MULTI QUEUE LINUX BLOCK DEVICE DRIVERS IN RUST

Storage Developer Conference 2023

Andreas Hindborg

Samsung GOST

AGENDA

- Why Memory Safety ~~in the Linux Kernel~~ **in General** is Important
- Memory Safety in **Rust**
- The Rust for Linux **Community**
- `blk-mq` Rust API
  - `null_blk`
  - `nvme`

WHY CARE ABOUT MEMORY SAFETY

- Microsoft: 70% of all security bugs are memory safety issues [1]
- Chrome: 70% of all security bugs are memory safety issues [2]
- 20% of bugs fixed in **stable Linux** Kernel branches for drivers are memory safety issues [4]
- 65% of recent **Linux kernel** vulnerabilities are memory safety issues [3]
- ASOP: Memory safety vulnerabilities disproportionately represent our most **severe vulnerabilities** [7]
- **41%** of fixes submitted to C `null_blk` are fixes for memory safety issues [6]

GOAL: PREVENT MEMORY SAFETY RELATED BUGS IN LINUX

WHY RUST  INSTEAD OF <LANG>

Rust is Much Like C:

- Ahead of time compiled
- Focus on maximum programmer control and zero runtime overhead
- Works well for bare metal work
- Statically typed
- Performance on par with C/C++
- Easy to link with C programs
- Basic control flow structures are the same (no throwing of exceptions)

DIFFERENCES BETWEEN RUST AND C

- Strong type system
- Module system (no include files)
- All statements including blocks evaluate to values
- All values have move semantics by default
- References - One mutable or many immutable
  - Static lifetime analysis
- Generic Types
- Macros (Simple expansion and AST Transforms)
- RAII is encouraged
- Async/Await primitives
- **Safe subset without UB through static analysis**

MEMORY SAFETY

MEMORY SAFETY IN RUST

Rust has a **safe** subset

- Memory safe
- Type safe
- Thread safe

In **safe** Rust

- No buffer overflows
- No use after free
- No dereferencing null or invalid pointers
- No double free
- No pointer aliasing
- No type errors
- No data races

THIS IS NOT UNSAFE BEHAVIOR IN RUST

- Deadlocks
- Race conditions
- Memory leaks
- Failing to call destructors
- Integer overflows (checked operations available)
- Program aborts
- Deletion of the production database (logic errors)

RUST IN THE LINUX KERNEL

CALLING C IS UNSAFE 😬

- We don’t want to rewrite Linux in Rust ➡️ we have to talk to C
- At FFI boundary we have to verify safety invariants **by hand**
- This is not as bad as it sounds
- The things we verify at FFI boundary are things C programmers should verify **always**
- We opt out of the **safe** subset with the `unsafe` keyword

UNSAFE RUST

In `unsafe` Rust we can:

- Dereference a raw pointer
- Call an **unsafe function** or method (including C functions)
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of unions

STRATEGY FOR DEPLOYING RUST

- Support driver implementations in **safe** Rust
- Constrain unsafe code to subsystem wrappers
- Keep unsafe blocks small and well documented
- Focus review bandwidth on **unsafe** blocks

COMMUNITY

THE RUST FOR LINUX COMMUNITY

- Part of Linux Kernel since **6.1**
- Zulip - https://rust-for-linux.zulipchat.com/
  - ~500 members
- List - rust-for-linux@vger.kernel.org
  - Send your rust-core **patches** here
  - But use relevant **subsystem list** for non-core patches
- WWW - https://rust-for-linux.com
  - Contributor guide: https://rust-for-linux.com/contributing
- Github - https://github.com/rust-for-Linux/linux
  - Used prior to merge - now primarily a **backlog**

THE ROAD SO FAR (HIGHLIGHTS)

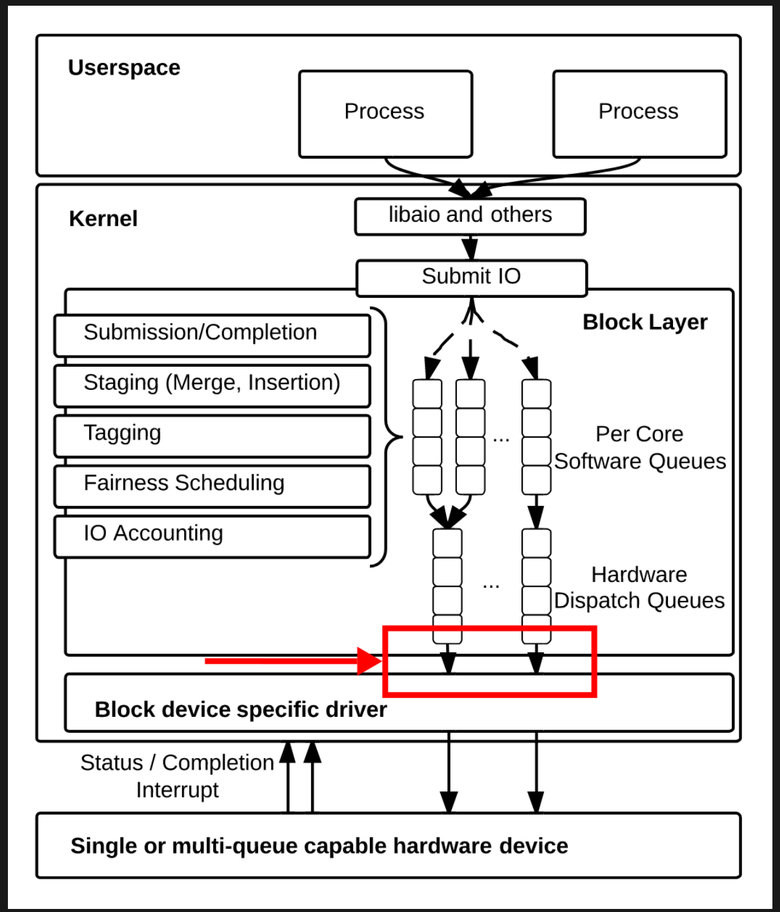
- 6.1
  - Kbuild support for rustc, bindgen
  - `alloc`
  - `printk`
  - Rust module
- 6.2
  - `#[vtable]`
  - Errors
  - Fallible constructors for containers
  - `BStr`, `CStr`
  - `Either`, `Opaque`
- 6.3
  - `Arc`
  - `ScopeGuard`
  - `ForignOwnable`
- 6.4
  - Pinned initialization (`pin-init`)
  - `sync` module with `Lock`, `CondVar`, `Mutex`, etc.
  - `uapi` crate
- 6.5
  - `rustc` 1.68.2
- 6.6
  - `rustc` 1.71.1
  - `bindgen` 0.65.1
- 6.x (Pending)
  - Workqueue

ONGOING PROJECTS

- (Android) Binder
- DRM API + Apple M1/M2 GPU Driver
- ENC28J60 SPI Ethernet
- V4L2
- In kernel TLS handshake
- netdev
- VirtIO
- PuzzleFS (Container FS)
- Kernel Sockets
- Device Mapper
- RCU
- VMBus
- **blk-mq**
  - nvme
  - null_blk

MULTI QUEUE BLOCK DEVICE DRIVERS

blk-mq


Userspace
Process
Process
Kernel
libaio and others
Submit IO
Block Layer
Submission/Completion
Staging (Merge, Insertion)
Tagging
Fairness Scheduling
IO Accounting
...
Per Core
Software Queues
...
Hardware
Dispatch Queues
Block device specific driver
Status / Completion
Interrupt
Single or multi-queue capable hardware device

[5]

BLK-MQ INTERFACE

```rust
#[macros::vtable]
pub trait Operations: Sized {
    type RequestData;
    type QueueData: ForeignOwnable;
    type HwData: ForeignOwnable;
    type TagSetData: ForeignOwnable;

    fn new_request_data(
        _tagset_data: <Self::TagSetData as ForeignOwnable>::Borrowed<'_>,
    ) -> Result<Self::RequestData>;

    fn init_request_data(
        _tagset_data: <Self::TagSetData as ForeignOwnable>::Borrowed<'_>,
        _data: Pin<&mut Self::RequestData>,
    ) -> Result {
        Ok(())
    }

    fn queue_rq(
        hw_data: <Self::HwData as ForeignOwnable>::Borrowed<'_>,
        queue_data: <Self::QueueData as ForeignOwnable>::Borrowed<'_>,
        rq: &Request<Self>,
        is_last: bool,
    ) -> Result;

    fn commit_rqs(
        hw_data: <Self::HwData as ForeignOwnable>::Borrowed<'_>,
        queue_data: <Self::QueueData as ForeignOwnable>::Borrowed<'_>,
    );

    fn complete(_rq: &Request<Self>);

    fn init_hctx(
        tagset_data: <Self::TagSetData as ForeignOwnable>::Borrowed<'_>,
        hctx_idx: u32,
    ) -> Result<Self::HwData>;


    fn poll(hw_data: <Self::HwData as ForeignOwnable>::Borrowed<'_>) -> i32 {
        unreachable!()
    }

    fn map_queues(tag_set: &TagSetRef) -> Result {
        unreachable!()
    }
}
```

```c
struct blk_mq_ops {

    blk_status_t (*queue_rq)(struct blk_mq_hw_ctx *,
                const struct blk_mq_queue_data *);

    void (*commit_rqs)(struct blk_mq_hw_ctx *);

    int (*poll)(struct blk_mq_hw_ctx *, struct io_comp_batch *);

    void (*complete)(struct request *);

    int (*init_hctx)(struct blk_mq_hw_ctx *, void *, unsigned int);

    void (*exit_hctx)(struct blk_mq_hw_ctx *, unsigned int);

    int (*init_request)(struct blk_mq_tag_set *set, struct request *,
                unsigned int, unsigned int);

    void (*exit_request)(struct blk_mq_tag_set *set, struct request *,
                 unsigned int);

    int (*map_queues)(struct blk_mq_tag_set *set);

};
```

queue_rq()

Rust

```rust
#[kernel::macros::vtable]
pub trait Operations: Sized {
    // ...
    type QueueData: ForeignOwnable;
    type HwData: ForeignOwnable;
    // ...
    fn queue_rq(
        hw_data: <Self::HwData as ForeignOwnable>::Borrowed<'_>,
        queue_data: <Self::QueueData as ForeignOwnable>::Borrowed<'_>,
        rq: &Request<Self>,
        is_last: bool,
    ) -> Result;
    // ...
}
```

C

```c
blk_status_t (*queue_rq)(struct blk_mq_hw_ctx *, const struct blk_mq_queue_data *);
```

IMPLEMENTING queue_rq()

```
#[kernel::macros::vtable]
impl mq::Operations for IoQueueOperations {
    // ...
    type QueueData = Box<NvmeNamespace>;
    type HwData = Arc<NvmeQueue<Self>>;
    // ...
     fn queue_rq(
        io_queue: ArcBorrow<'_, NvmeQueue<Self>>,
        ns: &NvmeNamespace,
        rq: &mq::Request<Self>,
        is_last: bool,
    ) -> Result {
        // ...
    }
    // ...
}
```

CALLING queue_rq()

```
unsafe extern "C" fn queue_rq_callback(
    hctx: *mut bindings::blk_mq_hw_ctx,
    bd: *const bindings::blk_mq_queue_data,
) -> bindings::blk_status_t {
    // SAFETY: `bd` is valid as required by this function.
    let rq = unsafe { (*bd).rq };

    // SAFETY: ...
    let hw_data = unsafe { T::HwData::borrow((*hctx).driver_data) };

    // SAFETY: `hctx` is valid as required by this function.
    let queue_data = unsafe { (*(*hctx).queue).queuedata };

    // SAFETY: ...
    let queue_data = unsafe { T::QueueData::borrow(queue_data) };

    // SAFETY: `bd` is valid as required by the safety requirement for this function.
    let ret = T::queue_rq(hw_data, queue_data, &Request::from_ptr(rq), unsafe {
        (*bd).last
    });
    if let Err(e) = ret {
        e.to_blk_status()
    } else {
        bindings::BLK_STS_OK as _
    }
}
```

SAFETY COMMENTS

```
// # Safety
//
// The caller of this function must ensure that `hctx` and `bd` are valid
// and initialized. The pointees must outlive this function. Further
// `hctx->driver_data` must be a pointer created by a call to
// `Self::init_hctx_callback()` and the pointee must outlive this function.
// This function must not be called with a `hctx` for which
// `Self::exit_hctx_callback()` has been called.
unsafe extern "C" fn queue_rq_callback(...) {

    // ...

    // SAFETY: The safety requirement for this function ensure that
    // `(*hctx).driver_data` was returned by a call to
    // `Self::init_hctx_callback()`. That function uses
    // `PointerWrapper::into_pointer()` to create `driver_data`. Further,
    // the returned value does not outlive this function and
    // `from_pointer()` is not called until `Self::exit_hctx_callback()` is
    // called. By the safety requirement of this function and contract with
    // the `blk-mq` API, `queue_rq_callback()` will not be called after that
    // point.
    let hw_data = unsafe { T::HwData::borrow((*hctx).driver_data) };

   // ...
}
```

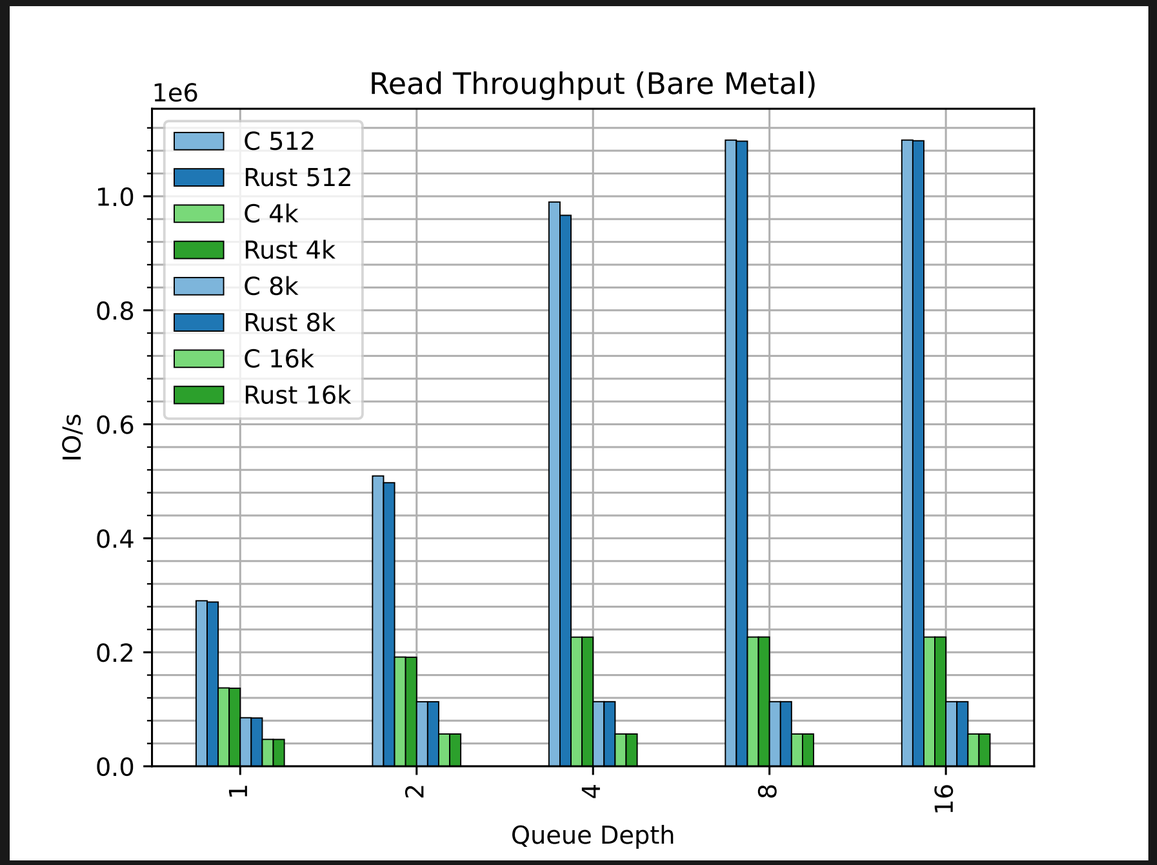
WHERE IS THE CODE?

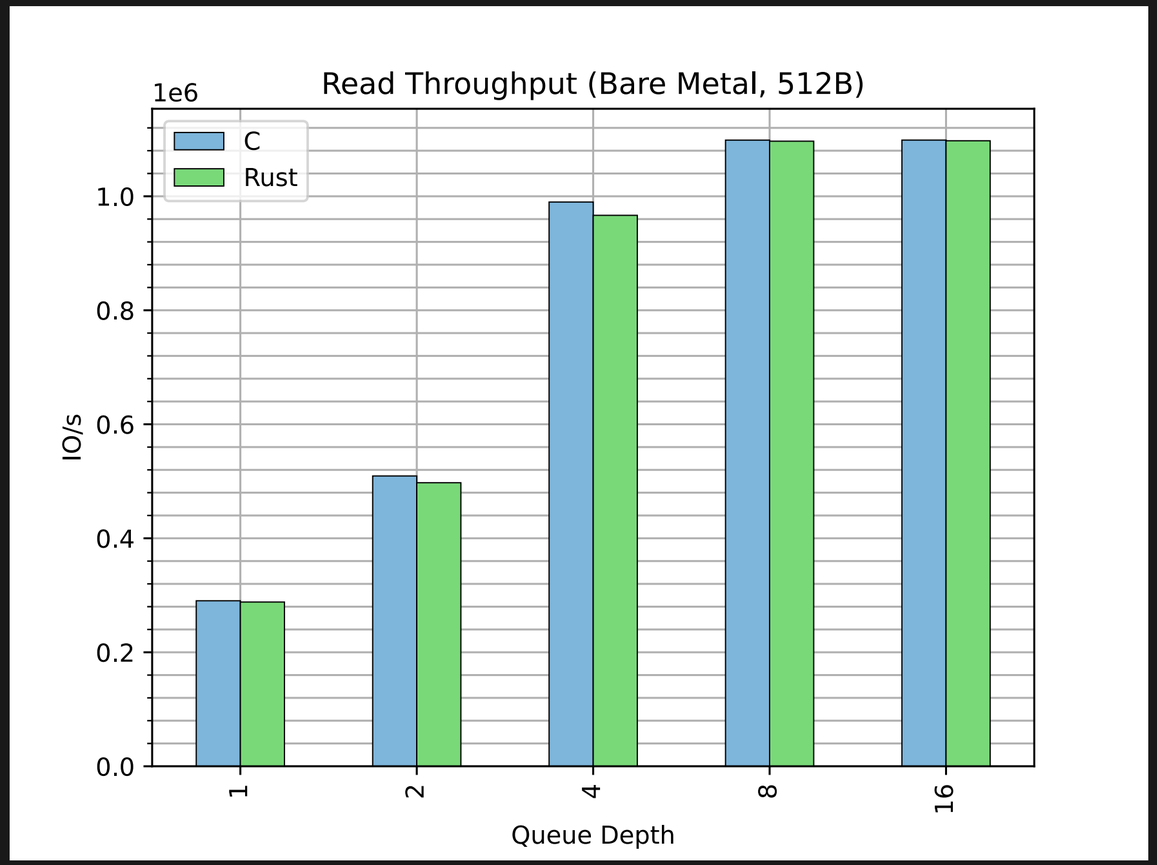
NVMe Driver

Null Block Driver

PERFORMANCE

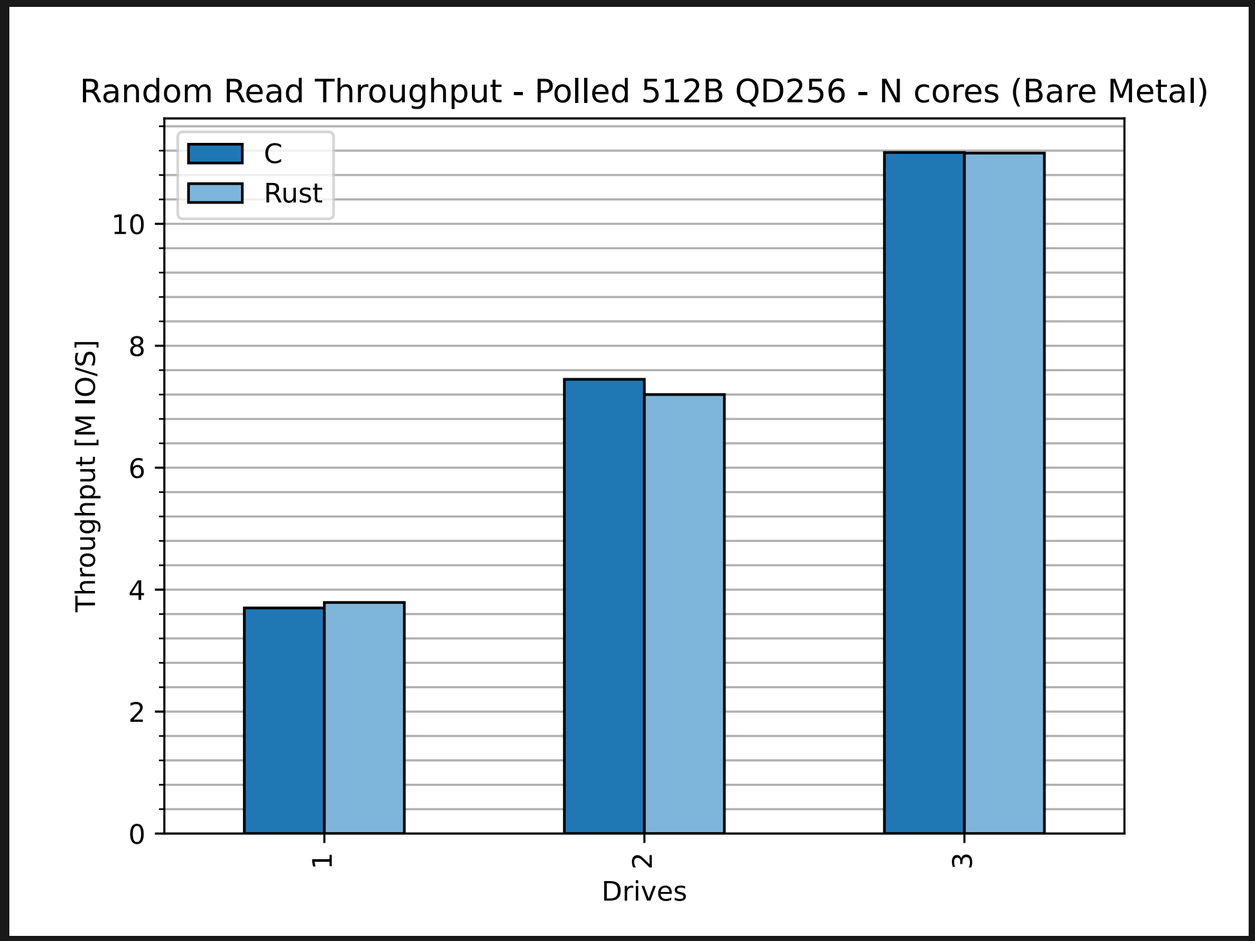
THROUGHPUT VS QUEUE DEPTH





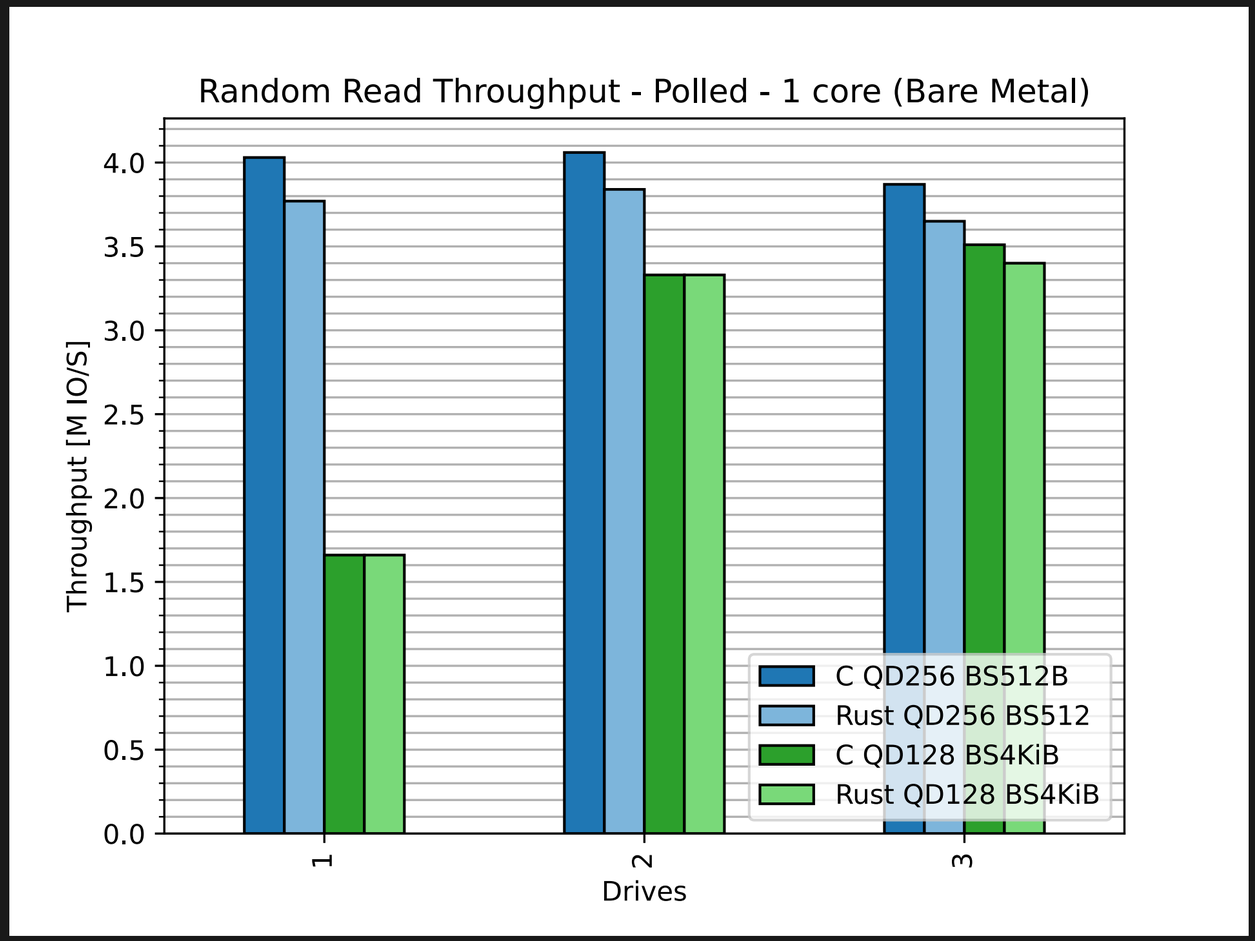
INTEL MEMPEK1W016GA, 12th Gen Intel(R) Core(TM) i5-12600, PCIe 3.0 x2, Linux 6.5-rc5+rust_next+rnvme

POLLED - HORIZONTAL SCALING



AMD EPYC 7313 3x INTEL P5800x 16GT/s x4 7.88 GB/s (PCIe 4), DATA FROM RUST/C NVME ON LINUX 6.1

POLLED - VERTICAL SCALING

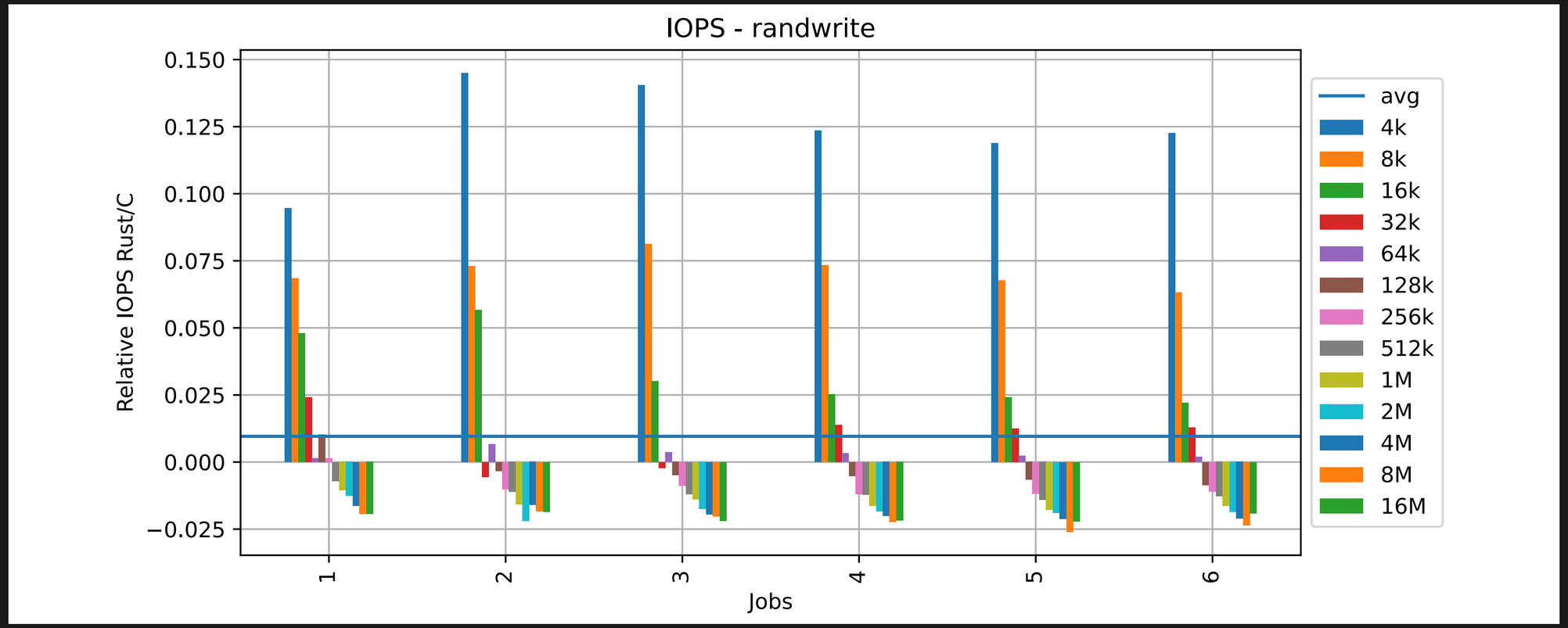


AMD EPYC 7313 3x INTEL P5800x 16GT/s x4 7.88 GB/s (PCIe 4), DATA FROM RUST/C NVME ON LINUX 6.1

RUST null_blk

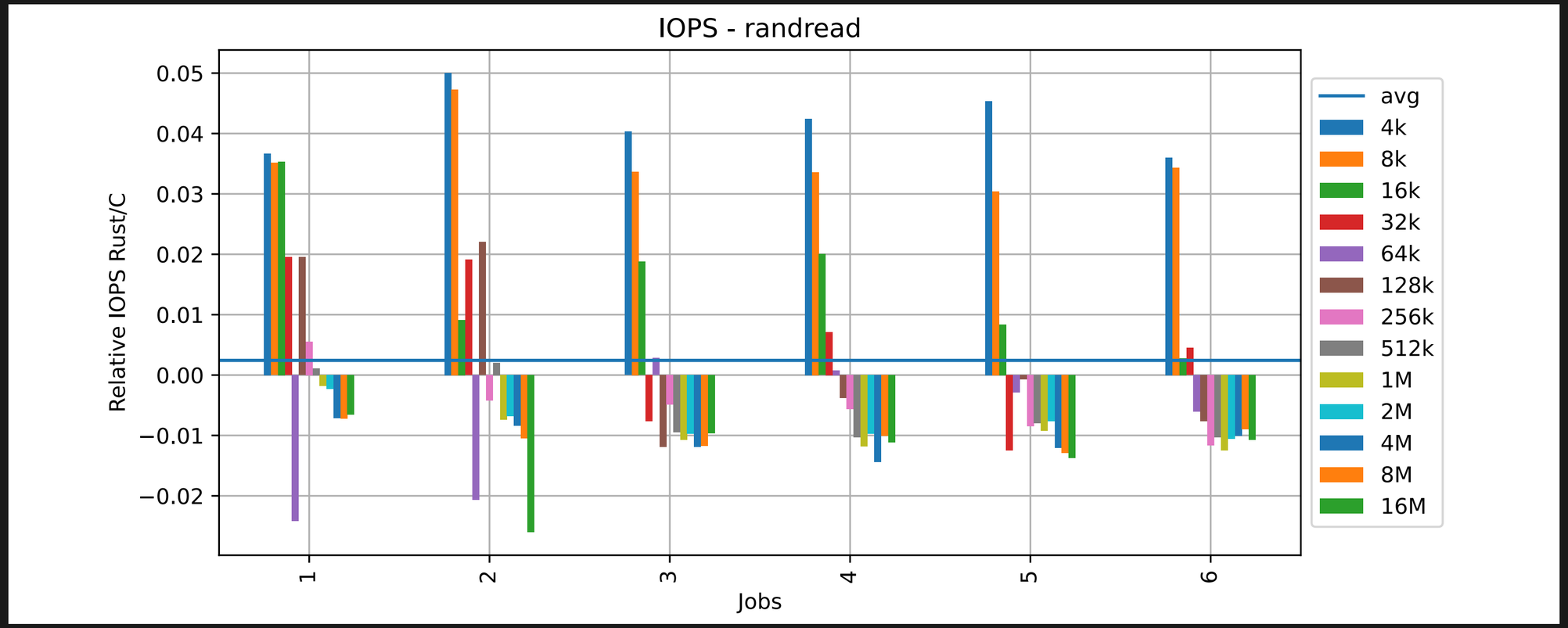
- Simple demonstrator for Rust block APIs
- Allow block community to ease into Rust
- Remove potential memory safety issues in the process:
  - C null_blk is 256 commits (as of 6.1)
  - 27% (68) are bug fixes
  - **41%** (28) of fixes are fixes for memory safety issues
- Demonstrator as submitted:
  - Limited features set - for now
  - Driver: 147 LoC (100% **safe Rust**)
  - Block API: 585 LoC + 252 LoC pages/radix_tree
  - Average performance over 5 synthetic benchmarks ▶ Better for small BS, worse for large BS

RANDOM WRITE



Intel Alder Lake workstation (i5-12600). 60s `fio` runs on bare metal, pinned workers, io_uring, bs 4k to 1M -> QD 128, bs >= 2M -> QD 64, batch submit/complete -> 16.

RANDOM READ



Intel Alder Lake workstation (i5-12600). 60s `fio` runs on bare metal, pinned workers, io_uring, bs 4k to 1M -> QD 128, bs >= 2M -> QD 64, batch submit/complete -> 16.

QUESTIONS?

REFERENCES

[1] *https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/*.

[2] *https://www.chromium.org/Home/chromium-security/memory-safety/*.

[3] *https://lssna19.sched.com/event/RHaT/writing-linux-kernel-modules-in-safe-rust-geoffrey-thomas-two-sigma-investments-alex-gaynor-alloy*.

[4] A. A. Vasilyev, “Static verification for memory safety of Linux kernel drivers,” Proceedings of ISP RAS, 30:6 (2018), 143–160: *http://dx.doi.org/10.15514/ISPRAS-2018-30(6)-8*.

[5] Linux block IO: introducing multi-queue SSD access on multi-core systems: *https://doi.org/10.1145/2485732.2485740*.

[6] [LSF/MM/BPF TOPIC] blk_mq rust bindings: *https://lore.kernel.org/all/87y1ofj5tt.fsf@metaspace.dk/*.

[7] Memory Safe Languages in Android 13: *https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html*.