Exploring Unikernel: An Empirical Comparison with Linux

Taekyung Kang, Kyungha Kim

8/12/2026

Presenters

- **Taekyung Kang**
  - Software Engineer at Boeing (Current)
    - Boeing OS
    - Previously: RBT
  - Systems Engineer at Agency for Defense Development
    - Unmanned Reconnaissance Vehicle systems
  - M.S. in Neuro-Electronics Engineering Lab, Ewha University
  - Lives in South Korea
- **Kyungha Kim**
  - Software Engineer at Boeing (Current)
    - Boeing OS
    - Previously: RBT
  - Researcher at Agency for Defense Development
    - HILS for Missile systems
  - M.S. in Astrodynamics and Control Lab., Yonsei University
  - Based in Seoul, South Korea

Outline

- Motivation
- Unikraft Overview
- Experimental Setup
- Empirical Comparison
- Conclusion
- Q&A

Motivation

The Paradigm of General-Purpose OS in Embedded Systems

- **The Strength of Linux**
  - Established ecosystem with full **POSIX compliance.**
  - Reliable performance for a **broad range of applications.**

- **Hidden Costs:**
  - **Large Footprint**: Hundreds of **unnecessary drivers and background services** for specialized tasks. Excessively **large binary sizes** for **resource-constrained** hardware.
  - **Performance Overhead**: Latency from frequent **User-to-Kernel transitions** (Expensive Syscalls) impacting time-sensitive execution.

- *“Can we eliminate these architectural costs while maintaining functionality?”*

→ This leads us to explore **Unikernels** - Application-Specific Operating Systems.

What is a Unikernel?

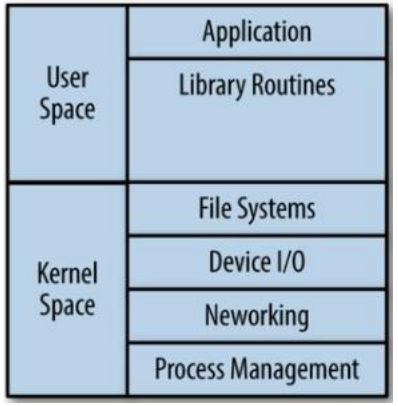
Unikernels

Unikernels take minimalism to the next level.

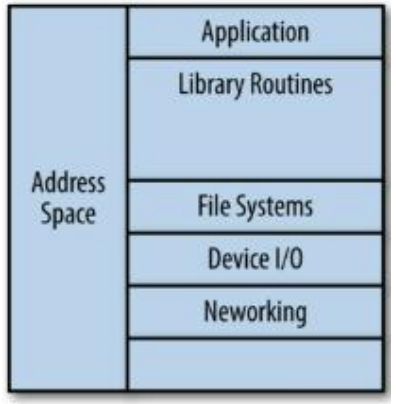
A unikernel is a **single-purpose, single-address-space** image that includes:

- Only the required parts of the OS.
- The application logic.
- No shells, no package managers, no unused ports.

> This makes them **ultra-secure** (smaller attack surface), blazing fast to **boot**, and extremely **lightweight**.



Traditional OS Architecture



Unikernel Architecture

In the unikernel model, things are drastically simplified.

There is no separation between user space and kernel space — instead, both the application and only the essential parts of the OS are compiled into a **single binary** that runs directly on hardware or a hypervisor.

Source: Manish Pillai, DEV Community (2025)

Unikernel Landscape

Feature	MirageOS	IncludeOS	OSv	Unikraft
Main Language	OCaml (Incompatible with C)	C++ (Requires C++ environment)	C++-based	**C/C++** (with support for additional language runtimes)
Supported Environments	Xen, KVM, Solo5	KVM, VirtualBox	Xen, KVM, VMware	**Xen, KVM, Firecracker, QEMU**
API & POSIX Compliance	Non-POSIX (Custom OCaml APIs)	Limited, custom C++ libraries	**POSIX-oriented**	**POSIX-oriented**
Architectural Modularity	Package-based	Source-level static linking	Relatively low (Monolithic structure)	**High modularity**
Legacy App Compatibility	Low (Requires total rewrite in OCaml)	Low (Restricted to modified C++)	Designed to support existing workloads	**Suitable for POSIX-oriented applications;** porting effort depends on workload
Project Status	Academic research project	Less active since 2021	Stable but legacy maintenance	**Highly Active (Linux Foundation project)**

This study was conducted using **Unikraft**.

https://uradical.io/latest-news/unikernels-in-2026

Unikraft Overview

Modular Build for Application-Specific Efficiency

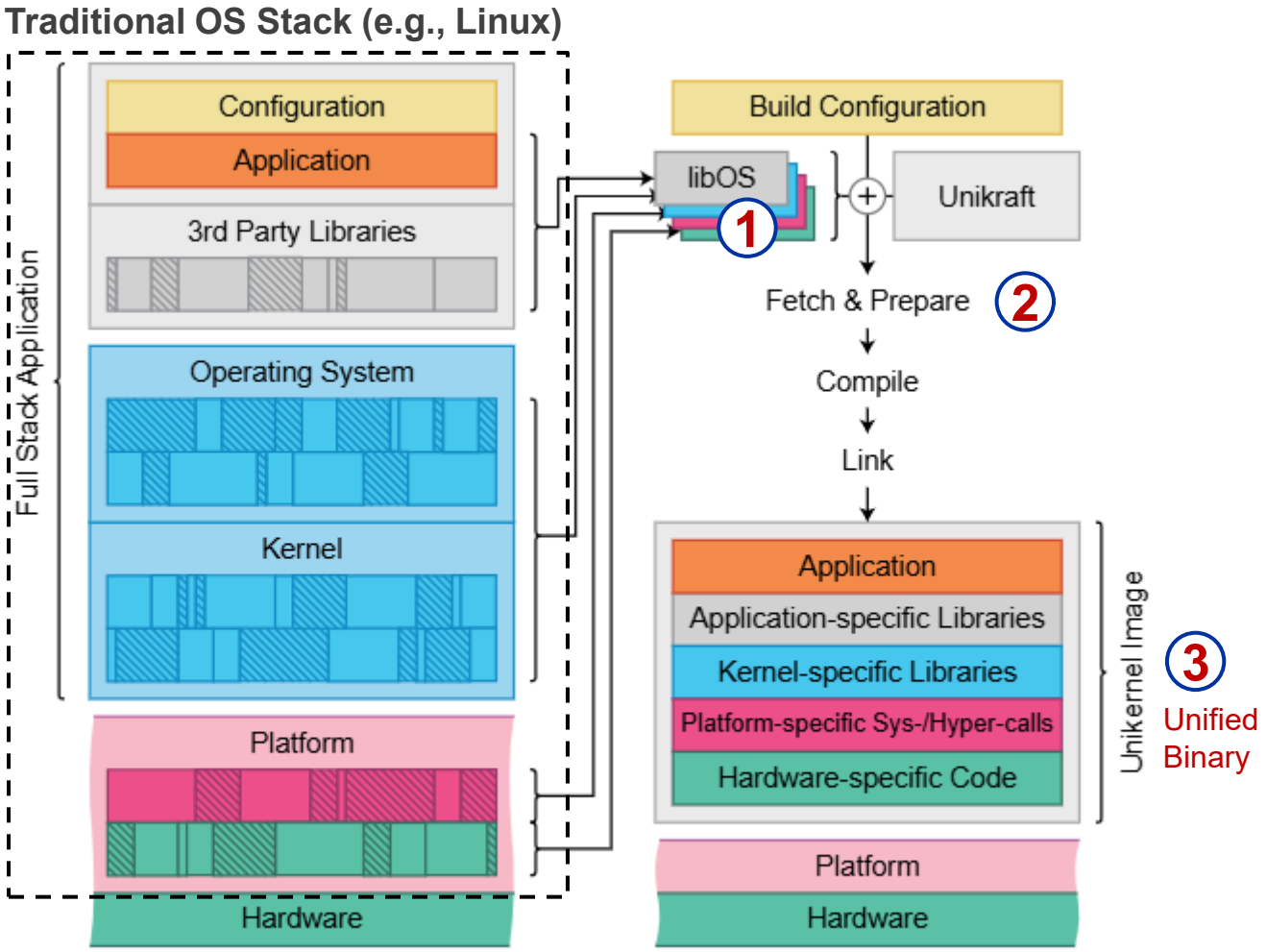
1. Modular Build (Kconfig-driven)

- **Micro-libraries**: OS as independent components.①
- **Selective Binding**: Compiles only essential code via Kconfig (make menuconfig). ②
- **Zero-Bloat**: Eliminates unused drivers and services (Size Reduction).

2. Flattened Architecture ③

- **Single Address Space**: Links App + OS into a **unified binary**.
- **No Boundaries**: Removes User-Kernel separation for a direct execution path.

*"To Verify the impact of Unikraft's modular and flattened architecture, we compare **binary sizes** and measure **execution times** in this study."*



Overview of the Unikraft build process.

(Image from https://unikraft.org/docs/internals/build-process)

Unikraft Micro-libraries

- **Unikraft repository structure**

```
unikraft/
├── Config.uk              ← Top-level Kconfig
├── Makefile               ← Main Makefile
├── Makefile.uk            ← Global compile flags
├── lib/                   ← Core micro-libraries
│   ├── uksched/
│   ├── ukboot/
│   ├── syscall_shim/
│   ├── posix-process/
│   ├── posix-time/
│   └── ...
├── plat/                  ← Platform micro-libraries
│   ├── xen/
│   ├── kvm/
│   └── ...
├── drivers/               ← Driver micro-libraries
│   ├── virtio/
│   ├── xen/
│   └── ...
└── arch/                  ← Architecture support
    ├── arm64/
    ├── x86_64/
    └── ...
```

- **Micro-library structure**

  (common pattern for all libraries)

```
<library-name>/
├── Config.uk
├── Makefile.uk
├── *.c
└── exportsyms.uk
```

Each library is independently compiled and selected at build time

→ Only the libraries required by the application can be included in the final image

Unikraft Micro-libraries

```
<library-name>/
  ├── Config.uk
  ├── Makefile.uk
  ├── *.c
  └── exportsyms.uk
```

- **Example: unikraft/lib/posix-time**

– **[Config.uk]** Defines build option and dependencies

```
config LIBPOSIX_TIME
       bool "posix-time: Time syscalls"
       default n
       select HAVE_TIME
       select LIBUKLCPU
```

– **[Makefile.uk]** Register sources and syscall entries

```
$(eval $(call addlib_s,libposix_time,$(CONFIG_LIBPOSIX_TIME)))

CINCLUDES-$(CONFIG_LIBPOSIX_TIME)    += $(LIBPOSIX_TIME_COMMON_INCLUDES-y)

LIBPOSIX_TIME_SRCS-y += $(LIBPOSIX_TIME_BASE)/time.c
LIBPOSIX_TIME_SRCS-y += $(LIBPOSIX_TIME_BASE)/timer.c

UK_PROVIDED_SYSCALLS-$(CONFIG_LIBPOSIX_TIME) += clock_gettime-2
UK_PROVIDED_SYSCALLS-$(CONFIG_LIBPOSIX_TIME) += nanosleep-2
UK_PROVIDED_SYSCALLS-$(CONFIG_LIBPOSIX_TIME) += gettimeofday-2
```

– **[time.c]** Source code, including syscall handlers

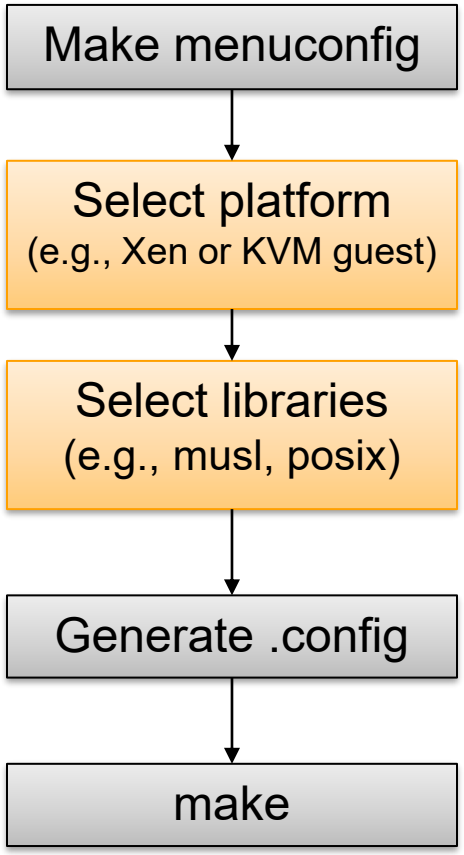
```
int uk_sys_clock_gettime(clockid_t clk_id,
                         struct timespec *tp)
{
    __nsec now;

    switch (clk_id) {
    case CLOCK_MONOTONIC:
        now = ukplat_monotonic_clock(); break;
    ...
    }

    tp->tv_sec = ukarch_time_nsec_to_sec(now);
    tp->tv_nsec = ukarch_time_subsec(now);
    return 0;
}

UK_SYSCALL_R_DEFINE(int, clock_gettime, clockid_t, clk_id,
                    struct timespec*, tp)
{
    return uk_sys_clock_gettime(clk_id, tp);
}
```

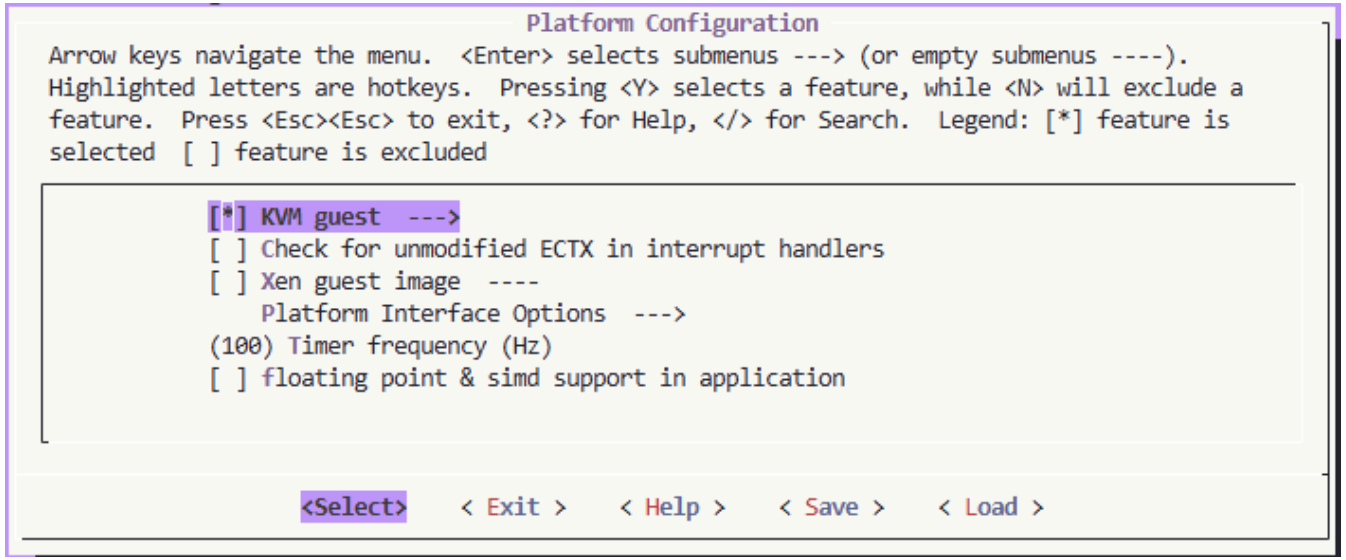
– **[exportsyms.uk]** Declares externally visible symbols

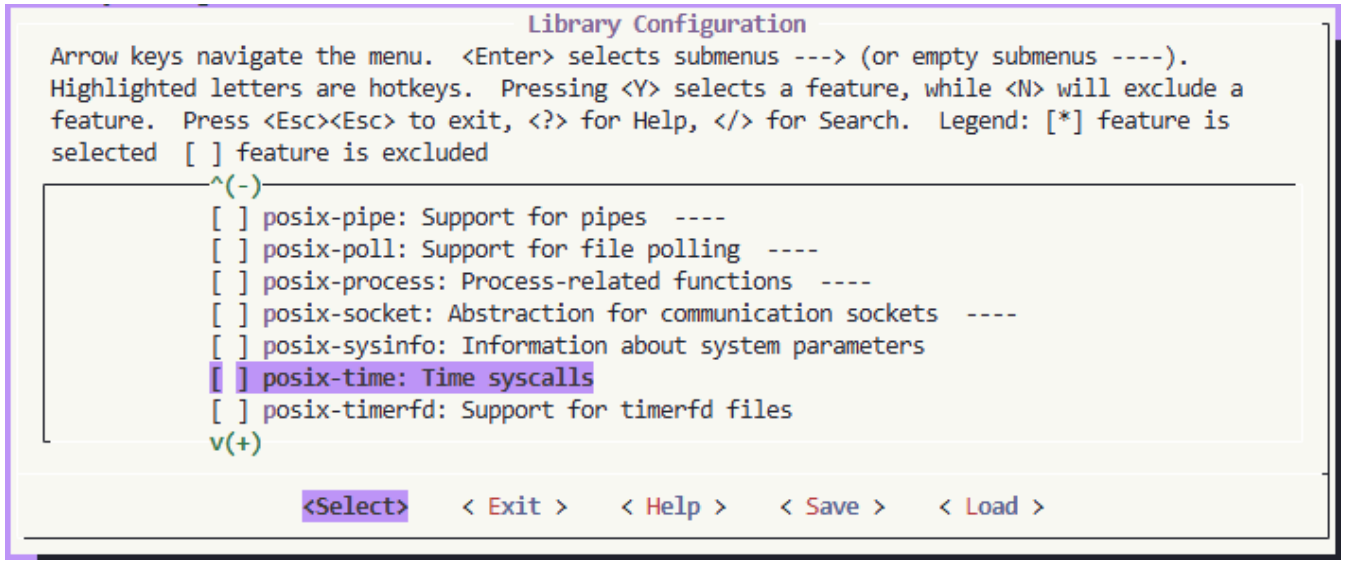
```
uk_sys_nanosleep
uk_sys_clock_gettime
uk_sys_gettimeofday
clock_gettime
nanosleep
```

Seletive Binding in Unikraft

- **Unikraft build workflow**







Run-time Efficiency via Syscall Shimming

- **Native Compatibility: Musl Libc Glue**
  - Supports standard POSIX/C for existing Linux apps.
  - Minimal to No Source Modification
  - *(Note: In this study, we used identical source code for a fair comparison)*

- **Direct Execution: Syscall Shim Layer**
  - Intercepts standard syscalls at runtime.
  - Bypasses expensive CPU traps (User ↔ Kernel)

- **Shortest Path: Direct Function Calls**
  - Routes directly to Unikraft Micro-libraries

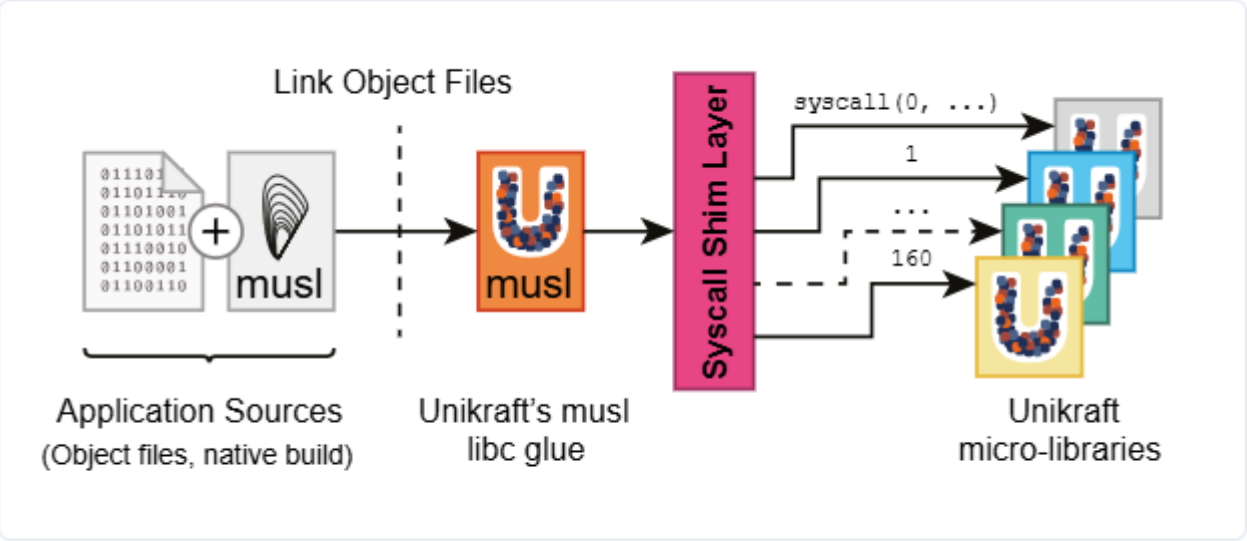
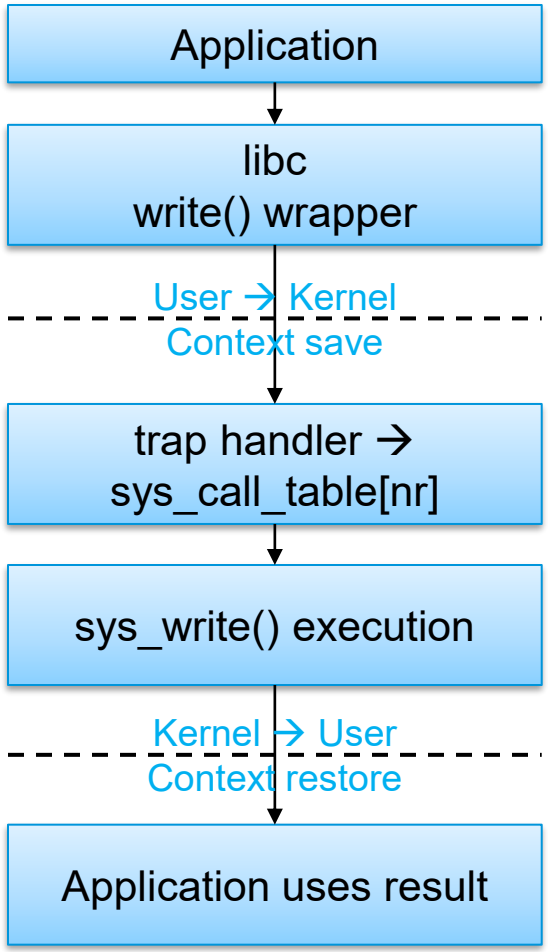


Figure 1: Unikraft avoids porting of applications by building them using their native build systems with musl and linking the object files into the Unikraft build.

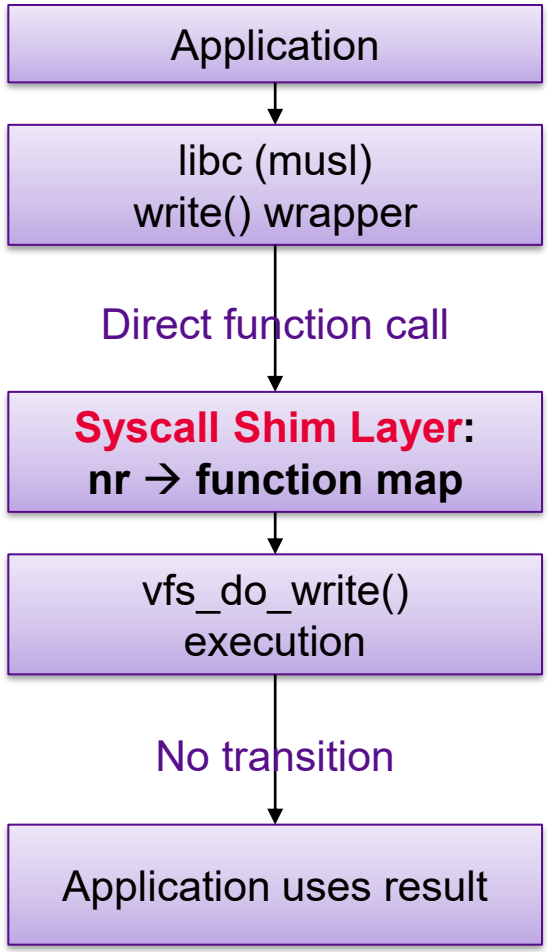
(Image from https://unikraft.org/docs/concepts/compatibility)

Syscall Processing – Linux vs Unikraft

- **Linux path**



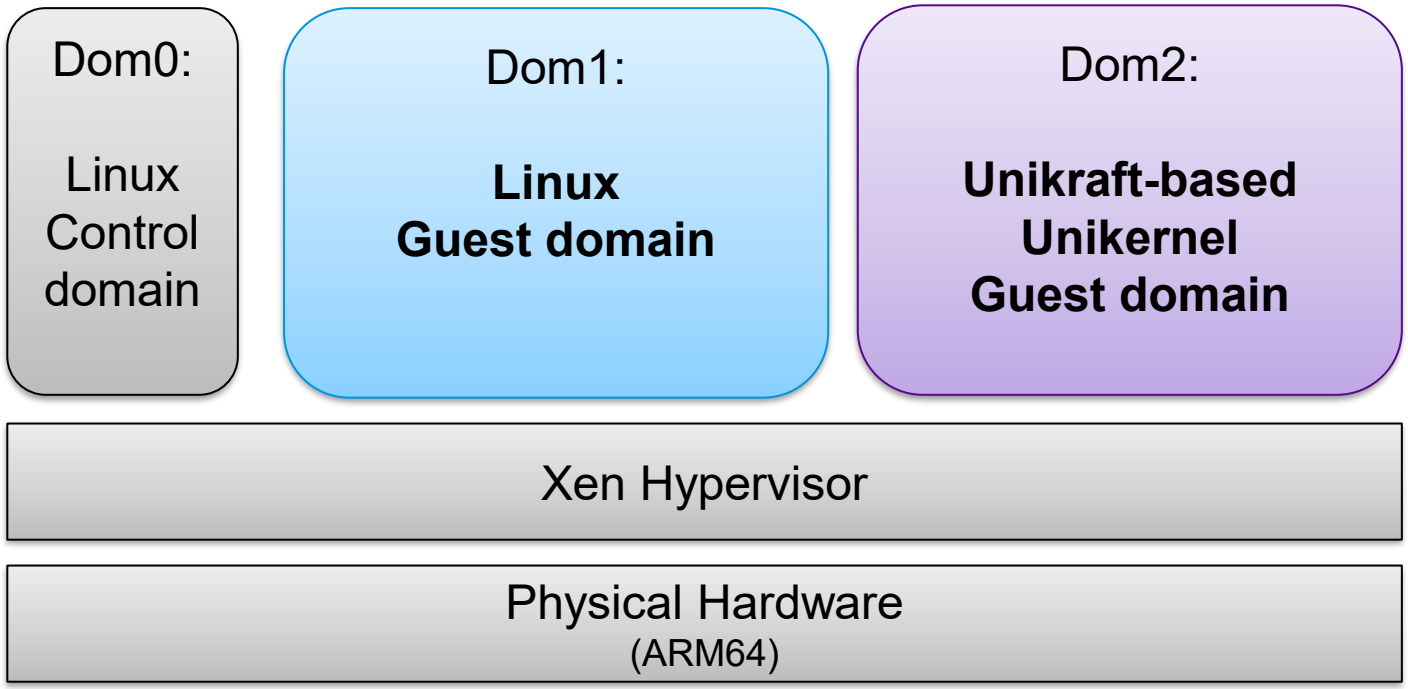
- **Unikraft path**



Linux resolves syscalls at runtime through **kernel traps**, whereas **Unikraft** preconfigures the mapping and handles calls through a lightweight **shim.**

Experimental Setup

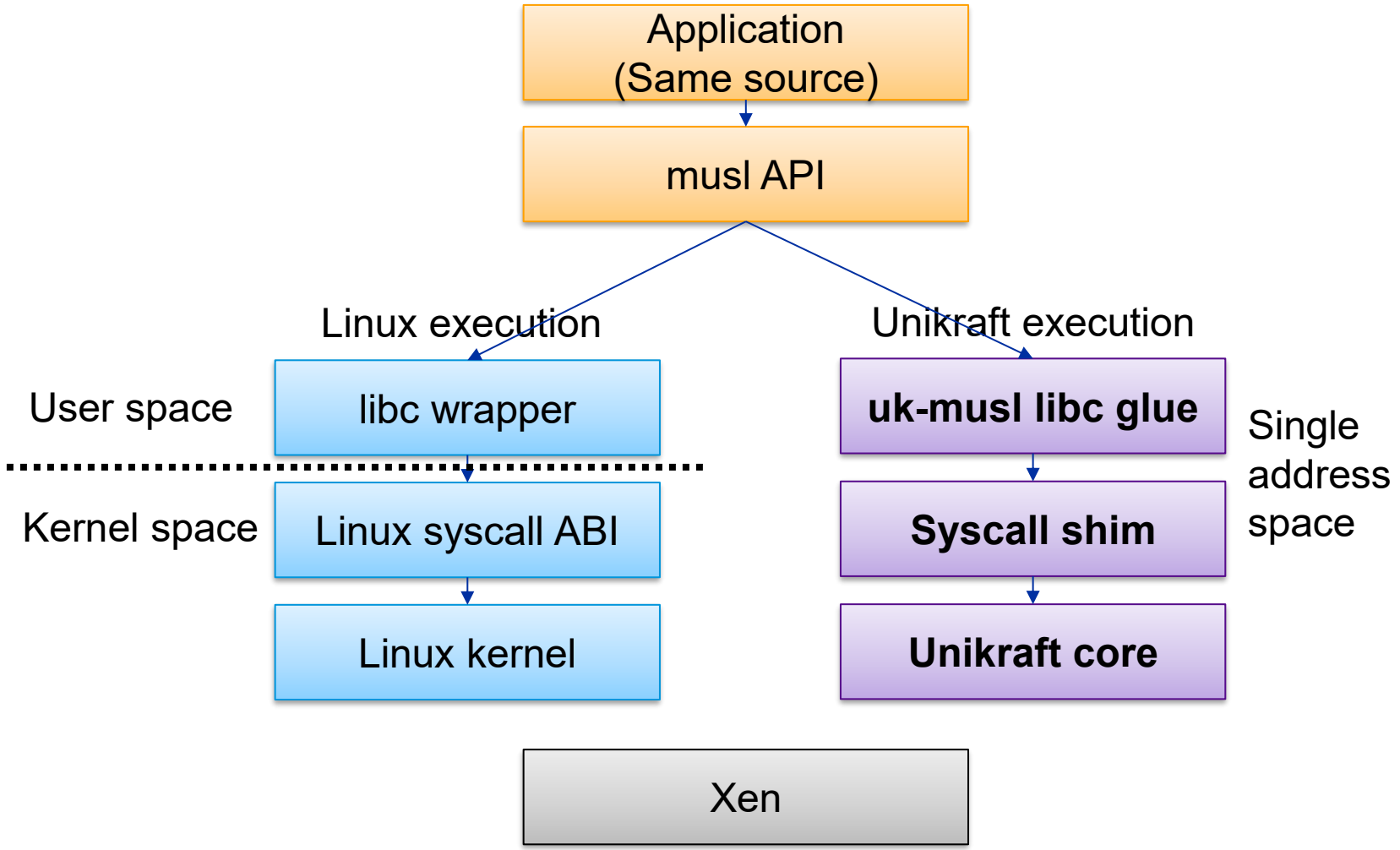
Experimental Setup:
Xen-based Linux vs. Unikraft Unikernel



- **Dom1: Linux Guest Domain**
  - Full Linux OS stack
  - Kernel + standard user space
- **Dom2: Unikraft Guest Domain**
  - Minimal Unikernel stack
  - Application-specific components only
- **Dom0: Linux Control Domain**
  - Privileged domain for hardware access and VM management

Both domains ran the same test application on the same Xen platform

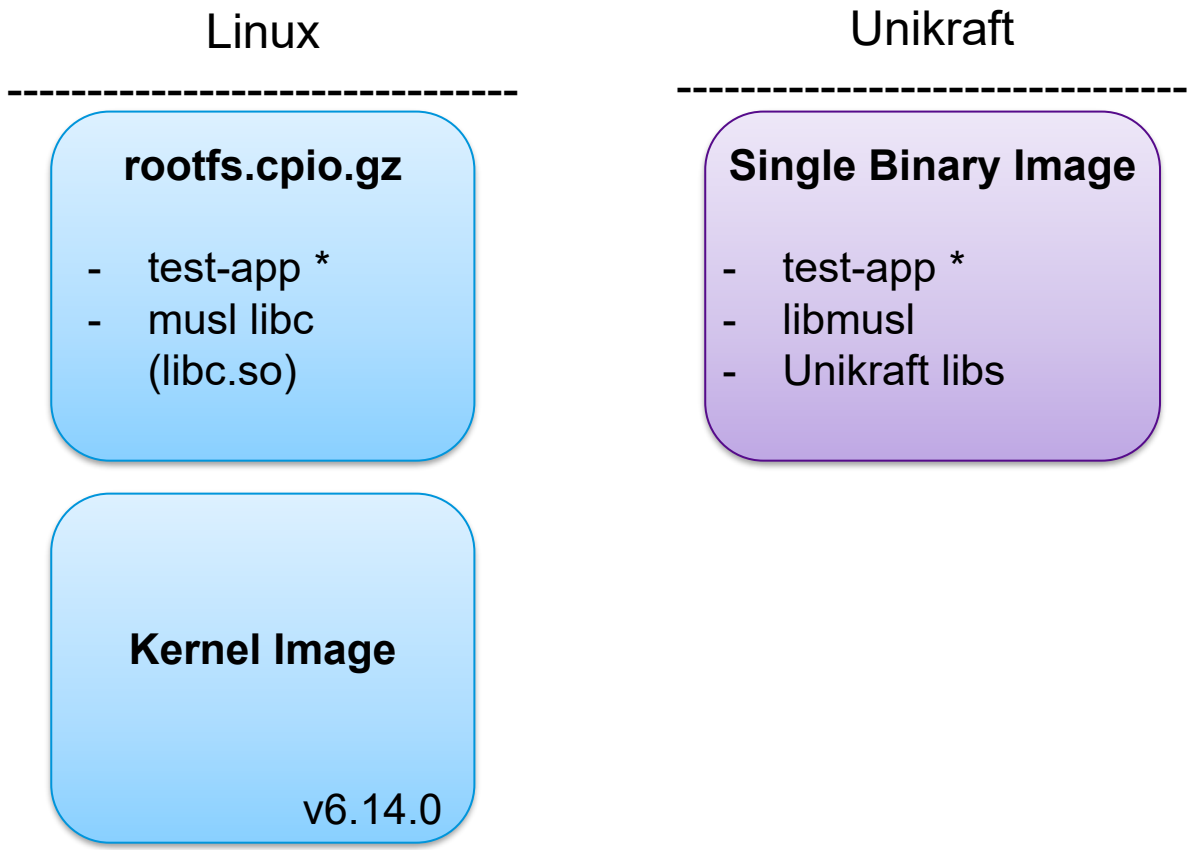
Experimental Setup: Execution Stack Comparison



- **Linux**
  - User/Kernel space isolation
  - Context switching via Syscall ABI
- **Unikraft**
  - Single address space; No boundaries
  - No mode switching: Replaces Syscall with direct function call (via Syscall shim)
  - Includes only essential Unikraft core components.

Note. musl is commonly used in unikernel environment due to its small size, simple design, and suitability for static linking.

Experimental Setup:
Build Environment & Artifacts Comparison



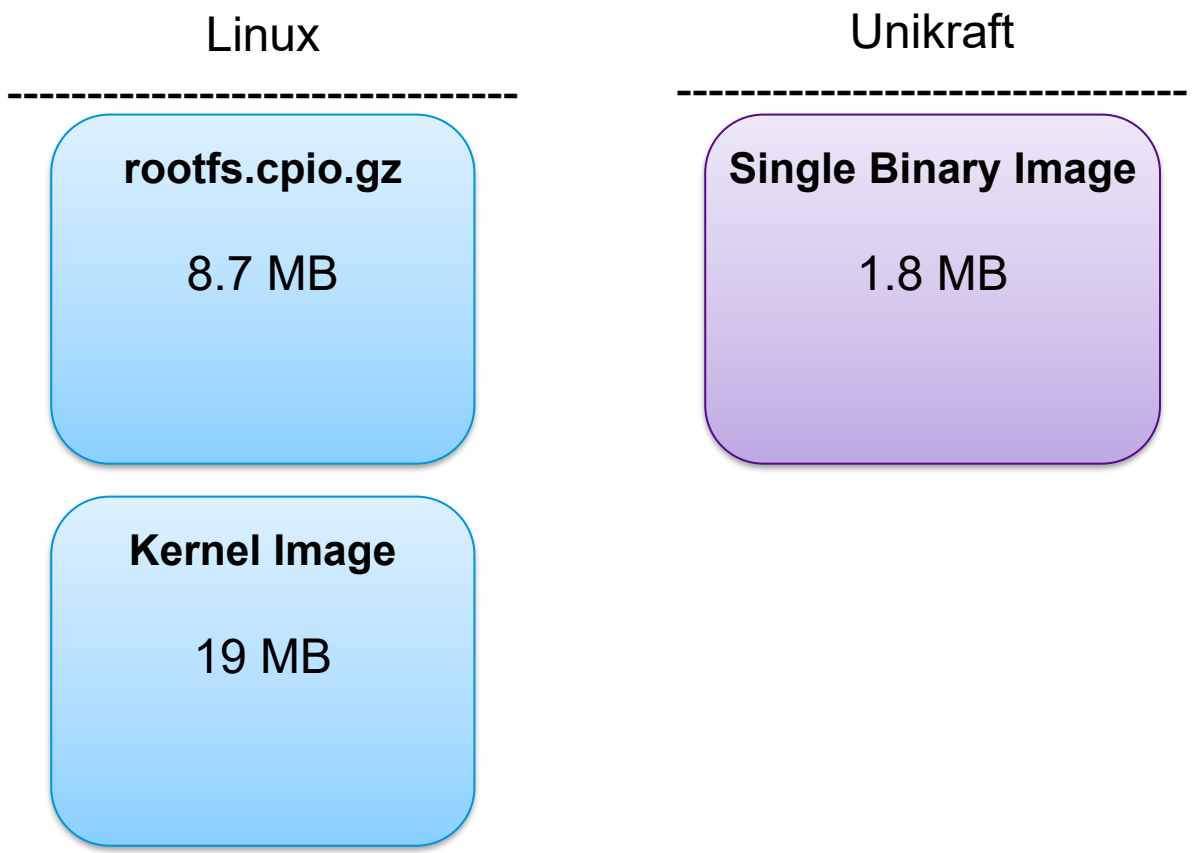
* test-app: Utilized identical C source code

- **Linux**
  - Build tool: Buildroot
  - musl version: 1.2.5
  - Compiler: GCC 14.2.0
  - Output: Kernel Image + root filesystem
- **Unikraft**
  - Build tool: Unikraft (v0.20.0)
  - musl version: 1.2.3
  - Compiler: GCC 14.2.0
  - Output: Single Binary Image

Empirical Comparison

- Build Time & Image Size
- libc Execution Latency

Build Time & Image Size



- **Linux**
  - Build Time: 10m33s
  - Image Size: 19 MB + 8.7 MB = 27.7 MB
- **Unikraft**
  - Build Time: 22s
  - Image Size: 1.8 MB

~28.8x Faster Build Time

~15.4x Smaller Binary Footprint

libc Execution Latency: Methodology

1. Measurement

- **Execution latency of libc functions** across Linux and Unikraft guest domains using clock_gettime()

```
for (int i = 0; i < ITERATIONS; i++) {
    clock_gettime(CLOCK_MONOTONIC, &start);
    pid = getpid();
    clock_gettime(CLOCK_MONOTONIC, &end);

    getpid_times[i] = diff_ns(start, end);
```

Per-iteration measurement
- Single-threaded environment
- ITERATIONS = 5,000
- Avg/Max/Min/Std

- **Target libc functions**
  - Syscall-based: getpid, mmap, write, read
  - User-space only: clock_gettime*, strncpy, malloc

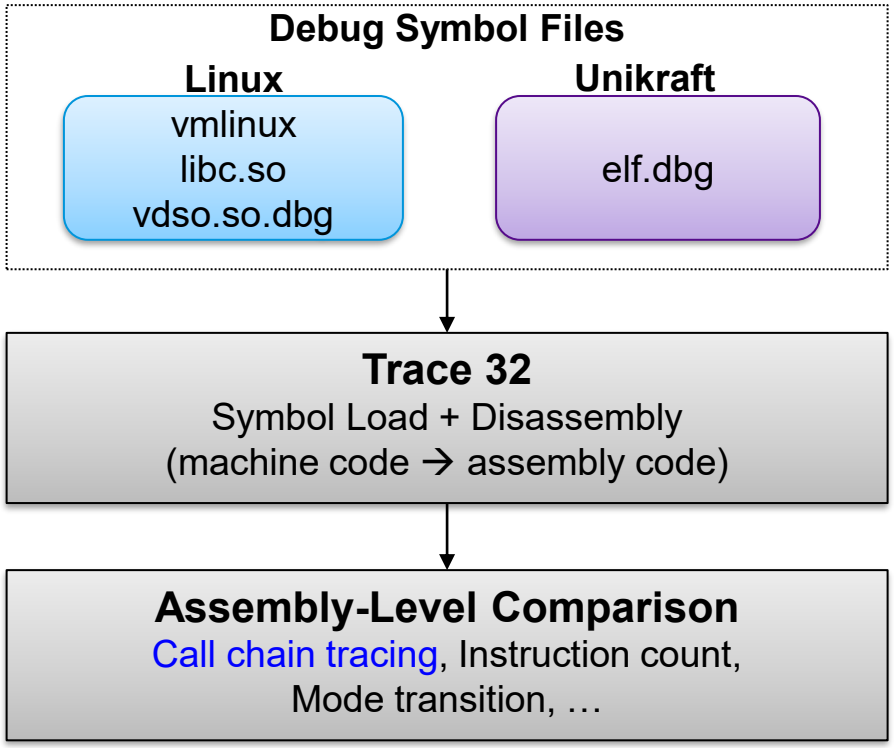
- **Data Refinement**
  - Excluded top 1% outliers (P99 cut-off) from the dataset.
  - To eliminate non-deterministic background noise (e.g., hardware interrupts, kernel background tasks).

libc Execution Latency: Methodology

2. Analysis: assembly-level code inspection

- **Object**
  - Identify root causes of latency differences between Linux and Unikraft

- **Method**



libc Execution Latency: clock_gettime()

1. Execution Latency

[ns]	Linux	Unikraft
avg	69	120

– Unikraft: **1.7x slower** execution latency

Note: Most libc functions were measured per iteration; **clock_gettime()** was measured using total elapsed time because it cannot directly measure its own overhead.

libc Execution Latency: clock_gettime()

1. **Execution Latency: Linux 69 ns < Unikraft 120 ns**
2. **Assembly-Level Analysis**

Linux

```
clock_gettime
    blr x2  ;vDSO __kernel_clock_gettime ptr
→ __kernel_clock_gettime (vDSO)
    mrs x2, CNTVCT_EL0
```

Utilizes vDSO (Virtual Dynamic Shared Object) to **bypass expensive system calls** by mapping kernel timing data into user-space and directly reads the hardware counter (CNTVCT_EL0) **without kernel entry**.

Unikraft

```
clock_gettime
→ uk_syscall_r_clock_gettime
    ├ _uk_syscall_wrapper_do_entertab
    ├ uk_sys_clock_gettime
    │  └ ukplat_monotonic_clock
    │      mrs x1, CNTVCT_EL0
    └ _uk_syscall_wrapper_do_exittab
```

Runs in a single address space (EL1) with no kernel/user context switch but keeps **generic syscall wrappers** (entertab/exittab).

– Unikraft's nested call stack (multiple bl instructions) leads to a longer execution path compared to Linux's direct read.

libc Execution Latency: clock_gettime()

1. **Execution Latency: Linux 69 ns < Unikraft 120 ns**
2. **Assembly-Level Analysis**
3. **Additional Test**
   - Call uk_sys_clock_gettime() directly.
     - No generic syscall wrappers.
     - Only Unikraft native function.

Unikraft

```
clock_gettime
→ uk_syscall_r_clock_gettime
    ├ _uk_syscall_wrapper_do_entertab
    ├ uk_sys_clock_gettime
    │  └ ukplat_monotonic_clock
    │      mrs x1, CNTVCT_EL0
    └ _uk_syscall_wrapper_do_exittab
```

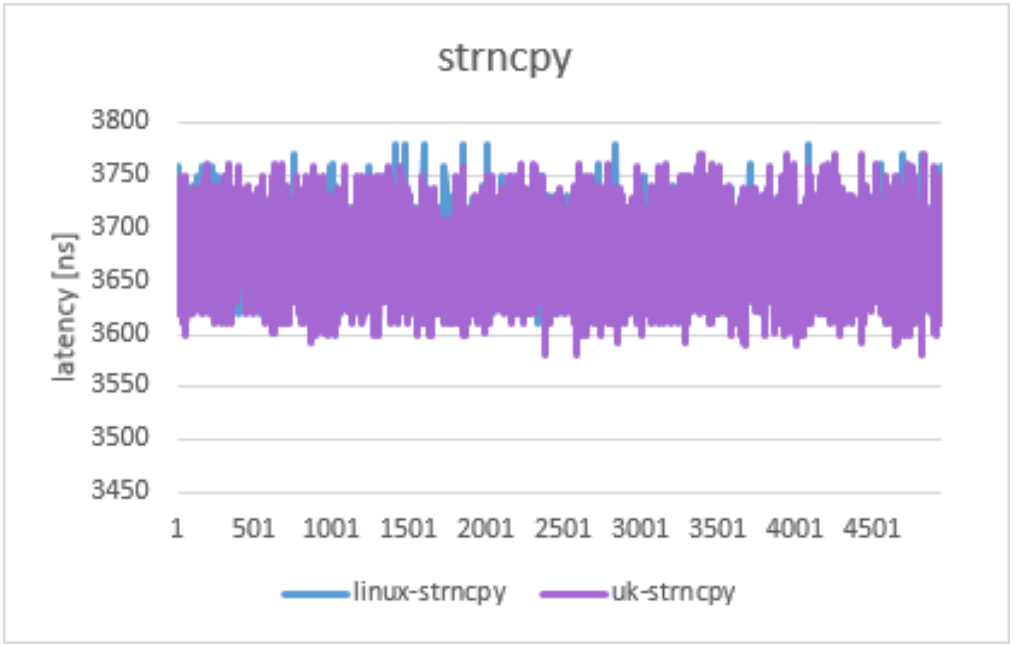
[ns]	**Linux** clock_gettime	**Unikraft** clock_gettime	**Unikraft** uk_sys_clock_gettime
avg	69	120	51

   - Unikraft's overhead comes from generic syscall wrappers, not from the core implementation

libc Execution Latency: strncpy()

1. Execution Latency

[ns]	Linux	Unikraft
avg	3,673	3,671
max	3,780	3,770
min	3,611	3,580
stdev	23	32



- Similar performance
- Strncpy is pure user-space computation, no syscall involved

libc Execution Latency: strncpy()

1. **Execution Latency:** Linux 3,673 ns ≈ Unikraft 3,671 ns
2. **Assembly-Level Analysis**

Linux

```
strncpy
→ stpncpy
→ memset
: 69 instruction counts
```

Unikraft

```
strncpy
→ stpncpy
→ memset
: 59 instruction counts
```

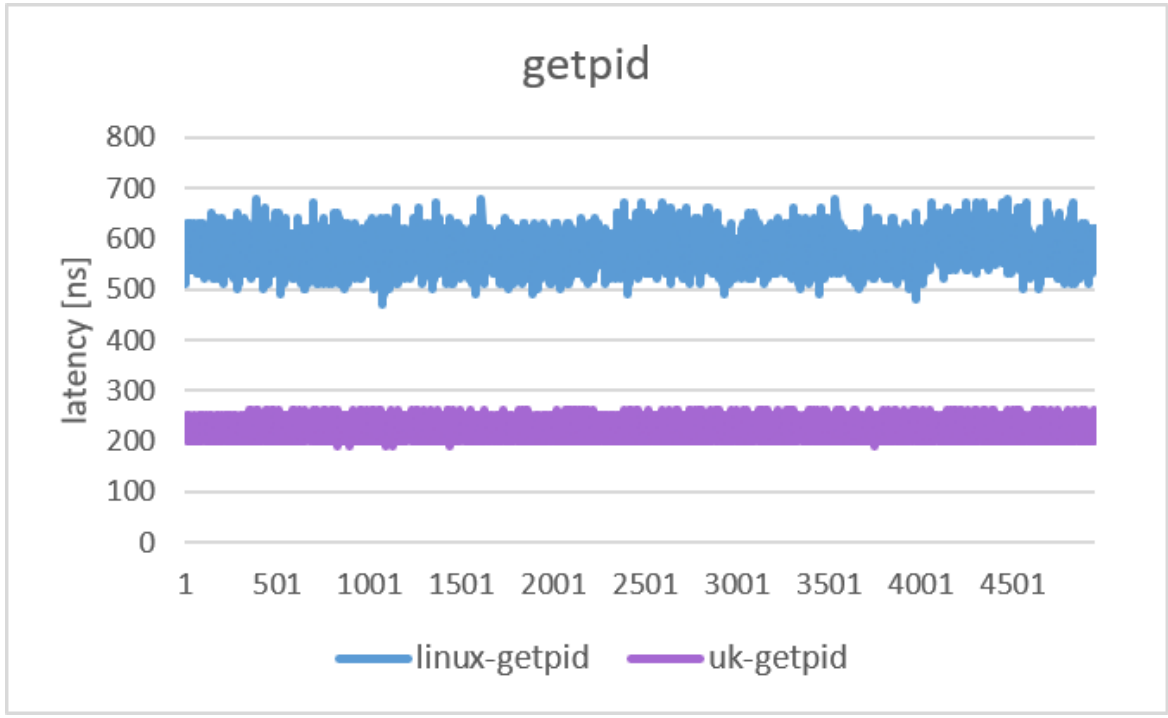
- No system call in strncpy() → Same call chain
- Core assembly is the same code with different register assignments.

libc Execution Latency: getpid()

1. Execution Latency

[ns]	Linux	Unikraft
avg	577	214
max	680	260
min	470	190
stdev	30	14

– Unikraft: 2.7x faster execution latency



libc Execution Latency: getpid()

1. **Execution Latency:** Linux 577 ns > Unikernel 214 ns
2. **Assembly-Level Analysis**

Linux

```
getpid
      svc #0x0 ;EL0 → EL1 exception
→ el0_svc
→ do_el0_svc
→ el0_svc_common
→ invoke_syscall
→ __arm64_sys_getpid
      blr x1 ;sys_call_table[172]
→ __task_pid_nr_ns
```

Requires a costly **EL0 ↔ EL1 transition** triggered by an exception (svc #0x0). Involves a full register save/restore sequence and **deep call chain**.

Unikraft

```
getpid
→ uk_syscall_r_getpid
      ├ _uk_syscall_wrapper_do_entertab
      ├ mrs x0, TPIDR_EL0 ;direct access to thread struct.
      └ _uk_syscall_wrapper_do_exittab
```

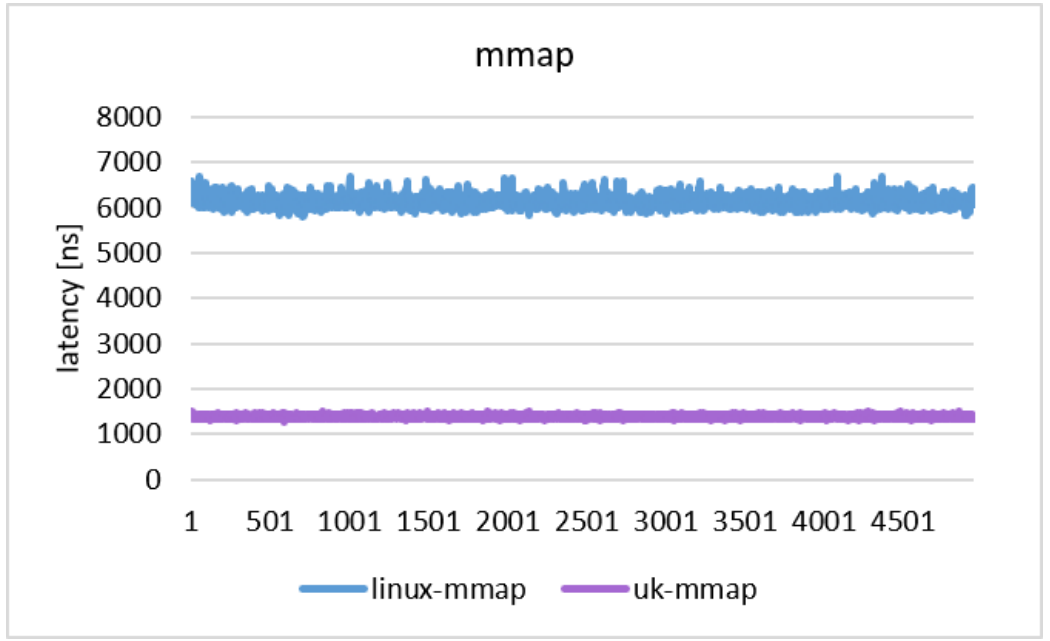
Directly reads the thread structure **within a single address** space at EL1, requiring only syscall wrappers (entertab/exittab).

- Unikernel is faster because it eliminates the **overhead of EL0 ↔ EL1 transition** by running in a single address space.

libc Execution Latency: mmap()

1. Execution Latency

[ns]	Linux	Unikraft
avg	6,126	1,386
max	6,680	1,500
min	5,791	1,281
stdev	101	30



- Unikraft:
  - 4.4x faster execution latency
  - 3.3x lower jitter → Predictable, deterministic behavior

libc Execution Latency: mmap()

1. **Execution Latency**: Linux 6,126 ns > Unikernel 1,386 ns
2. **Assembly-Level Analysis**

Linux

```
mmap → el0_svc (EL0 → EL1) → do_el0_svc → el0_svc_common
→ invoke_syscall → __arm64_sys_mmap → ksys_mmap_pgoff
→ vm_mmap_pgoff
   ├ down_write_killable: acquire mm->mmap_lock
   ├ do_mmap
   │ ├ get_unmapped_area
   │ │└ generic_get_unmapped_area_topdown
   │ │  └ unmapped_area_topdown: maple tree search
   │ └ mmap_region
   │   └ __mmap_region: creates a new VMA and integrates it
   │                    into the process's Maple Tree
   │     ├ mas_find x 2~3: lookup existing VMA in maple tree
   │     ├ vm_area_alloc: allocate VMA struct from slab cache
   │     ├ vma_merge: merge attempt with adjacent VMAs
   │     ├ mas_preallocate: pre-allocate maple tree node for split
   │     └ mas_store_prealloc: insert VMA into maple tree
   └ up_write: release mm->mmap_lock
```

Unikraft

```
mmap64 (Posix Interface Layer)
→ uk_syscall_r_mmap (Syscall Wrapper Layer)
→ __uk_syscall_e_mmap (Implementation Layer)
```

_uk_syscall_e_mmap main function:

1. **Memory allocation**
   *mem = uk_memalign(allocator, __PAGE_SIZE, len)
2. **Tracking structure allocation(Metadata)**
   new = uk_malloc(allocator, 24)
3. **Memory initiation** for the allocated memory address
   memset(mem, 0, len)
4. **Tracking info. Setup (Simple linked list)**
   new->begin = mem; //Start addr. of mem
   new->end = mem + len; // End addr. of mem
   new->next = NULL; // Initialize next node ptr.
5. **Linked list integration**
   if (!mmap_addr)
   mmap_addr = new; // Set as first node
   else
   last->next = new; // Append to existing list
6. **Address return**

libc Execution Latency: mmap()

1. **Execution Latency**: Linux 6,126 ns > Unikernel 1,386 ns
2. **Assembly-Level Analysis**
3. **Key Features**

Feature Category	Linux	Unikraft
Execution Mode	EL0 <--> EL1 mode switching	Single address space
Call Depth	Deep Execution Path (10+ layers)	Shallow call depth (3~4 layers)
Instruction Count (Asm)	5,000+ instructions	~290 instructions
Locking Mechanism	Heavyweight lock(rwsem), even for a single-threaded app	Lockless
Data Structure	Maple Tree	Simple Linked List
Validation Logic	Complex	Essential only
VMA Management	Complex Merge	Direct allocation without merge

libc Execution Latency: mmap()

1. **Execution Latency**: Linux 6,126 ns > Unikernel 1,386 ns
2. **Assembly-Level Analysis**
3. **Key Features**
4. **Limitations of Unikraft mmap()**
   - Lack of locking for multi-process support
   - Data Structure: Simple linked list
   - Security/Safety: Relaxed validation
   - File-backed mmap: Not yet supported on Xen/ARM64

*Note: This test used anonymous mmap only.

Execution Latency Comparison

Target Function	Category	Winner	Speedup (Average)	Jitter Reduction (Stdev)
mmap	System Call	Unikraft	4.4x faster	3.3x lower jitter
getpid	System Call	Unikraft	2.7x faster	2.2x lower jitter
write	System Call	Unikraft	1.8x faster	2.5x lower jitter
read	System Call	Unikraft	1.4x faster	2.2x lower jitter
clock_gettime	vDSO	Linux	1.7x faster	-
malloc	Library Call	Unikraft	3.8x faster	1.7x lower jitter
strncpy	Library Call	-	Similar performance	Similar performance

Conclusion

Conclusion

- **Strengths of Unikraft:**
  - Modular build with selective binding → significantly smaller image and faster build
  - Single address space: no kernel trap, no context save/restore
  - Syscall shim preserves libc compatibility while avoiding mode-switch
  - Further optimization possible by calling Unikraft APIs (uk_*) directly, bypassing shim entirely
  - Lower latency and more deterministic timing

- **Trade-offs of Unikraft:**
  - Functional Gaps: Lacks common OS features (e.g., full signal handling), and Incomplete platform/architecture coverage (e.g., Xen/ARM64)
  - Single-Thread Focus: Optimized for ”Single Address Space” to maximize raw speed (e.g., absence of mm_struct locking)
  - Multi-Process Early Stage: Support began in May 2025, but still requires further development

“Choice depends on application needs, deployment constraints, and operational considerations.”

Q & A