µFork

Supporting POSIX `fork` Within a Single-Address-Space Operating System.

What's a SASOS?

- The kernel and **all** user-space applications share one single massive address space. Virtual addresses are made unique. **Why?**

What's a SASOS?

- The kernel and **all** user-space applications share one single massive address space. Virtual addresses are made unique. **Why?**
  - **Lightweightness**: Context switches are instantaneous, TLB flushing is practically eliminated.

What's a SASOS?

- The kernel and **all** user-space applications share one single massive address space. Virtual addresses are made unique. **Why?**
  - **Lightweightness**: Context switches are instantaneous, TLB flushing is practically eliminated. IPC is *fast* (no kernel copy).

What's a SASOS?

- The kernel and **all** user-space applications share one single massive address space. Virtual addresses are made unique. **Why?**
  - **Lightweightness**: Context switches are instantaneous, TLB flushing is practically eliminated. IPC is *fast* (no kernel copy).
  - 64-bit virtual address space is massive

What's a SASOS?

- The kernel and **all** user-space applications share one single massive address space. Virtual addresses are made unique. **Why?**
  - **Lightweightness**: Context switches are instantaneous, TLB flushing is practically eliminated. IPC is *fast* (no kernel copy).
  - 64-bit virtual address space is massive
  - **Key Obstacle**: *Inherently* incompatible with `fork`.

What’s a SASOS?

- The kernel and **all** user-space applications share one single massive address space. Virtual addresses are made unique. **Why?**
  - **Lightweightness**: Context switches are instantaneous, TLB flushing is practically eliminated. IPC is *fast* (no kernel copy).
  - 64-bit virtual address space is massive
  - **Key Obstacle**: *Inherently* incompatible with `fork`.

Security?

Why's `fork` Important Anyways?

Why's `fork` Important Anyways?

Turns out that **around half** of the 50 most popular C repositories on GitHub and around half of the 50 most popular Debian packages use `fork`.

Why's `fork` Important Anyways?

Turns out that **around half** of the 50 most popular C repositories on GitHub and around half of the 50 most popular Debian packages use `fork`.

- `fork` for sub-process execution (bash)

Why's `fork` Important Anyways?

Turns out that **around half** of the 50 most popular C repositories on GitHub and around half of the 50 most popular Debian packages use `fork`.

- `fork` for sub-process execution (bash)
- `fork` for concurrency (nginx, apache)

Why's `fork` Important Anyways?

Turns out that **around half** of the 50 most popular C repositories on GitHub and around half of the 50 most popular Debian packages use `fork`.

- `fork` for sub-process execution (bash)
- `fork` for concurrency (nginx, apache)
- `fork` for privilege separation (OpenSSH, qmail)

Why's `fork` Important Anyways?

Turns out that **around half** of the 50 most popular C repositories on GitHub and around half of the 50 most popular Debian packages use `fork`.

- `fork` for sub-process execution (bash)
- `fork` for concurrency (nginx, apache)
- `fork` for privilege separation (OpenSSH, qmail)
- `fork` for on-demand resource duplication (Redis, lazy store on disk)

Implementation Challenges

Implementation Challenges

- POSIX processes are isolated by virtue of residing in different address spaces. We must provide an equally secure workaround.

Implementation Challenges

- POSIX processes are isolated by virtue of residing in different address spaces. We must provide an equally secure workaround.
- A SASOS `fork` must relocate the child to a *separate*, distinct virtual address region. We must **relocate absolute memory references**, so that they too point within the child.

Implementation Challenges

- POSIX processes are isolated by virtue of residing in different address spaces. We must provide an equally secure workaround.
- A SASOS `fork` must relocate the child to a *separate*, distinct virtual address region. We must **relocate absolute memory references**, so that they too point within the child. Pointer tracking is notoriously difficult (integer misidentification).

Implementation Challenges

- POSIX processes are isolated by virtue of residing in different address spaces. We must provide an equally secure workaround.
- A SASOS `fork` must relocate the child to a *separate*, distinct virtual address region. We must **relocate absolute memory references**, so that they too point within the child. Pointer tracking is notoriously difficult (integer misidentification). Must keep this transparent!

Past Solutions

Past Solutions

- Segment-Relative Addressing: Fast but gets messy when handwritten assembly, JIT runtimes and compiler integration are taken into account. (considerable engineering effort across the entire development toolchain)

Past Solutions

- Segment-Relative Addressing: Fast but gets messy when handwritten assembly, JIT runtimes and compiler integration are taken into account. (considerable engineering effort across the entire development toolchain) (Angel, 1992)

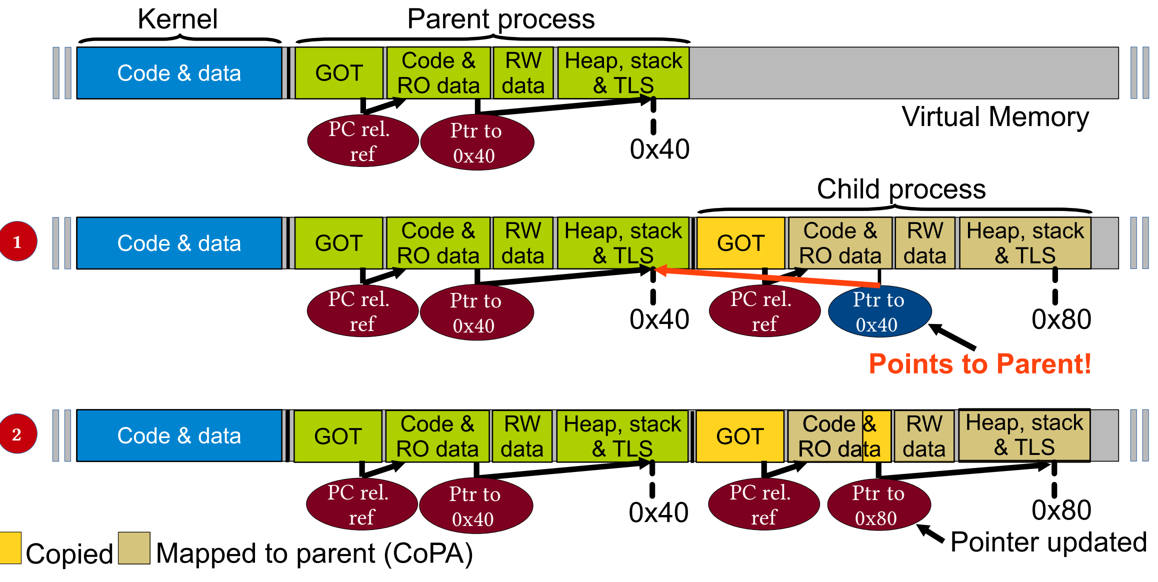
Past Solutions

- Segment-Relative Addressing: Fast but gets messy when handwritten assembly, JIT runtimes and compiler integration are taken into account. (considerable engineering effort across the entire development toolchain) (Angel, 1992)
- OS as a Process: Treat SASOS as a process and implement `fork` for the hypervisor. Circumvents most challenges. Clever, *slow* in practice (misses the entire point).

Past Solutions

System	SAS	Isolation	SC	IPCs	Seg	f+e only
Angel	Yes	Yes	Yes	Fast	Yes	No
Mungi	Yes	Yes	Yes	Fast	Yes	No
Nephele	No	Yes	No	Med	No	No
KylinX	No	Yes	No	Med	No	No
Graphene	No	Yes	No	Med	No	No
Graphene SGX	No	Yes	No	Slow	No	No
Iso-Unik	No	Yes	Yes	Med	No	No
OSv	Yes	No	Yes	Fast	No	Yes
Junction	Yes	No	No	Med	No	Yes
μFork	Yes	Yes	Yes	Fast	No	No

Overview of µFork



Overview of μFork

We'll leverage position-independent code (PIC) so that the majority of compiled memory references are made to be relative to the stack, base, or instruction pointers.

Overview of μFork

We'll leverage position-independent code (PIC) so that the majority of compiled memory references are made to be relative to the stack, base, or instruction pointers.

Crucially, Copy-on-Write must be **revisited**. We mustn't give read access to pages containing stale absolute parent addresses. We'll enforce **CoPA** (Copy-on-Pointer-Access).

Overview of µFork

We'll leverage position-independent code (PIC) so that the majority of compiled memory references are made to be relative to the stack, base, or instruction pointers.

Crucially, Copy-on-Write must be **revisited**. We mustn't give read access to pages containing stale absolute parent addresses. We'll enforce **CoPA** (Copy-on-Pointer-Access).

Now what? Who'll impose isolation? What about pointer tracking?

Overview of µFork

We'll leverage position-independent code (PIC) so that the majority of compiled memory references are made to be relative to the stack, base, or instruction pointers.

Crucially, Copy-on-Write must be **revisited**. We mustn't give read access to pages containing stale absolute parent addresses. We'll enforce **CoPA** (Copy-on-Pointer-Access).

Now what? Who'll impose isolation? What about pointer tracking?
→ **hardware**

CHERI 🍒

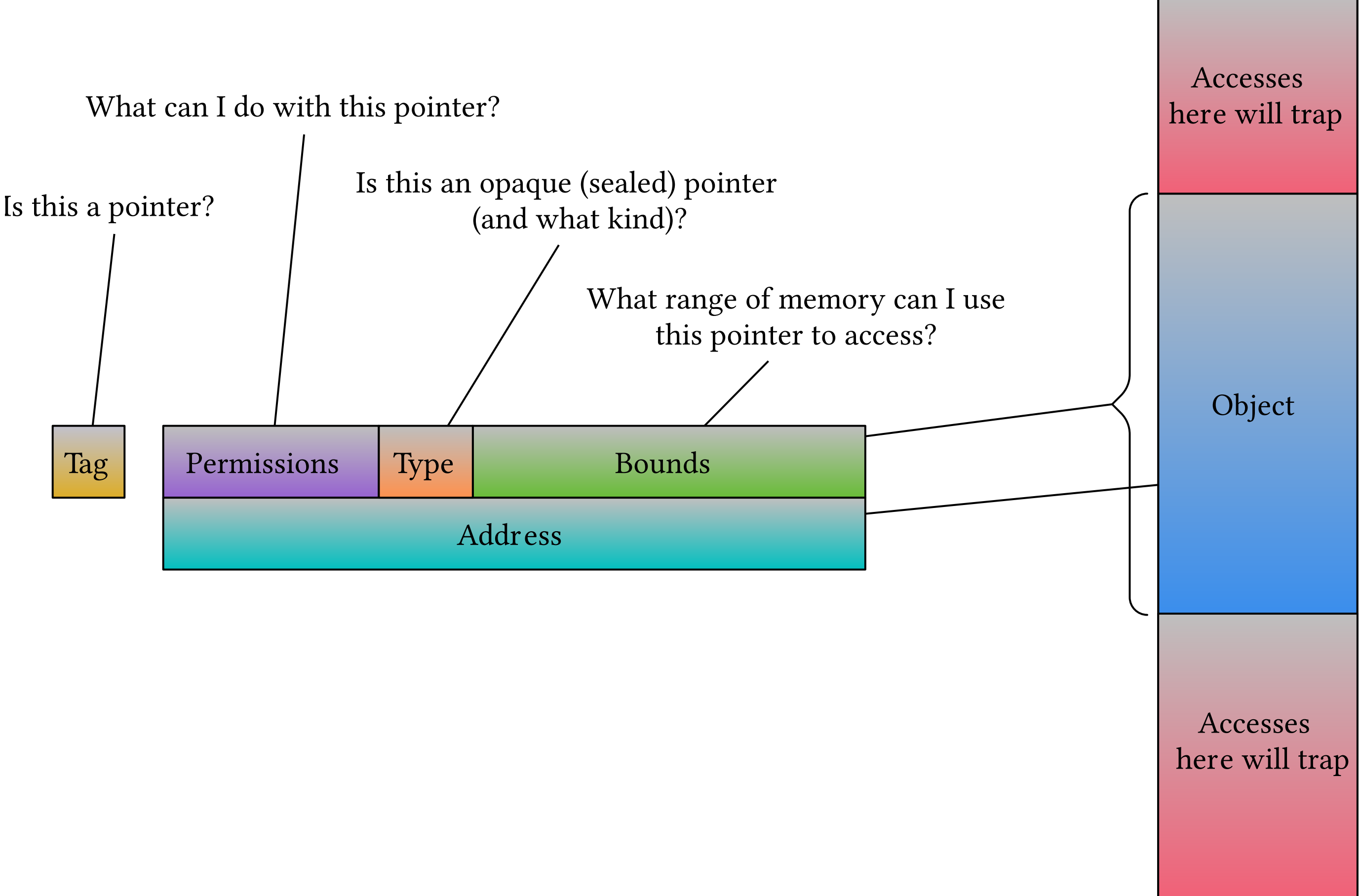
Elegant hardware technology designed to eliminate the majority of memory safety bugs in RISC processors.

CHERI

Elegant hardware technology designed to eliminate the majority of memory safety bugs in RISC processors.

- Pointers shall no longer be stored as raw integers (root of all evil). They're wrapped around **hardware-enforced *types*** called *capabilities* along with relevant metadata (tag bit integrity).

```
lw s0, 8(a1)
# CHERI (extended registers, explicit pointer arithmetic)
clw s0, 0(ca1)
lc cs3, 8(ca1)
```


Memory
Accesses here will trap
What can I do with this pointer?
Is this an opaque (sealed) pointer (and what kind)?
Is this a pointer?
What range of memory can I use this pointer to access?
Object
Tag
Permissions
Type
Bounds
Address
Accesses here will trap

CHERI

CHERI capabilities respect *monotonicity*.

CHERI

CHERI capabilities respect *monotonicity*. The hardware provides an almighty capability which is stripped down by the kernel at will.

CHERI 🍒

CHERI capabilities respect *monotonicity*. The hardware provides an almighty capability which is stripped down by the kernel at will. Isolation is guaranteed by the simple fact that capability transformations can only ever *reduce* privileges.

CHERI

CHERI capabilities respect *monotonicity*. The hardware provides an almighty capability which is stripped down by the kernel at will. Isolation is guaranteed by the simple fact that capability transformations can only ever *reduce* privileges.

- Principle of Least Privilege
- Principle of Intentional Use

Process-Kernel Isolation

Capabilities include an *object type* field. A nonzero value indicates a **sealed capability**.

Process-Kernel Isolation

Capabilities include an *object type* field. A nonzero value indicates a **sealed capability**. The kernel can essentially form armored system call entry point references (tokens) that can be called (`cinvoke`), duplicated and moved around user-space programs relentlessly.

Process-Kernel Isolation

Capabilities include an *object type* field. A nonzero value indicates a **sealed capability**. The kernel can essentially form armored system call entry point references (tokens) that can be called (`cinvoke`), duplicated and moved around user-space programs relentlessly. Crucially, these capabilities *cannot* be modified by anyone except the sealer.

Process-Kernel Isolation

Capabilities include an *object type* field. A nonzero value indicates a **sealed capability**. The kernel can essentially form armored system call entry point references (tokens) that can be called (`cinvoke`), duplicated and moved around user-space programs relentlessly. Crucially, these capabilities *cannot* be modified by anyone except the sealer.

→ Blazingly fast

Process-Kernel Isolation (Cont.)

> We implicitly refer to the Unikraft implementation. μFork is simply a **layer**.

Process-Kernel Isolation (Cont.)

> We implicitly refer to the Unikraft implementation. µFork is simply a **layer**.

Privileged instructions are safely prohibited (`access_system_registers` permission bit enforced on PPC).

Process-Kernel Isolation (Cont.)

> We implicitly refer to the Unikraft implementation. μFork is simply a **layer**.

Privileged instructions are safely prohibited (`access_system_registers` permission bit enforced on PPC). Everyone gets `EL1`! System calls are trivialized to external procedures.

Patching it Up

- **ASLR**?

Patching it Up

- **ASLR**? Randomize the base offset of the contiguous process segment

Patching it Up

- **ASLR**? Randomize the base offset of the contiguous process segment
- **Heap**?

Patching it Up

- **ASLR**? Randomize the base offset of the contiguous process segment
- **Heap**? VAS is massive, assume a safe upper-bound

Patching it Up

- **ASLR**? Randomize the base offset of the contiguous process segment
- **Heap**? VAS is massive, assume a safe upper-bound
- **SMP**?

Patching it Up

- **ASLR**? Randomize the base offset of the contiguous process segment
- **Heap**? VAS is massive, assume a safe upper-bound
- **SMP**? Serializing kernel code execution with a "big kernel lock" (they're working on it).

Patching it Up

- **ASLR**? Randomize the base offset of the contiguous process segment
- **Heap**? VAS is massive, assume a safe upper-bound
- **SMP**? Serializing kernel code execution with a “big kernel lock” (they’re working on it).
- **Fragmentation**?

Patching it Up

- **ASLR**? Randomize the base offset of the contiguous process segment
- **Heap**? VAS is massive, assume a safe upper-bound
- **SMP**? Serializing kernel code execution with a “big kernel lock” (they’re working on it).
- **Fragmentation**? Ignore (consider compaction)
- **TOCTTOU**? *Optional* system call parameter duplication

Clarifying CoPA

Exploit the CHERI page-table capability-was-accessed bit.

Clarifying CoPA

Exploit the CHERI page-table capability-was-accessed bit.

- The child page table entry is forced to point to a free physical page and remains inaccessible until the copying has finished (atomicity).

Clarifying CoPA

Exploit the CHERI page-table capability-was-accessed bit.

- The child page table entry is forced to point to a free physical page and remains inaccessible until the copying has finished (atomicity).
- The page is copied.

Clarifying CoPA

Exploit the CHERI page-table capability-was-accessed bit.

- The child page table entry is forced to point to a free physical page and remains inaccessible until the copying has finished (atomicity).
- The page is copied.
- The page is scanned in 16-byte increments (size of CHERI capability). Absolute memory references are identified by the presence of a valid CHERI tag and are transformed accordingly, in place.

Overview

(verbal)

Evaluation (Setup)

CHERI introduces a non-negligible overhead, so we'll compare µFork + Unikraft with **CheriBSD**, a monolithic FreeBSD kernel running on the Morello platform. µFork executes on top of the `bhyve` hypervisor (missing drivers) but nevertheless **outperforms all competition**.

Evaluation (Setup)

CHERI introduces a non-negligible overhead, so we'll compare µFork + Unikraft with **CheriBSD**, a monolithic FreeBSD kernel running on the Morello platform. µFork executes on top of the `bhyve` hypervisor (missing drivers) but nevertheless **outperforms all competition**.

Comparison with Nephele is unnecessary, it *will* be slower. Nephele is designed for `x86_64` only, so experimental data is directly extracted from their (possibly biased) paper. They still lose.

Evaluation (Shorthand)

- μFork + Unikraft running on top of the `bhyve` supervisor
- CheriBSD (FreeBSD) running on bare-metal
- `x86_64` Nephele (vitalization-based SASOS `fork` implementation)

1. Forking `hello_world.c`

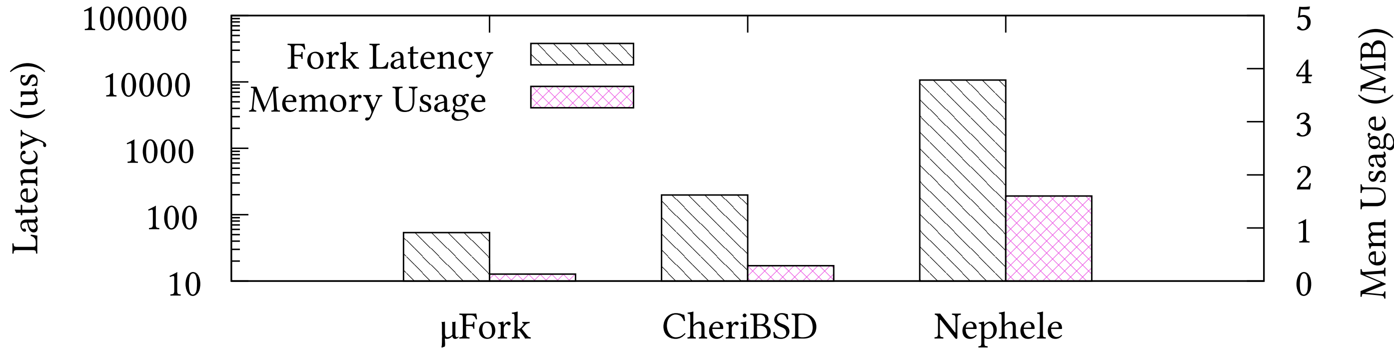


Figure 8. `fork` latency and memory usage (hello world).

1. Forking `hello_world.c`

Nephele: 10,700 µs
Creating a Xen domain is complicated

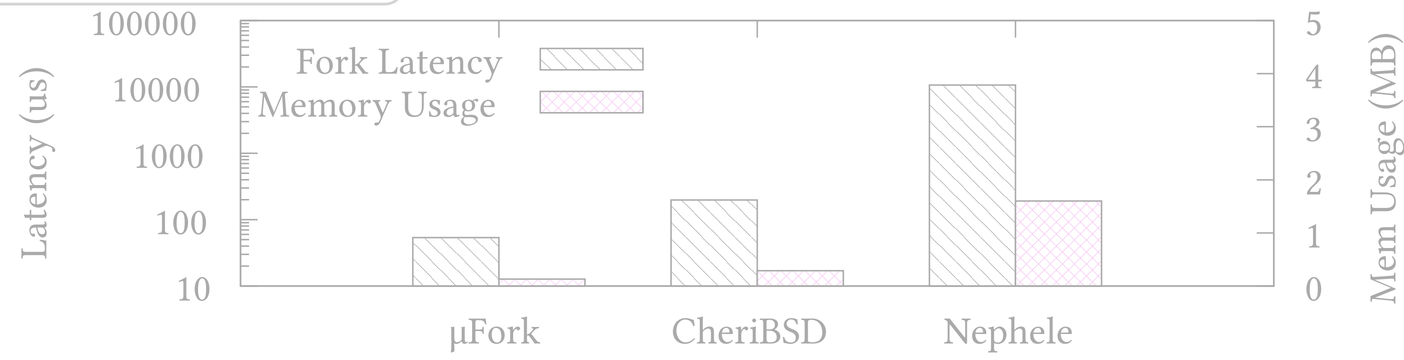


Figure 8. `fork` latency and memory usage (hello world).

1. Forking `hello_world.c`

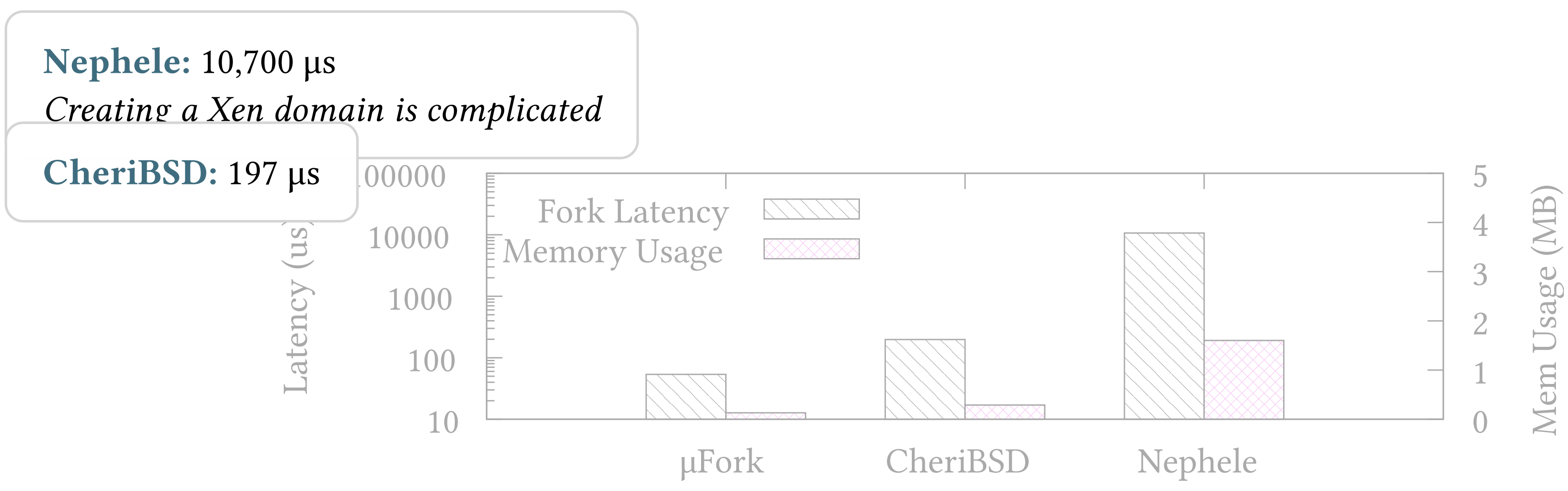


Figure 8. `fork` latency and memory usage (hello world).

1. Forking `hello_world.c`

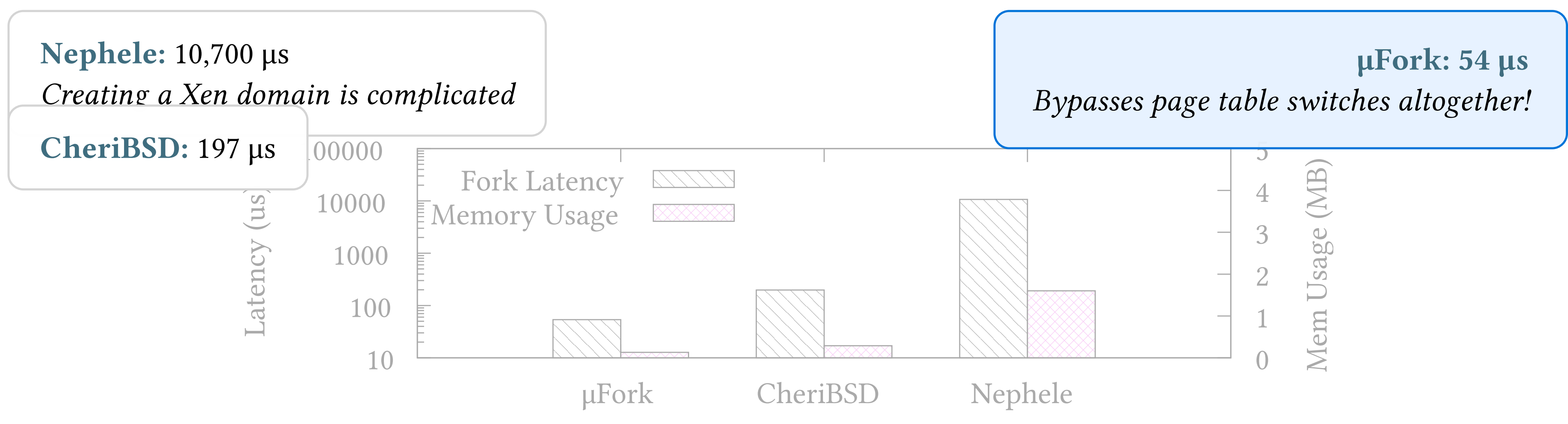


Figure 8. `fork` latency and memory usage (hello world).

2. Does it Actually Matter? (FaaS)

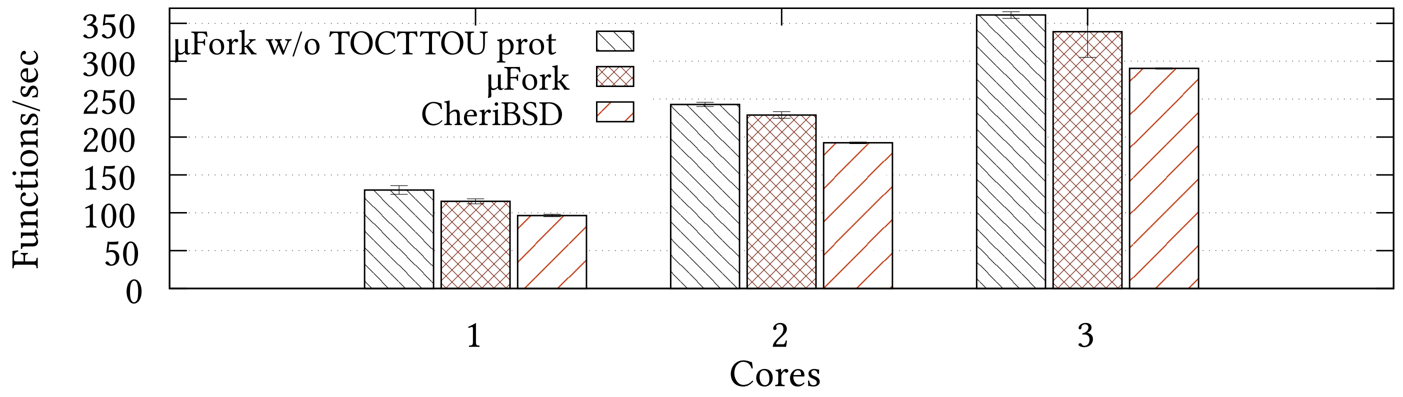


Figure 6. FaaS function throughput.

2. Does it Actually Matter? (FaaS)

FaaS functions are typically short-lived, with 50% of functions taking less than 1s to execute.
(Zygote language runtime pre-warming technique)

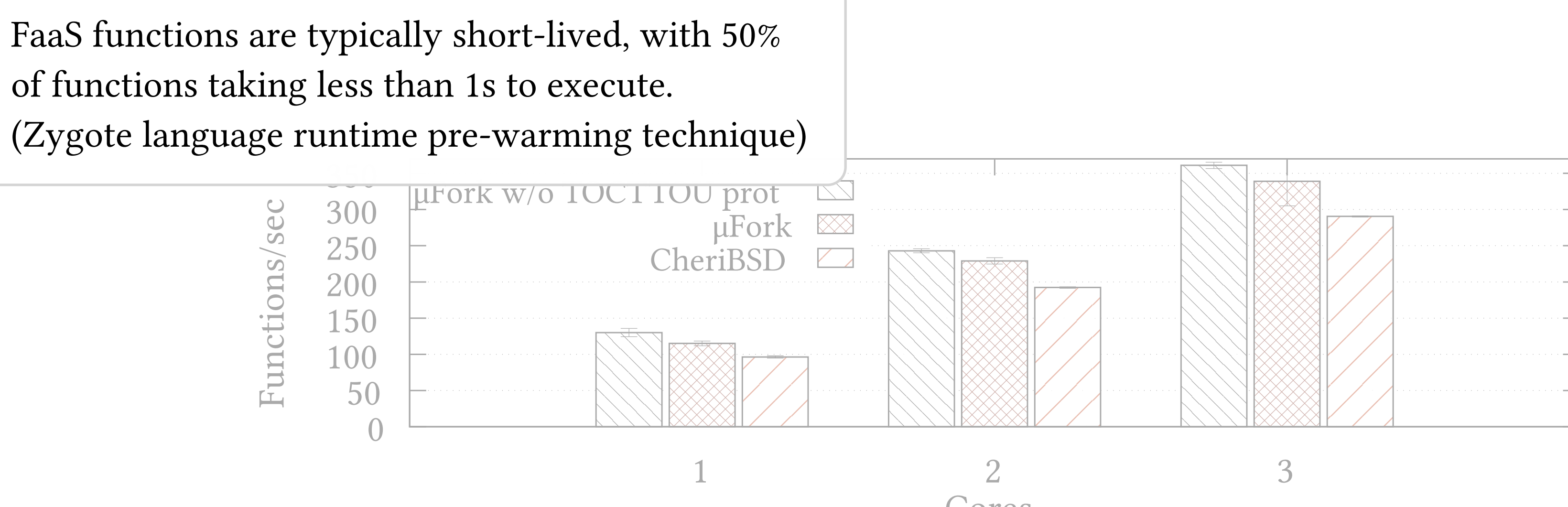


Figure 6. FaaS function throughput.

2. Does it Actually Matter? (FaaS)

FaaS functions are typically short-lived, with 50% of functions taking less than 1s to execute.
(Zygote language runtime pre-warming technique)

µFork handles **24% more requests per second** than CheriBSD across multiple cores
(TOCTTOU? `float_operation` **not** system-call intensive)

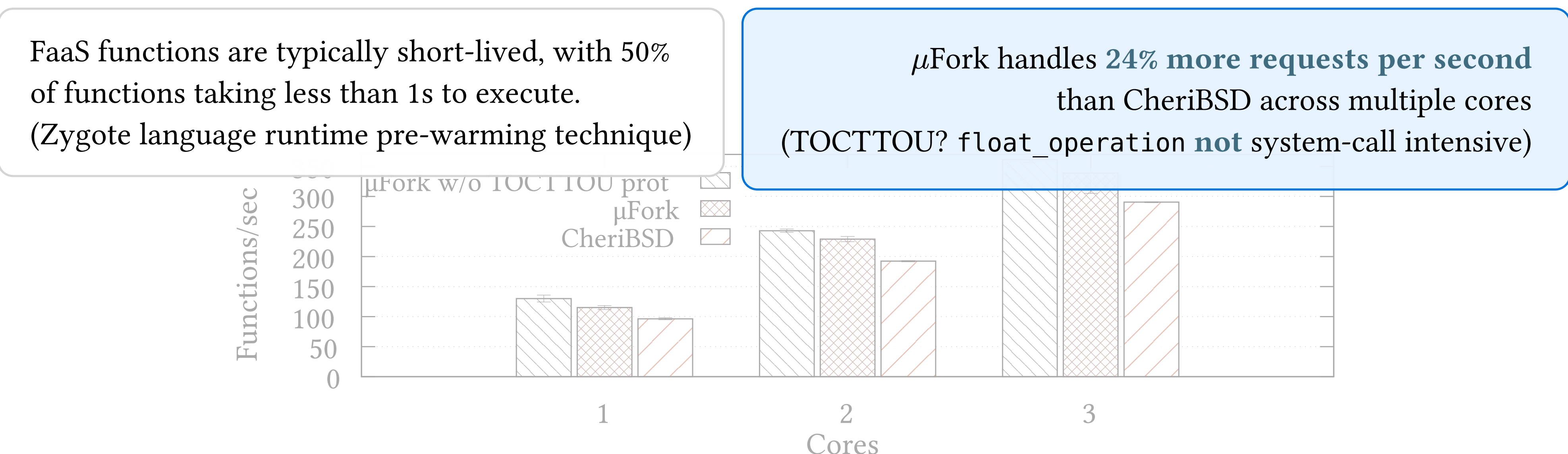


Figure 6. FaaS function throughput.

3. Evaluating CoPA Performance

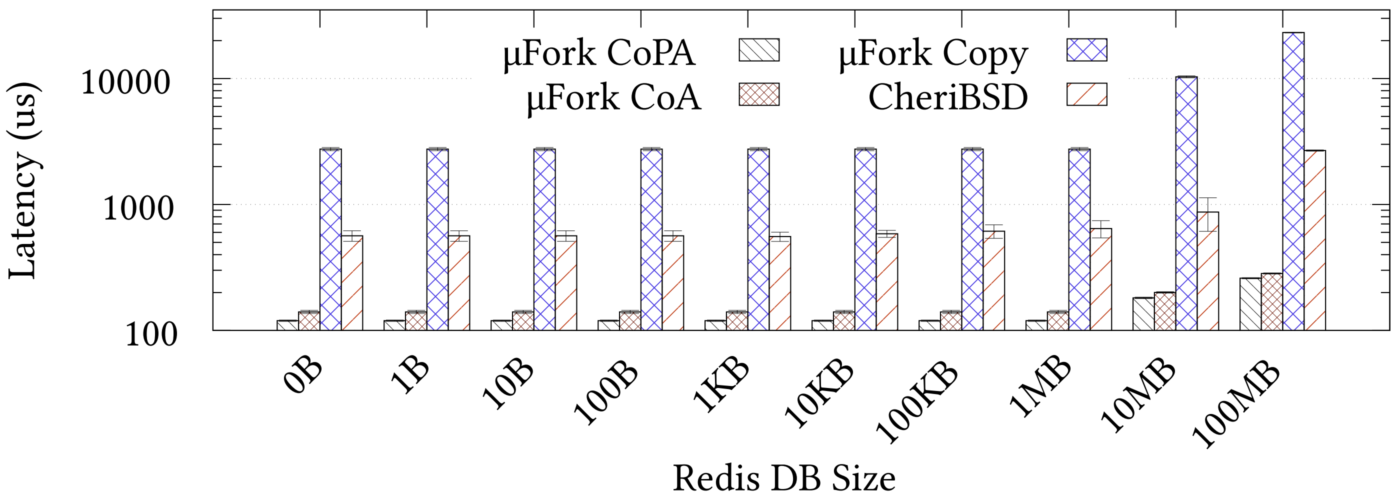


Figure 4. Redis `fork` latency (us).

3. Evaluating CoPA Performance

Fork latency and memory consumption are **analogous**.
The blue line is a full synchronous copy.
The usual bottleneck is the heap itself (static).

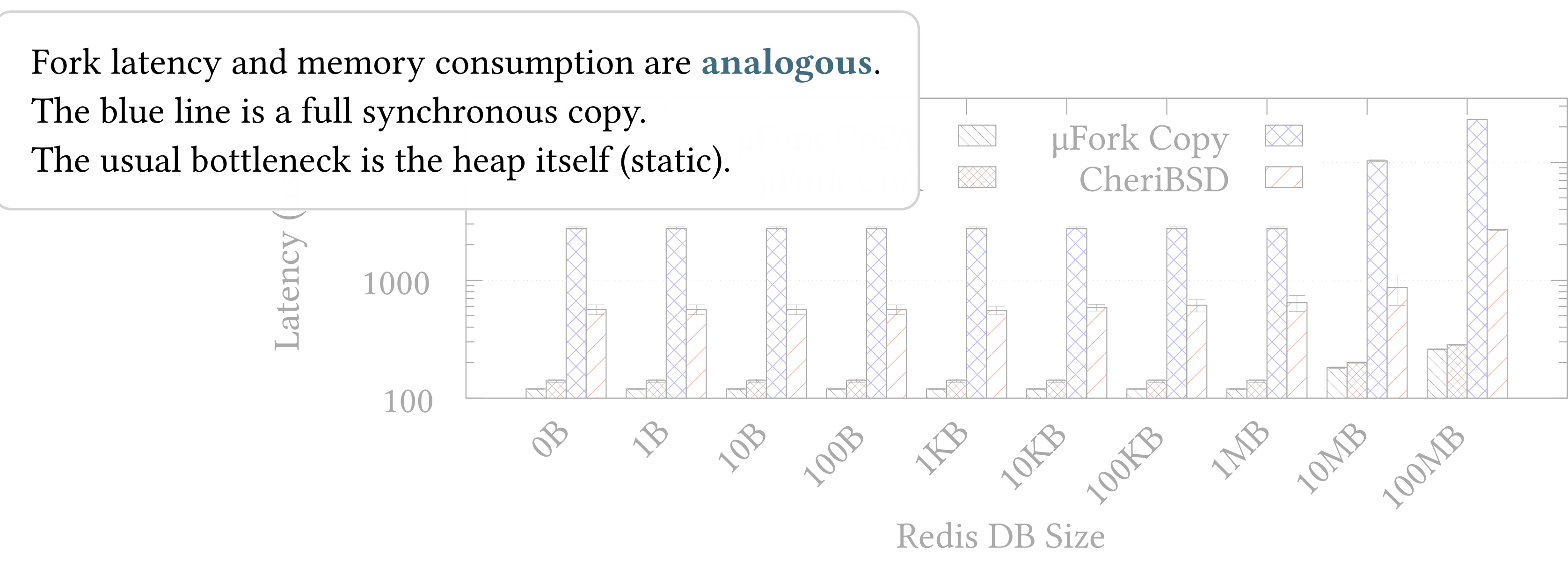


Figure 4. Redis `fork` latency (us).

3. Evaluating CoPA Performance

Fork latency and memory consumption are **analogous**.
The blue line is a full synchronous copy.
The usual bottleneck is the heap itself (static).

CoPA is **incredibly** performant.
μFork naturally outperforms traditional OSes.

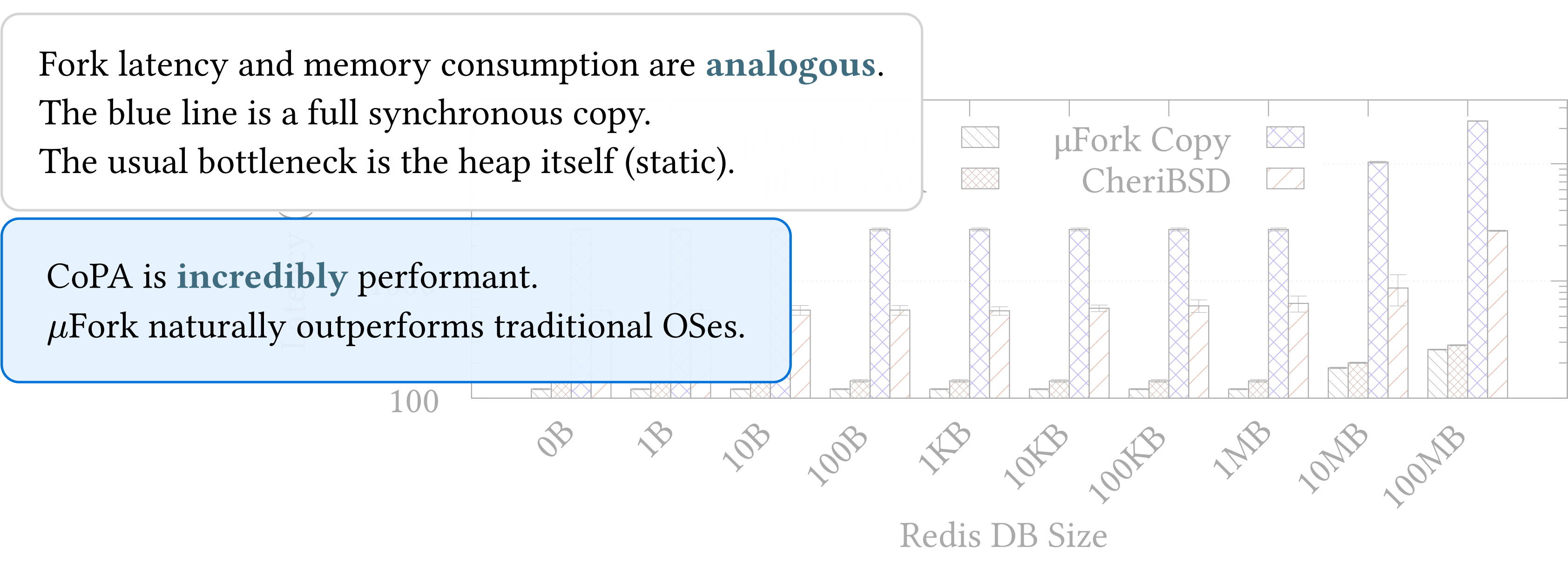


Figure 4. Redis `fork` latency (us).

Questions?