Integrating Unikernel Optimizations in a General Purpose OS

Ali Raza, Thomas Unger, Matthew Boyd, Eric Munson,
Parul Sohal, Ulrich Drepper, Richard Jones,
Daniel Bristot de Oliveira, Larry Woodman, Renato Mancuso,
Jonathan Appavoo, Orran Krieger

BOSTON UNIVERSITY

General Purpose Operating Systems

Advantages

- Rich application support
- Wide hardware support
- Vast ecosystem of tools, utilities
- Community of developers, operators and performance engineers

Limitations

- Designed for the general case
- Overhead of security and resource multiplexing

BOSTON UNIVERSITY

Unikernels

Advantages

- Designed to support a single application
- Optimizations include
  - no ring transition overheads
  - zero copy paths
  - custom scheduling and preemption policies
  - direct access to hardware etc.
- Lightweight and resource efficient

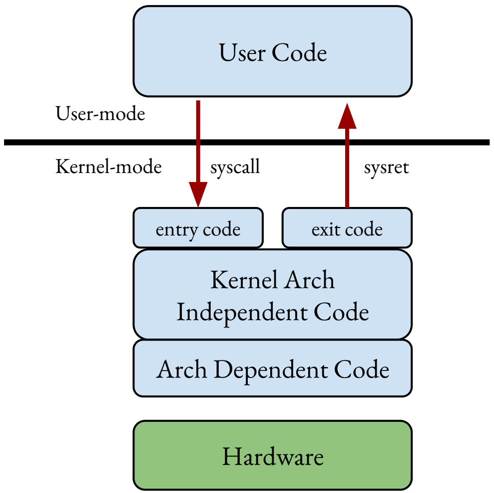
Limitations

- Small or non-existent community
- Lack of application and hardware support
- Mostly virtualized, single processor deployment
- Lack of support for utilities and tools
- Untested code base

BOSTON UNIVERSITY

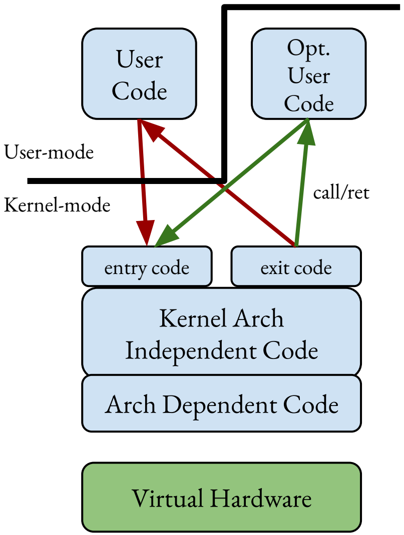
Research Space

General Purpose OS



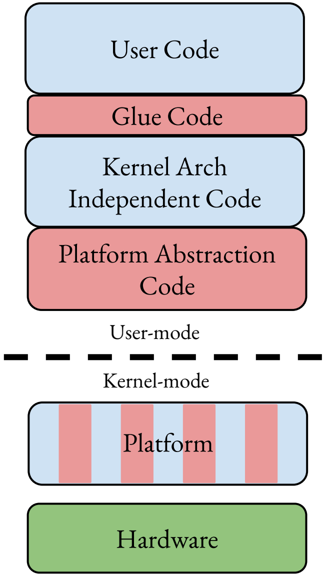
Linux, Windows, NetBSD etc.

Incremental Systems



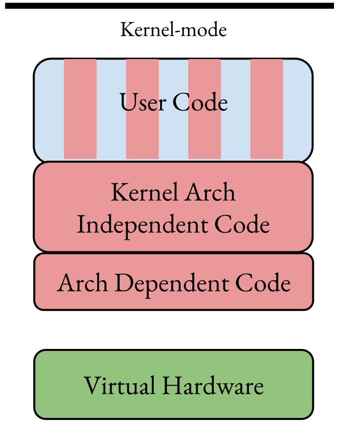
KML, Lupine Linux, X-Containers

Forks of General Purpose OS



LKL, Drawbridge, Rump Kernel

Clean Slate Unikernels

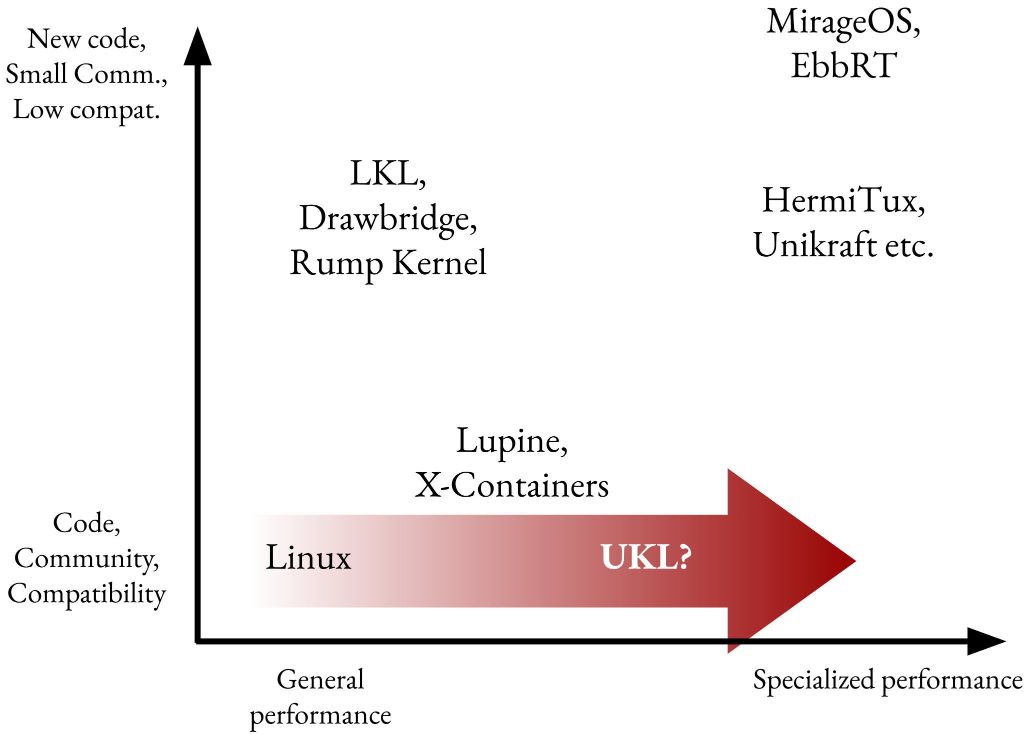


MirageOS, EbbRT, HermiTux, Unikraft etc.

Old code
New code

Motivation

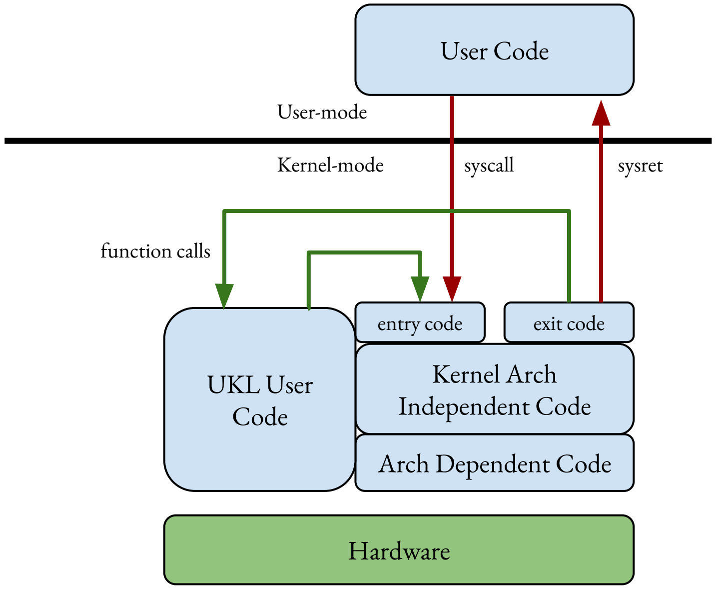
- Is it possible to integrate unikernel optimizations in a general purpose OS without forking the code base?
- Would it be possible to preserve the battle tested code, development community, application and hardware support?
- Would there be any performance benefit?



BOSTON UNIVERSITY

Design: UKL Base Model

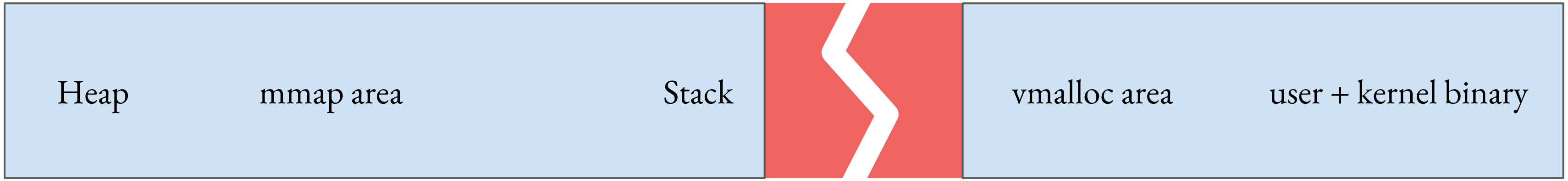
- Unikernel-aware version of Linux
- Geared towards functionality and stability, not necessarily performance
- Like unikernels
  - link user code with kernel code
  - replace syscalls with function calls
- Unlike unikernels
  - preserve application and hardware compatibility
  - preserve support for multiple processes
  - preserve address space layout
  - maintain distinct execution models for user code and kernel code



BOSTON UNIVERSITY

Design: UKL Base Model

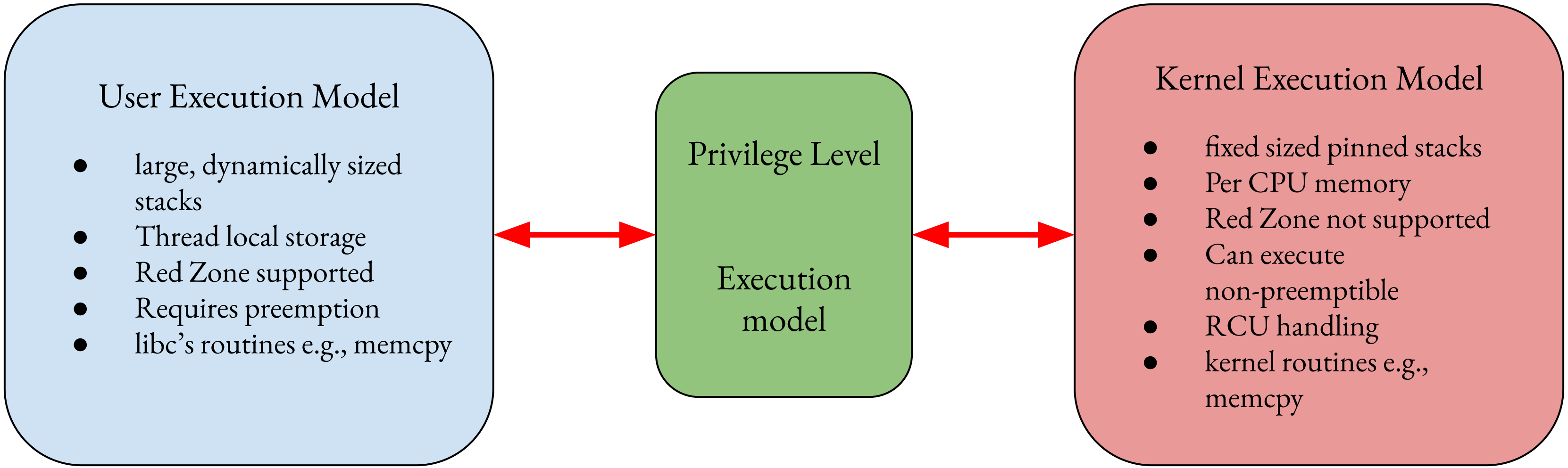
- Preserve address space layout
- unmodified user and kernel memory allocators
- UKL user ELF binary is loaded with the kernel



BOSTON UNIVERSITY

Design: UKL Base Model

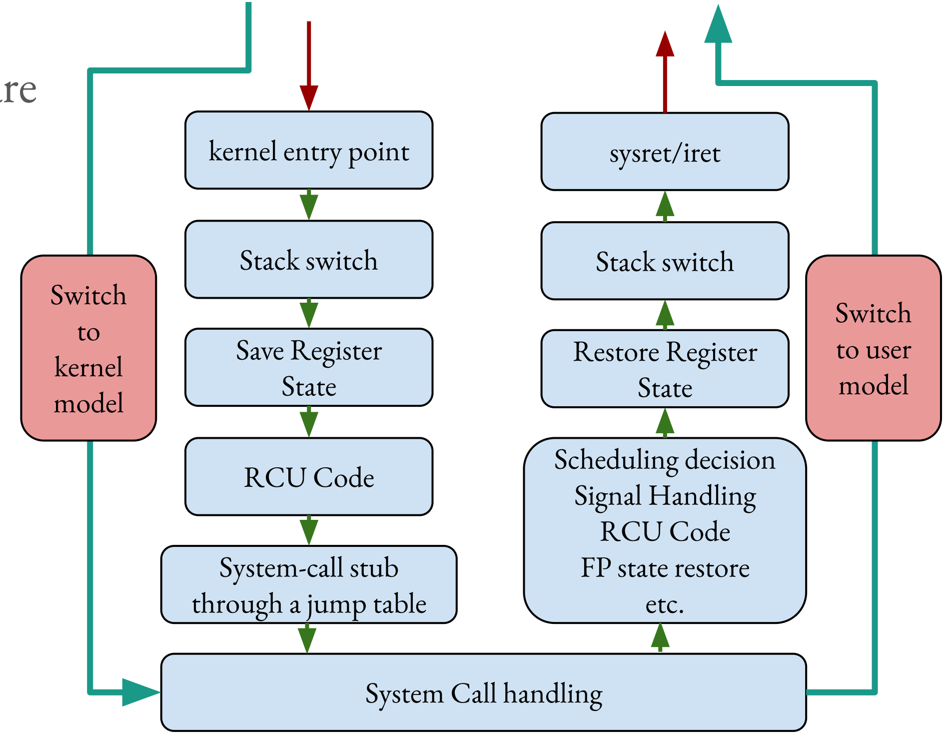
- Distinct execution models
- Decouple execution model from privilege level



BOSTON UNIVERSITY

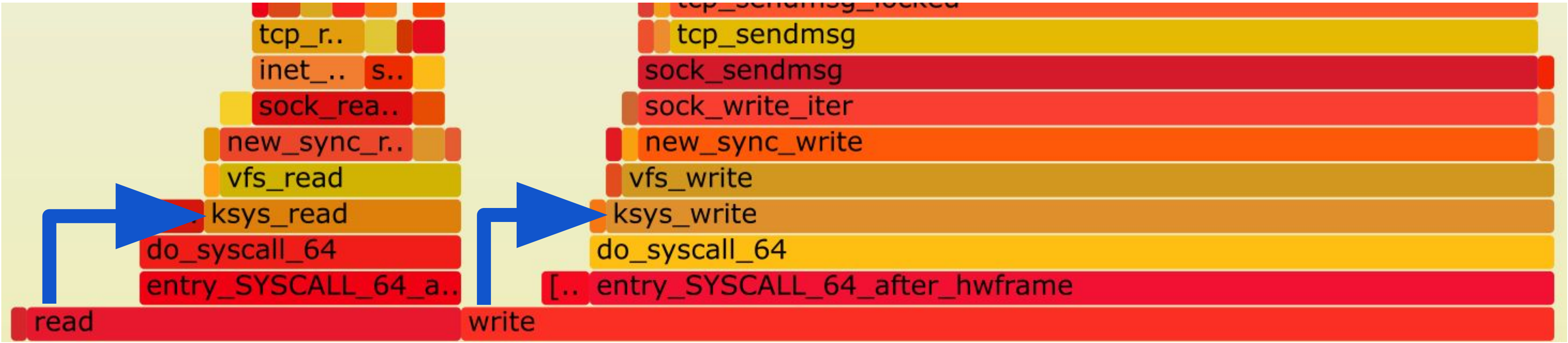
Unikernel Optimizations: Bypassing entry/exit code

- transitions between kernel and user code are expensive
- a configuration option to skip transition code
- do so while keeping a separation between execution models
- ability to selectively execute functionality e.g., stack switch, signal handling etc.
- automatic normal path execution after a number of bypasses



BOSTON UNIVERSITY

Unikernel Optimizations: Bypassing entry/exit code



BOSTON UNIVERSITY

Unikernel Optimizations: Avoid Stack Switches

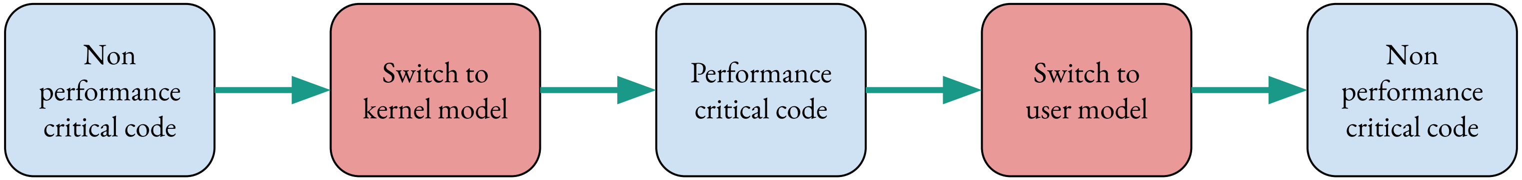
- Assembly code for transition breaks compiler view
- Stops cross layer optimization e.g., LTO
- Avoid stack switches, keep user stack for user and kernel code
- Kernel code cannot take page faults
- User stacks can get page faults
- Stack page fault aware fault handler
- Or use shared kernel stacks

Unikernel Optimizations: `ret` versus `iret`

- `iret` is used for interrupts, exceptions and faults
- an expensive instruction compared to `ret`
- but `iret` guarantees atomicity to
  - switch the stack
  - ring transition
  - update instruction pointer
  - restore flags
- use `ret` instead, while ensuring atomicity

Unikernel Optimizations: Kernel mode execution

- Kernel can run non-preemptible
- Applications can enter kernel mode of execution
- Allows ‘run-to-completion’

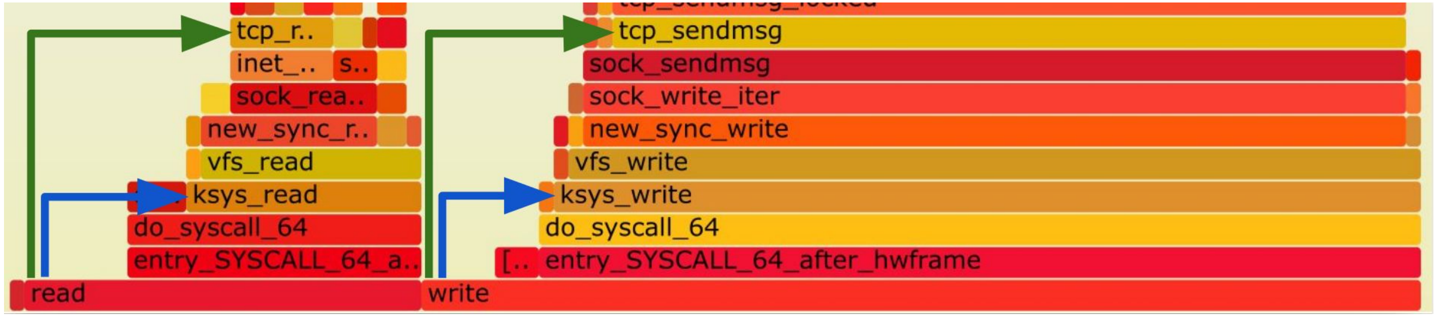


Unikernel Optimizations: Calling kernel routines

- Applications can call any internal kernel functions
- E.g., calling kernel memory allocator instead of user ones
- `vmalloc()` allocates pinned memory from kernel address range

Unikernel Optimizations: Deep shortcuts

- Instead of calling pre-written kernel functions
- Add application specific custom code to the kernel
- Use application knowledge to create custom paths in the kernel



BOSTON UNIVERSITY

Implementation

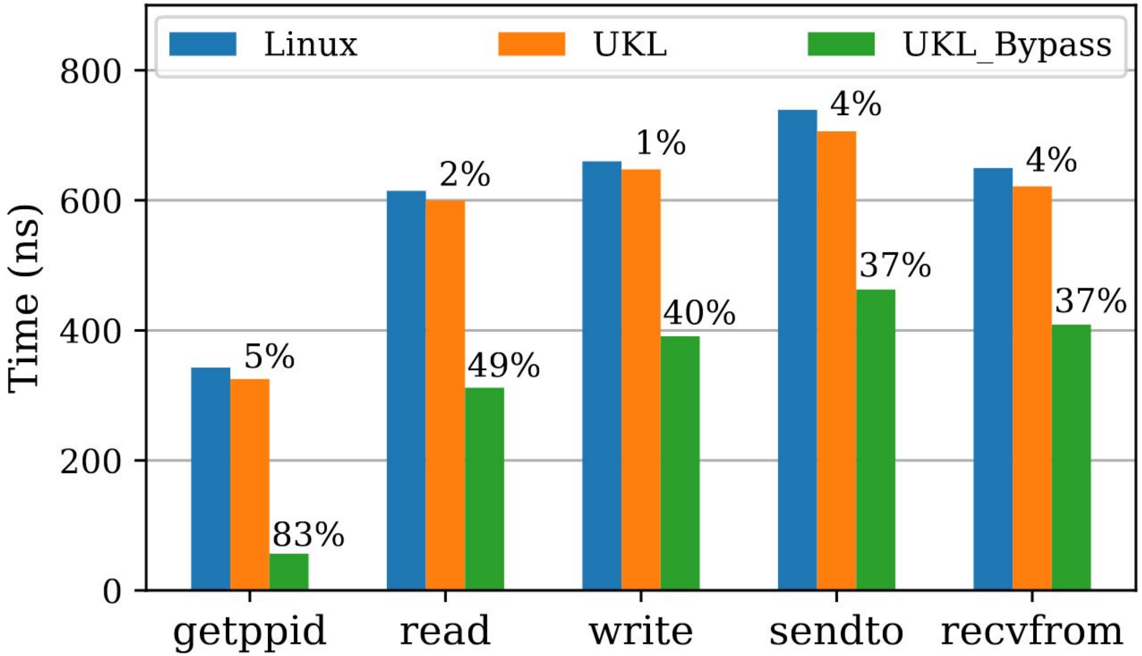
- Modifications to Linux and glibc
- Currently targets x86_64 architecture
- All code is protected by #ifdefs, so UKL can be configured out
- Prefixing all user symbols with `_ukl` to avoid name collisions
- Changes to bootstrapping, process creation and initialization paths
- Execution model tracking in transition code
- UKL base model ~550 LoC modified in Linux
- Total UKL patch to Linux is less than 1400 LoC

BOSTON UNIVERSITY

Evaluation: Code, Community and Compatibility

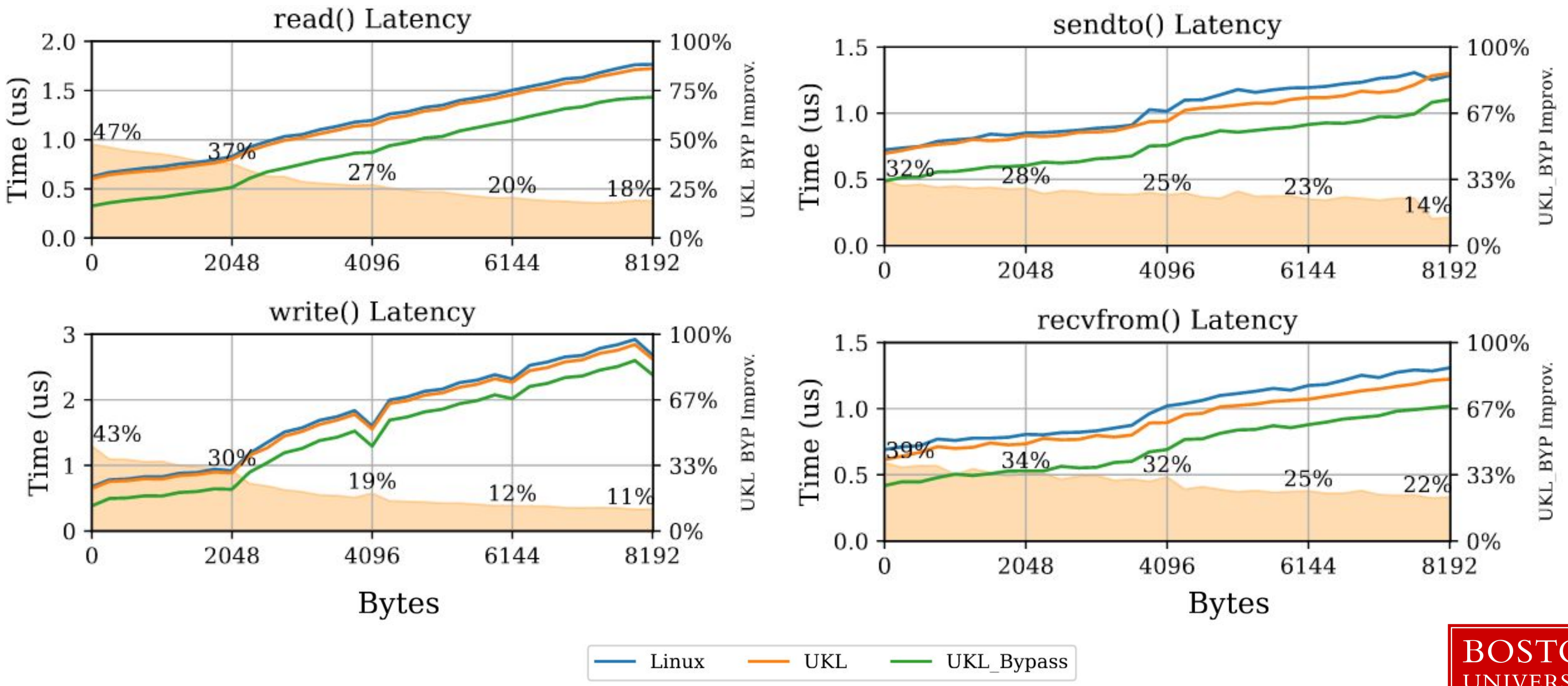
- Application support
- Hardware Support
- Ecosystem Support
- Do not support exec, dynamic loader or proprietary pre-compiled binaries

Evaluation: System call latency

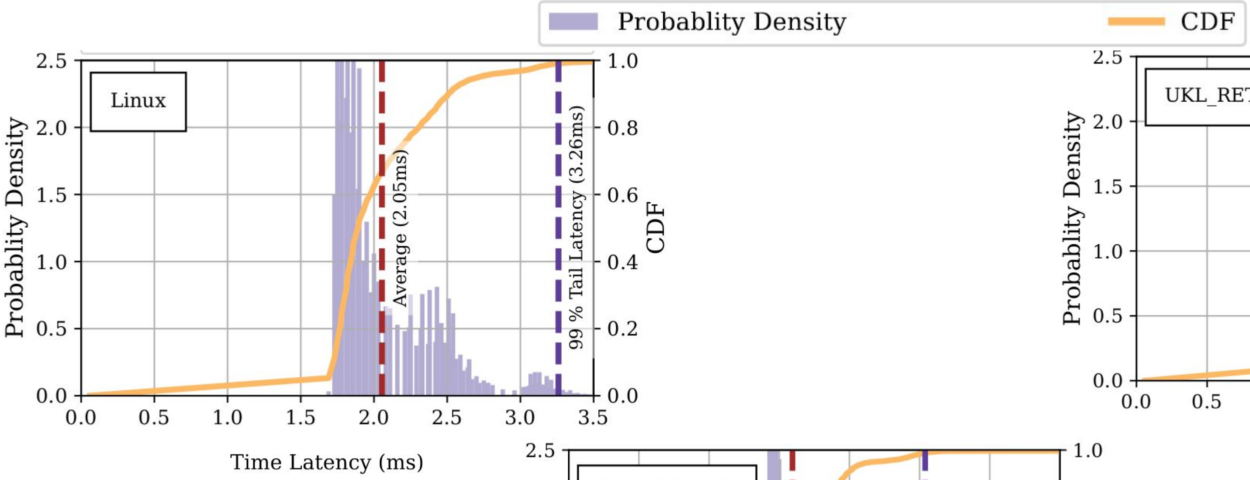


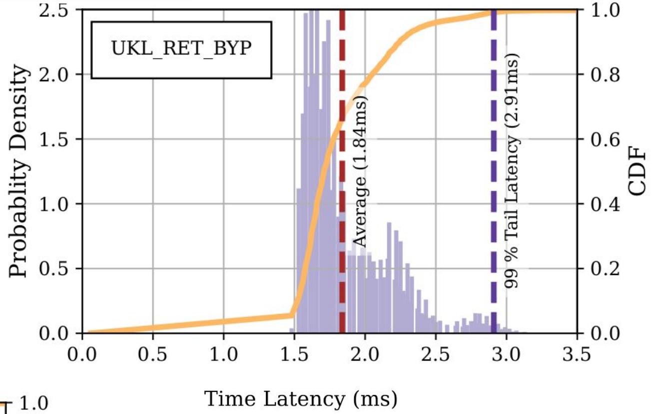
BOSTON UNIVERSITY

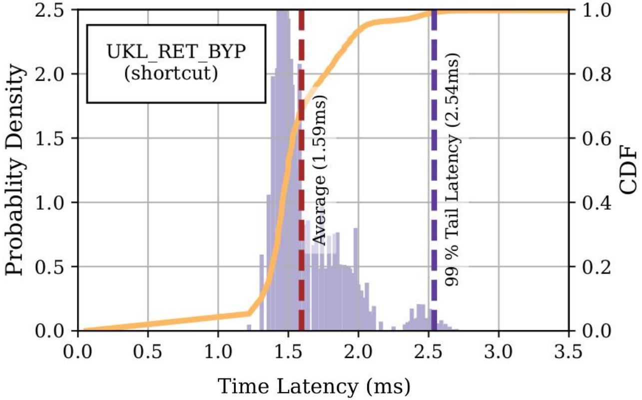
Evaluation: System call latency with larger payloads



Evaluation: Performance







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

BOSTON UNIVERSITY