MULTI QUEUE LINUX BLOCK DEVICE DRIVERS IN RUST

Linux Plumbers Conference 2023

Andreas Hindborg

Samsung GOST

PROJECT GOALS

- Allow implementation of Linux Kernel block device drivers in Rust 🦀
- Provide **reference** implementations
- Remove risks for early adopters

BLOCK LAYER RUST ABSTRACTIONS

- **Four** parts:
  - blk-mq abstractions
  - NVMe driver
  - Null block driver
  - Dependencies
- **Rebase** ~twice every kernel release cycle
  - After rust-next pull request
  - After kernel release
  - https://github.com/metaspace/linux
    - `null_blk`, `null_blk-6.6`
    - `nvme`, `nvme-6.6`

DEPENDENCIES FOR NVME

- `device::Device` - **Device** abstractions
- `driver::DriverOps` - **Bus** driver abstractions
- `irq::Registration` - **IRQ** abstractions
- `pci::Driver` - **PCI** abstractions
- `io_mem::Resource` - **IO Memory** abstractions
- `dma::Pool` - Coherent **DMA pool** abstractions
- `sync::revocable::Mutex` - **Revocable mutex**
- `revocable::Revocable` - **Revocable objects**
- `pages::Pages` - **Page** allocation (will move to Folios)
- `module_param::ModuleParam` - Module **Parameter** abstractions

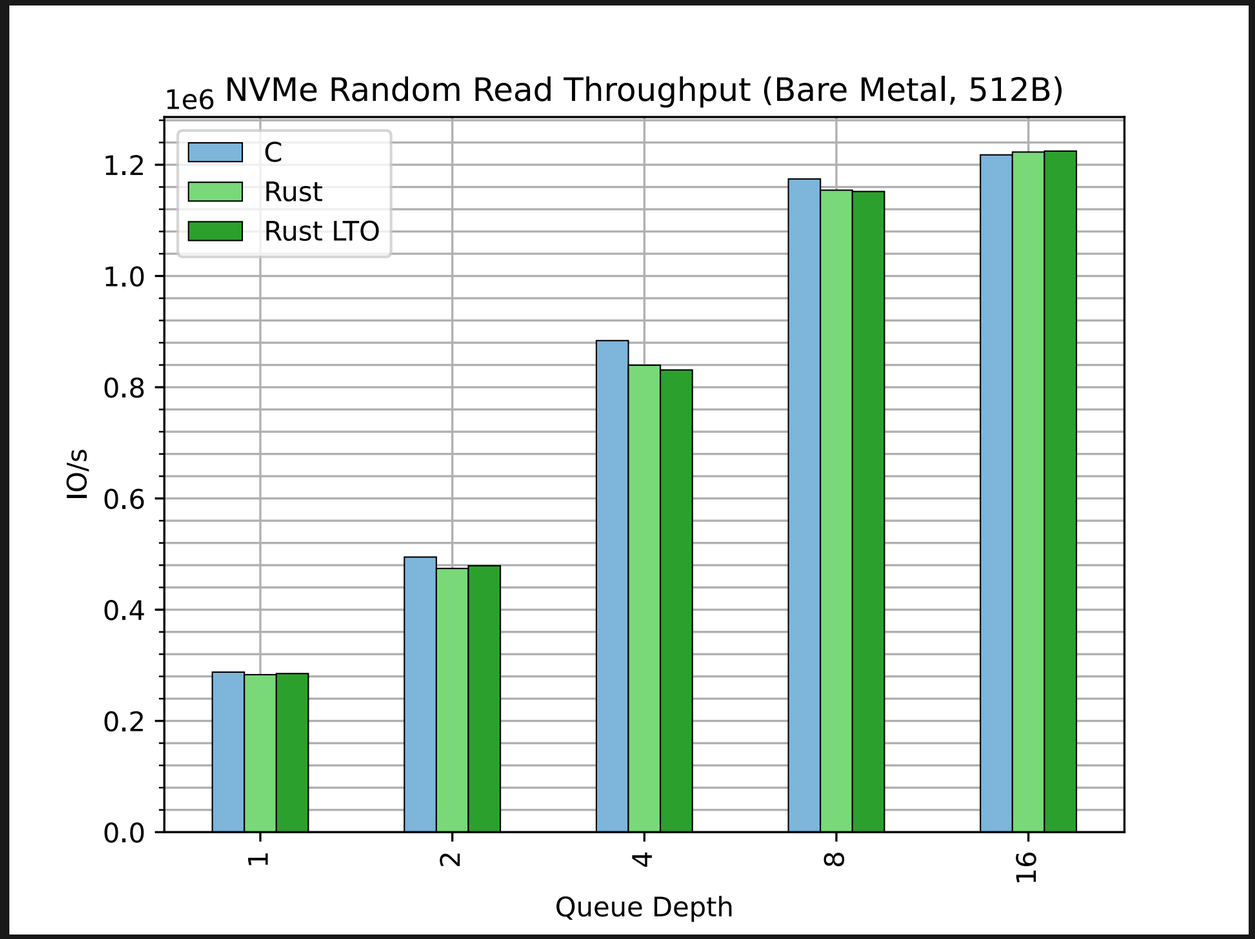
DEPENDENCIES FOR `null_blk`

- `radix_tree::RadixTree` - **Radix Tree** abstractions (may move to xarray)
- `hrtimer::Timer` - High resolution **timer** abstractions
- `pages::Pages` - **Page** allocation (will move to Folios)
- `module_param::ModuleParam` - Module **Parameter** abstractions

NVME

- **PCI** transport only
- Great reference for real driver
- Important for **shaping** `blk-mq` bindings
- In **maintenance** mode while `null_blk` work is ongoing

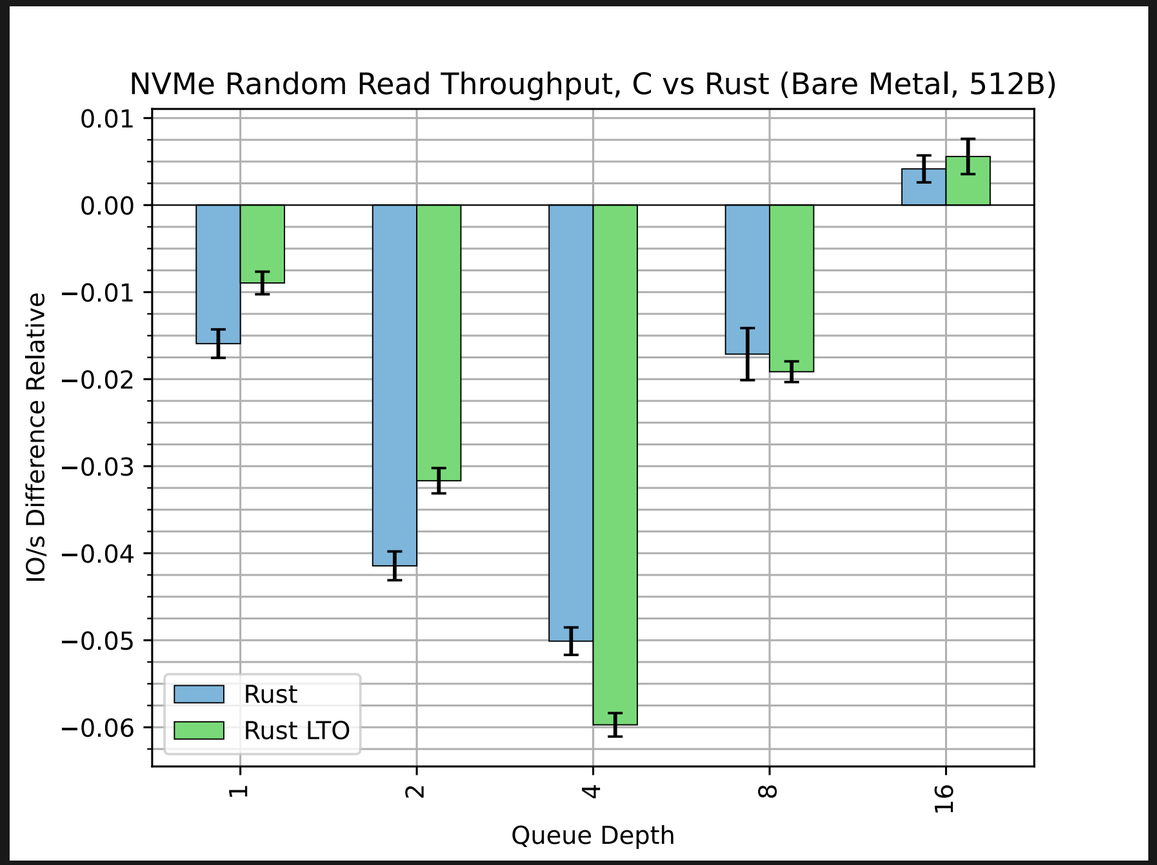
NVME PERFORMANCE



Results based on https://github.com/metaspace/linux/tree/nvme-next-for-6.6 - `rust-next` pull request for 6.6

NVME PERFORMANCE

- Results based on https://github.com/metaspace/linux/tree/nvme-next-for-6.6 - on top of 6.5
- 30 samples
- Difference of means modeled with t-distribution
- P99 confidence intervals



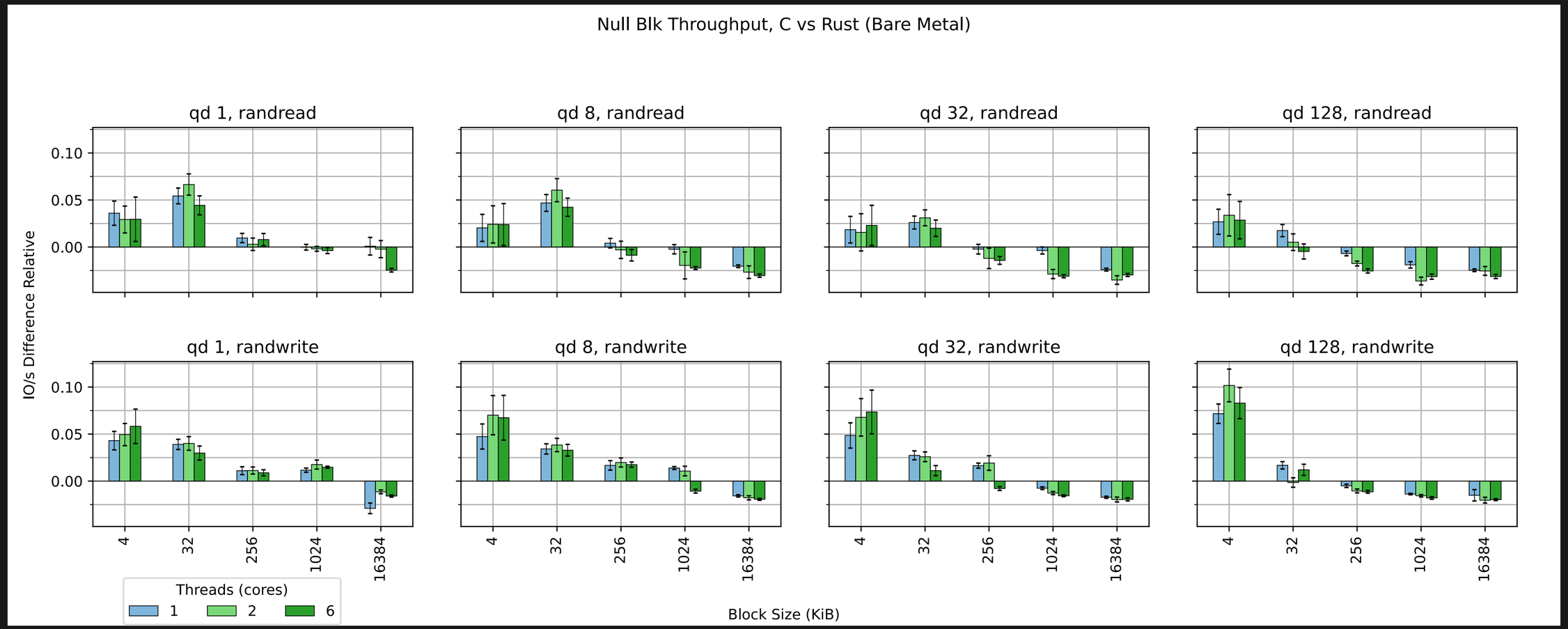
NULL BLOCK - IMPLEMENTED FEATURES

- `blk-mq` support
- Direct completion
- Soft IRQ completion
- Timer completion
- Read and write requests
- Memory backing
- Currently just **210 LoC**

NULL BLOCK - FUTURE WORK

- Bio-based submission
- NUMA support
- Block size configuration
- Multiple devices
- Dynamic device creation/destruction
- Queue depth configuration
- Queue count configuration
- Discard operation support
- Cache emulation
- Bandwidth throttling
- Per node hctx
- IO scheduler configuration
- Blocking submission mode
- Shared tags configuration (for >1 device)
- Zoned storage support
- Bad block simulation
- Poll queues

NULL BLOCK - PERFORMANCE



- *Results based on https://github.com/metaspace/linux/tree/null_blk-6.6 - on top of 6.6*
- *40 samples*
- *Difference of means modeled with t-distribution*
- *P95 confidence intervals*

WAYS TO CONTRIBUTE

- Reviews are appreciated
- Patches welcome - reach out so we can **coordinate**
- Lots of stuff to do - especially in **dependencies**
- Get in touch about shared dependencies

hrtimer API

DEFINE

```
#[pin_data]
struct IntrusiveTimer {
    #[pin]
    timer: Timer<Self>,
    flag: Arc<AtomicBool>,
}
```

INTRUSIVE MAGIC

```
impl_has_timer! {
    impl HasTimer<Self> for IntrusiveTimer { self.timer }
}
```

CALLBACK

```
impl TimerCallback for IntrusiveTimer {
    type Receiver<'a> = Pin<&'a mut IntrusiveTimer>;

    fn run<'a>(this: Self::Receiver<'a>) {
        pr_info!("Timer called\n");
        this.flag.store(true, Ordering::Relaxed);
    }
}
```

CONSTRUCT

```
impl IntrusiveTimer {
    fn new() -> impl PinInit<Self> {
        pin_init!(Self {
            timer <- Timer::new(),
            flag: Arc::try_new(AtomicBool::new(false)).unwrap(),
        })
    }
}
```

RUN

```
stack_pin_init!(let foo = IntrusiveTimer::new());
let flag = foo.flag.clone();
foo.schedule(200_000_000);
while !flag.load(Ordering::Relaxed) {
    //
}
pr_info!("Test OK\n");
```

QUESTIONS ?