
6.5 Single Threaded Application - Redis 97
6.5.1 Comparison with other Specialized Systems 97
6.5.2 Bare Metal Experiment 105
6.5.3 `perf` Analysis of Redis 108
6.6 Multithreaded Application - Memcached 116
6.7 Key Takeaways 120

7 Conclusion **121**

Curriculum Vitae **134**

List of Tables

4.1 UKL Configuration options 62

6.1 Comparison of UKL patch to 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. 84

6.2 Mean operation count and throughput in Mb/s of `fio` when run with Linux and UKL_RET_BYP. UKL showed a 36% improvement in operation count and throughput. 96

6.3 Throughput of Redis `SET`s and `GET`s in Million reqs/sec for Linux 5.14 and `UKL_RET_BYP (shortcut)` running bare-metal. `UKL_RET_BYP (shortcut)` shows more than 4% improvement for `SET`s and more than 8% improvement for `GET`s over Linux. `redis_benchmark` is used to generate traffic. 104

6.4 Redis throughput and latency improvements of UKL base model, `UKL_RET_BYP` and `UKL_RET_BYP (shortcut)` over Linux 108

6.5 Total number of instructions executed and CPU cycles (and time) taken by Redis running on Linux, UKL base model, `UKL_RET_BYP` and `UKL_RET_BYP (shortcut)`. The instructions per cycle ratio is also shown. 111

6.6 Branches encountered and mispredicted by Linux, UKL base model, `UKL_RET_BYP` and `UKL_RET_BYP (shortcut)` while running Redis. . . . 112

6.7 An analysis of L1 cache performance for Linux, UKL base model, UKL_RET_BYP and UKL_RET_BYP (shortcut) while running Redis. A breakdown into instruction and data cache accesses and misses is also shown. 113

6.8 Data TLB performance of Redis running on Linux, UKL base model, UKL_RET_BYP and UKL_RET_BYP (shortcut). 115

6.9 Last-level cache accesses and misses for Redis running on Linux, UKL base model, UKL_RET_BYP and UKL_RET_BYP (shortcut). The CPU cycles and time taken for each system, already shown in table 6.5, is provided again for quick reference. 116

6.10 99th percentile tail latency, in msec, of Memcached running on Linux and UKL_RET_BYP (shortcut). Percentage improvement of UKL_RET_BYP (shortcut) over Linux is also shown. UKL_RET_BYP (shortcut) gets upto 10% tail latency improvement over Linux, even as the load on the Memcached server increases. 119

List of Figures

5·1 Part of a flame graph generated after profiling Redis with UKL base model with `perf` [9]. 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., bypassing the Linux VFS [8] layer and calling TCP functions directly from Redis code. . . 79

6·1 Comparison of Linux, UKL base model, and UKL_BYP for simple system calls. With modern hardware, the UKL advantage of avoiding the system call overhead is modest (<5%). However, there appears to be a significant advantage for simple calls with `UKL_BYP`, which bypasses the entry and exit code on transitions between application and kernel. 89

6·2 Comparison of Linux, UKL base model, and UKL with bypass configuration for `read` and `write` system calls. With increasing payload for each system call, the UKL base model (orange line) shows modest improvement over Linux, but there is a significant advantage for `UKL_BYP` (green line). The percentage improvement for `UKL_BYP` (shaded area) over Linux decreases as payload increases but is still significant for 8KB payload. 91

6·3 Comparison of Linux, UKL base model, and UKL with bypass configuration for `sendto` and `recvfrom` system calls. With increasing payload for each system call, the UKL base model (orange line) shows modest improvement over Linux, but there is a significant advantage for `UKL_BYP` (green line). The percentage improvement for `UKL_BYP` (shaded area) over Linux decreases as payload increases but is still significant for 8KB payload. 92

6·4 Latency of stack page faults; UKL_PF_DF handles problematic page faults on a double fault stack, and UKL_PF_SS handles all page faults on a dedicated stack showing slight improvement over Linux. UKL_RET_PF_DF is further configured to use `ret` instead of `iret` when returning from page faults and shows a higher improvement over Linux. 94

6·5 Redis SET throughput comparison of UKL with Unikraft [54] and Lupine [56]. We provide Linux 4.0 as a baseline comparison for Lupine and Linux 5.14 as a baseline comparison for UKL. We reuse the setup done by Unikraft[54] for this experiment where the Redis server was running on these systems inside a single-core virtual machine. The client ran `redis-benchmark` outside the virtual machine, on the host, with 30 connections, 100k requests, and pipelining 16 requests. 98

6·6 Redis GET throughput comparison of UKL with Unikraft [54] and Lupine [56]. We provide Linux 4.0 as a baseline comparison for Lupine and Linux 5.14 as a baseline comparison for UKL. We reuse the setup done by Unikraft[54] for this experiment where the Redis server was running on these systems inside a single-core virtual machine. The client ran `redis-benchmark` outside the virtual machine, on the host, with 30 connections, 100k requests, and pipelining 16 requests. 99

6·7 Repeat of the experiment in fig. 6·5 (Redis SET throughput), except with Linux 5.14 and `UKL_RET_BYP (shortcut)` run in a 2 core virtual machine, with one core isolated through `isolcpus` boot parameter. The Redis server, which is single-threaded, was pinned on this isolated core through `taskset` utility. Providing an extra core improves Linux 5.14 and `UKL_RET_BYP (shortcut)` performance. 101

6·8 Repeat of the experiment in fig. 6·6 (Redis GET throughput), except with Linux 5.14 and `UKL_RET_BYP (shortcut)` run in a 2 core virtual machine, with one core isolated through `isolcpus` boot parameter. The Redis server, which is single-threaded, was pinned on this isolated core through `taskset` utility. Providing an extra core improves Linux 5.14 and `UKL_RET_BYP (shortcut)` performance. 103

6·9 Probability Density (purple bars) and CDF (orange line) of Redis deployed on Linux, UKL, `UKL_RET_BYP` and `UKL_RET_BYP (shortcut)` and tested with the Memtier benchmark. Average latency (broken red line) and 99th percentile tail latency (broken purple line) are also shown. 107

6·10 A description of the node we use for Redis experiments, created through the `lstopo` utility, which shows a single NUMA node with 2 sockets (or packages), each containing 8 cores. Each core has separate 32KB L1 instruction and data caches and a 256KB unified L2 cache. All 8 cores on a socket share a 20MB L3 or last-level cache. . . 109

6·11 99% tail latency for Memcached against increasing load. Memcached is configured to run with 4 threads, pinned to 4 cores, and deployed inside a 6 core VM. The client, i.e., `memtier_benchamrk`, was also configured with 4 threads, each pinned to a separate core and deployed natively on a different physical node, and both nodes connected through a physical network. This figure shows the 99th percentile tail latency of Memcached on different systems as the number of connections per thread in `memtier_benchmark` was increased to simulate increasing load. `UKL_RET_BYP (shortcut)` shows lower latency than Linux, even at a higher load (see table 6.10 for details on latency numbers and percentage improvement). UKL base model and `UKL_RET_BYP` also show a smaller improvement over Linux in most cases. 117

List of Abbreviations

ABI	.....	Application Binary Interface
API	.....	Application Programming Interface
eBPF	.....	extended Berkeley Packet Filter
BSD	.....	Berkeley Software Distribution
CDF	.....	Cumulative Distribution Function
CPU	.....	Central Processing Unit
DBMS	.....	Database Management System
DMA	.....	Direct Memory Access
DPDK	.....	Data Plane Development Kit
DRAM	.....	Dynamic Random-Access Memory
ELF	.....	Executable and Linkable Format
EXT4	.....	Fourth Extended Filesystem
FIO	.....	Flexible I/O Tester
FTP	.....	File Transfer Protocol
HCL	.....	Hardware Compatibility List
HPC	.....	High Performance Computing
IDT	.....	Interrupt Descriptor Table
I/O	.....	Input-Output
IOMMU	.....	Input–Output Memory Management Unit
IP	.....	Internet Protocol
IPC	.....	Inter-Process Communication
IST	.....	Interrupt Stack Table
JVM	.....	Java Virtual Machine
KML	.....	Kernel Mode Linux
KVM	.....	Kernel-based Virtual Machine
L1	.....	Level 1
L2	.....	Level 2
L3	.....	Level 3
LKL	.....	Linux Kernel Library
LLC	.....	Last Level Cache

LOC	.....	Lines of Code
LTO	.....	Link Time Optimization
LWN	.....	Linux Weekly News
MPM	.....	Multiprocessor Module
NIC	.....	Network Interface Card
NUMA	.....	Non-Uniform Memory Access
OS	.....	Operating System
PAL	.....	Platform Abstraction Layer
PIC	.....	Programmable Interrupt Controller
POSIX	.....	Portable Operating System Interface
PSL	.....	Protected Shared Libraries
QEMU	.....	Quick EMUlator
RAM	.....	Random-Access Memory
RCU	.....	Read-Copy Update
RDMA	.....	Remote Direct Memory Access
RFC	.....	Request For Comments
RPC	.....	Remote Procedure Call
SATA	.....	Serial Advanced Technology Attachment
SMP	.....	Symmetric Multiprocessing
SPDK	.....	Storage Performance Development Kit
SR-IOV	.....	Single Root I/O Virtualization
STL	.....	Standard Template Library
TAL	.....	Typed Assembly Language
TCP	.....	Transmission Control Protocol
TLB	.....	Translation Lookaside Buffer
TLS	.....	Thread-Local Storage
UDP	.....	User Datagram Protocol
UKL	.....	Unikernel Linux
VDSO	.....	Virtual Dynamic Shared Object
VFS	.....	Virtual File System
VLAN	.....	Virtual Local Area Network
VM	.....	Virtual Machine
VMA	.....	Virtual Memory Area

Chapter 1

Introduction

General-purpose operating systems, like Linux, offer enormous value in the data center for a number of reasons. First, they provide features like backward compatibility, a stable API, security and isolation, and services applications depend on for execution, e.g., threading, scheduling, timekeeping, synchronization, and memory management. Second, many applications, tools, utilities, debuggers, profilers, etc., have been written for general-purpose operating systems, allowing developers and operators to share a collective knowledge of designing, deploying, and performance-tuning applications. Third, the vast hardware compatibility list (HCL) of general-purpose operating systems like Linux allows applications to be deployed on many different architectures and take advantage of a diverse range of accelerators that are becoming increasingly important. Fourth, they offer some specialization, i.e., for performance or functional reasons, applications can, up to a degree, modify the operating system's behavior and code, e.g., in Linux, through build-time configuration options, run-time policy management, insertable modules, or recent developments like eBPF. Fifth, these operating systems have a vast community of developers, especially around Linux, which keeps maintaining and fixing bugs in the existing code base and adding new features and device drivers.

However, there is a growing mismatch between the design of general-purpose operating systems and the requirements of some of today's cloud workloads. Operating systems like Linux were designed assuming that multiple non-trusting applications

would share the same system. Under these assumptions, it is challenging to optimize for a single application, and security, isolation, and privacy come at the cost of performance. The general-purpose operating system APIs were designed to abstract away hardware devices and multiplex them between many applications. However, with the advancements in virtualization technology, non-trusting applications don't have to share the same system anymore; instead, they are now deployed separately in single-user, and often single-application, virtual machines. Applications deployed on virtual or dedicated physical machines do not need to share resources and should not have to pay this penalty.

The design of general-purpose operating systems has also not evolved at the same pace as advancements in hardware. With the emergence of high throughput network and storage I/O devices, e.g., 400 Gbit/s Ethernet cards, general-purpose operating systems have become I/O bottlenecks. The socket API, and even new interfaces like io_uring, provided by Linux, cannot keep up with the requirements of many modern workloads [99]. Consequently, there is a need for specialized behavior from the operating system.

There has been a massive body of work exploring operating system specialization, from extensible operating systems - which allowed applications to insert custom code into the kernel (§ 3.1), to library operating systems and unikernels (§ 3.2) - which involved developing new or adapting existing operating systems according to application needs, and modern-day kernel bypass mechanisms (§ 3.3) - which, as the name suggests, bypass existing operating systems on performance-critical paths.

Extensible systems [90] allowed applications to insert custom code into the kernel, thus extending the standard interface, to gain performance improvements. These systems were developed in the 80s and 90s, before the advent of virtualization technology and cloud computing, when the hardware and operating system had to be

shared between multiple applications. Consequently, these operating systems could not be specialized fully to the needs of any single application, and security became a major concern, giving birth to the idea that *"Extensible Kernels are Leading OS Research Astray"* [34].

Recently, after advancements in virtualization technology removed the requirement for multiple applications to share hardware, there has been a resurgence of research systems exploring the idea of a library operating system, or a *unikernel*, where an application is linked with a specialized kernel and deployed directly on (virtual) hardware [35]. Unikernels are mostly designed and written from scratch according to the needs of a class of applications. They have shown tremendous improvements over general purpose operating systems like Linux in boot time [63, 54], security [100, 76], resource utilization [29], and I/O performance [88, 52].

Despite all these advantages, unikernels have not been widely adopted. As with any operating system, widespread adoption of a unikernel will require significant and ongoing investment by a large community. Justifying this investment is difficult because unikernels target only niche portions of the broad use cases of general-purpose operating systems. Many only run a single application [63, 88, 54], whereas many workloads require secondary applications to run, e.g., loggers and profilers. Most unikernels only support virtualized deployments [88, 54, 76, 88] and often a single processor core [54, 63], whereas many workloads are often deployed bare-metal, e.g., Ceph [96] or are multi-threaded and require multiple cores. Unikernels do not support accelerators (e.g., GPUs and FPGAs) that are increasingly critical to achieving high performance in a post Dennard scaling world. Some unikernels do not support any backward compatibility [63] and require the applications to be fully ported to the new APIs they provide.

Some unikernel projects have tried to remedy the lack of application and hardware

compatibility of from-scratch design by taking general-purpose operating systems and modifying them. Examples include NetBSD-based Rump Kernel [49], Windows-based Drawbridge [80], and Linux-based Linux Kernel Library (LKL) [83]. These projects, however, make significant changes to the base general-purpose operating systems, resulting in a fork of the code base and community. As a result, ongoing investments in the base operating system are typically not incorporated into these forked projects.

To avoid the investment required to maintain a new or significantly modified operating system, the recent Lupine [56] and X-Containers [91] projects exploit Linux's innate configurability to enact application-specific customization. These projects avoid the hardware overhead of system calls between user and kernel mode, but they do not explore deeper optimizations that unikernels written from scratch do. Essentially these projects preserve the API between the application and the underlying operating system, giving up on unikernel performance advantages that depend on specializing kernel code to the application.

Despite the challenges, operating system specialization research points us to the potential of specialization. Based on application knowledge, special-purpose code paths can switch to polling if frequent I/O is expected [98, 25, 29, 43, 65, 77, 79, 81, 88], the complex state machines in the general purpose operating systems can be flattened and unneeded conditions can be removed [88, 52], scheduling policies can be modified [92, 81, 44, 77, 37], multiple implementations of different features can co-exist [38, 82, 17, 88, 54], I/O paths can be highly customized [88, 54], zero-copy networking can be implemented [88, 52], etc.

The influence of operating system specialization can also be felt in general-purpose systems. While some specialization can be very specific, if it solves a critical problem and doesn't have intrusive changes, it can become part of the general purpose code base, maintained and extended as more applications use it [4, 3]. Examples of mech-

anisms that enable specialization include modularity with well-defined interfaces and virtual functions to select context-appropriate code paths, dynamically inserting or removing modules, a massive list of compile-time and run-time configuration options, and most recently, eBPF to download custom code into the Linux kernel to be executed in a virtual machine, etc. While continuing to be general purpose, with one code base, Linux has become increasingly specializable, to the extent some believe it is all that is needed [56]. The existing specialization techniques in Linux already require an administrator to enable them, e.g., to configure/build the system or to download the customized eBPF code.

General purpose operating system specialization suffers from some of the same problems that earlier extensible operating systems faced; general purpose operating systems are general, designed to provide performance to different applications, and thus cannot be fully specialized to a single application. Modular code paths and virtual functions still need to cater to broad use cases, e.g., TCP v UDP traffic, instead of application-specific cases. Loadable modules and compile-time and run-time configurations can only be implemented by kernel developers, posing a massive barrier to entry for application developers who better understand the needs of their applications. Further, although eBPF suggests that significant gains can be obtained if in-kernel specialization can be enabled with application code, the problem with eBPF is that it is limited in the kinds of optimizations possible, and a totally new programming model needs to be adopted. This means user-level libraries won't work, and application programmers have to rewrite their code completely.

In this thesis, we propose the idea of having a system that, at compile time of the kernel, can be specialized with regular application code, adopting the same requirement specializable systems have, as well as Linux currently has with root privileges, that there is one critical application to run. Imagine a spectrum where the

same kernel can be configured as a general-purpose system on one end (sacrificing performance to preserve application and hardware compatibility) and as a highly specialized system on the other end (optimizing performance, even at the cost of application and hardware compatibility). At the general-purpose operating system end of the spectrum, the unmodified application, linked into the kernel, enjoys broad application and hardware compatibility, and an ecosystem of tools and utilities, while other applications can run alongside it. Can we start on this end of the spectrum and then move towards the specialized end, based on application needs, by turning on optimizations explored by earlier specializable systems?

Taking Linux as a starting point, in terms of design and implementation, is it possible to build such a system that can be adopted across the generality-specialization spectrum? Would the resulting changes be so many that we essentially end up with a forked code base? Will we be able to preserve Linux's battle-tested code base, its developer community, its application compatibility and HCL, and its ecosystem of tools and utilities? Is it possible to integrate optimizations explored by specializable systems into Linux? Would these optimizations provide any performance benefits? Will unmodified applications be able to take advantage of this system? Can we enable developers to design custom optimizations by modifying the applications and the kernel together? This thesis is an exploration of these questions. We chose Linux as a starting point since it is the defacto operating system for cloud deployments today, and countless applications, tools, and utilities have been written for it. It has broad hardware support, and a large community of operators and engineers share its collective knowledge and understanding.

If we can build a system that can be adopted across the generality-specialization spectrum, we hypothesize the following will be possible. At the general purpose end, the system will just work on all hardware, with all applications, configured and de-

ployed using standard techniques that the community knows about. Applications can use standard tools to identify the performance-critical optimizations that could potentially be adopted. New configuration options could be employed incrementally and tested thoroughly to identify the workloads and scenarios that they are appropriate for. Application and user libraries could be written to improve performance only where it matters, solving one problem at a time rather than having to write an entire system. In contrast to eBPF, the application developer can do the work instead of the kernel expert, and all user libraries will just work. Since it is one code base, the entire community of the general-purpose operating system can be involved, and there is never the need to make a massive investment to re-produce what the general-purpose system already does well. Just like systems have adopted increasing modularity, they can incrementally start adding new checks, internal interfaces, and options to tie into application code for critical workloads. As one moves towards a more and more specialized system to improve performance, the system will be suitable for fewer applications, but it is the decision of the developer if it is justified.

Our research has demonstrated that building a system that can be moved across the generality-specialization spectrum is possible. In our system *'Unikernel Linux'* (UKL), like many unikernels, a single application is statically linked with the kernel and executed in supervisor mode. This base model of UKL preserves most of the invariants and design of Linux, including a separate pageable application portion of the address space and a pinned kernel portion, distinct execution environments for application and kernel code, and the ability to run multiple processes. As a result, this base model provides an avenue toward supporting all hardware and applications of the original Linux kernel and the entire Linux ecosystem of tools for deployment, debugging, and performance tuning. The changes to Linux to support the UKL base model (550 LoC) and the resulting performance improvement (e.g., 5% for syscall)

are, as expected, modest.

Once an application runs in the UKL base model, a developer can move along the spectrum towards the specialized end by adopting additional configuration options and/or modifying the applications to invoke kernel functionality directly. These steps may improve performance but may only work for some applications. Example configuration options we have explored include avoiding costly transition checks between application and kernel code, using simple return (rather than `iret`) from page faults and interrupts, and using shared stacks for application and kernel execution. Application modifications can, for example, defer scheduling and exploit application knowledge to reduce the overhead of synchronization and polymorphism. Experiments show up to 83% improvement in system call latency and substantial performance advantages for real workloads, e.g., 26% improvement in Redis throughput while improving tail latency by 22%. The full UKL patch to Linux, including the base model and all configurations we have developed so far, is 1250 LoC.

This thesis answers some research questions and explores others. It proves that it is possible to link an unmodified application with the kernel and execute it at the kernel privilege level. It also demonstrates that the changes are practical, the patch set is modest in size, and there are significant performance gains. To answer the question of how far this system can go to adopt the kind of optimizations explored by earlier systems, this is intrinsically a many-year process. This thesis adds the base capability and enables application developers to pursue potentially many different optimizations.

This thesis makes the following contributions:

1. Proposed the design of a system that can be adapted from a general-purpose operating system to a specialized system instead of being a fixed point on the generality-specialization spectrum.

2. An existence proof that optimizations explored by specializable systems can be integrated into a general-purpose operating system in a fashion that preserves its compatibility and application execution environment.

3. A demonstration that a single kernel can be adopted across a spectrum between a general-purpose operating system, where it enjoys complete application and hardware compatibility, and the ecosystem of tools, utilities, and community, and a specialized unikernel, which is specialized to the requirements of a single application.

4. A demonstration that performance advantages are possible; although unmodified applications achieve modest gains, co-optimizing the application and kernel code can achieve more significant gains.

The rest of this thesis is organized as follows. Chapter 2 provides the relevant background and motivation for this research. Chapter 3 discusses previous research in the area of operating system specialization and, based on that research, builds the case for the generality-specialization spectrum. Chapter 4 presents the goals for UKL and its design to meet those goals. Chapter 5 discusses UKL's implementation details and Chapter 6 presents the evaluation. Finally, Chapter 7 concludes this thesis and discusses potential future work.

Chapter 2

Background and Motivation

Modern data center workloads, deployed on dedicated machines, don't share the system with non-trusting applications. This relaxes their security requirements from the operating system and provides an opportunity for specialization (§ 2.1.1). Further, the high-performance requirements of some of today's use cases necessitate specialization (§ 2.1.2). We briefly discuss the research efforts in the space of specialization and their impact on general-purpose operating systems (§ 2.2). Then we discuss how these specializable systems lack wide application and hardware compatibility, large ecosystem of developers, tools, and utilities of general-purpose operating systems, and how it is a significant impediment to wide adoption (§ 2.3). Finally, we discuss a way forward and the motivation for this thesis (§ 2.4).

2.1 Operating System Specialization

With the advancement in virtualization technology and the advent of cloud computing over the last decade, the way we deploy applications and what is required of the operating systems is changing. General-purpose operating systems have not kept up with these recent trends. This section discusses the modern data center workload requirements and the challenges general-purpose operating systems face.

2.1.1 Opportunity for Specialization

General-purpose operating systems are designed to allow multiple non-trusting applications to share the same system. They maintain strict isolation and separation between different applications and also between applications and the kernel. To execute any privileged operation, the kernel ensures that the application is allowed to do that operation and then executes it on behalf of the applications. For example, to send a buffer over the network, Linux ensures that the buffer exists in the memory region designated for that application, uses specific paths to copy over that buffer to its own memory, and then directs the NIC to transfer it.

In numerous modern use cases, data center workloads do not share the machine with non-trusting workloads, e.g., Memcached [69] and Redis [85] are often deployed on thousands of dedicated virtual machines. Infrastructure workloads, e.g., Ceph [96], are usually deployed on dedicated bare-metal servers. These applications can significantly benefit if the operating system could be specialized to the needs of the application and allows optimizations like direct access to device buffers and zero-copy I/O paths etc.

2.1.2 Need for Specialization

General-purpose operating systems were designed at a time when storage was backed by slow disks and network bandwidth was limited. There was no need to develop faster abstractions or APIs because the operating system was not the limiting factor. Today, we have fast storage through highspeed drives and non-volatile memory, and high bandwidth, low latency networks coupled with highspeed NICs, especially in data centers. General-purpose operating systems and their APIs have emerged as the new I/O bottlenecks. Studies have shown that the Linux API is not suited to the new highspeed I/O devices [99]. The need for operating system specialization to remove

performance bottlenecks is paramount.

Without specialization, operating systems are becoming prohibitively slow for many performance-sensitive applications. These applications are now being adapted to bypass the operating system and directly access hardware devices for faster I/O through frameworks like DPDK [2] and SPDK [15]. Although the Linux community is adding newer APIs of its own, e.g., io_uring, kernel bypass techniques, and userspace I/O stacks show higher performance improvements [67].

2.2 Specialization Research and its Impact

In response to the mismatch between the design of general-purpose operating systems and the high-performance requirements of some applications and use cases, operating system specialization has always been a focus of research. A large body of work in extensible operating systems explored specialization, dating back to the 80s and 90s [92, 35, 27, 72]. Since these systems came before the advances in virtualization technology, multiple applications had to share the same system. This meant that extensible operating systems could not be totally specialized to a single application, since the system still needed to ensure that application-specific specialization could not compromise the security and isolation of other applications.

With the advancement in virtualization technologies, applications did not have to share the machine. This led to operating system specialization focusing on a single application; unikernel [63, 88, 54, 52, 76] and kernel bypass research [50, 43, 81, 44, 77, 37, 99]. Unikernels are specialized operating systems, often written from scratch [63, 88, 54], designed according to the requirements of specific classes of applications [52, 54, 88]. These systems show immense performance improvements over, e.g., Linux [52, 54, 88] but have yet to see widespread adoption. Major challenges to their adoption include the lack the applications and hardware support,

mature codebase, and ecosystem of tools, developers, and community of general-purpose operating systems. Other research has focused on bypassing the general-purpose operating systems [24, 79, 25] and accessing I/O devices directly [87, 24, 43] to gain huge performance benefits. These kernel bypass techniques require specialized NICs [61, 81, 32, 46], offload security and isolation to hardware devices, and have poor integration with existing applications. Moreover, many kernel bypass projects are implemented as library operating systems [99, 25, 79, 81], and essentially have the same challenges as other library operating systems.

Despite failing to achieve wide-scale adoption, specializable systems have impacted the design of general-purpose operating systems. Extensible operating systems demonstrated that substantial performance gains are possible if applications can insert extensions into the kernel instead of forcing all applications to use the standard interfaces. Loadable modules and API extensions like io_uring [4] in Linux harken back to the era of extensible operating systems. Exokernel [35] pointed to the gains possible by packet filters, i.e., allowing application code to be downloaded into the operating system. Unikernels have shown how zero-copy networking [88], lock-free implementations [52], and customized code paths [88] can provide high-speed I/O to applications. Kernel bypass techniques have shown that if the complex state machine of the general-purpose network stack can be bypassed, it results in huge performance gains. eBPF now allows similar capabilities in Linux. These lessons have pushed general-purpose operating systems like Linux to be more specializable, especially if administrator privileges are used to optimize for a single application, e.g., through loadable modules, injecting application code into the kernel through eBPF, and compile-time and run-time configurations. Although there's a limit to this specialization, as general-purpose operating systems have to support a wide range of applications, the impact of specialization points to the need for more.

2.3 Problems with Specializable Systems

Specializable systems show performance improvements over general-purpose operating systems, but lack their application (§ 2.3.1) and hardware compatibility (§ 2.3.2) and their ecosystem of developers, tools, and utilities (§ 2.3.3). This makes the large-scale adoption of specializable systems hard. This section discusses these problems in more detail.

2.3.1 Application support

General-purpose operating systems provide an application programming interface (API), e.g., POSIX. Over the years, there have been many additions to the API of operating systems like Linux, e.g., new system calls or different programming models like io_uring, but the Linux community has maintained backward compatibility, ensuring that applications written for Linux are always supported. This has allowed a large body of applications, shared libraries, tools, and utilities, e.g., debuggers, and profilers, to be written for Linux. Linux also provides applications with complex services, e.g., scheduling, timekeeping, threading, memory management, etc. Applications don't have to be programmed differently for various hardware devices; instead, general-purpose operating systems abstract all the device idiosyncrasies behind the same API and abstractions, such as sockets for networking and files for storage. Further, applications don't have to protect themselves against other non-trusted applications on the same system; general-purpose operating systems ensure applications' isolation, security, and privacy.

Today's specializable systems give up on this generality to provide custom code paths for target applications. They either do not support standard APIs and services or implement only the parts needed to run the target application. Most only support a single process and, consequently, cannot run tools and utilities that the target

application might require.

2.3.2 Hardware support

A wide range of infrastructure applications (e.g., storage systems, schedulers, networking toolkits) that are typically deployed bare-metal depend on the vast Hardware Compatibility List (HCL) of Linux. In a post-Dennard scaling world, where performance depends on taking advantage of the revolution of heterogeneous computing (e.g., GPUs, TPUs, FPGAs), Linux enables the use of these accelerators. Linux has support for different architectures, e.g., x86_64, arm, etc., and drivers for a wide range of peripheral devices. This encourages developers to write their applications for Linux, especially if they need support for a particular hardware device. This further incentivizes hardware vendors to provide drivers and libraries for Linux for their devices.

Specializable systems cannot support the entire HCL of general-purpose operating systems; that would be a momentous engineering task, even if these systems had large developer communities. Many only provide support for virtualized deployments. Consequently, direct access to hardware and I/O devices or support for accelerators is unavailable. Kernel bypass mechanisms also only support a small list of devices that allow access from userspace and have vendor-supplied libraries.

2.3.3 Ecosystem

General-purpose operating systems like Linux have huge development communities that keep integrating new features, fixing bugs, adding device drivers, and ensuring application compatibility. The Linux community also keeps adding new features to the Linux API, e.g., io_uring [4] and eBPF [3], etc., which offer higher performance benefits. Engineers, developers, and operators share the collective knowledge of building, debugging, and performance-tuning applications on Linux. The massive

body of compatible software and wide hardware compatibility list, along with a huge developer community, has created a vast ecosystem around Linux that is extremely important for today's cloud infrastructure.

Many specializable systems are limited to only research endeavors or are unmaintained after a few years since they lack the developer communities of general-purpose operating systems like Linux. Consequently, they have a limited or non-existent set of tools, debuggers, and profilers required by applications.

2.4 A Path Forward

Specializable systems have acknowledged the problem of adoption, and many have tried to address elements of it in different ways. Some have tried to offload all functionality to a general-purpose operating system and only implemented performance-critical code paths [88, 17]. Kernel bypass systems also only implement the data plane and use Linux as the control plane, i.e., the application can use Linux for all management and profiling tasks. Some systems even support features typically not supported by specializable systems, e.g., fork [76, 100], multiprocessing [93], or bare metal deployment [48, 88]. While these efforts have demonstrated that any one of the limitations can be addressed, it will be a heroic task to address all of them. Moreover, these systems lack the larger ecosystem of the general-purpose operating system, i.e., helper applications, tools, and utilities for logging, debugging, profiling, and management tasks, and a community of developers and experienced system administrators. On the contrary, general-purpose operating systems like Linux, although specializable to a degree, are still general by design and cannot be fully specialized only to one application.

There is a need to preserve Linux's application and hardware compatibility, battle-tested code base, and its ecosystem while also enabling the

system to be more specialized to one single application by adopting optimizations explored by specializable systems. UKL aims to explore this research problem.

Chapter 3

Related Work

Unikernel Linux (UKL) aims to integrate application-specific optimizations into Linux, a general-purpose operating system. This allows applications to start with Linux's full hardware and software compatibility and its ecosystem of tools, utilities, and developers. Optimizations explored by specialized systems can then be added to Linux according to application needs. As optimizations are added to Linux, the target set of applications those optimizations are appropriate for decreases but the performance gains for that set increase. This generality-specialization spectrum between Linux and highly specialized systems is central to UKL research. Earlier research systems can be placed on this spectrum, and this thesis adopts many valuable lessons from them. This chapter discusses the earlier research and how it has motivated and impacted UKL.

In Section § 3.1, we discuss how Extensible operating systems of the 80s and 90s were designed to allow multiple non-trusting applications to securely customize the behavior of the underlying operating system. Recent unikernels (Section § 3.2) exploit the fact that applications don't have to share the system with non-trusting applications, and have explored specialized paths for application performance. Finally, in Section § 3.3, we discuss how kernel bypass systems provide great insights into what parts of Linux are performance bottlenecks and how applications can benefit from direct access to hardware devices. The advantages and drawbacks learned from kernel bypass research in creating custom code paths inspire the deep optimiza-

tions explored in UKL research. Below we discuss all of this previous work, with descriptions of individual projects, and summarize the insights and lessons learned from them.

3.1 Extensible Operating Systems

There has been a huge body of work in the 90s that explores extensible operating systems [90]; these are systems where *"applications can interact with the OS through additional extension interfaces"* [34], i.e., these operating systems allow application developers to extend the standard interface by adding custom high-performance code to the kernel [92, 27]. Extensible operating systems were developed before the advent of virtualization under the assumption that multiple non-trusting applications would share the system. Under these assumptions, it was imperative to ensure that extensions inserted into the operating system by one application did not affect another application, the security and trust boundaries between applications were not compromised, extensions inserted by an application could be reversed once that application finished executing, and the operating system could arbitrate between conflicting extensions of different applications [89]. Solving these self-created problems meant that the operating system could not be specialized to a single application, and led some to argue the *"Extensible Kernels are Leading OS Research Astray"* [34].

3.1.1 Vino [92]

Vino takes database management systems (DBMS) as the case study and argues that services provided by the operating system might not be suited to application needs, resulting in performance loss. Resource-intensive systems like DBMS often avoid the operating system services and implement their own in userspace to improve performance, e.g., avoiding the filesystem and writing to the raw disk directly. In other cases, it is impossible to avoid operating system services, e.g., virtual memory

management. This can lead to a doubling of effort, i.e., DBMS and the operating system implementing their own custom policies.

To fix this mismatch between DBMS requirements and operating system services, Vino makes application-specific tailoring of services a norm, not an exception. All policies can be modified at the application's discretion, and the operating system arbitrates the policy of competing applications. Vino provides an extensible interface for services with a default implementation that can be modified fully or partly by applications. Extensions, or *"grafts"*, are added dynamically to the kernel to avoid privilege switch costs. Vino ensures safety through static type checking of grafts and sandboxing them at runtime. Vino ensures fairness in resource utilization by implementing timeouts. Many recent unikernels follow the same philosophy of customizable services, e.g., Unikraft [54] and language level protection mechanisms, e.g., MirageOS [63]. UKL is also motivated by the disconnect between application requirements and Linux design for some use cases.

3.1.2 Cache Kernel [30]

Cache kernel argues that microkernels are not the solution to the limitations of monolithic general-purpose operating systems. Although microkernels provide modularity, reliability, and security improvements, they suffer from some major drawbacks; they are slow due to privilege separation between servers and the kernel, bloated because virtual memory management and support for different hardware devices have to be part of the kernel, and non-extensible because they don't provide sufficient control over resource management to applications. This paper introduces Cache Kernel, a minimal supervisor mode kernel that caches the OS objects, e.g., threads, address spaces, etc., while the rest of the functionality is implemented in application kernels in userspace. Cache kernel exports an interface to allow application kernels to load and unload objects. Applications run on top of application kernels in the same or

separate address spaces. Application kernels reduce supervisor-level complexity and allow application-level resource management and application control over exception handling.

Application kernels load objects into the cache kernel that need to be executed. When an application faults or traps, the cache kernel catches that and notifies or gives control to the application kernel, providing application-level control of exception handling. A multiprocessor module (MPM) is a collection of nodes/CPUs and their caches. One cache kernel runs on one MPM along with a resource management server. Applications run on MPMs by getting resources from the resource manager. Running multiple cache kernels provides parallelism and fault containment, i.e., if an MPM/cache kernel fails, it does so independently of other MPMs or cache kernels. This model allows applications to manage hardware, enables application-specific exception handling, and reduces the complexity of the supervisor. Recent systems like Graphene [93] use similar ideas of providing multiprocessing by running multiple instances of the system. The difference between Exokernel [35] and Cache kernel [30] is philosophical. Cache kernel, unlike the Exokernel, imposes a hard boundary between the supervisor and application kernels, and does not allow the inserting of packet filters or safe user code into the kernel.

3.1.3 Spin [27]

Spin aims to provide high performance to performance-sensitive applications by allowing safe extensions. Spin argues that the interface exported to the applications should provide fine-grain access to system services and allow security control at the same granularity. The communication between the extensions and the kernel should be low latency. To ensure this, Spin dynamically loads extensions into kernel virtual address space, which are accessible through simple procedure calls. Extensions are written in a type-safe language to ensure safety, and further isolation is provided

by using namespaces. Recent systems like MirageOS [63] and EbbRT [88] also use language-level primitives for isolation and performance.

3.1.4 Exokernel [35]

The exokernel authors argue that operating systems limit the flexibility and performance of applications because they export a rigid, non-extensible interface. They hide information about the machine behind high-level abstractions, and their implementation cannot be replaced or modified. New abstractions can only be added as awkward emulation on top of existing ones.

They argue that the lower the level of a primitive, the more efficient its implementation and the more latitude it grants higher-level abstractions. The ideal interface is the hardware. Exokernel is a minimal kernel that securely multiplexes the hardware resources and exports a very low-level interface to library operating systems, which are specialized for a single application and implement all of the system services like virtual memory and IPC etc. Although most of the functionality is implemented in library operating systems, it might be performant at times to download code into the kernel for execution, e.g., for packet filtering. Type-safe languages, static checking, and sandboxing may be used to run downloaded code safely. The exokernel, which runs at a higher privilege, can repossess all the resources of an uncooperative library operating system.

Exokernel has been the architectural model of almost all recent unikernels and library operating systems. Through the advancements in virtualization technology, hypervisors, e.g., Xen [21] and Qemu-KVM [10, 5] are modern manifestations of the Exokernel [35] and expose a very low-level machine interface. Unikernels [63, 52, 54, 66, 28, 55, 100] run on this exposed interface instead of being deployed bare metal.

3.1.5 Scout [72]

The layered system model is fundamental to how operating systems have been structured and built and allows us to manage complexity, isolate failures and enhance configurability. This research introduces the 'path' abstraction, which can be understood as a logical channel through a multi-layered system over which I/O data flows. Many operating system optimizations can be understood and explained through this path abstraction. This is because paths expose and exploit non-local context, allowing optimizations based on a global context. This global context is unavailable inside any single layer and can improve resource allocation, scheduling decisions, and code quality.

The contributions of this research include developing the Scout operating system with an explicit path abstraction and demonstrating how this path abstraction leads to advantages in resource allocation and scheduling. Examples of path-based optimizations which depend on global context include zero copy paths, real-time scheduling, and early decision-making. Examples of code quality improvements based on path abstraction include compiler optimizations, e.g., constant folding and propagation, dead-code elimination, etc., and eliminating redundant work by merging per-layer operations. Modern kernel bypass techniques [2, 15] follow a similar philosophy by creating direct custom paths from applications to hardware devices. The entire layered stack of traditional operating systems is replaced by a global context, leading to performance benefits.

3.1.6 Flux OSkit [36]

This research notes that building an operating system is hard work. Before researchers can work on 'interesting' components, they have to implement functionality like early bootstrap or device initializations, etc. Adapting an existing operating system for

research purposes can be complicated and full of dependencies.

Flux OSKit provides a framework to use a set of modularized library code with simple well-documented interfaces for operating system construction, e.g., bootstrapping, a POSIX environment, memory management, etc. Each library either has a single goal, e.g., drivers, or a set of functions that can be used by other components, e.g., a C library. All these components can optionally also be taken from other well-developed operating systems, improving overall code quality and maturity. Flux OSKit provides a kernel support library, which contains functionality to deal with the underlying architecture, e.g., to initialize segmentation, page tables, install IDT and traps, etc. Recent systems like Unikraft [54] and EbbRT [88] follow a similar modular approach.

3.1.7 Protected Shared Libraries [20]

Protected Shared Libraries, or PSL, is based on the premise that the flexibility and extensibility of a system depend on its modular design. Modularity can be achieved through a microkernel design or language primitives, e.g., object-oriented design. But these approaches have limitations; the modularity of microkernel design is limited to user kernel separation, and language-based primitives require building a new operating system altogether.

PSL adopts a library model as opposed to a process model. In PSL, modularity comes from passive protection domains through shared libraries, and efficiency comes from cross-domain interactions with shared data. PSL is implemented by dividing the address space into different regions, each with its own sharing permissions. When switching between libraries, memory regions can be mapped and unmapped. The linker and loader are responsible for dividing the code into separate sections and loading them into the correct regions in memory, respectively. Shared libraries allow zero-copy code, which is efficient and easier for the programmer to understand, and

sharing of code allows the same symbols to be called from different domains and still be resolved correctly. For protection, libraries can only be entered from secure entry points, so the code and data of the previous library can be unmapped, and the code and data of the new library can be mapped. Recent systems like Kylinx [100] also take a similar approach by supporting dynamic libraries which can be updated at runtime.

3.1.8 SLIC [39]

This work aims to add new efficient extensions to commodity operating systems through interposition. Interposition means capturing events crossing an interface boundary and forwarding those events to an extension. The extension can call the original interface or do something new. Previous approaches do a from-scratch implementation to create extensible systems, but the initial cost of implementation of an operating system is prohibitive and requires many developers and a considerable amount of time. Re-engineering available operating systems to be extensible is another approach, but since operating systems are complex, it can quickly turn into redesigning and reimplementing the operating system, which is, again, complex and very costly.

This paper takes the third approach, i.e., adding functionality with minor modifications to the underlying operating system and no changes to the applications. Interposition can be done through two approaches; modifying jump tables or making direct calls to kernel functions through binary rewriting. Extensions implemented in user space can take advantage of standard development and debugging techniques and are safe from malicious applications through process isolation. But user-level extensions suffer from the overhead of boundary crossing and context switching. Kernel-level extensions can be invoked directly from the kernel and have better performance. These extensions are protected from malicious applications by being in kernel space, but the

kernel is not protected against malicious or faulty extensions. UKL takes lessons from all the extensible systems but is very similar to SLIC in that it also tries to minimize the code changes to an existing operating system. UKL takes it further by having the option of modifying applications to exploit deeper shortcuts into the kernel.

3.1.9 K42 [53]

K42 was designed to address scalability and dynamic customization of operating system functionality on multiprocessor machines. The project noted that, to achieve scalability, it has to minimize sharing of operating system structures and instead improve locality. To that end, K42 uses an object-oriented design; system services consist of objects, i.e., process, file, etc., and instead of having global objects and locks, each processor has its own instance of those objects. In order to improve customizability, K42 moves kernel functionality into servers and application libraries, encouraging custom code paths and avoiding common implementations.

Each processor has its own pool of memory from which processor-specific objects are allocated. Any faults or client requests are handled at the same processor to improve locality. This object-oriented design also allowed applications to replace objects with new ones to provide better performance and code updates. This gave a lot of flexibility, where objects could be replaced based on the requirements of the applications at runtime, giving an adaptive nature to operating system design.

3.1.10 Discussion

Extensible operating systems were designed to integrate application-specific optimizations into general-purpose operating systems while maintaining security and isolation guarantees between non-trusting workloads. Although extensible operating systems have not been widely adopted, many lessons learned from them have made their way into Linux and modern specializable systems. Linux provides specialization up to

a degree by allowing loadable modules, newer APIs like io_uring, and allowing custom code to be downloaded into the kernel, i.e., eBPF [3]. All of these mechanisms essentially *extend* Linux's interface.

Compared to extensible operating systems, modern unikernels § 3.2 and library operating systems that exploit kernel bypass techniques § 3.3 solve a more straightforward problem. They also aim to provide workload-specific optimizations, but since workloads don't have to share machines, they don't have the security and isolation requirements of extensible operating systems. Modern specializable systems [88, 54, 63] take inspiration from the optimizations explored by extensible operating systems, e.g., custom code paths, language level primitives, pushing functionality into application libraries, etc.

3.2 Unikernels and Library Operating Systems

With the advent of virtualization and large-scale adoption of the cloud, the way workloads were deployed changed. Modern workloads are often deployed on single applications, single-user virtual machines. The relaxed security requirements due to not having to share the machine with other workloads provide the opportunity to optimize the performance of a single workload. Added to this, modern I/O-bound workloads require specialized behavior. These factors have led to modern unikernels and library operating systems that optimize for a specific workload. We divide library operating systems or unikernels into Clean-slate systems, split-Design systems, and incremental systems. Below is an explanation of each category and descriptions of a representative set of projects from each.

3.2.1 Clean-Slate Systems

These are systems written from scratch or *almost* written from scratch, i.e., some might use a minimal bare-bones kernel for bootstrapping, etc. A lot of clean-slate

systems, e.g., ClickOS [66], MiniCache [55], and KylinX [100] etc., build on MiniOS, which ships with Xen [21] and is a minimal kernel.

3.2.1.1 MirageOS [63]

MirageOS aims to build **Unikernels** and defines them as *"single-purpose appliances that are compile-time specialized into standalone kernels, and sealed against modification when deployed to a cloud platform."* Unikernels run applications written in a high-level language, run on a hypervisor and provide a reduction in image size, improved efficiency and security, and better performance. MirageOS is written in OCaml and uses language-level and compiler-based techniques to create unikernels that are more robust to vulnerabilities. Single-purpose image means fewer lines of code, so the attack surface is minimized. After compilation, it is sealed to further changes. MirageOS runs on top of Xen [21] hypervisor and builds on the MiniOS kernel that ships with Xen. MirageOS showed that security and efficiency benefits come at no performance degradation compared to traditional kernels and even improve performance in some cases, e.g., thread creation. MirageOS does not support hardware deployment and runs in virtualization over Xen. Applications must be rewritten for MirageOS; consequently, it does not have a large ecosystem of developers, tools, and operators.

3.2.1.2 EbbRT [88]

EbbRT aims to reduce the effort required to build application-specific library operating systems without sacrificing performance or optimization. General-purpose operating systems are not optimized for each application, and it is hard to customize or construct an operating system for each app. To remedy this, EbbRT consists of building blocks that can be extended, replaced, or discarded to create specialized systems. EbbRT allows hardware access with minimal abstraction due to a lightweight

execution model. EbbRT allows functionality to be offloaded to a general-purpose OS to ease engineering efforts. It has two modes of execution; native runtime - a lightweight bootable library operating system runtime, and hosted runtime - a user-level library that can be linked into a process of a general-purpose operating system. Native runtime allows apps to be written directly to hardware and provides basic functionality, i.e., clocks, networking, memory allocation, etc. Any unimplemented and non-performance critical functionality can be offloaded to the hosted runtime, allowing EbbRT applications to integrate with legacy software. EbbRT library operating systems show performance improvements compared to Linux, e.g., Memcached implemented for EbbRT delivers more than two times higher throughput. EbbRT's behavior is fixed after the build and has to be recompiled to make changes.

3.2.1.3 HermitCore [58]

HermitCore is a unikernel created for High Performance Computing (HPC) applications. It brings the multikernel operating system design [22] to unikernels by running a separate unikernel on each core and communicating through RPC over IP. HermitCore implements only 17 syscalls. Any pre and post-processing is done through a Linux instance running on one of the cores. Linux's hotplugging mechanism separates cores that boot up HermitCore unikernels. HermitCore allows behavior changes at runtime, but for code changes, it has to be recompiled. HermitCore runs bare metal but requires Linux for management. Applications have to be recompiled and linked with modified libraries to use HermitCore.

3.2.1.4 HermiTux [76]

HermiTux aims to reduce the effort required to port applications to a unikernel setting. It provides binary compatibility with existing Linux applications by supporting the syscall instruction to achieve its goal. Further, through binary analysis and

rewriting techniques, syscalls can be replaced by function calls in static binaries. For applications that use dynamic libraries, those libraries can be replaced with unikernel-aware libraries that already make function calls instead of syscalls. HermiTux uses a custom hypervisor called uhyve, an extension of ukvm [97] hypervisor. It extends HermitCore unikernel [58] to provide more syscall coverage, and its current prototype provides 83 syscalls. HermiTux shows performance benefits, e.g., more than five times lower syscall latency than Linux. HermiTux has to be recompiled for changes to manifest and runs in virtualization, so it does not target hardware support. Applications must be recompiled and linked with modified libraries to run on HermiTux.

3.2.1.5 Unikraft [54]

Unikraft aims to lower the difficulty of creating application-specific unikernels. It has a modular design where all operating system services, e.g., memory allocator, scheduler, etc., are implemented as micro-libraries. These micro-libraries can be replaced, skipped, or extended as required. It uses compiler features such as link-time optimization and dead-code elimination to achieve a smaller code size. It provides POSIX compatibility but allows non-POSIX APIs for performance improvement. It provides a syscall shim layer to translate syscalls made by libraries into function calls at build time. Due to the required engineering effort, it only supports the commonly used 146 syscalls. Unikraft performs 1.7 times to 2.7 times better than Linux for Nginx, SQLite, and Redis. Unikraft unikernels must be recompiled to make any changes and run in virtualization, so they don't target hardware support. Unikraft allows unmodified applications to run after recompilation and linking with modified libraries. It also allows applications to be modified for more significant gains. It has a thriving community of developers, but still extremely small compared to Linux.

3.2.1.6 Libra [17]

Libra is a system based on the exokernel and library operating system model [35]. It uses the Xen [21] hypervisor in the role of the exokernel, and Libra assumes the role of the library operating system. Library operating system development is complex due to the engineering effort required, so Libra only provides performance-critical features and offloads unimplemented functionality to a Linux domain. This reduces the cost of library operating system development. Libra ports a JAVA runtime so that applications written in JAVA can use the performance advantage of a specialized system. Libra needs to be rebuilt for changes to occur, and unmodified applications can run with Libra after linking with modified libraries. It has no hardware support and runs in virtualization.

3.2.1.7 Xax [31]

Xax notes that web apps enjoy host independence and isolation by running in a browser and aims to provide the same for desktop applications with native code performance. It creates a *picoprocess* abstraction, a lightweight process that executes native code with a narrow syscall interface. Desktop applications run in the picoprocess, which also includes a platform abstraction layer or PAL. PAL provides an operating system independent ABI to the applications and interfaces with the Xax monitor, which mediates access to the outside world and runs in a browser environment. The browser acts as a sandbox environment providing security, and PAL provides operating system independence. The implementation details of Xax reveal the difficulty in providing compatibility to applications in a restricted setting. Applications must be modified to remove unneeded dependencies, syscalls are rejected, or error values returned as long as the application does not exit. Some syscalls are emulated internally, and a few are provided through calls to the host. Understandably,

libraries such as pthreads are not supported. Like many clean-slate systems, Xax does not support runtime customization or codebase modifications. It runs in a browser, is hardware-independent, and has no application compatibility, i.e., applications must be rewritten.

3.2.1.8 ClickOS [66]

ClickOS takes the idea of operating system specialization and applies it towards a specific use case, i.e., provide a high-performance, virtualized software middlebox platform. Hardware-based middleboxes are expensive and difficult to manage, and software-based middleboxes have performance limitations. ClickOS uses a software router Click, along with the minimal library operating system MiniOS which ships with Xen to create ClickOS. It runs virtualized on Xen, and the project also modifies the Xen code base to optimize I/O for performance improvements. ClickOS has to be recompiled to make changes, does not target broad application compatibility, and only operates in virtualization.

3.2.1.9 OSv [52]

OSv is an operating system designed for cloud deployments. It's based on the premise that cloud deployments are single-application virtual machines, and a general-purpose operating system designed to multiplex resources is not suitable. Also, support for multiple devices is irrelevant to a virtualized operating system. OSv's goals are to run existing applications and frameworks like JVM faster than Linux, provide optional faster non-POSIX interfaces and create small images that can boot quickly. OSv is written from scratch in C++ and uses lock-free techniques, zero-copy mechanisms, and paravirtualized drivers for performance improvement. OSv has to be recompiled for changes to occur and runs only in virtualization, so it does not target hardware compatibility. It is almost binary compatible with existing Linux applications which

has allowed some adoption in cloud deployments.

3.2.1.10 KylinX [100]

KylinX is another point in the Xen/MiniOS-based unikernels area. It notes that despite having excellent isolation and performance benefits, unikernels don't have a process like abstraction, which limits flexibility and applicability. Its process-like VM (pVM) abstraction allows unikernels to call hypervisor-assisted fork to create new unikernel instances. It allows applications running as unikernels to communicate with each other through abstractions similar to interprocess communication. It will enable the hypervisor to support dynamic libraries in unikernel address space with the option of live library update. Kylinx allows modification in behavior and codebase at runtime. It does not target hardware compatibility and runs only in virtualization, and it also has no compatibility with existing Linux applications.

3.2.1.11 Discussion

Clean-slate systems have the freedom to design and optimize for performance. These projects show substantial performance benefits over traditional operating systems, e.g., Memcached for EbbRT shows more than two times higher throughput than Linux. For performance, many clean-slate systems use paravirtualized drivers, e.g., Virtio [54, 88] for I/O.

Regarding implementation, clean-slate systems require considerable effort because the operating system has to be built from scratch. As Ford et al. note [36] that *"any realistic operating system, in order to be useful even for research, must include many largely uninteresting elements such as boot loader code, kernel startup code, various device drivers, kernel printf and malloc code, and a kernel debugger"*. Clean-slate systems must make a significant engineering effort to write the 'uninteresting' code before the research contributions manifest. Consequently, these systems make some

trade-offs. For instance, many only support a single processor model [63, 54] or mostly run only in virtualization [63, 54, 76, 88]. Running in virtualization helps with device drivers. Hypervisors expose simple virtual devices, which frees the developers from writing drivers. Further, this gives clean-slate systems host independence, i.e., any hypervisor running on any hardware only needs to expose a simple machine interface for these systems to run. A few clean-slate systems are cognizant of the considerable engineering effort required. Unikraft [54] and Ebbrt [88] have a modular design, so building operating system appliances is easier. Developers can use different modules together to build the required kernel. These modules can be replaced, removed, or edited as needed. Further, EbbRT [88] allows offloading unimplemented functionality to a general-purpose operating system. This means the developers can start with a model where all I/O and other functionality are routed through a general-purpose operating system and then implement the most performance-critical parts of the system using EbbRT's modular design. This iterative development lets developers focus on the system's most critical features first.

Having the freedom of not having legacy kernel code, clean-slate systems can choose their level of compatibility with existing applications. MirageOS [63] offers no compatibility; it requires applications to be written in OCaml. EbbRT [88] also requires applications to be reimplemented or modified to take advantage of its event-driven execution and low latency I/O code paths. Unikraft [54] allows API-level compatibility with the help of modified C libraries. Further, Unikraft has the option of non-POSIX-Linux API for better performance, i.e., applications can be modified to use the new optimized API. Unikraft does not provide full API coverage yet (146 syscalls [54]), but providing the rest of the syscalls is an engineering effort and not a design limitation. HermiTux [76] goes all the way and provides full binary compatibility for Linux applications. Further, it provides binary rewriting for static binaries

and C library replacement for dynamic binaries to use the faster function call mechanism instead of the syscalls. HermiTux also only supports partial API coverage, i.e., 83 syscalls [76].

Clean-slate systems are primarily run only in virtualization. This allows them to offload security and isolation to the hypervisor. Further, hardware-based virtualization means these systems are isolated from the host. MirageOS [63] uses language-level safety through OCaml, a high-level statically typed language. This makes it robust to vulnerabilities. Further, MirageOS [63] appliances are secure by design. Their code and behavior can only be modified at build time, not run time.

Operating systems need decades to build communities for development, support, and bug fixes. No matter how specialized or performant, new operating systems cannot match long-running projects like Linux in this regard. Clean-slate systems have the biggest disadvantage here, having little or no developer support and a new codebase that is not battle-tested.

Clean-slate systems are designed to explore application-specific optimizations, e.g., MirageOS [63] and EbbRT [88] force the developers to use the optimized execution model and internal APIs fully since applications have to be rewritten for them from scratch. EbbRT [88] and Unikraft [54] help per application optimizations further through their modular design. Modules can be replaced based on application needs or developed just for a single application without modifying the rest of the system.

Clean-slate systems have the advantage of starting from scratch and designing the system without legacy baggage. These systems can achieve diverse goals, e.g., performance, language level safety, minimalism, small memory footprint, etc. Nevertheless, the factors holding them back are massive development effort, small or non-existent developer community, and lack of battle-tested codebase.

3.2.2 Split-Design Systems

These systems split a general-purpose operating system and use its architecture-independent parts as a library in userspace. Since this library cannot be used without the architecture-dependent code, these systems write new code to act as glue between the different parts and to emulate the personality and functionality of the original general-purpose operating system.

3.2.2.1 Linux Kernel Library (LKL) [83]

Linux Kernel Library or LKL was created to allow applications to reuse Linux code by porting architecture-independent parts of the Linux kernel into a userspace library that can then be linked with the application. LKL requires memory management, timers, and support for threading, interrupts, and idling from the underlying host. Glue code is also necessary to make all the different parts of LKL run. LKL does not contain architecture-specific code so that it can run on different host operating systems. Applications need to be ported to use LKL code, so only a few data points are available. An FTP server implemented in userspace using LKL shows comparable performance to an FTP server using native services. LKL does not target hardware compatibility (designed to run in userspace). LKL has remained unmaintained for years now.

3.2.2.2 Drawbridge [80]

Drawbridge aims to create a library operating system from the Windows operating system. Its goal is not performance, but security, host independence, and easy migration. Drawbridge imports parts of the Windows operating system and emulates the rest of the functionality to provide the expected Windows personality and interface to applications. A platform adaptation layer is also provided, which implements the functionality required by the library operating system and translates the library

operating system ABI to host ABI. A security monitor lives in the host, virtualizing host resources such as threading, virtual memory, filesystem, and networking support for the library operating system while maintaining a separation between the library operating system and the host. Communication with the applications running inside the library operating system happens through a remote desktop protocol. Drawbridge allows unmodified applications to run with comparable performance to native execution. It does not target hardware compatibility and is a proprietary solution, so the actual state of its support and ecosystem is unknown.

3.2.2.3 Rump Kernel [49]

Rump kernel is an effort to allow NetBSD drivers to run in userspace. The project notes that driver development can lead to an unstable system. Hence, drivers need to be developed and run in a protected environment. It introduces the idea of partial paravirtualization, hence the name *'Rump'*. It only implements the glue code required to run the drivers in userspace, and requests services like scheduling, threading, memory allocation, etc., from the host underneath. Rump has a very fluid design. Each driver can exist as a separate server, giving a microkernel personality. Or all the drivers can exist together with the application in a single server, providing a library operating system personality or any combination between these extremes. Rump allows applications to exist on a separate and/or the same host. Further, Rump allows the reimplementation of the host-specific layer so that it can run on a general-purpose OS as a process or Xen as a paravirtualized system.

Rumprun unikernel [48] is the continuation of the Rump kernel project. It allows Rump kernels to run on KVM or bare-metal. As noted earlier, Rump kernels have a host-dependent layer to interact with the host. Rumprun provides a small kernel, written from scratch, to live beneath that host-dependent layer. It is aptly named 'bare metal kernel' or bmk. Bmk provides Rump kernels with the required

bootstrapping code, threading support, interrupts, page-level memory management, etc. It only supports cooperative scheduling and does not support virtual memory or signals.

Rump allows binary compatibility and must be recompiled for changes to occur. It has been used in many other research systems. The community around Rump is mainly academic and research-focused, but significant additions to Rump have not been made in years.

3.2.2.4 LibrettOS [75]

LibrettOS combines microkernel and library operating system principles to achieve performance and intra-kernel isolation. LibrettOS builds on top of Rumprun [48] and uses Xen [21] as the hypervisor. Performance-sensitive applications are run in a library operating system fashion, with direct access to virtualized hardware through SR-IOV and IOMMU support. Other applications which are not performance sensitive or cannot be allowed direct access to hardware due to security concerns use microkernel-style servers. LibrettOS uses NetBSD drivers for servers, which run atop Rumprun in separate Xen domains. The compatibility and ecosystem of LibrettOS are the same as Rump's.

3.2.2.5 Graphene [93]

Graphene is the next step in the Drawbridge [80] and picoprocess [31] ecosystem. It notes that modern applications comprise multiple processes, but library operating systems run only a single process. So Graphene is a Linux-compatible library operating system where multiple library operating systems collaboratively implement the POSIX abstraction, yet appear to the application as a single operating system. Graphene library operating system is based on Drawbridge [80] and runs on top of a platform abstraction layer (PAL). Functionality is emulated in the library operating

system, so only a tiny number of syscalls go through the PAL to the host kernel. A security monitor secures this interface. Library operating system instances communicate and coordinate shared state through RPC. Reference monitors can sandbox different library operating system instances together, providing isolation to applications. Graphene has to be recompiled for changes to occur. It runs on top of a general-purpose OS, and unmodified applications can run on top of it after a recompilation and linking with modified libraries. There is not a huge ecosystem and community around Graphene, but it has been used in other research systems [94].

3.2.2.6 Discussion

Split-Design systems struggle with performance issues. Unlike traditional monolithic operating systems, split-Design systems operate in user mode and pay the ring transition cost for calls to the host. Further, these systems implement glue code to allow imported parts of the general-purpose operating system to function. This glue code is written for compatibility and emulation, not for performance, e.g., Drawbridge, LKL, etc. Finally, split-Design systems depend on services from the host, e.g., threading or scheduling, etc. Any functionality split-Design systems provide will always include the latency already added by the host. This poses enormous performance challenges for split-Design systems. Rump kernels are a little more nuanced. They have modes where no general-purpose host operating system is involved, i.e., Rumprun unikernel. Rumprun has a small, bare-bones kernel to act as the host. It is geared towards providing the bare minimum functionality, not necessarily performance. So performance-wise, split-Design systems lag behind general-purpose operating systems.

These systems also have a considerable implementation effort involved to emulate the functionality and personality of the original operating system. On top of that, other goals, e.g., security, host independence, etc., require still more new code to be written. Drawbridge [80] implements a security monitor which resides on the host

operating system and virtualizes resources for the library operating system. Rump kernels [49] have multiple layers, e.g., the library that provides platform abstraction or the minimal kernel layer that allows Rumprun unikernel [48] to run bare metal or on KVM, etc. Similarly, LKL [83] also requires a lot of new code to allow imported parts of the Linux kernel to be useful as a library. Split-Design systems must implement the platform abstraction layer for each underlying platform to achieve host independence.

Regarding compatibility, split-Design systems are highly compatible with existing applications. Drawbridge [80] has binary compatibility with Windows applications, primarily because it presents itself as a Windows-like operating system with the same personality, not as a library. Rump [49] allows API-level compatibility with modifications to the C library. Applications can be modified to use internal APIs for better performance. Since LKL [83] is a library, applications and other user libraries need to be modified to use its functionality instead of a host kernel's.

Split-Design systems are mostly not built for a machine interface. It is harder to secure custom interfaces like the syscall interface compared to a machine interface. Drawbridge [80] runs on top of a host operating system on a syscall interface. Its security monitor also lives on the host. In this deployment, the host becomes part of the trusted computing base and isolates multiple instances of Drawbridge. In another deployment, Drawbridge can run on the Hyper-V hypervisor and make hypercalls to the host. That is a more secure deployment, especially if hardware virtualization is available. Rump [49] can be deployed in different ways; when deployed as a process, the isolation from other processes is offloaded to the host operating system. The security guarantees are much more substantial when deployed on top of Xen, KVM, or bare metal.

Although split-Design systems inherit the codebase from general-purpose operating systems, they don't inherit the community because split-Design systems differ

significantly from their original donor operating systems. They contain new glue code for emulation and connecting different imported parts. Having little or no developer support means the new parts of their codebase are not battle-tested.

General optimizations and speed-ups are not usually available to split-Design systems because they run on a general-purpose operating system in user mode. These easy gains are possible in other deployments where a general-purpose operating system is not involved, e.g., a split-Design system running on a hypervisor or bare metal. Rump [49] allows per-application optimizations due to the modular design it shares with its donor operating system NetBSD.

Split-Design systems result from the exokernel movement [35], where a library operating system runs on an interface exposed by the exokernel. The split-Design systems require a rich interface, e.g., threading support or memory management, which is contrary to the Exokernel idea that an ideal interface is a machine interface or one that can be as low and as minimal as possible. In this sense, clean-slate systems that run in virtualization better represent the Exokernel model, with the hypervisor acting as the exokernel. Split-Design systems also suffer in performance and functionality because the donor operating systems are not designed to be split down the middle. Those monolithic operating systems have huge dependencies on the low-level architecture-dependent code and other parts of the system. It is almost impossible to cut out self-contained pieces. That is why split-Design systems must write massive amounts of glue code to get a functional system out of the imported parts. Rump [49] fares better mainly because NetBSD is already a modular operating system with fewer dependencies across the different parts. This has enabled Rump kernels to be used in other projects as well [75, 1].

Split design systems try to import kernel code to userspace and end up creating a huge implementation effort and little performance gains. UKL, on the other hand,

imports application code into the kernel and presents a cleaner design and large performance benefits.

3.2.3 Incremental Systems

These systems patch a general-purpose operating system to expose internal interfaces to applications. These systems take an incremental approach to specialization while retaining the benefits of having a general-purpose operating system.

3.2.3.1 X-containers [91]

X-containers project proposes deploying *"Single Concern"* containers in a library operating system setting on top of an exokernel which provides isolation. It uses Xen [21] as the exokernel and Linux as the library operating system. For performance, applications run at the same privilege level as the Linux library operating system. Further, the Linux library operating system provides binary compatibility and multiprocessing. Xen exokernel gives virtual machine-based isolation and ease of deployment without requiring hardware virtualization support. Despite being a general-purpose operating system, this work notes that Linux is highly customizable through build time config options and boot parameters. It creates a small and highly optimized Linux library operating system. It modifies Xen so that the Linux library operating system and its user processes run at the same privilege level. Any syscalls made are trapped by Xen and then forwarded to the library operating system to achieve binary compatibility. Further, binary rewriting allows using direct function calls instead of syscalls. X-containers show significant performance improvements compared to Docker and other container solutions for cloud workloads, e.g., Memcached. X-containers allows behavior and codebase changes at runtime by virtue of being based on Linux and offers full binary compatibility. X-Containers has evolved into a proprietary solution [14].

3.2.3.2 Kernel Mode Linux (KML) [64]

Kernel Mode Linux (KML) is an out-of-tree patch of the Linux kernel (the latest patch was for kernel version 4.0, which is more than seven years old). It was motivated by slow kernel entry/exit mechanisms, i.e., int80 and sysenter/sysexit. KML runs user applications at the kernel privilege level so that applications can make a function call to the kernel entry point. The gains of this feature have decreased over time as faster syscall and sysret instructions have been introduced. Also, KML belonged to an era of systems where single application deployments were not very common, so it justifiably talks about security concerns of running applications at the highest privilege level. It provides a typed assembly language (TAL) to statically analyze the application's instructions to ensure it is not misbehaving. KML has lived on in specialized systems lore and has been used recently again by Lupine. KML allows complete hardware and application binary compatibility by being a patch of Linux. However, it has not seen any development after Linux v 4.0 due to the original patch not making it into the mainline and a lack of community to keep working on it.

3.2.3.3 Lupine Linux [56]

Lupine Linux aims to build a unikernel-like Linux kernel through configurations and syscall overhead elimination. Lupine builds on top of Kernel Mode Linux (KML) [64]. Lupine Linux [56] notes that Linux is highly customizable despite being the giant monolith general-purpose operating system. It meticulously prunes thousands of Linux config options, choosing just over 300 options to create an extremely lightweight and optimized Linux kernel. When run on a lightweight monitor like uhyve [76] or solo5 [13], it gives memory and boot times comparable to specialized unikernels. Further, Lupine uses the KML [64] patch to eliminate the syscall overhead. Lupine shows up to 40% improvement in syscall latency, but this gets amortized to only a 4%

improvement in macrobenchmarks. Lupine further discusses how the config options can be modified to move away from a bare-bones unikernel personality that runs a single application and only runs on a single processor etc., towards a more general-purpose operating system personality. Lupine inherits all the traits of KML. It also does not have any community around it, and because KML was only developed for Linux v 4.0 at the latest, that's the Linux version Lupine supports.

3.2.3.4 Discussion

Incremental systems have some performance benefits compared to the general-purpose operating systems they are based on. X-containers [91] and Lupine [56] avoid the ring transition overhead by running applications in kernel mode and making function calls instead of syscalls. X-containers gets 27x better syscall latency than Docker for selected syscalls but does not show the results of increasing the payload for those syscalls. Lupine gets 40x better syscall latency than Linux for extremely small requests. For Lupine, these benefits exponentially decrease as the syscall payload increases.

The least amount of implementation is required in incremental systems. These systems do not aim to fundamentally modify the execution model or structure of the general-purpose operating system. The implementation primarily focuses on exposing the kernel API for function calls instead of syscalls. Lupine [56], which uses the KML [64] patch of the Linux kernel, only exposes the syscall entry points. It modifies a C library to make function calls to those entry points instead of syscalls. X-containers modifies both Linux and Xen to allow applications and Linux kernel to operate in kernel mode.

Incremental systems provide a very high degree of compatibility with the least effort. X-containers [91] is fully binary compatible with existing Linux applications because syscalls are trapped by Xen and forwarded to the guest Linux kernel. It

allows binary rewriting as well. Lupine [56] only offers API-level compatibility with modifications to the C library.

These systems are interesting because they allow multiple processes to co-exist on the same operating system. Although Lupine [56] restricts itself to a single process as a design decision, the underlying KML [64] patch allows co-running multiple processes. X-containers [91] allows multiple processes and decouples security and isolation from the process abstraction. The same idea can be applied to incremental systems in general, i.e., multiple trusting processes can be deployed together, e.g., a primary workload and helper utilities for logging and profiling, etc. This way, the isolation can be done externally, e.g., at the hypervisor level in the case of both Lupine and X-containers.

Operating systems need decades to build communities for development, support, and bug fixes. No matter how specialized or performant, new operating systems cannot match long-running projects like Linux in this regard, resulting in small or non-existent developer communities. This is one of the more significant obstacles in the widespread adoption of operating systems; production deployments choose operating systems backed by large developer communities instead of betting on a promising new operating system but almost no developer support. Lack of production deployment means the code is not as well tested as Linux.

Incremental systems, in theory, have an easier route to inheriting the community from Linux; although KML [64] and, by extension, Lupine [56], and X-containers [91] modify Linux, they don't do it as significantly as split-Design systems, e.g., LKL [83]. With the help of the Linux developer community, these patches can potentially be merged upstream, but in reality, there has been no upstream activity on these patches for years, or they were never submitted for review in the first place.

Incremental systems have the most potential for compatibility and low engineering

effort. These systems only expose the external kernel API to applications but can potentially do further specialization. These systems do not have to worry about device drivers because they inherit them from the base operating system they modify, and these systems already support a POSIX-Linux environment. It is hard for these systems to make massive design and code changes and not risk losing application compatibility. Getting performance speed-ups is more challenging for these systems because they don't link applications and the kernel together, so they can't explore deep application-specific optimizations.

3.3 Kernel Bypass

The availability of hardware, and related libraries, that allow applications direct access to low-level I/O buffers, and extremely high-speed I/O requirements of some modern workloads have led to kernel bypass research. General-purpose operating systems like Linux don't provide the performance and isolation guarantees required in some cases; key-value stores have microsecond-scale tail latency requirements, and the performance of some workloads drops due to interference by sharing resources with co-running applications. Kernel bypass techniques have addressed these issues by splitting the workloads [25, 79] into the performance-critical data plane (fast-path) and the control plane (slow-path), which includes setup, management, and clean-up operations, etc. The general-purpose operating system services the slow path, and the fast path is customized to the workload requirements for high performance through kernel bypass. Using the general-purpose operating system for non-performance critical paths eases the engineering burden and provides some level of compatibility with the existing ecosystem. The specialized performance-critical paths often require specialized hardware and libraries, or new code needs to be inserted into the general-purpose operating system, e.g., kernel modules in Linux.

Some earlier works [87, 25] argue that hardware-based kernel bypass techniques have some drawbacks, e.g., the inherent inflexibility of requiring special hardware, lack of security guarantees as isolation and protection are offloaded to hardware, and poor integration with existing applications. Works that have used commodity hardware either allow userspace network stacks direct access to NIC buffers or insert special modules into Linux that provide applications with specialized APIs and batch packet processing. Examples include Chronos [50], netmap [87], mTCP [43], Sandstorm [65], Dune [24] and Snap [67]. Similarly, UKL does not require any specialized hardware. Speed-ups provided by specialized hardware [41] are orthogonal to UKL design.

Hardware-based kernel bypass mechanisms either depend on DPDK and require modern specialized DPDK-enabled NICs or depend on RDMA and require specialized Inifiniband NICs. DPDK [2] provides userspace libraries for packet processing, bypassing the Linux network stack. Examples of works that depend on DPDK, or those which don't require it but have optional support for it, include mTCP [43], IX [25], MICA [61], ZygOS [81], Shinjuku [44], Shenango [77], TAS [51], eRPC [45], Caladan [37] and Demikernel [99], etc. RDMA-enabled specialized NICs allow kernel bypass and provide the required userspace libraries, e.g., FaRM [32], HERD [46], RAMCloud [78], FaSST [47], LITE [95], and eRPC [45], etc. A lot of previous work has used RDMA to build abstractions to improve the performance of RPCs [32, 46, 47, 95], etc.

The unpredictable and uncontrolled tail latency offered by Linux [50] has been the motivation for many research projects. These projects use Linux as a control plane and bypass its network stack for performance-critical paths or the data plane. Works that target Linux's tail latency issues include Chronos [50], MICA [61], and ZygOS [81], etc. These works have implemented their own network stacks in userspace and show significant performance benefits over Linux. UKL also takes motivation

from these research works and shows improvement in tail latency for cloud applications, i.e., Redis. Further, the deep shortcuts explored in UKL, where applications can reach into the Linux network stack and call functions directly, can be seen as a kernel bypass but within the kernel.

Previous research has also used kernel bypass mechanisms to show improvement in specific use cases. Some works have focused on TCP performance in Linux and bypassed Linux's TCP stack to show performance benefits. mTCP [43] argues that short TCP connections are becoming pervasive in cloud settings, but they suffer from high system call overhead and latency issues caused by the Linux network stack. TAS [51] further argues that assuming a data center setting can greatly simplify the TCP stack and provide performance benefits over Linux. Other works like MICA [61] and HERD [46] focus on using kernel bypass to improve the performance of key-value stores. Sandstorm [65] argues that kernel bypass mechanisms can provide further performance benefits if they are designed to optimize for application requirements. Other works like FaRM [32] and RAMCloud [78] use kernel bypass to fully benefit from the cheap and widely available DRAM to accelerate message passing and build a memory-based storage system, respectively. eRPC [45] and FaSST [47] improve RPC performance using kernel bypass. All of these systems help UKL make the point that each application and workload has its own specific optimizations, and Linux should be equipped to provide the developers with a way of integrating those application-specific optimizations.

Linux kernel schedulers cannot provide microsecond-scale latencies [44, 81]. So systems like ZygOS [81], Shinjuku [44], Shenango [77] and Caladan [37] have focused on bypassing the Linux kernel scheduler and implemented userspace schedulers with policies which allow microsecond-scale tail latency guarantees. This work is highly relevant to UKL since it provides optimizations that can be integrated into UKL to

provide interference reduction [37] and microsecond-scale tail latency guarantees [81]. A run-to-completion optimization currently explored in UKL, which allows bypassing the Linux scheduling for performance-critical paths and shows tail latency improvement, also falls in this research area.

Recently, projects like Demikernel [99] and PIX [42] aim to provide usable APIs and abstractions for hard-to-use kernel bypass mechanisms. This shows that kernel bypass research is coming the full circle, i.e., arguing that existing APIs are too slow because they abstract away the hardware, bypassing those APIs for direct hardware access, realizing that directly programming individual devices is tedious and non-scalable, providing new APIs that abstract hardware devices for ease of development. This greatly motivates UKL research because, through UKL, developers can keep using the existing APIs and gain performance benefits or explore newer APIs by calling functions deep into the kernel and bypassing the complex state machine of the Linux network stack.

3.4 Generality-Specialization Spectrum

To rectify the disconnect between application requirements and operating system design, it is helpful to take a bird's eye view of the research space and find the common theme between general-purpose operating systems and specializable systems. There exists a generality-specialization spectrum between general-purpose operating systems and specializable systems. General-purpose operating systems like Linux lie at the generality end of this spectrum where the top priority is preserving application and hardware compatibility, and the ecosystem of tools, utilities, and community, while specialized systems lie at the other end where the top priority is to optimize performance.

At the general-purpose end of this spectrum, properties such as application com-

patibility and hardware support are the main priority. Other features are by-products of this compatibility, e.g., the interdependence of application and shared libraries is only possible if there is some standardized API, e.g., the Linux API. Even software compiled earlier can run on newer systems because Linux ensures application binary interface or ABI compatibility. A massive list of devices is supported, and hardware vendors can distribute compiled binaries of userspace libraries and kernel modules, and they just work without the need for recompilation. The vast community of application programmers and Linux kernel development community has been growing over the years. They can provide support, reason about bugs, and add new features because they agree on the standards of compatibility. After compatibility, performance is the second priority. A lot of effort is spent on making features perform better, but never at the expense of breaking API or ABI compatibility. Linus Torvalds pressed on this issue when he said, "If a change results in user programs breaking, it's a bug in the kernel." [11]

A closer look at general-purpose operating systems like Linux will show that it is a specializable system in itself, albeit the range of specialization is limited. Linux allows loadable modules, custom code injection into the kernel through eBPF, and compile-time and runtime configurability. But since it is a general-purpose operating system designed to concurrently support multiple applications that might have conflicting goals, it cannot be entirely specialized for a single application. Linux can be imagined as occupying a small range on the generality-specialization spectrum, lying at the generality end but able to oscillate a little, and provide some specialization when fully pushed towards the specialization end of its range.

If one starts moving on this spectrum towards the specialization end, other systems begin to appear which are also occupying small portions of this spectrum, e.g., systems that harvest architecture-independent code from the operating systems, e.g.,

LKL [83]. LKL packages some Linux kernel code as a library to which applications can link. The final binary still executes in userspace, as it requires all the services, e.g., threading, timekeeping, and memory management, from the underlying general-purpose operating system. Some operations serviced by the LKL library might be faster since the system call overhead is avoided. Linux provides the environment for this to execute, but one cannot expect the Linux community to update the LKL library every time a new kernel version comes out because the LKL library provides a non-standard function call API instead of a system call API promised by Linux.

Also on this spectrum are systems like KML [64] and Lupine [56] etc. These systems break the compatibility by executing applications at the kernel privilege level through a non-standard function call API. Otherwise, these systems try to preserve the rest of the compatibility of Linux. As a result, Linux applications are compatible with these systems, but there are no security guarantees anymore, and the Linux kernel development community will not provide support to ensure isolation from non-trusting applications which execute at higher privilege. These systems trade off some compatibility for some performance gains.

Kernel bypass frameworks [99] are also a great example of workload-specific optimization. The library operating systems that exploit kernel bypass mechanisms require Linux to set up the device and execute the application in userspace. Once that is done, the application, programmed to a non-standard API, accesses the device directly. Linux does not support this mode of operation as it breaks compatibility, and the Linux kernel development community does not provide support for this. Kernel bypass systems trade-off a lot of compatibility, and if applications can afford this loss of compatibility, the performance gains are incredibly high.

Further along this spectrum are systems that trade off more and more of the compatibility of Linux to gain extremely high performance. Clean slate unikernels,

which lie on the performance end of this spectrum, do precisely that. Some systems provide full or almost full Linux ABI compatibilty [52, 76] so Linux applications can run. However, if support is required, these systems have a very small or nearly non-existent community. Further, some systems, e.g., Mirage [63] and EbbRT [88], don't even provide API compatibility, and applications must be rewritten for them. Significant performance advantages over Linux replace the loss of compatibility in these systems. Clean-slate systems occupy a small range on the spectrum, lying at the specialization end.

Given the generality-specialization spectrum framing of this research space, all the systems discussed fall into small regions on this spectrum because they were not originally designed with this spectrum in mind. The primary motivation for many of these systems was to get performance advantages over Linux. In hindsight, their adoption has been limited because they lack application and hardware compatibility and the vast ecosystem of general-purpose operating systems.

While some systems explicitly target just one class of workload [55, 66], many envision covering a broad set of applications and being specialized to any target application. Even for systems that have gone past a research prototype [52, 54] used by just the researchers that developed it, they have ended up occupying just a small range on the generality-specialization spectrum, suited to solving one task for specific applications and hardware. Preferably, an operating system needs to occupy a broad range; ideally, the entire spectrum, i.e., support many applications and be able to specialize to them. Specializable systems are in a catch-22 situation, they have to be successful before they can build the community, which adds features needed to occupy a broad part of the spectrum, but without already occupying a broad part of the spectrum, they can't build the community.

This generality-specialziation spectrum based frmaing of the problem enables us

to compile a list of goals for this research and then design the system to achieve those goals; it should ideally start at the general-purpose end of this spectrum and naturally inherit application and hardware compatibilty, ecosystem, and large community and, according to the application's needs, move along this specialization to optimize for performance. Along the way to specialization, as optimizations are added, the system will become specialized to a smaller set of applications and hardware, but, hopefully, retain the community. Unlike previous systems, where the designers of those systems decided which features of the general purpose operating systems were essential for applications and thus needed to be kept and what were not, application developers should make this decision according to their individual needs. Instead of either fully adopting a new system or sticking with Linux, the application developers should pick and choose the unique optimizations they prefer and apply them on top of Linux. The design of UKL is an attempt to achieve these goals.

Chapter 4

Architecture

In § 4.1, we describe UKL's goals and how they impact UKL's architecture. Section 4.2.1 describes the design of the UKL *base model*, which sits at the general purpose end of the generality-specialization spectrum. The optimizations that can be applied on top of this model to move the system towards the specialization end are discussed in § 4.2.2.

4.1 Goals

UKL explores the generality-specialization spectrum between Linux and specializable systems. The goal is to start at the generality end with Linux and its support for a broad class of applications and hardware and move towards the specialization end of the spectrum by adding optimizations. As we move along this spectrum, we go towards a highly optimized unikernel which may be specialized down to the specifics of a single application and platform. In doing this, we aim to enable optimizations (§ 4.1.1) demonstrated by earlier systems while preserving an application execution environment's broad application support (§ 4.1.2), broad hardware support (§ 4.1.3), and the ecosystem of developers, tools, and operators (§ 4.1.4). We describe each of these four goals.

4.1.1 Optimizations

At the generality end of the spectrum (§ 3.4), general-purpose operating systems like Linux are limited in their range of optimizations. This is because, unlike many specialized systems (§ 3.2.1), they do not link the application and kernel code together and execute them in different address spaces. This limits them to optimizations like loadable modules, inserting application code into the kernel through eBPF [3], and compile-time and run-time configurability. Even incrementally specialized systems like X-containers (§ 3.2.3.1) and Lupine Linux (§ 3.2.3.3) only leverage executing applications in kernel privilege mode and the compile time configurability of Linux to produce lightweight binaries. UKL needs to remove the separation between application and kernel code to explore further optimizations.

In order to be able to move across the generality-specialization spectrum (§ 3.4), starting with a general-purpose operating system and adding optimizations to move towards a highly specialized system, it is necessary to link application and kernel code together and execute in a single address space. Through this design, UKL will be able to explore optimizations that previous specialized systems have adopted, like the following:

- avoiding ring transition overheads (§§ 3.2.1.4 and 3.2.1.5)
- exploiting the shared address space to pass pointers rather than copying data (§ 3.2.1.2)
- exploiting fine-grained control over scheduling decisions, e.g., deferring preemption in latency-sensitive routines (§ 3.2.1.9)
- enabling interrupts to be efficiently dispatched to the application code (§ 3.2.1.2)
- exploiting knowledge of the application to remove code that is never used (§ 3.2.1.1)

- employing kernel-level mechanisms to optimize locking and memory management (§ 3.2.1.9), for instance, by using Read-Copy-Update (RCU) [68], per-processor memory (§ 3.1.9), and DMA-aided data movement (§ 3.3)
- enabling compiler (§ 3.2.1.2), link-time, and profile-driven optimizations between the application and kernel code
- exploring novel code paths for performance-critical parts of the application (§§ 3.2.1.5 and 3.2.1.9) and direct hardware access to applications for high-speed I/O (§ 3.3).

Ultimately our goal with UKL is to explore the full spectrum between Linux and specialized systems. For this thesis, our goal is to enable applications to be linked into the Linux kernel, explore at least some of the above optimizations and measure the performance improvements that can be achieved, first in the context of an unmodified application source plus a recompilation and link step, then by modest changes to the application and general-purpose system.

4.1.2 Application Support

One of the fundamental problems with specializable systems is the limited set of applications that they support. Only executing a single process excludes any application that requires helper processes, scripts, etc. Moreover, moving away from backward application compatibility and standard APIs typically requires substantial porting effort for any application and libraries the application uses.

UKL seeks to enable optimizations while remaining broadly applicable. Our goal is to allow any unmodified Linux application and library to use UKL with a recompilation, as long as only one application needs to be linked into the kernel. Depending on the application, the developer can gradually move toward the performance end, i.e., incrementally enable optimizations. 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 specialization end.

4.1.3 Hardware Support

Another fundamental problem with specializable systems is the lack of support for physical machines and devices. While recent research has mostly focused on virtual systems, some recent [88, 48] and previous [70, 30, 35, 59, 18] systems have demonstrated the value of per-application specialized operating systems on physical machines. Unfortunately, even these systems were limited to specific hardware platforms with a restricted set of device drivers. The lack of hardware support 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 an environment capable of supporting the complete hardware compatibility list (HCL) of Linux, allowing applications to exploit any hardware (e.g., GPUs, TPUs, FPGAs) enabled in Linux. Although the implementation efforts in this research are focused only on Intel x86_64 architecture, the lessons learned are generalizable across other platforms as well.

4.1.4 Ecosystem

While application and hardware support are typically considered the fundamental barriers to moving along the spectrum toward the specialization end, the problem is much larger. Linux has a vast community of developers and 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. Most of all, the base changes required 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.

As optimizations are added to the system, the ecosystem support will understandably decrease, e.g., if an application developer implements a highly specialized path inside the kernel, it might not be accepted upstream due to lack of broad applicability, or if the application requires no other process execute alongside it, the tools and utilities cannot be run.

4.2 Design

As stated above, at the general-purpose end of the spectrum, UKL aims to support all applications (§ 4.1.2), hardware (§ 4.1.3), and ecosystem (§ 4.1.4) of Linux. At this end, UKL is in what we call its ***base model***. Importantly, the base model preserves the (known and unknown) invariants and assumptions of applications and Linux, except syscalls are replaced by function calls, and application code now executes in supervisor mode in the kernel address space. From this starting point, an expert programmer can adopt specific optimizations, explored by specializable systems, that are valuable and safe for their particular application by choosing (additional) configuration options and/or modifying the application to invoke kernel functionality directly, thus moving towards the performance or unikernel end of the spectrum. Section 4.2.1 describes the base model, and Section 4.2.2 describes various optimizations we have

explored that specialized the system for some applications.

4.2.1 Base Model

UKL base model can be best understood as a standard Linux kernel, except one application is linked with the kernel and executes at the kernel privilege level. That particular application does not experience any difference than if it were running on unmodified Linux, except a very slight speed up due to the replacement of `syscall/sysret` instructions with `call/ret`. Other applications running on UKL run precisely as they would on unmodified Linux, and the UKL base model supports all the HCL of unmodified Linux. Below we describe the design of the UKL base model.

4.2.1.1 Support for multiple processes

One key area where UKL differs from other unikernels is that, while only one application can be linked into the kernel, UKL supports a full userspace and enables other applications to run unmodified on top of the kernel. Support for multiple processes is critical to run many applications that are logically composed of various processes (§ 4.1.2), standard configuration and initialization scripts for device bring-up (§ 4.1.3), and the tooling used for operations, debugging and testing (§ 4.1.4). UKL supports multithreaded applications.

It is important to note that while UKL supports multiple processes, other processes are not protected from the performance-optimized one linked into the kernel. Similar to unikernels, our security model assumes other workloads that need to be protected are isolated by a hypervisor or by techniques for securing bare metal machines [71, 16].

4.2.1.2 Address space layout

UKL preserves the standard Linux virtual address space split between the 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 allocated in the kernel portion of the virtual address space.

We found it necessary to adapt this address space layout because Linux performs a check to see if an address being accessed is pinned or not; modifying this layout would have resulted in changes that may have been difficult to upstream (§ 4.1.4). Unfortunately, this layout has two negative implications for application compatibility. First, (see § 5.2) 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 now pinned.

4.2.1.3 Execution models

Even though the application and kernel are linked together, UKL differs from other 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 source modification (§ 4.1.2).

On the other hand, Kernel code 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 may prohibit acceptance by

the community (§ 4.1.4).

On the 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` flag to identify the current mode of the thread so that subsequent interrupts, faults, and exceptions will go through regular transition code when resuming interrupted application code, minimizing affecting kernel invariants in the base model(§ 4.1.4).

4.2.2 Optimizations

While preserving existing execution environments enables most applications to run with no source 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 optimizations, that have been explored by specializable systems, are possible. First, a developer can apply configuration options that may improve performance. Second, a knowledgeable developer can improve performance by modifying the application to call internal kernel routines and violating, in a controlled fashion, the normal assumptions and invariants of kernel versus application code. Expertise is needed to perform these customizations. For example, the kernel will fail 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.

In § 4.2.2.1, we discuss the configuration options we have explored so far, that don't require application modification. In § 4.2.2.2, we discuss the optimizations that

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 4.1: UKL Configuration options

require applications to be modified but provide larger performance benefits.

4.2.2.1 Configuration Options

Once an application is running, a developer can easily explore a number of configuration options that, while not safe for **all** applications, may be safe and offer performance advantages for their application.

Bypassing entry/exit code: On Linux, on transitions between application and kernel environments through system calls, interrupts, and exceptions, some entry and exit code is executed; this 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 environments. As we will see, this model results in significant performance gains for applications that make many small kernel requests.

Avoiding stack switches: Linux runs applications on dynamically sized user stacks, and kernel code on fixed-sized, pinned kernel stacks. Every time kernel functionality is invoked, this stack switch breaks the compiler's view and limits cross-layer

optimizations, e.g., link-time optimizations [1]. The developer can select between two UKL configurations that avoid stack switching (UKL_NSS and UKL_NSS_PS); UKL_NSS skips the jump to kernel stacks when transitioning from application to the kernel execution environment and allows applications that need large stacks to benefit from this option. However, this prevents normal user space applications (e.g., tools, utilities, debuggers) to co-run with the UKL application (see § 5.10 for details). UKL_NSS_PS solves this limitation by skipping the jump to kernel stacks and allocating user stacks in the pinned kernel portion of the address space. These are non-extensible stacks, so applications that need to extend stacks dynamically cannot use this option.

ret versus iret: Linux uses `iret` when returning from interrupts, faults and exceptions. `iret` is an expensive instruction that automatically changes the privilege level, instruction pointer, stack pointer, etc. The UKL_RET configuration option switches from kernel to application stack and uses `ret` instead of `iret` to return to application code. To ensure that the system never lands in an undefined state, interrupts are enabled only after returning to the application stack and code.

4.2.2.2 Application Modifications

Along with the above configurations, developers can modify the applications to 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 regular application routines. Using such memory results in less overhead because, unlike user memory, all of its pages are pinned and thus avoid subsequent faults when kernel interfaces have to copy data to and from that memory.

[1] today LTO in Linux is only possible with CLANG while `glibc` can only be compiled with gcc. Efforts are underway in the community to enable `glibc` to be compiled with CLANG and to enable Linux LTO with gcc. We are excited to explore the advantages of LTO as soon as one of these external efforts completes.

Developers can explore deeper optimizations by taking advantage of application knowledge. For example, they may be able to assert that only one 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.

The UKL base model ensures that the application and kernel execution environments stay separate, with proper transitions between the two. But applications may find it beneficial to adopt some features, even for short times, of the kernel execution environment like non-preemption. Applications can toggle a per-thread flag which switches on features of the kernel execution environment, allowing application threads to be treated as kernel threads in specific cases, e.g., so they won't be preempted. This can be used as a 'run-to-completion' mode where performance-critical paths of the application can avoid perturbation.

Chapter 5

Implementation

This chapter talks about UKL's implementation details, i.e., the overview of changes to Linux and `glibc` (§ 5.1), linking an application and its required libraries into the Linux kernel and building a bootable binary (§ 5.2), boot-up and initialization process (§ 5.3), enabling the UKL base model, i.e., ensuring that applications run without modifications (§§ 5.4 to 5.6), process creation (§§ 5.7 and 5.8), configuration based optimizations (§§ 5.9 to 5.12), and optimizations which involve application modification (§ 5.13).

5.1 Changes to Linux and `glibc`

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 in table 4.1) is 1250 lines. The vast majority of these changes are target-specific, i.e., in the x86 architecture directory. A newer version of the UKL base model patch, i.e., for Linux kernel version 6.0, has been shared with the Linux community [62]. This chapter, however, discusses the changes to Linux kernel version 5.14, which was used to evaluate UKL (chapter 6).

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 UKL changes are wrapped in macros which can be turned on or off through kernel and `glibc` build time config options; they are

compiled out when Linux and `glibc` are configured for a different target.

We found that the UKL patch can be small due to many favorable design decisions by the Linux community. For instance, Linux's low-level transition code has recently undergone massive changes to reduce assembly code and rewrite functionality in C. Further, the remianing assembly code has been refactored to removr code duplication. This has allowed UKL changes to also be localized to the newly refactored code. Further, the ABI for application threads dedicates a register (`fs`) to point to thread-local storage. In contrast, kernel threads have no such concept but instead dedicate a register (`gs`) to point to processor-specific memory. If a register were 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 5,439 lines of code are added or changed in `glibc`. This 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 sub-directory and then modified. The actual number of lines changed in `glibc` is 1,720. All the UKL changes are well contained in a separate directory. The `glibc` build process, configured for UKL, first searches the UKL specific directory for a target file at build time before searching the default location.

The different configurations of UKL (Table table 4.1) involve changes to the transitions between application and kernel code. All changes were made through Linux `SYSCALL_DEFINE` and `glibc INLINE_SYSCALL` macros. This ensured that the changes were limited to header files, keeping the LoC changed to a minimum.

5.2 Building UKL

Our current implementation does not support dynamically loaded libraries, requiring the application code and associated user libraries to be compiled and statically linked

with the kernel. This code is built with two special flags. The first flag disables the red zone optimization (`-mno-red-zone`) as is standard when building kernel code. Red zone optimization allows the compiler to use the stack without decrementing the stack pointer in leaf functions, i.e., functions that do not call any other function. This optimization works for applications running in user mode because any interrupt, exception, or fault will trigger a privilege change and a switch to a dedicated kernel stack, leaving the user stack intact. This optimization cannot be used for an application running in kernel privilege level because, since there is no privilege change, the switch to a dedicated stack does not happen and the interrupt or fault servicing code uses the same, i.e., user stack. This means that any data on the stack under the stack pointer will be garbled, leading the application to break once control returns to it. The second kernel memory model flag (`-mcmodel=kernel`) enables the generated code to be loaded into the highest 2GB of address space instead of the lower 2GB that is the default for user code.

While a limitation of our current implementation, that attempts to minimize changes to Linux, we do believe future work could enable binary compatibility for non performance-critical parts of application code and user libraries. The kernel already supports dynamic linking, and extending the dynamic linker to support application libraries seems feasible. Previous work [76] has shown how all faults, interrupts, and exceptions can be made to use dedicated stacks through the Intel interrupt stack table (IST); such an approach would eliminate the need to disable red zones. Finally, an new research direction would be to only link performance-critical parts of the application and user libraries with the kernel, and ensure binary compatibility for all other application code and user libraries by loading them in the user part of address space and using binary re-writing to resolve addresses.

After the application and libraries are compiled, a modified kernel build system

combines them and the kernel into a final `vmlinux` binary which can boot bare-metal or virtual. Before linking the application and kernel together, all application symbols (including library ones) are prefixed with `ukl_` to avoid name collisions using the `objcopy` utility. 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 and C++ constructors and destructors work. Appropriate changes to the kernel loader are also made to load the new ELF sections along with the kernel.

5.3 Boot up and Initialization

UKL ensures that all Linux's boot-up and initialization code is executed without modification. This allows UKL to inherit Linux's hardware and application support; Linux's early boot-up code does hardware discovery and device set-up. Once the system boots up, the user space is initialized as it would in unmodified Linux. This provides the expected execution environment for `glibc` and other libraries and applications to run without modifications.

There are, however, some modifications to enable UKL applications to run properly. Even though we added the thread local storage (TLS) section to kernel's vmlinux binary, through the changes to the linker script described above (§ 5.3), the kernel loader is still oblivious to this change. We modify the Linux loader so that, as Linux boots and decompresses its binary, the loader loads all the sections at their correct places in memory, including the PT_TLS or thread local storage section.

5.4 Environment Tracking

On unmodified Linux, on every switch between application and kernel execution environments (system calls, interrupts, or exceptions), entry and exit code is executed.

Linux tracks whether a process is running in the application or kernel execution environment through the `CS` register (which actually tracks the hardware privilege level). UKL-optimized applications always run at the kernel privilege level, so the `CS` register never changes and cannot be used to track the execution environment. UKL tracks the execution environment with a flag (`ukl_mode`) added to the kernel's thread control block, i.e., `task_struct`.

5.5 Transition between execution environments

If a UKL application makes a system call, it uses a procedure call rather than a `syscall` instruction. We modify `glibc`'s `INLINE_SYSCALL` macro to replace `syscall` instruction with a `call` instruction to the UKL system call entry point, `ukl_entry_SYSCALL_64`. This entry point is similar to Linux kernel's unmodified system call entry point, i.e., `entry_SYSCALL_64`, except in the handling the difference between `call` and `syscall` instructions. A `syscall` does not modify the application stack, but `call` pushes the return address to the application stack. Also, `syscall` puts the return address and flags into specific registers, while `call` does not. UKL's system call entry point `ukl_entry_SYSCALL_64` fixes this discrepancy, i.e., puts the return address into `RCX` register and flags into `R11` register. This is important because the following code saves the entire register state on the kernel stack (and this state is used later on by kernel code, e.g., all the arguments to the system call are contained in it). If this register state is incorrect, the kernel code can break unpredictably.

The UKL entry point also differs from Linux's in how the execution environment is tracked, as discussed in § 5.4. UKL toggles the `ukl_mode` flag to kernel value, so that kernel entry code is not executed on any subsequent interrupts (because the system call already switched the execution environment to kernel) until the system call returns.

On the return path, `ukl_entry_SYSCALL_64` switches the `ukl_mode` flag back to application value and executes a `ret` instruction instead of a `sysret` or `iret` instruction. The return address pushed on the applications stack by `call` will now be popped by the `ret` so that the stack is intact when control returns to the application. Details of how we keep the interrupts disabled on the return path can be found in § 5.9.

5.6 Permissions Checking

The Linux kernel normally has several checks to ensure that applications and kernel threads behave as expected. UKL defies many of the assumptions regarding permissions, so these checks need to be made UKL-aware as well. As an example, application threads are not allowed to access kernel memory. But UKL threads need to access memory in the kernel portion of the address space since their text and data sections reside there due to being linked with the kernel. In another example, kernel threads are not meant to suffer page faults in the application's part of the address space. Since UKL always executes at the kernel privilege level, it is considered a kernel thread for these checks. Since UKL threads have their heaps, stacks, and memory-mapped areas in the application's part of the address space, page faults are bound to occur. For these scenarios, we modify the header files where the permission-checking functions and macros are defined. Here we add checks for the `ukl_mode` flag (a non-zero value means it is a UKL thread) so that in case of a UKL thread, the correct permissions are returned. For example, permission is granted for the UKL thread to access the kernel's part of the address space. In all the places where permissions are checked, these inline functions and macros are called, so making changes here allows us to keep the overall patch size small.

5.7 Changes to `execve`

User-space processes running on UKL can `exec` as expected. The UKL process is started by invoking `exec` with a program name specified by a configuration option. While most of `execve` is unmodified, we skip loading the (nonexistent) binary and jump directly to the `glibc` entry point. `glibc` initialization happens almost as expected, but when initializing thread-local storage, changes had to be made to read symbols set by the kernel linker script instead of an ELF binary. C and C++ constructors run the same way as in a normal process. Command-line parameters to `main` are extracted from a part of the Linux kernel command line.

Details about `fork` and `clone` in UKL, which would create another UKL process or thread, respectively, can be found in § 5.8. While we have not yet done so, we believe that, once UKL calls `fork`, only modest effort is required to enable the new UKL process to change into a non-UKL process through `exec`, and then execute some normal user-space binary. That would entail enabling the `exec` code in the kernel to, based on some UKL flag, set the `ukl_mode` flag to zero. This would mean that the process is now a non-UKL process, and the rest of the `exec` code will treat it as such. It will be able to run a normal userspace binary because UKL changes don't modify `execve` handling of non-UKL processes.

5.8 Fork and Clone

Non-UKL user-space processes running on UKL can fork as expected. UKL processes can also call `fork`, or `clone` with specific flags so that it acts as `fork`, but there are some caveats. Process creation, or `fork`, only makes copies of the user part of the address space (or copy-on-write to be exact). The kernel page tables are not duplicated because they don't need to be. For UKL processes, the application text and data sections are present in the kernel part of the address space (due to the

application being linked with the kernel). This means that when a UKL process forks, the text and data sections are not duplicated, and child and parent UKL processes share them. Text section can be shared between multiple UKL processes, but sharing the data section can lead to an undefined state. So `fork` may not function as expected, depending on the application.

Ignoring this issue, we have enabled this *"fork"* of a UKL process, and it works for many applications. A change needs to be made in the page-fault handling code. UKL process never faults on instructions because its text section is always pinned due to being in the kernel part of the address space. Also, the parent does not fault on the "virtual dynamic shared object" or VDSO pages, which are located in the user part of the address space, because `execve` code faults that in before handing control off to the UKL application. The child, however, needs to fault those VDSO pages in when it tries to access them. This fires a page fault, which fails because, normally, a process executing in kernel mode (which UKL always does) should not be accessing text pages in the user part of the address space. The fault handling code needs to be made UKL-aware so that it allows UKL to fault in VDSO pages.

We have used this new `fork` to run multiple UKL processes for the LEBench microbenchmark [86], which, we have found, does not rely on the data section. However, this new `fork` will not work for real-world applications which rely on the data section and use both `fork` and `exec`. Although we have not explored it yet, one way of enabling full `fork` support would be to borrow the idea from `glibc`'s handling of the thread-local storage (TLS) [33]. This would mean keeping the original data section that lives in the kernel part of the address space intact, and for every process, including the first one, copy it into that process's user part of the address space and use that copy.

As far as using `clone` to create new threads is concerned it fully works with UKL.

To create UKL threads, the user-space pthread library calls `pthread_create`, which further calls `clone`. We modified the pthreads library to pass a new flag `CLONE_UKL` to ensure the correct initial register state is copied into the new task. This is important because if shared stacks are enabled (§ 5.10), the register state would be found on the user stack, and if not, it will be found on the kernel stack. We have run multi-threaded real-world applications like Memcached without any problem.

5.9 Kernel to Application Transition with `UKL_RET`

In UKL, returning from system calls always uses `ret` instruction instead of `sysret` or `iret` instructions. By turning on the `UKL_RET` configuration option, this becomes true for all transitions from kernel code to application code, e.g., on returning from interrupts, exceptions, and faults. In doing so, UKL must ensure that the system does not land in an undefined state, in case of an interrupt. The `sysret` or `iret` instructions perform a number of operations atomically; updating the instruction pointer and stack pointer, restoring flags (which restarts interrupts), and changing privilege level. The atomicity of these instructions ensures that when interrupts are enabled, the control has cleanly returned from the kernel to the application. The `ret` instruction only pops the return address from the stack and updates the instruction pointer. Additional instructions are required to restore the flags (which enables interrupts) and change the stack pointer. If an interrupt arrives before all of these instructions are executed, the system can be in an undefined state. UKL ensures that interrupts are not enabled before all of these instructions are executed. UKL prepares the user stack for the return path earlier when going from application to kernel code. The `call` instruction puts the return address on the user stack. Once in the system call entry point (see 5.5), UKL pushes flags onto the user stack as well. On the return path, it first updates the stack pointer from the kernel to the user

stack and then restores flags (which restarts interrupts). Taking interrupts here is safe because the instruction pointer was already correct, and now the stack pointer is also correct. At this point, the `ret` instruction can return control back to the application.

5.10 Shared Stacks with `UKL_NSS` and `UKL_NSS_PS`

In the UKL base model, a switch between the user and kernel stack occurs when code transitions between the two domains, limiting cross-layer compiler optimization. We have developed two configurations to avoid these stack switches.

With the `UKL_NSS` configuration option, UKL avoids switching from application to kernel stack in the system call entry point (see § 5.5), which results in the kernel also using the application stack. Similarly, the reverse switch from kernel to application stack on the return path is unnecessary and thus avoided. However, this configuration option restricts other non-UKL applications to only run before or after the UKL application, but not concurrently with it, because doing so can result in page faults that cannot be handled. To understand this, consider the following example; in the case of an inter-processor interrupt, e.g., for a TLB invalidation, before interrupting the other processor, Linux allocates some data structures on the current process's kernel stack. These data structures are meant for the remote processor to access, but in the case of `UKL_NSS` configuration option, the current process's kernel stack is actually its user stack, which is naturally not mapped in the page tables of any other process. When the remote processor is interrupted, it inherits the page tables of the process running on it at that time. It would try to access the data structures, but since the user stack of the original process is not mapped in its page tables, it would result in a page fault that cannot be handled. When only the UKL process is running on the system, this won't be a problem because UKL shares the kernel's page tables,

and the remote processor would be running an idle kernel thread or another UKL thread. The page tables are shared in both cases, and page faults won't occur.

This restriction is solved by the `UKL_NSS_PS` configuration option, which allocates fixed-sized stacks in the kernel part of the address space, and shares those between the kernel and application. This means that every processor, when interrupted, can access those shared stacks used by the interrupting processor. We currently use a fixed size for these pinned stacks (the largest size `glibc` allows application stacks to grow to), but this size can be configurable in the future. This configuration allows multiple processes to run concurrently but is impractical for applications that create a large number of threads that use huge stacks.

5.11 Page Faults with `UKL_PF_DF` and `UKL_PF_SS`

Normally, if a user stack suffers a page fault, hardware switches to the kernel stack and pushes a frame on that kernel stack. The switch from user to kernel stack only happens if a privilege switch from user to kernel mode also occurs. But UKL threads always execute in kernel mode, so when the user stack suffers a page fault, no privilege switch occurs, and consequently, no switch to a pinned kernel stack happens. When the hardware tries to push state on the already faulted user stack, it faults again, generating a double fault. A stack switch to a new pinned special stack always occurs in case of a double fault through the Interrupt Stack Table (IST) mechanism. The double fault prints a panic message, call stack, and register values to help with debugging, and the system then panics.

UKL addresses this stack page fault issue in two ways; with `UKL_PF_DF` and `UKL_PF_SS` configuration options. If `UKL_PF_DF` is used, it is assumed that the only reason to land in the double fault handler is through the stack page fault. Since the double fault uses a new pinned stack, it can be used to handle the original page fault as

well. So a jump is made to the page fault handler, which extends the user stack and returns the control to the application as if no double fault ever occurred. The `UKL_PF_SS` configuration option fixes this issue by using the Interrupt Stack Table (IST) mechanism to create a new stack for page faults. The page fault entry in the Interrupt Descriptor Table (IDT) is then updated so that every page fault is serviced on the new stack, regardless if it is a stack-based page fault or otherwise or if it occurred to a UKL process or non-UKL process. The jump to this new stack also happens if the execution was already in kernel mode, i.e., no privilege switch occurred. This way, as soon as any page fault occurs, the hardware switches to the new stack, pushes the state on it and then handles the page fault.

The `UKL_NSS` configuration option creates a new complication for the page fault handler because if `UKL_NSS` configuration option is turned on and a page fault occurs, a deadlock can occur. For example, if a thread is executing some memory management function, e.g., `mmap`, and needs to modify the page tables, it has to take the lock on the memory control struct (`mm_struct`). While it has this lock, if a page fault occurs, the page fault handler will run. This handler will again try to take the lock on `mm_struct` and fail because the lock is already taken (by the same thread earlier!). Unable to take the lock, the thread will wait, resulting in a deadlock. To address this problem, when a UKL thread or process is created, UKL saves a reference to the user stack virtual memory area or VMA. In case of any page fault, the faulting address is first compared to the saved user stack VMA to check if the page fault is a user stack page fault. If so, UKL skips trying to retake the lock and handles the fault. If the faulting address does not belong to the user stack, normal page fault handling occurs, i.e., try to take the lock and then handle the fault.

5.12 Bypassing Application-Kernel Transitions with `UKL_BYP`

When the transition happens from application code to kernel code, some entry code is executed, including stack switch, saving register state, and RCU handling. Some exit code is executed on the return path, including a stack switch, restoring register state, RCU handling, and calls to the scheduler, signal handler, etc., based on flags (see §§ 4.2.2.1 and 5.5. Since this entry and exit code takes time, especially if the scheduler is called or signals are handled, bypassing it results in a performance advantage. To enable bypassing the entry and exit code, UKL has `UKL_BYP` configuration option. We modify the Linux `SYSCALL_DEFINE` macro to add a stub that simply calls the actual system call function. We also modify `glibc INLINE_SYSCALL` macro so that if the `UKL_BYP` configuration option is turned on, `glibc` calls the stub in Linux source instead of going through the normal path and calling UKL's system call entry point (see § 5.5). This allows the entry and exit code to be bypassed, and control goes from `glibc` directly to the Linux functions and back.

The entry and exit code that is bypassed must be run at some point; otherwise, the system can break, e.g., if the scheduler is never called, a thread would never give up the CPU and result in a hung multi-threaded application, or an application that depends on signals might also break because signals are not handled. To address this, we further modify `glibc INLINE_SYSCALL` macro so that it only bypasses the entry and exit code for a certain number of transitions before automatically going through the regular route once. This number is also configurable. UKL also ensures that the entry and exit code is bypassed on interrupts and faults as well to ensure correct behavior.

5.13 Optimizations based on Application Modifications

In addition to the configuration options (See table 4.1) for performance improvement, UKL also allows application modification for further gains (See § 4.2.2.2). Developers can modify their applications to call internal kernel routines directly, e.g., `vmalloc`. To ensure this happens safely, the execution environment needs to be tracked carefully (See § 5.4). Environment tracking is easy at pre-defined points, e.g., system calls, interrupts or faults, etc., but it is not tracked if the application starts calling kernel functions directly. This would result in kernel functions being executed in the application execution environment and can result in unforeseen issues, e.g., it can be problematic if an interrupt occurs while kernel code is running in the application execution environment and the scheduler is called when exiting the interrupt handler. This is because the execution could be that part of the kernel that cannot be preempted and could take the system to an undefined state. To address this, UKL provides applications with two functions, i.e., `enter_user` and `exit_user`. If applications call any kernel functionality after calling `exit_user` first, and once they return from the kernel routine, they call `enter_user`, they are guaranteed to be in the safe state. These functions toggle the `ukl_mode` flag and changes are made to entry and exit functions to check this flag before executing.

5.14 Optimizations based on Application and Kernel Co-modifications

In addition to allowing developers to modify applications to call existing kernel functions directly, UKL also enables developers to implement custom code paths inside the kernel and then modify the application to call those code paths. To ensure this happens safely, the same entry and exit functions are used as described in § 5.13. To illustrate this optimization, consider the following example.

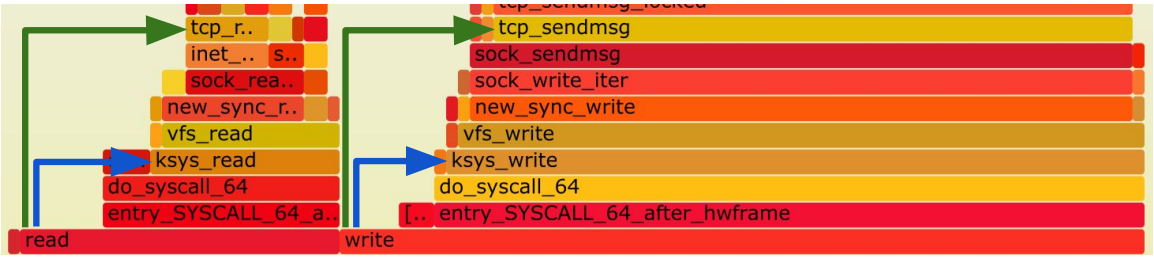

Figure 5·1: Part of a flame graph generated after profiling Redis with UKL base model with `perf` [9]. The `read` and `write` functions at the bottom reside in Redis code. Blue arrows show the code bypassed in UKL_BYP, and green arrows show deeper shortcuts, i.e., bypassing the Linux VFS [8] layer and calling TCP functions directly from Redis code.

While running Redis with UKL (§ 6.5), we wanted to understand if further specialization is possible, on top of the configurations. We built Redis with the UKL base model, deployed it bare metal, and profiled it using the Linux `perf` [9] utility. Figure 5·1 shows part of a flame graph [40] we generated from the `perf` output. The blue arrows in fig. 5·1 show how `UKL_BYP` configuration option bypasses Linux entry and exit code and invokes the system call routines directly, i.e., `read` to `ksysread` and `write` to `ksyswrite`. Figure 5·1 further shows that both `read` and `write` paths have to go through the Virtual File System (VFS) [8] layer before reaching the TCP functionality. VFS layer, which is *"an abstraction within the kernel which allows different filesystem implementations to coexist"* [8], allows applications to use generic system calls like `read` and `write`, and the kernel, based on the file descriptor, invokes the appropriate functions. We can see that there is significant overhead due to the polymorphism, and Redis, for specific read and writes, always invokes TCP functionality. An obvious specializaiton then was to introduce special purpose code that can bypass the VFS layer and invoke the underlying TCP functionality directly. Green arrows on fig. 5·1 show our intended shortcuts, i.e., `read` to `tcp_recvmsg` and `write` to `tcp_sendmsg`.

Due to arguments mismatch, `read` and `write` functions in Redis could not be

simply replaced with `tcp_recvmsg` and `tcp_sendmsg` functions, respectively; `read` and `write` take a file descriptor argument whereas `tcp_recvmsg` and `tcp_sendmsg` take a socket struct as one of the arguments. A colleague [84] implemented stubs in the kernel which translated the file descriptor to the corresponding socket struct and called relevant kernel TCP function. We modified Redis to call these stubs instead of `read` and `write` functions. UKL provides functions that need to be called before and after invoking kernel functions directly (§ 5.13) to ensure the execution environment is correctly tracked. These changes required only 10 LOC to be modified in Redis. We call this optimization `UKL_RET_BYP (shortcut)`.

Chapter 6

Evaluation

The main goals of this thesis are to move a general-purpose operating system across the generality-specialization spectrum, i.e., integrate optimizations explored by different specializable systems into a general-purpose operating system while preserving its application and hardware compatibility and its ecosystem of tools, utilities and its community of developers. In this chapter, we want to understand if UKL can achieve those goals and if there are any performance advantages compared to Linux and other specializable systems.

In Section 6.1, we discuss if UKL preserves the generality of Linux. Our experiences show that UKL runs many unmodified Linux applications (§ 6.1.1) after recompiling them and supports unmodified Linux binaries as normal userspace processes. UKL maintains Linux's hardware compatibility (§ 6.1.2), and we have been able to deploy it virtualized and bare-metal on different hardware. UKL can be deployed, debugged, tuned, and profiled with standard Linux utilities and tools, preserving its ecosystem (§ 6.1.3). The code changes required for UKL are also minimal.

In Section 6.3, we aim to understand, at a fine granularity, what impact different optimizations have on performance. We run a series of microbenchmarks and find out that, for various system calls, the UKL base model performs very similarly to Linux. Adding more optimizations on top of the UKL base model results in significant improvements over Linux.

In Section 6.5, we take Redis as a case study and explore different aspects of UKL.

We compare UKL to other specializable systems and learn that UKL performs within 10% of a recent clean-slate unikernel designed for performance. On top of that, UKL preserves the applications and hardware compatibility of Linux, and its ecosystem, which unikernels written from scratch cannot. We benchmark Redis in virtual and bare-metal environments and see up to 22% improvement in latency and up to 26% improvement in throughput compared to Linux. To further understand the reasons for UKL's performance improvement over Linux, we profile UKL, analyze the output, and find out that cache efficiency plays a significant role in UKL's gains.

6.1 Preserving Generality

One of the central goals of UKL is to preserve, at least at the general-purpose end of the spectrum (§ 3.4), the application and hardware compatibility of Linux, as well as its ecosystem of tools, utilities, and developers. This section presents a qualitative analysis of how well UKL achieved this goal. We share our experiences of optimizing different applications with UKL (§ 6.1.1), deploying UKL on different virtual and physical hardware (§ 6.1.2), and running various applications, tools, and utilities as normal user space processes and sharing the UKL patch with the Linux community (§ 6.1.3).

6.1.1 Application support

We expected no significant challenges in running different unmodified applications as optimization targets with the UKL base model (§ 4.2.1, after compilation and linking. Our hypothesis was largely true. We tested dozens of unmodified applications without any additional UKL-specific effort, including Memcached [69], Redis [85], Nginx [74], FIO [19] (a filesystem benchmark), a multi-party computation benchmark [60], a small TCP echo server, simple programs to test C++ constructors and the C++ Standard Template Library (STL), the GAP Benchmark Suite [23] (a complex C++

graph based benchmark suite), LEBench [86] (a Linux system call benchmark), and a large number of standard `glibc` and pthread unit test programs.

Although the experience of running different applications with UKL was largely smooth, we did experience three (anticipated) challenges. First, it can be difficult to re-recompile and statically link applications with complex dependencies and Makefiles. Second, we have hit a number of programs that by default invoke `fork` followed by `exec` (see §§ 5.7 and 5.8), e.g., Postgres, or are dependent on the dynamic loader. Third, we have run into issues with proprietary applications available in only binary form, e.g., user-level libraries for GPUs. UKL needs the source code to recompile and link with the kernel.

6.1.2 Hardware support

One of our goals with UKL was to support the entire hardware compatibility list (HCL) of unmodified Linux (§ 4.1.3). We ensured that Linux boot-up process (which does hardware discovery and device setup) was not modified, except at specific points to enable UKL functionality (§ 5.3). Our experience with running UKL shows that this goal was achieved; we have not run into any compatibility issues and have booted or `kexec`ed to UKL on a wide variety of x86-64 servers, virtualization platforms, and laptops with different Intel and Brocade and virtual NICs, as well as NVMe, SATA controllers, and virtual block devices. The scripts and tools used to deploy and manage regular Linux machines were used for UKL deployments without modification. We have not been able to exploit GPUs without access to the source code for key libraries.

6.1.3 Ecosystem

UKL aims to preserve Linux's ecosystem of tools, utilities, and its community (§ 4.1.4), and our experience shows that this goal was also achieved. As expected, we

Project	**LoC**	**Files**	**Subsystems**	**Outcome**
Popcorn	7,763	64	14	Out of tree
NetGPU	3,827	45	14	Rejected
DAMON	3,805	24	3	Accepted
KML	3,177	70	16	Out of tree
BPFStruct	2,639	32	10	Accepted
BPFDump	2,343	32	8	Accepted
ArmMTE	1,764	63	14	Accepted
NFTOffload	1,579	56	24	Accepted
UKL	**550**	**33**	**10**	-
KRSI	1,085	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 6.1: Comparison of UKL patch to 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.

have had no difficulty running hundreds of unmodified binaries as normal user-level processes, including all the standard UNIX utilities, bash, different profilers, `perf`, and eBPF tools. 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. 5·1).

The UKL patch size for the base model is around 550 LoC, and the full UKL patch with all the configurations we have explored so far is 1250 lines. We have spent several months discussing and presenting the concept of unikernels and the UKL approach to kernel developers within Red Hat. We posted the base model as an RFC to the public Linux kernel mailing list in October 2022 [12]. We had several commenters with specific technical suggestions that can be readily addressed and one maintainer with extensive and constructive feedback that we will be incorporating. Only one maintainer seemed strongly opposed for philosophical reasons, but even he did not reject the patch.

To provide context for the size of the UKL patch, table 6.1 compares the base UKL patch to a selection of Linux features described in Linux Weekly News (LWN) [73] articles in 2020. UKL's patch size is smaller, and it modifies fewer files and Linux subsystems than many other patches which have been accepted into Linux. For comparison, the KML[64] patch, used in the recent Lupine work, which runs applications in kernel mode is 3177 LoC, a complexity that may have contributed to the patch not being accepted upstream. While UKL goes beyond the optimizations that KML does, the UKL base model can be compared to KML. The UKL base model and KML both run applications in kernel mode and replace `syscall`s with function calls. UKL base goes beyond this and also links the applications with the kernel. So one question we had was why the implementation of the UKL base model was so

much simpler compared to KML. After reviewing the code, we realized this simplicity is due to three fortuitous changes since KML was introduced. First, the UKL base model takes advantage of recent changes to the Linux kernel that make the changes to assembly much less intrusive. Second, the UKL base model supports only x64-64, while KML was introduced when it was necessary to support i386 to be relevant. Third, the UKL base model does not deal with older hardware, like the i8259 PIC, that had to be supported by KML. All the optimizations explored in UKL so far go beyond KML, enabling the broader specializations we talk about later, but the patch for them is still only 1250 LoC, well below KML's.

6.2 Experimental Setup

Unless otherwise stated, all experiments in this thesis are run on Dell R620 servers configured with 128G of RAM across two sockets on a single NUMA node. Each socket contains an Intel Xeon CPU E5-2660 0 @ 2.20GHz with 8 cores. The processors are configured to disable Turbo Boost, hyper-threads, sleep states, and dynamic frequency scaling. They are connected through a 10Gb link and use Broadcom NetXtreme II BCM57800 1/10 Gigabit Ethernet NICs. Multi-node experiments use identically configured nodes attached to the same top-of-rack switch to reduce external noise. We use identical application versions, Linux kernel v5.14 and `glibc` v2.31 for Linux and UKL for all experiments. Further, we use similar Linux configurations (modulo the UKL options) and boot command line options for Linux and UKL.

6.3 Microbenchmarks

We want to investigate the performance benefits of each individual optimization explored in UKL so far. In this section, we use microbenchmarks to analyze and attribute performance benefits to different optimizations. We analyze the latency of

simple system calls (§ 6.3.1), the effect on latency if input size to those system calls is increased (§ 6.3.2), and the latency of page faults with different configuration options for UKL (§ 6.3.3). We compare different configuration options for UKL against unmodified Linux, which sits at the general-purpose end of the generality-specialization spectrum, to see if these optimizations move the system towards the specialization end. We find that the advantage of one of the fundamental characteristics of unikernels, i.e., linking the application into the kernel, is, with modern hardware, in fact, small. Second, if we bypass Linux entry and exit code on system calls, the performance advantages can be significant, even for system calls that do substantial work; more than 80% for smaller system calls, e.g., `getppid` and 24% for system calls which spend more time in the kernel, i.e., 8KByte `recvfrom()` system calls. Third, optimizing page faults for applications linked in the kernel address space can be a significant advantage.

We use a Linux benchmark suite called LEBench [86] to evaluate the effect of different optimizations. LEBench measures the latency of many commonly used Linux system calls (e.g., `read`, `write`, `mmap`) as well as the latency of common operations like page fault handling. We modified LEBench to make the results repeatable across runs, add more determinism and improve its output. As one example of the changes we made, originally, LEBench used `malloc` to allocate different buffers, e.g., the buffer to read data into from a file to measure the latency of the `read` system call. Using `malloc` could affect reproducibility because it uses `brk` or `mmap` to allocate buffers depending on their size. Smaller buffers, allocated on the heap through `brk` call, have a higher chance of being allocated on pages already accessed earlier, and thus already faulted in, compared to larger buffers allocated through `mmap` call. When data is read into the buffers allocated through `mmap`, additional time is spent in page fault handling every time a new page is accessed, which alters the results of the `read`

system call. We modified LEBench to use `mmap` for all allocations. We also ensured all buffers were allocated at the same location in memory and that they were faulted into memory by accessing them beforehand.

6.3.1 System call base performance

We measure the latency of five commonly used system calls, i.e., `getppid`, `read`, `write`, `sendto`, and `recvfrom`. We compare the UKL base model and `UKL_BYP` to unmodified Linux. We hypothesize that the UKL base model (§ 4.2.1) would perform only slightly better than unmodified Linux because, on modern hardware, the benefit of replacing `syscall` instruction with `call` instruction is not expected to be large, while `UKL_BYP` (§ 5.12) would offer more significant performance gains due to bypassing Linux entry and exit code.

We use LEBench [86] to measure the latency of each system call for each system at least 10,000 times. To ensure that we only measure the intended operation, all buffers to store timing results are pre-allocated and pre-faulted into memory. We do the same with buffers required for reading and writing data for `read`, `write`, `sendto`, and `recvfrom` system calls. For `read` system call, we open a file and read 1 byte into a buffer. The file is located in a memory-based file system to avoid the cost of disk access. For `write` system call, we write 1 byte from a buffer into a file. For `sendto` system call, we call `fork` to create a child process. The parent and child open sockets for communication and transmit 1 byte in each run, and we measure the latency on the sender's side. For `recvfrom` system call, we again `fork` a child and open sockets. But the sender sends all the data before we measure the latency of `recvfrom` at the receiver's side. We do several runs before collecting numbers to warm up the system.

Figure fig. 6·1 shows the results. As expected, the advantage of the UKL base model over Linux is marginal (less than 5%) because the `syscall` instruction on modern systems is so optimized that replacing it with `call` instruction does not

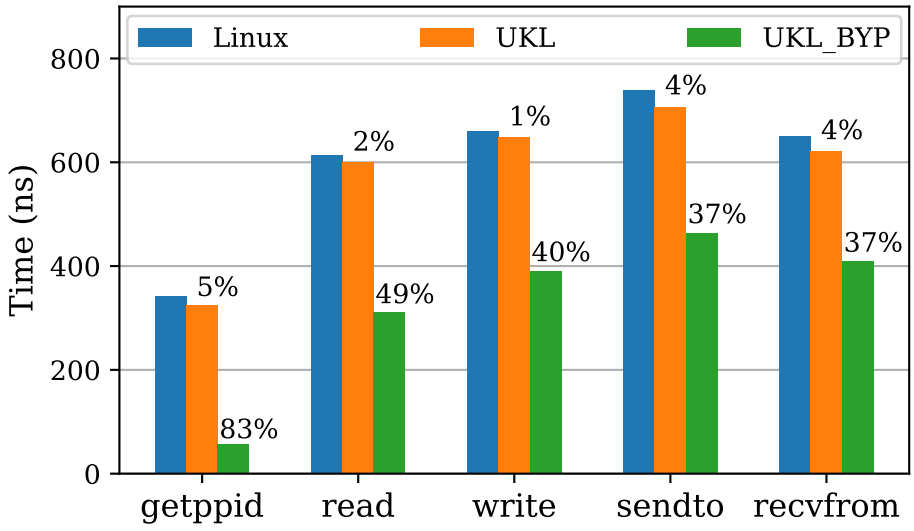

Figure 6·1: Comparison of Linux, UKL base model, and UKL_BYP for simple system calls. With modern hardware, the UKL advantage of avoiding the system call overhead is modest (<5%). However, there appears to be a significant advantage for simple calls with `UKL_BYP`, which bypasses the entry and exit code on transitions between application and kernel.

provide huge benefits. The real win comes with the UKL_BYP configuration option, which shows, compared to Linux, 83% improvement in getppid, 49% for read, 40% for write, and 37% for both sendto and recvfrom system calls. These results show that Linux entry and exit code is the primary source of latency in system calls, as opposed to the hardware cost of the syscall instruction, and UKL_BYP can greatly benefit workloads that make many small system calls.

6.3.2 Large requests

After evaluating base system call latency in § 6.3.1, we now want to measure the effect of increasing the input size on the latency of read, write, sendto, and recvfrom system calls. Based on results in § 6.3.1, we hypothesize that the UKL base model would have a negligible advantage over Linux, but UKL_BYP may show some improvement over Linux.

We again use LEBench [86] to measure the latency. Description of how LEBench measures the latency of read, write, sendto, and recvfrom system calls is provided in the previous section (§ 6.3.1). We increase the input size from 1 byte to 8 Kbytes, with 256-byte increments. The experiment is repeated 10,000 times for each size for each system (Linux, UKL base model, and UKL_BYP). The results for read and write system calls are shown in fig. 6·2 and results for sendto and recvfrom system calls are shown in fig. 6·3.

Figures 6·2 and 6·3 show that the UKL base model provides, as expected, negligible performance improvement over Linux, and UKL_BYP offers some performance improvement. The performance improvement given by UKL_BYP is due to bypassing the entry and exit code and is not affected by the system call input size. So UKL_BYP offsets the system call latency by the same amount irrespective of input size, resulting in a diminishing percentage gain as larger input sizes increase the time spent in kernel code. Figures 6·2 and 6·3 show the percentage improvement of UKL_BYP over Linux in

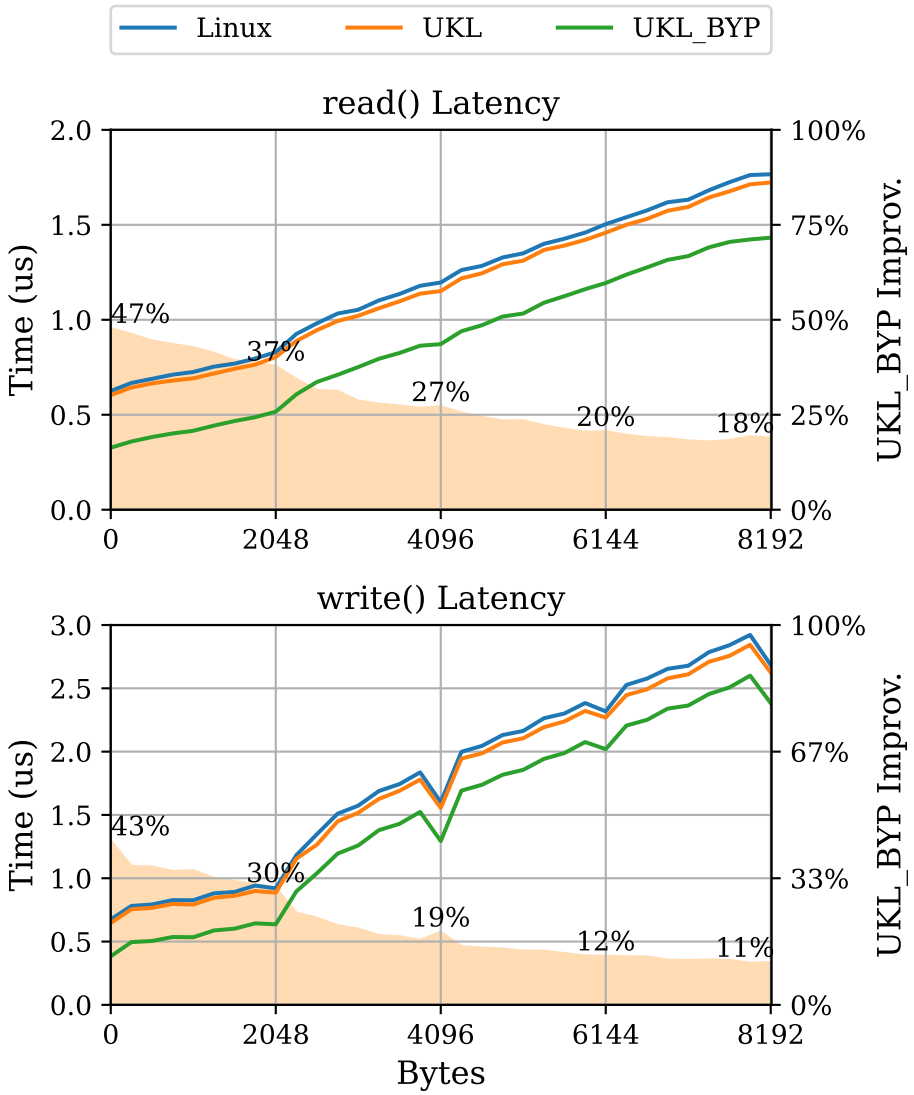

Figure 6·2: Comparison of Linux, UKL base model, and UKL with bypass configuration for `read` and `write` system calls. With increasing payload for each system call, the UKL base model (orange line) shows modest improvement over Linux, but there is a significant advantage for `UKL_BYP` (green line). The percentage improvement for `UKL_BYP` (shaded area) over Linux decreases as payload increases but is still significant for 8KB payload.

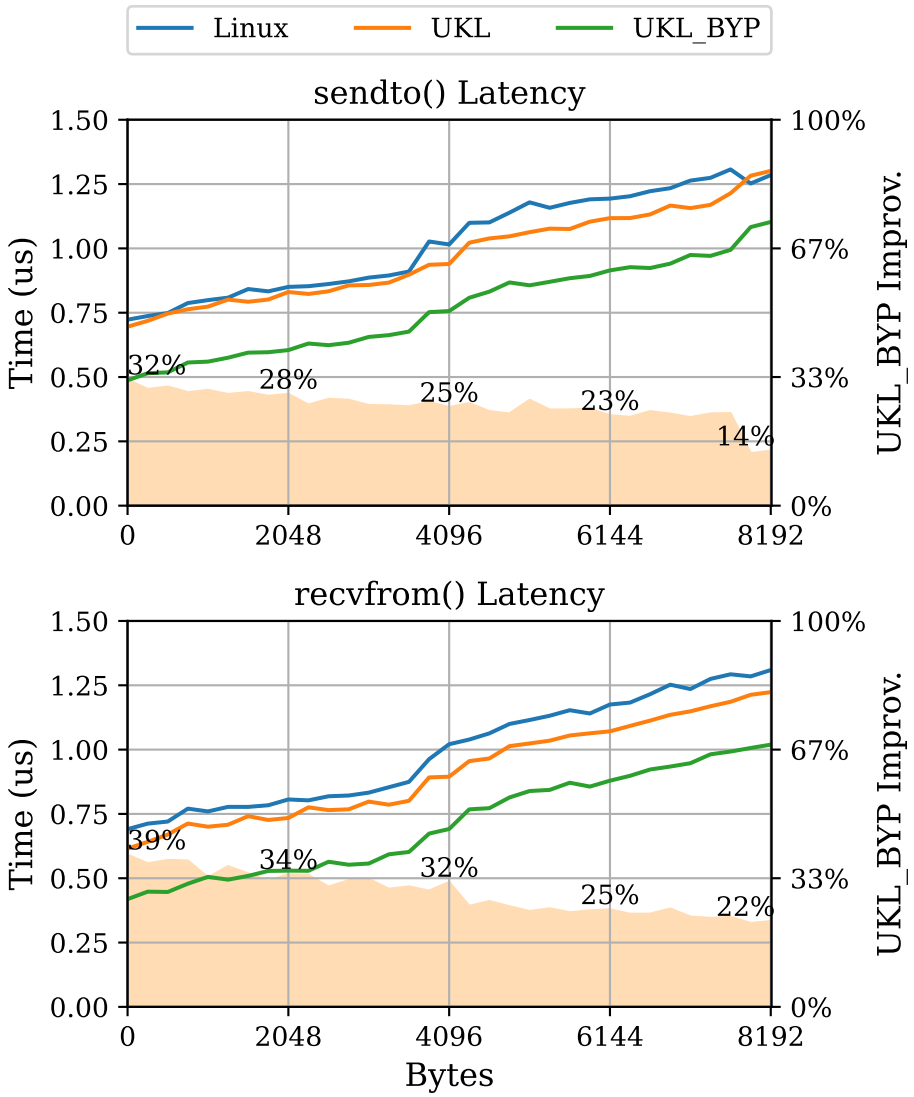

Figure 6·3: Comparison of Linux, UKL base model, and UKL with bypass configuration for `sendto` and `recvfrom` system calls. With increasing payload for each system call, the UKL base model (orange line) shows modest improvement over Linux, but there is a significant advantage for `UKL_BYP` (green line). The percentage improvement for `UKL_BYP` (shaded area) over Linux decreases as payload increases but is still significant for 8KB payload.

the shaded region and right size vertical axis. We can see the percentage improvement decreasing as the input size increase, but even for sizes up to 8 Kbytes, the percentage improvement is still significant, i.e., between 11% and 22%. This means that UKL_BYP can also benefit workloads that make system calls with larger payloads.

It is interesting to contrast our results with those from the recent Lupine § 3.2.3.3 Linux, which shows (like us) that the benefit of replacing `syscall` instruction with `call` instruction is minimal (less than 5%). From these results, authors of Lupine Linux conclude that the benefit of the transition between application and kernel code is minimal. But 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. Reducing this overhead significantly impacts even expensive system calls.

6.3.3 Page Fault handling

UKL provides two configuration options to handle stack page faults, i.e., UKL_PF_DF and UKL_PF_SS (see § 5.11 for details). UKL_PF_DF handles stack page faults on a double fault stack and does not change the handling of non-stack page faults. UKL_PF_SS, on the other hand, handles all stack and non-stack page faults on a dedicated stack. We want to see if these options affect the latency of page fault handling compared to unmodified Linux. Added to these, the UKL_RET configuration option (see § 5.9) optimizes the return path from all interrupts, exceptions, and faults. It would be interesting to see its effect on page fault latency. We hypothesize that there would not be a significant difference between Linux, UKL_PF_DF and UKL_PF_SS because all three involve a stack switch to a dedicated stack; Linux switches from user stack to kernel stack, UKL_PF_DF switches from user stack to double fault stack, and UKL_PF_SS switches from user stack to a dedicated stack (see § 5.11 for details). We expect UKL_PF_DF and UKL_PF_SS to perform slightly better than Linux because these don't

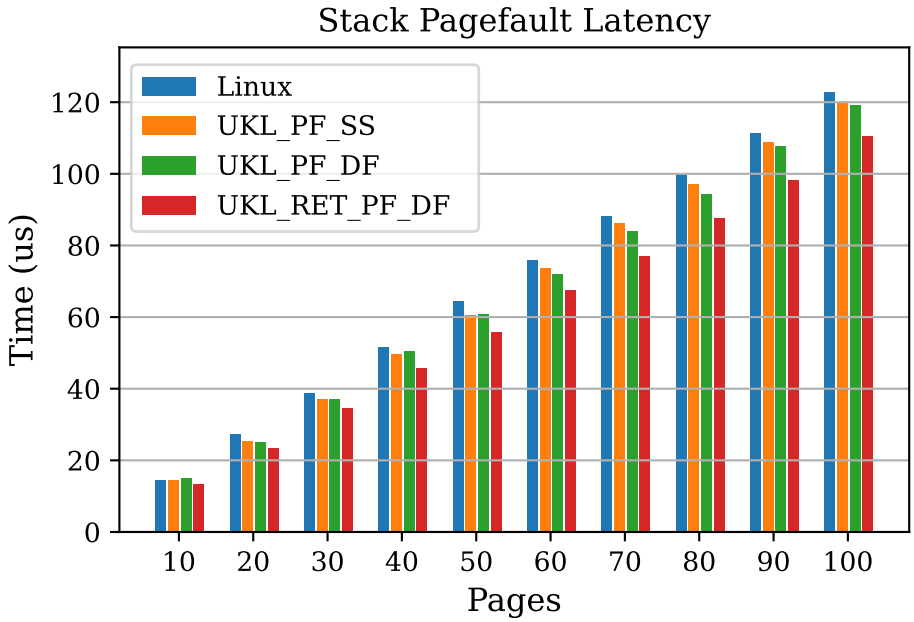

Figure 6·4: Latency of stack page faults; UKL_PF_DF handles problematic page faults on a double fault stack, and UKL_PF_SS handles all page faults on a dedicated stack showing slight improvement over Linux. UKL_RET_PF_DF is further configured to use `ret` instead of `iret` when returning from page faults and shows a higher improvement over Linux.

involve a privilege switch on page faults, while Linux does. `UKL_RET` is expected to show a bigger performance advantage because the return path of Linux, `UKL_PF_DF` and `UKL_PF_SS` involves a `iret` while `UKL_RET` involves a `ret`.

We use LEBench [86] to measure the stack page fault latency. We allocate a `char` array on the stack and measure the latency of writing only one `char` after every 4096 bytes (page size), i.e., the latency of faulting in each page of the stack. If we simply repeat this experiment, we would not be measuring stack page fault latency because, after the first run, the initial stack pages would already be faulted in memory. With increasing array size, only one extra page fault will be incurred. To ensure that we measure the latency of faulting in each page of the stack every time we repeat the experiment, we call `fork` each time to create a new process, and then we measure

the stack page fault latency in the child process. We use pipes to send data from the child to the parent after the run is over, after which the child exits. We repeat this experiment for array sizes ranging from 10 pages to 100 pages, with an increment of ten pages.

Figure fig. 6·4 the latency of stack page faults in Linux, `UKL_PF_DF`, `UKL_PF_SS` and `UKL_RET_PF_DF` which is `UKL_RET` configuration option added on top of the `UKL_PF_DF`. Linux, `UKL_PF_DF` and `UKL_PF_SS`, all include a stack switch on page faults. `UKL_PF_DF` and `UKL_PF_SS` show lower latency than Linux, as expected, due to kernel mode execution, i.e., not having to switch the privilege level on page faults. `UKL_PF_DF` and `UKL_PF_SS` should theoretically show the same latency, but `UKL_PF_DF` shows slightly lower latency for larger number of faults. A reason for this can be additional non-stack related page faults that might occur while running the benchmark. These non-stack page faults are handled on the stack already in use in `UKL_PF_DF`, but for `UKL_PF_SS`, all page faults, stack or non-stack, involve a switch to dedicated IST stack. Care has been taken to ensure that these non-stack faults are not too frequent, which is why `UKL_PF_DF` shows only marginally lower latency than `UKL_PF_SS`. The real advantage comes by using the `UKL_RET` configuration option. It optimizes the return path by using `ret` instead of `iret`. When it is applied on top of `UKL_PF_DF`, shown as `UKL_RET_PF_DF` in fig. 6·4, it shows lower latency compared to Linux, `UKL_PF_DF` and `UKL_PF_SS`, especially if the number of page faults increases. `UKL_RET` would also improve the latency of other interrupts (e.g., timer interrupt) that might occur while the benchmark is running, further improving performance.

6.4 I/O Latency Benchmark

Applications which are latency sensitive, e.g., high-frequency trading, require high-speed I/O. Section 6.3 showed the impact of different optimizations on system call

	Read		Write	
System	Mean	Throughput (Mb/s)	Mean	Throughput (Mb/s)
Linux	324K	42.14	323K	42.09
UKL_RET_BYP	441K	57.37	439K	57.2
Improvement	36.1 %		35.9 %	

Table 6.2: Mean operation count and throughput in Mb/s of `fio` when run with Linux and UKL_RET_BYP. UKL showed a 36% improvement in operation count and throughput.

latency. This section explores how those optimizations, and the resulting speed-up in system call latency, impact I/O latency.

To study I/O latency with UKL, we used `fio` [19], a flexible I/O benchmarking tool. We configured `fio` with an I/O depth of 1, so that any speed-ups directly translate to latency gains, performing randomly interleaved 4Kb reads and writes using direct I/O to an 8GB file. This experiment was done on a 2021 Lenovo X1 laptop with 64G of RAM and a 1T NVMe disk formatted with EXT4. Since each request has to wait for the prior one to finish, improvement in the latency of the requests directly translates into an increase in the number of requests serviced in a fixed period of time. We compare the throughput of Linux with `UKL_RET_BYP`. As with earlier experiments, the hypothesis was that `UKL_RET_BYP` would outperform Linux.

The results are shown in table 6.2. We can see that `UKL_RET_BYP` provides a 36% performance advantage over Linux for read and write operations. This shows that UKL can have a large impact on latency-sensitive applications.

6.5 Single Threaded Application - Redis

In this section, we want to evaluate the performance benefits of UKL and its optimizations on real-world applications. We select Redis [85], a widely used in-memory database, as our target application because it enables comparison to other unikernels. Redis can be configured to run as a single-threaded application; it does not use `fork` or `exec` and does not have any dependencies on any complex library except `glibc`. Due to these reasons, Redis has been widely used to test specializable systems [54, 56].

6.5.1 Comparison with other Specialized Systems

In this section, we compare UKL's performance against other specializable systems. We reuse the experimental setup done by Unikraft[54] for this experiment; the Redis server was deployed on Unikraft, Lupine, and various configurations of UKL inside a single-core virtual machine. Implementation details of `UKL_RET_BYP` can be found in §§ 5.9 and 5.12 and those of `UKL_RET_BYP (shortcut)` can be found in § 5.14. To replicate the full setup, `redis-benchmark` was used to generate workload and was deployed outside the virtual machine, on the host, with 30 connections, 100k requests, and pipelining of 16 requests. We also include Linux 4.0 as the baseline comparison for Lupine since Lupine uses the KML [64] patch, which was last implemented for Linux 4.0, and we include Linux 5.14 as a baseline comparison for UKL because that's the version of Linux we use for UKL. Figure 6·5 shows Redis throughput for `SET`s and fig. 6·6 shows Redis throughput for `GET`s.

For the virtualized, single-core experiments, we expect Unikraft to outperform other systems because it is a unikernel written from scratch, designed for performance. We expect Lupine to not show a significant advantage over Linux 4.0 because, in our experience, substituting `syscall` instructions with `call` and kernel mode execution provides only marginal improvement. UKL base model labeled UKL in the figs. 6·5

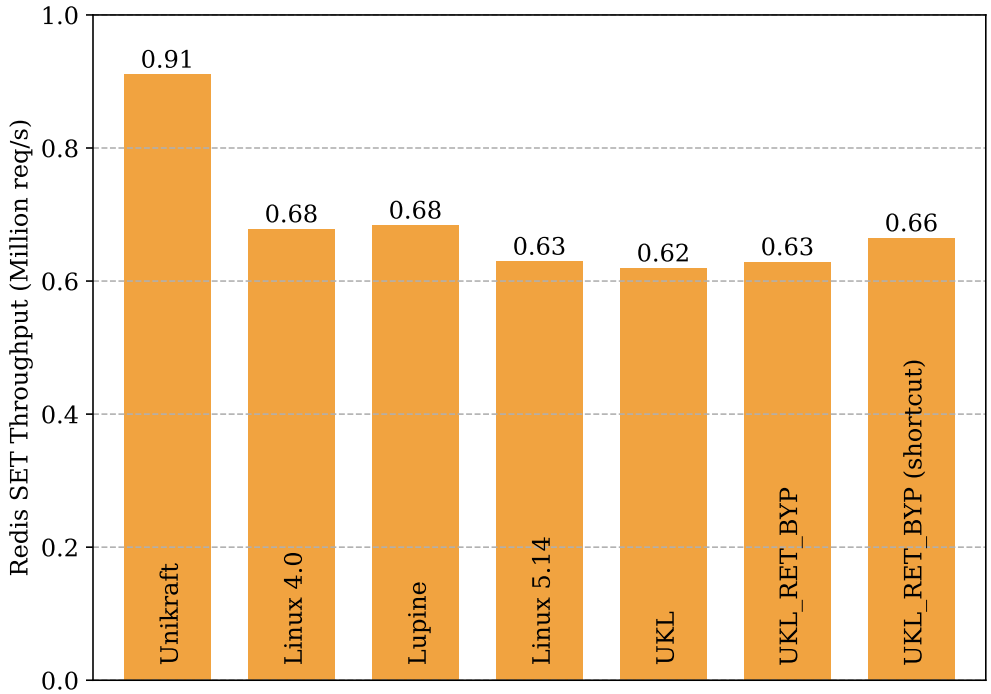

Figure 6·5: Redis SET throughput comparison of UKL with Unikraft [54] and Lupine [56]. We provide Linux 4.0 as a baseline comparison for Lupine and Linux 5.14 as a baseline comparison for UKL. We reuse the setup done by Unikraft[54] for this experiment where the Redis server was running on these systems inside a single-core virtual machine. The client ran `redis-benchmark` outside the virtual machine, on the host, with 30 connections, 100k requests, and pipelining 16 requests.

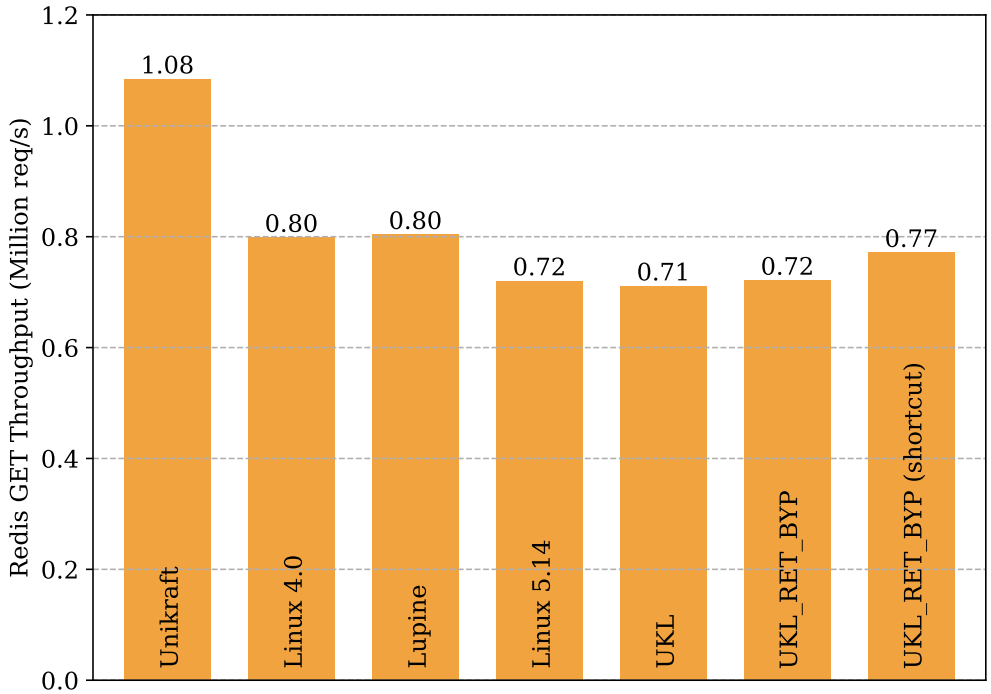

Figure 6·6: Redis GET throughput comparison of UKL with Unikraft [54] and Lupine [56]. We provide Linux 4.0 as a baseline comparison for Lupine and Linux 5.14 as a baseline comparison for UKL. We reuse the setup done by Unikraft[54] for this experiment where the Redis server was running on these systems inside a single-core virtual machine. The client ran `redis-benchmark` outside the virtual machine, on the host, with 30 connections, 100k requests, and pipelining 16 requests.

and 6·6 is similar to Lupine in this regard, except it also links the application and kernel code together. Similarly, we expect the UKL base model not to show colossal performance improvement over Linux 5.14. We expect `UKL_RET_BYP` to show some improvement over Linux 5.0 because it bypasses the Linux entry and exit code (§§ 5.9 and 5.12), and `UKL_RET_BYP (shortcut)` to show a more significant advantage over Linux 5.14 because it bypasses not just the entry and exit code, but also the VFS layer (§ 5.14).

Figures 6·5 and 6·6 show the results. As expected, Unikraft shows high throughput (0.9M req/s for `SET`s and 1M req/s for `GET`s), better than any other system. The reason for this performance (§ 3.2.1.5) is the specialized libraries, developed from scratch, for optimized I/O, memory allocation, scheduling, and synchronization. We see, as expected, Lupine and the UKL base model show no advantage over their respective comparison points. Linux 5.14 underperforms compared to Linux 4.0, and there can be many reasons for this; the configuration options used to build Linux 4.0 are the same as the ones used for Lupine (to ensure fair comparison). These configuration options are pruned for performance, i.e., no SMP support and turning off all options related to bare-metal boot. This makes Linux 4.0 much lighter than Linux 5.14, which is configured with the default Linux configuration options, allowing it to boot bare-metal. Further, the difference in kernel versions 4.0 and 5.14 is huge, and the code changes can also be responsible for the difference in performance. Further, Redis deployed on Linux 4.0 uses `musl` C library, whereas Redis deployed on Linux 5.14 uses `glibc`, which can also be a reason for the performance difference. The unexpected result is that `UKL_RET_BYP` shows no improvement over Linux 5.14, although it bypasses the Linux kernel entry and exit code. This means that the cost of Linux entry and exit code in a virtualized setup is not high enough to register a performance improvement if bypassed, but this requires further investigation. As

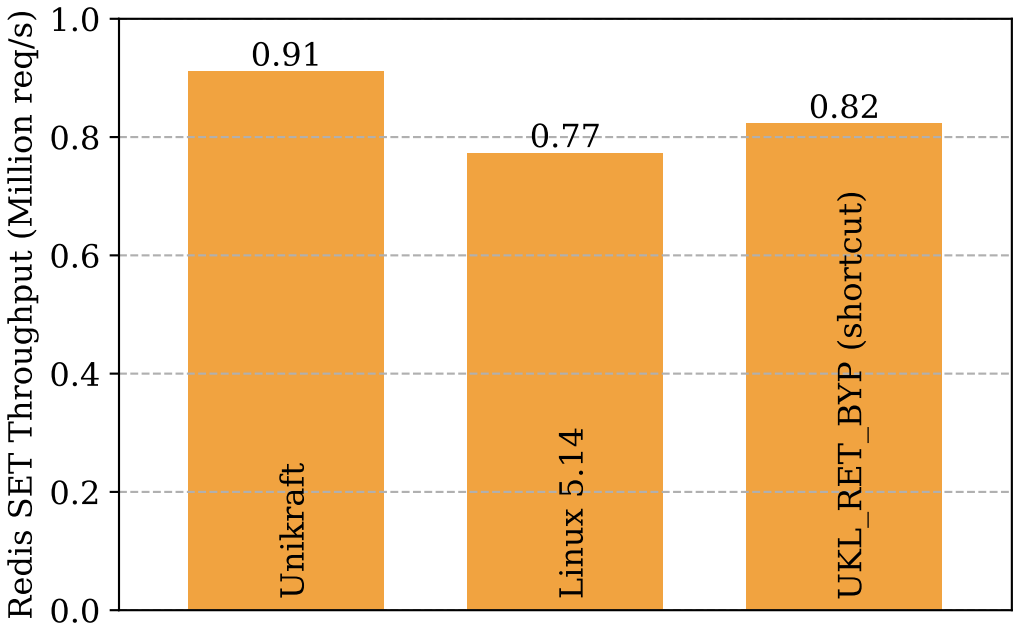

Figure 6·7: Repeat of the experiment in fig. 6·5 (Redis SET throughput), except with Linux 5.14 and `UKL_RET_BYP (shortcut)` run in a 2 core virtual machine, with one core isolated through `isolcpus` boot parameter. The Redis server, which is single-threaded, was pinned on this isolated core through `taskset` utility. Providing an extra core improves Linux 5.14 and `UKL_RET_BYP (shortcut)` performance.

expected, `UKL_RET_BYP (shortcut)` shows around 5% improvement over Linux 5.14 for `SET`s and around 7% improvement for `GET`s. This shows that specializing a general-purpose operating system can buy back 5% to 7% performance improvement that a highly specialized system, written from scratch, can provide. On top of this, `UKL_RET_BYP (shortcut)` also preserves application and hardware compatibility, and the ecosystem of tools, utilities, and the developer community, that a from-scratch system cannot.

Further, figs. 6·7 and 6·8 point to an interesting bit about the experimental setup and the systems being compared together. Unikraft (§ 3.2.1.5) only supports a single

core and a single process/thread. Similarly, Lupine is configured to have no SMP support, i.e., it also only supports a single core. UKL, on the other hand, is built with configuration options that allow the same kernel to be deployed bare metal, use multiple cores, and run multiple processes, which can be multi-threaded (building UKL with minimal configuration options is possible, but orthogonal to this research, because we want to preserve the generality as much as possible while specializing the system). Using the `ps` utility on UKL showed a large number of kernel background threads that contend with Redis for CPU time. To remove this contention, we reran the same experiment but provided two cores. Through Linux boot parameter `isolcpus`, we isolated one of those cores so nothing would be scheduled on that core, and using the `taskset` utility, we pinned Redis to that core.

Figures 6·7 and 6·8 show the results of these runs. Unikraft or Lupine don't use extra cores, and when run with an extra core, they showed no performance difference. Linux 5.14 running on two cores with Redis pinned to a dedicated core shows 0.77M reqs/s `SET` throughput and 0.94M reqs/s `GET` throughput. This shows that when there is no contention for CPU, unmodified Linux can perform better, and that leaves Unikraft, a highly specialized system written from scratch, only 18% better for `SET`s and 14% better for `GET`s. It must be noted here that all this performance advantage is not just due to removing CPU contention; some of it might be due to interrupt processing parallelism, i.e., network interrupts can now be services on two cores instead of one. For `UKL_RET_BYP (shortcut)`, we similarly isolate a core and pin Redis to that core, and get 0.82M reqs/s throughput for `SET`s and 1M reqs/s throughput for `GET`s, which is more than 6% better in both cases compared to Linux 5.14 running in this setup. Compared to `UKL_RET_BYP (shortcut)`, Unikraft's throughput is around 11% better for `SET`s and 8% better for `GET`s. This shows the huge opportunity UKL provides, i.e., with a few optimizations, the performance is within 10% of a highly

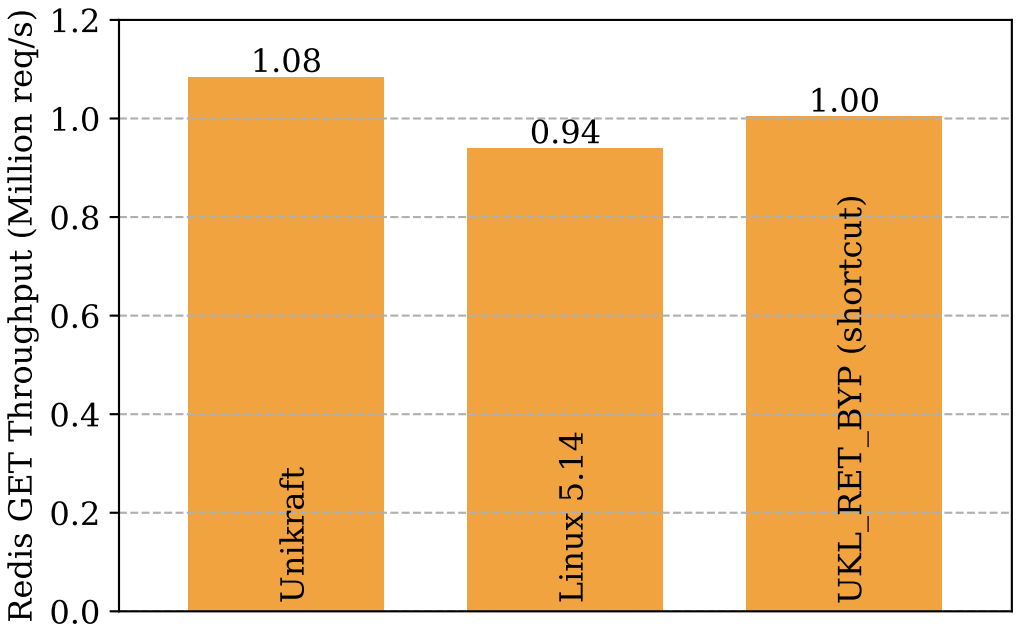

Figure 6·8: Repeat of the experiment in fig. 6·6 (Redis GET throughput), except with Linux 5.14 and `UKL_RET_BYP (shortcut)` run in a 2 core virtual machine, with one core isolated through `isolcpus` boot parameter. The Redis server, which is single-threaded, was pinned on this isolated core through `taskset` utility. Providing an extra core improves Linux 5.14 and `UKL_RET_BYP (shortcut)` performance.

	Linux 5.14	UKL_RET_BYP (shortcut)	**% Improv.**
SET	0.81	0.84	4.73%
GET	0.92	1.00	8.53%

Table 6.3: Throughput of Redis SETs and GETs in Million reqs/sec for Linux 5.14 and UKL_RET_BYP (shortcut) running bare-metal. UKL_RET_BYP (shortcut) shows more than 4% improvement for SETs and more than 8% improvement for GETs over Linux. redis_benchmark is used to generate traffic.

specialized unikernel written from scratch. If further optimizations are explored, the performance of UKL will increase further. All of these optimizations are done with minimal changes to Linux. In contrast, if a from-scratch system like Unikraft needs to explore further changes, it requires a higher time and engineering investment. Due to UKL preserving generality, even when specialized, we can use Linux utilities like taskset etc., which is not possible on other specializable systems. This provides UKL a huge advantage, i.e., familiar tools and utilities can be used to manage and tune UKL instances.

An operator, developer, or system administrator who knows how to deploy and manage Linux instances can use UKL for performance improvements while still using all the familiar tools and utilities. A small amount of specialization over Linux brings the performance within around 10% of a highly specialized unikernel written from scratch. The trade-off is getting 10% extra performance but throwing out Linux's battle-tested code, its community, and application and hardware compatibility. UKL shows that an incremental approach that preserves generality can provide a significant gain even with modest changes.

In § 6.5.2 we present a detailed analysis of Redis running on UKL bare-metal by using the memtier_benchmark, a benchmarking tool which, like redis-benchmark,

is also developed by RedisLabs [57], the developers of Redis. We don't use `redis-benchmark` because, according to its documentation [26], *"The redis-benchmark program is a quick and useful way to get some figures and evaluate the performance of a Redis instance on a given hardware. However, by default, it does not represent the maximum throughput a Redis instance can sustain."* Memtier benchmark, on the other hand, is recommended for benchmarking Redis [7].

But before we change benchmarks, we quickly wanted to gauge UKL's performance with `redis-benchmark`. Table 6.3 shows the performance of Linux 5.14 and `UKL_RET_BYP (shortcut)`, with both systems running on 2 physical cores and one of those cores isolated for Redis. We can see that `UKL_RET_BYP (shortcut)` provides more than 4% improvement for `SET`s and more than 8% improvement for `GET`s over Linux. As we will see later in § 6.5.2, UKL shows even higher improvement over Linux when using the `memtier_benchmark`.

6.5.2 Bare Metal Experiment

To evaluate the bare-metal performance advantages of Redis with UKL, we compare the UKL base model (§ 4.2.1), `UKL_RET_BYP` (§§ 5.9 and 5.12) and `UKL_RET_BYP (shortcut)` (§ 5.14) with unmodified Linux. For this experiment, instead of `redis-benchmark`, used in § 6.5.1, we use the `memtier_benchmark` which generates more realistic load than `redis-benchmark` [7], and can provide the best latency and throughput that Redis can provide.

Based on our microbenchmark (§ 6.3) results, we expect the UKL base model to show a marginal advantage over Linux and `UKL_RET_BYP` to outperform the UKL base model. Further, we hypothesize the `UKL_RET_BYP (shortcut)` will outperform `UKL_RET_BYP` because of bypassing the VFS layer.

We build and deploy unmodified Linux on a bare metal node that will be the server (see § 6.2 for hardware and network description) and connect it over a VLAN to

another bare metal node running Fedora 30, which will host the Memtier benchmark (client). We build Redis as a regular userspace process, link it statically with the identical versions of `glibc` and other libraries used for UKL builds, and deploy it on the server. We configure the memtier_benchmark to create three threads, each creating 100 clients. Each client generates 100,000 requests for the Redis server. We also tried many other configurations but had to find the one that would drive the Redis server as much as possible without saturating the 10Gb network link between the client and the server (otherwise, the results would be affected by network delays). For the UKL experiments, we kept the Memtier client configured the same and built and deployed the server with Redis compiled with UKL base model, `UKL_RET_BYP` and `UKL_RET_BYP (shortcut)`.

Figure 6·9 shows the results of our experiment; we plot the probability density and cumulative density function for Linux, UKL base model, `UKL_RET_BYP` and `UKL_RET_BYP (shortcut)`. Average and 99th percentile tail latency are also shown. As expected, the UKL base model shows a negligible advantage over Linux. `UKL_RET_BYP` outperforms Linux and UKL base model, both in average and tail latency. Table 6.4 shows that `UKL_RET_BYP` has an 11% better tail latency and a 12% improvement in throughput over Linux. Also, `UKL_RET_BYP (shortcut)` outperforms both `UKL_RET_BYP` and Linux. `UKL_RET_BYP (shortcut)` shows a 22% improvement in tail latency and 26% improvement in throughput over Linux (table 6.4). These results show that the shortcuts identified in § 5.14 provided huge performance improvements. Also, these results show that although configuration-based optimizations like `UKL_RET_BYP` improve performance, more significant gains can be had if applications are modified to use custom paths inside the kernel, e.g., `UKL_RET_BYP (shortcut)`. These results show the value of having a system that can run tools like `perf` to understand exactly where performance gains are possible. Deeper specialization can then be

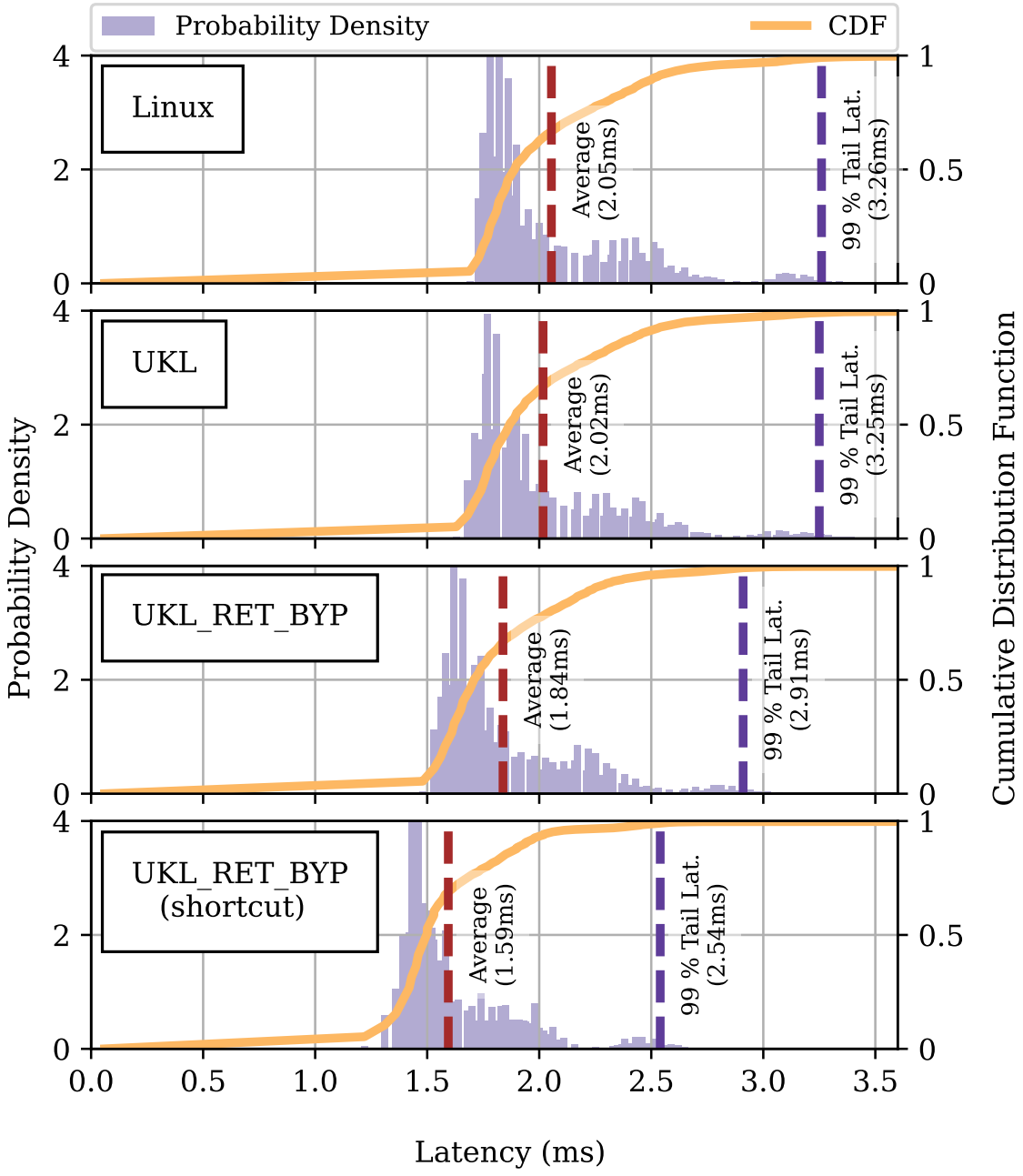

Figure 6·9: Probability Density (purple bars) and CDF (orange line) of Redis deployed on Linux, UKL, `UKL_RET_BYP` and `UKL_RET_BYP (shortcut)` and tested with the Memtier benchmark. Average latency (broken red line) and 99th percentile tail latency (broken purple line) are also shown.

	99 percentile tail latency		Throughput	
System	**(ms)**	**Improv.**	**(Kb/s)**	**Improv.**
Linux	3.26	-	6375.20	-
UKL base model	3.25	0.3%	6479.20	1.6%
`UKL_RET_BYP`	2.91	11%	7154.68	12%
`UKL_RET_BYP (shortcut)`	2.54	22%	8022.54	26%

Table 6.4: Redis throughput and latency improvements of UKL base model, `UKL_RET_BYP` and `UKL_RET_BYP (shortcut)` over Linux

identified for an application where we know a-priori that we can call a specific internal function and avoid all the general purpose code in the kernel. The kernel can be specialized according to the application's needs and pull the entire system towards the specialization end of the generality-specialization spectrum (§ 3.4).

6.5.3 `perf` Analysis of Redis

To better understand where the gains in Redis results (§ 6.5.2) come from, we use `perf` [9] to profile Redis running on unmodified Linux, UKL base model, `UKL_RET_BYP` and `UKL_RET_BYP (shortcut)`. We re-run the same bare metal experiment described in § 6.5.2. We run the experiment on the nodes described in § 6.2. To recap, fig. 6·10 shows the high-level architecture of the node. It has a single NUMA node with two sockets (or packages), and each socket contains 8 cores. We disable hyperthreading, so each core runs a single thread. Each core has a 32KB L1 instruction cache, a 32KB L1 data cache, and a unified 256KB L2 cache. All 8 cores on a socket share a 20MB last-level cache. We pin the Redis server on one of these cores through the `taskset` utility. In this context, we present the `perf` output in tables 6.5 to 6.9, and discuss it below.

Figure 6·10: A description of the node we use for Redis experiments, created through the `lstopo` utility, which shows a single NUMA node with 2 sockets (or packages), each containing 8 cores. Each core has separate 32KB L1 instruction and data caches and a 256KB unified L2 cache. All 8 cores on a socket share a 20MB L3 or last-level cache.

From table 6.5, the first thing to note is the total number of instructions executed. We see a tiny increase in instructions executed between Linux and UKL base model (0.57%), which makes sense because, while the UKL base model replaces the `syscall` instructions with `call` instructions, it actually adds a few new instructions in the transition code between application and Linux kernel for environment tracking (see § 5.4) and to ensure that the register state being stored on the stack is correct (see § 5.5) during those transitions. Then we see a slight decrease in instructions executed in `UKL_RET_BYP` compared to Linux (0.21%); this is due to bypassing the transition code between applications and the kernel (see § 5.12). We see a much more significant reduction in the number of instructions executed in `UKL_RET_BYP (shortcut)` (12.72%); this is because here we bypass not just the transition code between the application and kernel, we also bypass the VFS layer and call the relevant TCP send and receive functions directly from Redis send and receive functions, respectively. Since sends and receives are the majority of work that Redis does, optimizing these paths significantly reduces the total number of instructions executed.

The number of CPU cycles taken by the UKL base model compared to Linux follows the same trend as the total number of instructions executed, i.e., a slight increase (0.44%). The reduction in CPU cycles taken by `UKL_RET_BYP` (8.36%) compared to Linux is much more significant than the reduction in the number of instructions. This points to the fact that the instructions that were bypassed, i.e., the entry and exit code in transitions, were extremely inefficient. Bypassing that code has given a considerable performance advantage. Similarly, `UKL_RET_BYP (shortcut)`, when compared to Linux, shows a disproportionally more significant reduction of CPU cycles taken (21.86%) than the reduction in instructions executed (12.72%). This includes all the efficiency gotten by `UKL_RET_BYP` and further includes the efficiency gained by bypassing the VFS layer. This reduction in CPU cycles translates directly

	Linux	**UKL**	UKL_RET_BYP	UKL_RET_BYP (shortcut)
Instructions	358.61 B	360.64 B	357.84 B	313.01 B
Reduc. vs Linux		-0.57%	0.21%	12.72%
CPU Cycles	438.19 B	440.12 B	401.55 B	342.39 B
Reduc. vs Linux		-0.44%	8.36%	21.86%
Instructions/Cycle	0.82	0.82	0.89	0.91
Imp. over Linux		0.12%	8.89%	11.71%
Time (s)	200.87	201.85	184.04	156.93
Reduc. vs Linux		-0.49%	8.38%	21.87%

Table 6.5: Total number of instructions executed and CPU cycles (and time) taken by Redis running on Linux, UKL base model, UKL_RET_BYP and UKL_RET_BYP (shortcut). The instructions per cycle ratio is also shown.

to a decrease in time taken, as shown in table 6.5. And since CPU cycles taken by UKL_RET_BYP and UKL_RET_BYP (shortcut) decrease more than the reduction in instruction executed, we see improved instructions per cycle ratio for UKL_RET_BYP, which has 0.89 instructions per cycle (an improvement of 8.89% over Linux), and UKL_RET_BYP (shortcut), which has 0.91 instructions per cycle (an improvement of 11.71% over Linux). We will discuss the possible reasons for this jump in efficiency for UKL_RET_BYP and UKL_RET_BYP (shortcut) in the paragraphs below.

In table 6.6, we can see how all these systems fared as far as branch prediction is concerned, to discuss if it can be one of the factors for improved efficiency of UKL_RET_BYP and UKL_RET_BYP (shortcut). UKL base model shows a slight increase in the number of total branches encountered compared to Linux, which can be explained by the slight increase in the number of instructions executed. UKL_RET_BYP encountered the same number of branches as Linux and had the same percentage of branches mis-

	Linux	**UKL**	UKL_RET_BYP	UKL_RET_BYP (shortcut)
Branches	72.02 B	72.37 B	72.09 B	63.16 B
Reduc. vs Linux		-0.47%	-0.09%	12.31%
Branches Mispred.	0.34 B	0.42 B	0.34 B	0.09 B
% Mispred.	0.47%	0.58%	0.47%	0.15%

Table 6.6: Branches encountered and mispredicted by Linux, UKL base model, UKL_RET_BYP and UKL_RET_BYP (shortcut) while running Redis.

predicted as Linux (0.47%). This means that branch prediction did not play a role in the efficiency gotten by, at least, UKL_RET_BYP. UKL_RET_BYP (shortcut) encountered 12.31% fewer branches than Linux and had one-third the number of mispredictions (0.15% compared to 0.47% for Linux). The lower percentage of mispredicted branches for UKL_RET_BYP (shortcut) can be a factor in its improved efficiency, especially because branch misprediction can cause a pipeline flush and cost many cycles. But we have to be careful because the percentage of mispredicted branches in Linux is already very low (0.47%), so the room for improvement for UKL_RET_BYP (shortcut) is naturally minuscule.

Moving on to L1 cache performance, we can see from fig. 6·10 that the core has two separate 32KB L1 caches for instruction (icache) and data (dcache). Table 6.7 shows that UKL has better L1 icache efficiency Linux. UKL base model has 6.46% fewer icache misses compared to Linux, although it has been close to Linux, or slightly worse, in all metrics discussed earlier. This shows that not switching privilege domains every time a system call or interrupt occurs improves icache efficiency. UKL_RET_BYP shows 2.28% fewer icache misses than Linux, but not as good as the UKL base model. That might be due to a run-to-run perturbation, for the lack of a better explanation.

	Linux	**UKL**	UKL_RET_BYP	UKL_RET_BYP (shortcut)
L1 iCache Load Misses	10.28 B	9.62 B	10.05 B	5.21 B
Reduc. vs Linux		6.46%	2.28%	49.32%
L1 dCache Loads	106.92 B	107.86 B	106.16 B	92.05 B
L1 dCache Load Misses	6.75 B	7.26 B	7.07 B	6.14 B
L1 dCache Stores	61.94 B	62.83 B	59.2 B	49.72 B
L1 dCache Store Misses	1.87 B	1.95 B	1.91 B	1.82 B
L1 dCache Total	168.86 B	170.69 B	165.37 B	141.77 B
Reduc. vs Linux		-1.08%	2.07%	16.04%
L1 dCache Tot. Misses	8.62 B	9.21 B	8.98 B	7.96 B
Miss %	5.10%	5.40%	5.43%	5.61%

Table 6.7: An analysis of L1 cache performance for Linux, UKL base model, UKL_RET_BYP and UKL_RET_BYP (shortcut) while running Redis. A breakdown into instruction and data cache accesses and misses is also shown.

The real result is `UKL_RET_BYP (shortcut)` which shows almost 50% fewer icache misses than Linux. This means that after bypassing the entry and exit code and the VFS layer, `UKL_RET_BYP (shortcut)` not only has 12.72% fewer instructions to execute (table 6.5) than Linux, it also does so by fitting those instructions in the icache better, i.e., it misses the icache half as much as Linux.

Table 6.7 gives us interesting insights into L1 dcache performance as well. UKL base model accesses the L1 dcache slightly more times than Linux, as expected, due to a higher number of total instructions. `UKL_RET_BYP` shows a 2% lesser number of L1 dcache accesses compared to Linux. This might be because fewer instructions touch fewer kernel data structures, leading to fewer accesses to dcache. Again the real improvement is in `UKL_RET_BYP (shortcut)`, which accesses L1 dcache 16% less than Linux. We know that the VFS layer touches a lot of kernel data structures to translate a generic file descriptor to a network socket and route the code path from generic reads and writes to TCP sends and receives, respectively. Bypassing all that code also means that lesser data needs to be accessed, leading to fewer dcache accesses in `UKL_RET_BYP (shortcut)`. Although the L1 dcache miss percentage is almost the same for all four systems, because `UKL_RET_BYP (shortcut)` has fewer L1 dcache accesses, to begin with, the actual number of times L1 dcache misses is lower than Linux, i.e., 7.96 B compared to 8.62 B. Due to the fewer misses, `UKL_RET_BYP (shortcut)` saves the cost of accessing the L2 cache. So better L1 cache efficiency can be a factor in improved `UKL_RET_BYP (shortcut)` performance.

Table 6.8 shows that the number of data TLB accesses for all the systems is almost the same as the total L1 dcache accesses of those systems (from table 6.7). This is because the L1 cache is virtually indexed, physically tagged, so each L1 access would also involve a TLB access. Since the actual number of dTLB misses is the same for `UKL_RET_BYP (shortcut)` as compared to Linux, this does not explain the improved

	Linux	**UKL**	UKL_RET_BYP	UKL_RET_BYP (shortcut)
dTLB Accesses	169.08 B	170.89 B	165.56 B	141.96 B
Reduc. vs Linux		-1.07%	2.08%	16.04%
dTLB Misses	0.64 B	0.82 B	0.85 B	0.65 B
Miss %	0.38%	0.48%	0.51%	0.46%

Table 6.8: Data TLB performance of Redis running on Linux, UKL base model, UKL_RET_BYP and UKL_RET_BYP (shortcut).

performance of UKL_RET_BYP (shortcut).

Table 6.9 shows the number of last-level cache access and misses. The CPU cycles and time taken for each system, earlier shown in table 6.5, are provided again for quick reference. From table 6.7, we know that UKL_RET_BYP and especially UKL_RET_BYP (shortcut) had better L1 instruction and data cache efficiency. Table 6.9 shows the effect of that on the last-level cache, which is combined for instructions and data. We see a similar total number of accesses in Linux and the UKL base model. UKL_RET_BYP shows an 8.7% reduction in the total number of last-level memory accesses compared to Linux. This means that, due to bypassing the entry and exit code, UKL_RET_BYP had fewer instructions (better L1 icache efficiency), which touched fewer kernel data (better L1 dcache efficiency). That improved the L2 cache efficiency as well, i.e., better L2 cache hit rate and thus lower last-level cache accesses. The same is the case of UKL_RET_BYP (shortcut); better L1 cache efficiency led to better L2 cache efficiency, which meant 21.28% lower last-level cache accesses. Better L1 and L2 efficiency and a lower total number of accesses to the last-level cache translate directly to fewer CPU cycles and less time taken. To be precise, 8.77% fewer last-level cache accesses led to 8.36% improvement in UKL_RET_BYP and 21.28% fewer last-level cache accesses led to 21.86% improvement for UKL_RET_BYP (shortcut). These numbers

	Linux	**UKL**	UKL_RET_BYP	UKL_RET_BYP (shortcut)
LLC Accesses	6.07 B	6.16 B	5.54 B	4.78 B
Reduc. vs Linux		-1.44%	8.77%	21.28%
LLC Misses	1.41 B	1.4 B	0.85 B	0.79 B
Miss %	23.16%	22.72%	15.38%	16.59%
CPU Cycles	438.19 B	440.12 B	401.55 B	342.39 B
Reduc. vs Linux		-0.44%	8.36%	21.86%
Time (s)	200.87	201.85	184.04	156.93
Reduc. vs Linux		-0.49%	8.38%	21.87%

Table 6.9: Last-level cache accesses and misses for Redis running on Linux, UKL base model, `UKL_RET_BYP` and `UKL_RET_BYP (shortcut)`. The CPU cycles and time taken for each system, already shown in table 6.5, is provided again for quick reference.

suggest that UKL's gains come from better L1 and L2 cache efficiency compared to Linux.

6.6 Multithreaded Application - Memcached

So far, we have tested Redis, a single-threaded application. We now want to evaluate UKL's performance when targeting a more complex application, Memcached [69]. Memcached is a multi-threaded key-value store that relies heavily on the pthreads library and `glibc`'s internal synchronization mechanisms. It is also dependent on libevent [6], an event notification library that must be compiled and linked with. Memcached is an interesting application because unikernels generally don't support applications that require complex features, and some systems [88] require Memcached to be ported.

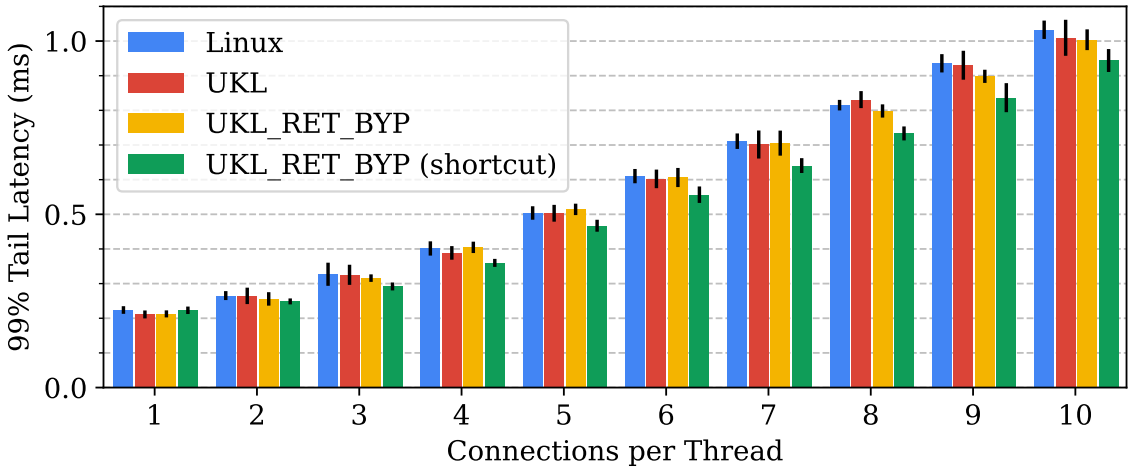

Figure 6·11: 99% tail latency for Memcached against increasing load. Memcached is configured to run with 4 threads, pinned to 4 cores, and deployed inside a 6 core VM. The client, i.e., `memtier_benchamrk`, was also configured with 4 threads, each pinned to a separate core and deployed natively on a different physical node, and both nodes connected through a physical network. This figure shows the 99th percentile tail latency of Memcached on different systems as the number of connections per thread in `memtier_benchmark` was increased to simulate increasing load. `UKL_RET_BYP (shortcut)` shows lower latency than Linux, even at a higher load (see table 6.10 for details on latency numbers and percentage improvement). UKL base model and `UKL_RET_BYP` also show a smaller improvement over Linux in most cases.

We deploy Memcached like it is mostly deployed in data centers, i.e., in a VM, and the client deployed on a separate physical node, sending requests over a physical network. We deploy Memcached inside a 6 core VM; we ensure that all the vCPUs are pinned to separate physical cores on the host. We configure Memcached to run with 4 threads, each thread pinned to a separate vCPU. Each of the systems, i.e., Linux and different configurations of UKL, are built with `virtio` network para-virtualization drivers. On the host end, we use the `vhost` mechanism to share network queues between the host and the guest, taking the QEMU userspace out of the critical path. This gives the VM a very high-speed network. Inside the VM, we pin the `virtio` network queues to the separate vCPUs as well to avoid cross-core interference due to network interrupts.

On a separate physical node, we run the `memtier_benchmark` on Fedora 30. We configure the benchmark to also have 4 threads, each pinned to a separate physical core, and generate a traffic of 100,000 requests. We increase the number of connections per thread from 2 to 10, to increase the load on Memcached server and measure the 99th percentile tail latency. We repeat each experiment 10 times and show an average in fig. 6·11 with error bars. The detailed breakdown of Linux and `UKL_RET_BYP (shortcut)` latency is shown in table 6.10, along with the percentage improvement over Linux.

From fig. 6·11 and table 6.10 we can see that UKL, especially `UKL_RET_BYP (shortcut)` shows upto 10% improvement over Linux, even when experiencing higher load. UKL base model and `UKL_RET_BYP` are very close to Linux in tail latency, especially at lower loads. This can be because when the network is not fully saturated, the improvement of the UKL base model and `UKL_RET_BYP` get amortized. `UKL_RET_BYP` shows improvement over Linux when the load on the system increases, especially after 8 connections per thread. This experiment shows us that UKL, unlike many other

Conns. per Thread	**Linux**	`UKL_RET_BYP (shortcut)`	**% Improv.**
1	0.22	0.22	0.36%
2	0.27	0.25	6.33%
3	0.33	0.29	10.76%
4	0.40	0.36	10.36%
5	0.50	0.47	7.30%
6	0.61	0.56	8.78%
7	0.71	0.64	9.90%
8	0.82	0.73	10.01%
9	0.94	0.84	10.60%
10	1.03	0.94	8.60%

Table 6.10: 99th percentile tail latency, in msec, of Memcached running on Linux and `UKL_RET_BYP (shortcut)`. Percentage improvement of `UKL_RET_BYP (shortcut)` over Linux is also shown. `UKL_RET_BYP (shortcut)` gets upto 10% tail latency improvement over Linux, even as the load on the Memcached server increases.

specializable systems, is capable of running complex multi-threaded applications and also provides performance improvement for them.

6.7 Key Takeaways

With UKL, we preserve Linux's application and hardware compatibility, along with its ecosystem of tools, utilities and community of developers and operators. UKL base model can be deployed wherever regular Linux can, and we have run many unmodified applications with it after a recompilation and relinking step. The changes required in Linux and `glibc` are minimal, which allows us to try for its upstream acceptance so the community can maintain and extend it. Although the performance improvements are negligible with the UKL base model, the real win is preserving the generality, something many specializable systems cannot achieve. As optimizations are added to UKL, the set of target applications shrinks, but the remaining applications experience increasing performance gains. Configuration-based optimizations can be turned on for many applications and provide decent gains. Significant performance improvements are possible by co-optimizing the application and the kernel together. Through this evaluation section, we have demonstrated that a general-purpose operating system can be specialized while preserving its application and hardware compatibility and the ecosystem around it.

Chapter 7

Conclusion

UKL explores the generality-specialization spectrum between general-purpose operating systems and specializable systems. Its most basic version links a target application with the Linux kernel, executes it in kernel mode, and replaces system calls with function calls. It shows minimal performance benefits but retains Linux's entire application and hardware compatibility, along with its battle-tested code base, community, and ecosystem of tools and utilities. Further, it integrates optimizations explored by specializable systems to Linux, e.g., faster transitions between application and kernel code, avoiding stack switches, run-to-completion modes, and allowing applications to bypass kernel state machines and directly call kernel functions. The total number of code changes to Linux and `glibc` are modest, and 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. Operators can configure and control UKL using the same tools they are familiar with, and developers can use standard Linux kernel tools like `eBPF` and `perf` to analyze their programs.

UKL has the word *unikernel* in its name but differs in several 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 preserving the often unwritten invariants of the Linux kernel. Second, UKL enables a knowledgeable 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.

While the set of changes to create UKL ended up being small, it has taken several years to get to this point. The design decisions are a result of multiple, typically much more pervasive, changes to Linux. The experience gained through those iterations resulted in several direction changes and learning how the desired optimizations could be integrated into Linux. Interestingly, the very modularity of Linux that enables a broad community to participate makes it very difficult to understand how to incorporate a change like UKL, but it can also be harnessed to enable the change in a very small number of lines of code.

This thesis is just the start of integrating performance optimizations in UKL. Through experience and knowledge of Linux, a series of simple optimizations that can be readily adopted have become apparent beyond the current efforts. For example, introducing and exploiting zero-copy interfaces to the applications, reducing some of the privacy assumptions implicit in the BSD socket interface when only one application consumes incoming data, etc.

These kernel-centric optimizations are just the start. From an application perspective, UKL will provide a natural path for improving performance and reducing the complexity of concurrent workloads. Often the burden falls onto the user code. It is hard to determine whether synchronization is needed from the user level, while the controlling 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.

This is just the beginning of this research direction, and the possibilities are limitless. UKL can serve as the go-to way applications are deployed in the cloud, starting with full compatibility of Linux and adding optimizations specific to the workloads. Developers would not have to choose between the compatibility benefits of Linux and the performance advantages of specializable systems. UKL can serve as a framework where new optimizations in Linux can be implemented and tested. One does not have to be a kernel developer; optimizations can be implemented as regular userspace code and linked into the kernel. UKL can also allow driver development in userspace and enable application developers to dive deep into the kernel and call functions directly instead of being limited to the system call API. UKL can be how specializable systems finally get mainstream adoption, ushering in the era of specialization while standing on the shoulders of generality.

Bibliography

[1] dead or alive: Linux libos project in 2016. https://github.com/thehajime/blog/issues/1. Accessed on 2021-10-7.

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

[3] ebpf. https://ebpf.io/. (Accessed on 1/18/2023).

[4] Efficient io with io_uring. https://kernel.dk/io_uring.pdf. (Accessed on 1/18/2023).

[5] Kernel virtual machine (kvm). https://www.linux-kvm.org/page/Main_Page. (Accessed on 11/25/2022).

[6] libevent – an event notification library. https://libevent.org/. (Accessed on 1/27/2023).

[7] memtier_benchmark: A high-throughput benchmarking tool for redis & memcached. https://redis.com/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/. (Accessed on 2/15/2023).

[8] Overview of the linux virtual file system. https://www.kernel.org/doc/html/latest/filesystems/vfs.html. (Accessed on 1/27/2023).

[9] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page. (Accessed on 1/27/2023).

[10] Qemu - a generic and open source machine emulator and virtualizer. https://www.qemu.org/. (Accessed on 11/25/2022).

[11] Re: [regression w/ patch] media commit causes user space to misbahave (was: Re: Linux 3.8-rc1). https://lkml.org/lkml/2012/12/23/75. (Accessed on 12/19/2022).

[12] [rfc ukl 00/10] unikernel linux (ukl). https://lore.kernel.org/lkml/20221003222133.20948-1-aliraza@bu.edu/. (Accessed on 12/28/2022).

[13] Solo5 - a sandboxed execution environment for unikernels. https://github.com/solo5/solo5. Accessed on 2021-10-7.

[14] X-containers. https://www.x-containers.org/home. (Accessed on 12/14/2022).

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

[16] Amazon. https://aws.amazon.com/ec2/nitro/, 2022. (Accessed 10/19/2022).

[17] 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.

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

[19] Jens Axboe. https://fio.readthedocs.io/en/latest/fio_doc.html. (Accessed on 10/13/2022).

[20] Arindam Banerji, John M Tracey, and David L Cohn. Protected shared libraries—a new approach to modularity and sharing. In *Proceedings of the 1997 USENIX Technical Conference*, pages 59–75, 1997.

[21] 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.

[22] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44, 2009.

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

[24] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged {CPU} features. In *10th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 12)*, pages 335–348, 2012.

[25] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. {IX}: A protected dataplane operating system for high throughput and low latency. In *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, pages 49–65, 2014.

[26] Redis Benchmark. https : / / redis . io / docs / management / optimization / benchmarks/. (Accessed on 01/28/2023).

[27] Brian N Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility safety and performance in the spin operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 267–283, 1995.

[28] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *2015 IEEE 7th international conference on cloud computing technology and science (cloudcom)*, pages 250–257. IEEE, 2015.

[29] 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.

[30] 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.

[31] 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.

[32] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. {FaRM}: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[33] Ulrich Drepper. Elf handling for thread-local storage. Technical report, Technical report, Red Hat, Inc., 2003. URL http://people. redhat. com . . . , 2005.

[34] Peter Druschel, Vivek S Pai, and Willy Zwaenepoel. Extensible kernels are leading os research astray. In *Proceedings. The Sixth Workshop on Hot Topics in Operating Systems (Cat. No. 97TB100133)*, pages 38–42. IEEE, 1997.

[35] 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.

[36] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The flux oskit: A substrate for kernel and language research. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pages 38–51, 1997.

[37] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[38] Benjamin Gamsa, Orran Krieger, Jonathan Appavoo, and Michael Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. In *OSDI*, volume 99, pages 87–100, 1999.

[39] Douglas P Ghormley, Steven H Rodrigues, David Petrou, and Thomas E Anderson. Slic: An extensibility system for commodity operating systems. In *USENIX Annual Technical Conference*, volume 98, 1998.

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

[41] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM Computer Communication Review*, 40(4):195–206, 2010.

[42] Rishabh Iyer, Katerina Argyraki, and George Candea. Performance interfaces for network functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 567–584, 2022.

[43] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. {mTCP}: a highly scalable user-level {TCP} stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, 2014.

[44] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for {μsecond-scale} tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, 2019.

[45] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter {RPCs} can be general and fast. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 1–16, 2019.

[46] Anuj Kalia, Michael Kaminsky, and David G Andersen. Using rdma efficiently for key-value services. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, pages 295–306, 2014.

[47] Anuj Kalia, Michael Kaminsky, and David G Andersen. {FaSST}: Fast, scalable and simple distributed transactions with {Two-Sided}({{{{{RDMA}}}}}) datagram {RPCs}. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 185–201, 2016.

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

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

[50] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. Chronos: Predictable low latency for data center applications. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–14, 2012.

[51] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[52] 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.

[53] Orran Krieger, Marc Auslander, Bryan Rosenburg, Robert W Wisniewski, Jimi Xenidis, Dilma Da Silva, Michal Ostrowski, Jonathan Appavoo, Maria Butrico, Mark Mergen, et al. K42: building a complete operating system. *ACM SIGOPS Operating Systems Review*, 40(4):133–145, 2006.

[54] 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.

[55] 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.

[56] 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.

[57] Redis Labs. https://redis.com/. (Accessed on 01/28/2023).

[58] Stefan Lankes, Simon Pickartz, and Jens Breitbart. Hermitcore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*, pages 1–8, 2016.

[59] 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.

[60] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. Secrecy: Secure collaborative analytics in untrusted clouds. *to apear NSDI 2023*, 2023.

[61] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. {MICA}: A holistic approach to fast {In-Memory}{Key-Value} storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[62] LKML. https://lkml.org/lkml/2022/10/3/1087, 2022. (Accessed 10/19/2022).

[63] 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.

[64] 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.

[65] Ilias Marinos, Robert NM Watson, and Mark Handley. Network stack specialization for performance. *ACM SIGCOMM Computer Communication Review*, 44(4):175–186, 2014.

[66] 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.

[67] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C Evans, Steve Gribble, et al. Snap: A microkernel approach to host networking. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 399–413, 2019.

[68] 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.

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

[70] 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.

[71] Amin Mosayyebzadeh, Apoorve Mohan, Sahil Tikale, Mania Abdi, Nabil Schear, Trammell Hudson, Charles Munson, Larry Rudolph, Gene Cooperman, Peter Desnoyers, and Orran Krieger. Supporting security sensitive tenants in a Bare-Metal cloud. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 587–602, Renton, WA, July 2019. USENIX Association.

[72] David Mosberger and Larry L Peterson. Making paths explicit in the scout operating system. In *OSDI*, volume 96, pages 153–167, 1996.

[73] Linux Weekly News. https://lwn.net/. (Accessed on 05/30/2022).

[74] Nginx. https://nginx.org/. (Accessed on 10/13/2022).

[75] 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.

[76] 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.

[77] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, 2019.

[78] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[79] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. Arrakis: The operating system is the control plane. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 1–16, 2014.

[80] 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.

[81] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 325–341, 2017.

[82] Calton Pu, Tito Autrey, Andrew Black, Charles Consel, Crispin Cowan, Jon Inouye, Lakshmi Kethana, Jonathan Walpole, and Ke Zhang. Optimistic incremental specialization: Streamlining a commercial operating system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 314–321, 1995.

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

[84] Ali Raza, Thomas Unger, Matthew Boyd, Eric Munson, Parul Sohal, Ulrich Drepper, Richard Jones, Daniel Bristot de Oliveira, Larry Woodman, Renato Mancuso, Jonathan Appavoo, and Orran Krieger. Integrating unikernel optimizations in a general purpose os, 2022.

[85] Redis. https://redis.io/. (Accessed on 05/30/2022).

[86] 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.

[87] Luigi Rizzo. netmap: a novel framework for fast packet i/o. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, 2012.

[88] 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.

[89] Margo I Seltzer, Yasuhiro Endo, Christopher A Small, and Keith A Smith. Issues in extensible operating systems. *Harvard Computer Science Group Technical Report TR-18-97*, 1997.

[90] Margo I Seltzer, Christopher A Small, and Keith Smith. The case for extensible operating systems. 1996.

[91] 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.

[92] Christopher A Small and Margo I Seltzer. Vino: An integrated platform for operating system and database research. 1994.

[93] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[94] Chia-Che Tsai, Donald E Porter, and Mona Vij. {Graphene-SGX}: A practical library {OS} for unmodified applications on {SGX}. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, 2017.

[95] Shin-Yeh Tsai and Yiying Zhang. Lite kernel rdma support for datacenter applications. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 306–324, 2017.

[96] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D.E. Long, and Carlos Maltzahn. Ceph: A Scalable, High-Performance Distributed File System. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[97] Dan Williams and Ricardo Koller. Unikernel monitors: extending minimalism outside of the box. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[98] Jisoo Yang, Dave B Minturn, and Frank Hady. When poll is better than interrupt. In *FAST*, volume 12, pages 3–3, 2012.

[99] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, pages 195–211, 2021.

[100] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. Kylinx: a dynamic library operating system for simplified and efficient cloud virtualization.

In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 173–186, 2018.

CURRICULUM VITAE