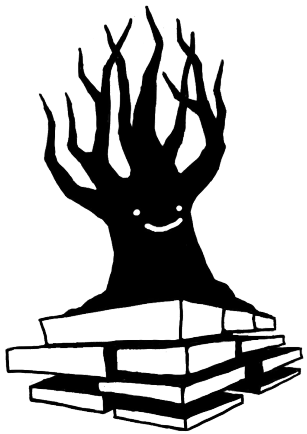
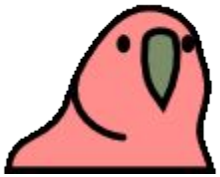


sled and rio

Rust DB + io_uring =



who am I

- building Rust databases since 2014
- previously worked at some social media & infrastructure companies
- for fun, I build and destroy distributed databases
- also for fun, I teach Rust workshops
- lol work

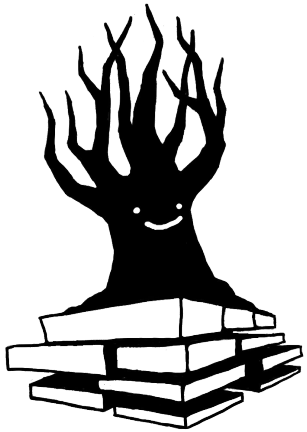
I like databases because they often involve many interesting engineering techniques

common database techniques

- lock-free programming
- replication, consensus, eventual consistency
- correctness testing
- self-tuning systems
- performance work

I started sled to have a single project where I could implement papers I read

sled acts like a concurrent BTreeMap that saves data on disk



```rust
let db = sled::open(path)?; // as in fs::open
db.insert(k, v)?;           // as in BTreeMap::insert
db.get(&k)?;                // as in BTreeMap::get
for kv in db.range(k..) {}  // as in BTreeMap::range
db.remove(&k)?;             // as in BTreeMap::remove
drop(db);                   // fsync and close file
```

Rust is the best DB language

1. Rust will approach Fortran performance in many cases. C/C++ is really limited by aliasing. More compile-time info => better optimizations.
2. Correctness. When there's a segfault, I have a very small set of unsafe blocks to audit to quickly narrow my search down.
3. Compatibility with the great C/C++ perf/debugging tools
4. I can accept code in pull requests with a small fraction of the mental energy as I would need to put into auditing C/C++ due to the compiler's strictness

fast to compile, low friction dev

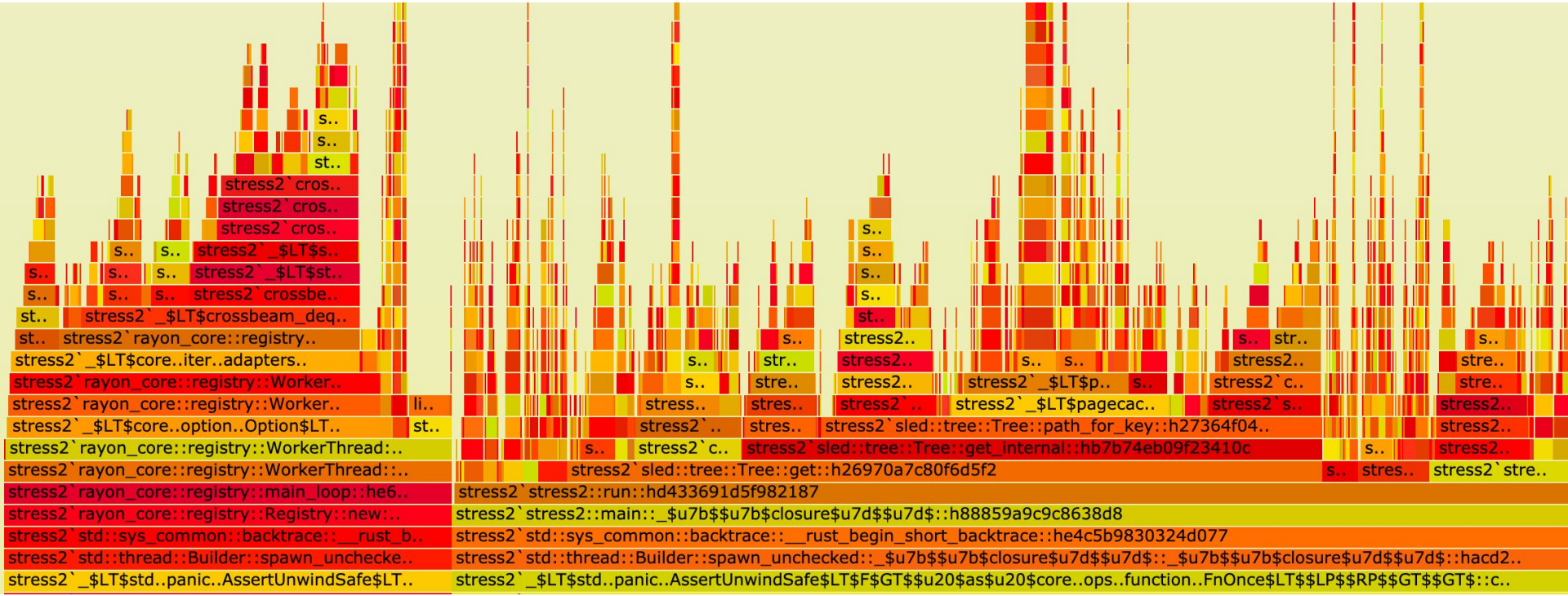
```
src/sled [master •]
λ cb --features=io_uring
   Compiling semver-parser v0.7.0
   Compiling autocfg v0.1.7
   Compiling libc v0.2.66
   Compiling cfg-if v0.1.10
   Compiling byteorder v1.3.2
   Compiling scopeguard v1.0.0
   Compiling crc32fast v1.2.0
   Compiling smallvec v1.1.0
   Compiling lazy_static v1.4.0
   Compiling log v0.4.8
   Compiling lock_api v0.3.2
   Compiling semver v0.9.0
   Compiling crossbeam-utils v0.7.0
   Compiling crossbeam-epoch v0.8.0
   Compiling rustc_version v0.2.3
   Compiling memoffset v0.5.3
   Compiling fxhash v0.2.1
   Compiling parking_lot_core v0.7.0
   Compiling rio v0.9.2
   Compiling fs2 v0.4.3
   Compiling parking_lot v0.10.0
   Compiling sled v0.31.0 (/home/t/src/sled)
    Finished dev [unoptimized + debuginfo] target(s) in 6.91s
```

built-in profiler

- easy to answer "why is this slow?"

```
benchmarks/stress2 [master •]
λ cargo flamegraph -- --duration=5
    Finished release [optimized + debuginfo] target(s) in 0.02s
did 1960859 ops, 132mb RSS
did 2047248 ops, 140mb RSS
did 2011877 ops, 140mb RSS
did 2027944 ops, 144mb RSS
did 2010145 ops, 144mb RSS
did 10058073 total ops in 5 seconds. 2011614 ops/s
pagecache profile:
             op |   min (us) |   med (us) |    90 (us) |    99 (us) |  99.9 (us) | 99.99 (us) |   max (us) |      count |    sum (s)
----------------------------------------------------------------------------------------------------------------------------------
tree:
            get |        1.1 |        2.4 |        2.9 |        3.6 |       31.3 |       40.9 |      936.6 |    9460885 |     24.174
       traverse |        0.9 |        1.9 |        2.2 |        2.7 |       28.9 |       37.0 |      847.5 |   10051577 |     19.923
            set |        3.8 |        5.9 |        9.0 |       19.3 |       93.0 |      609.3 |     2870.5 |     199486 |      1.391
          merge |        3.8 |        6.0 |        9.1 |       19.1 |      104.8 |      551.3 |     1591.2 |      99375 |      0.700
            del |        3.3 |        5.3 |        8.4 |       18.2 |       83.3 |      729.4 |     1575.4 |      99103 |      0.633
          start |   499940.7 |   499940.7 |   499940.7 |   499940.7 |   499940.7 |   499940.7 |   499940.7 |          1 |      0.499
           scan |        0.2 |        0.4 |        3.2 |        4.4 |       13.1 |       37.4 |       68.2 |     347324 |      0.349
            cas |        1.5 |        2.8 |        5.2 |       10.3 |       35.6 |      202.8 |      680.1 |      99889 |      0.334
       rev scan |        0.2 |        0.4 |        2.5 |        3.2 |        5.3 |       34.9 |      488.9 |     349224 |      0.245
tree contention loops: 1
tree split success rates: child(175/175) parent(175/175) root(0/0)
----------------------------------------------------------------------------------------------------------------------------------
pagecache:
            get |        0.1 |        0.2 |        0.5 |        0.7 |        2.3 |       30.9 |      798.1 |   30189291 |      8.724
           link |        1.2 |        2.7 |        5.6 |       10.0 |       84.1 |      591.3 |     2870.5 |     409833 |      1.507
       snapshot |   426021.3 |   426021.3 |   426021.3 |   426021.3 |   426021.3 |   426021.3 |   426021.3 |          1 |      0.425
        replace |        2.2 |        5.5 |        7.2 |       32.9 |      219.7 |      673.3 |     1591.2 |      37430 |      0.259
           pull |        1.0 |        1.8 |        4.1 |        6.3 |       17.5 |       49.0 |      263.0 |     108751 |      0.250
        rewrite |        3.9 |       19.1 |       59.9 |      334.4 |      334.4 |      334.4 |      334.4 |         66 |      0.002
          merge |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |          0 |      0.000
       page_out |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |          0 |      0.000
----------------------------------------------------------------------------------------------------------------------------------
serialization and compression:
    deserialize |        0.0 |        0.1 |        1.6 |        3.4 |       14.0 |       43.5 |      252.7 |     108751 |      0.048
      serialize |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |          0 |      0.000
       compress |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |          0 |      0.000
     decompress |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |          0 |      0.000
----------------------------------------------------------------------------------------------------------------------------------
log:
     reserve sz |       32.1 |      144.5 |      346.2 |     1042.1 |     1435.6 |     1701.8 |     2078.7 |     410430 | 80125290.000
   written bytes |   568069.0 |  2326788.6 |  2928496.4 |  2928496.4 |  2928496.4 |  2928496.4 |  2928496.4 |         33 | 71098964.000
           read |        0.1 |        0.3 |        1.0 |        2.0 |        3.6 |        7.9 |       48.5 |     911284 |      0.377
    reserve lat |        0.2 |        0.5 |        0.7 |        1.4 |        6.4 |      493.9 |      806.1 |     410430 |      0.288
          write |     1639.7 |     4732.7 |     6452.6 |     6716.0 |     6716.0 |     6716.0 |     6716.0 |         33 |      0.142
    make_stable |        0.0 |        0.0 |        0.0 |        0.1 |        0.1 |     5178.4 |     5553.8 |     108784 |      0.121
   assign offset |       18.2 |       42.2 |      196.8 |      233.3 |      233.3 |      233.3 |      233.3 |         33 |      0.002
 assign spinloop |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |          0 |      0.000
     res cvar r |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |          0 |      0.000
     res cvar w |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |        NaN |          0 |      0.000
log reservations: 410430
log res attempts: 417727
----------------------------------------------------------------------------------------------------------------------------------
segment accountant:
          hold |        0.1 |        0.2 |        0.7 |        4.4 |       11.7 |       96.8 |     1544.2 |     447937 |      0.262
       acquire |        0.0 |        0.0 |        0.0 |        2.3 |       24.3 |      302.5 |     1607.2 |     447937 |      0.120
       replace |        0.3 |        3.0 |        4.1 |        5.8 |       38.6 |      308.7 |      488.9 |      37673 |      0.121
          link |        0.1 |        0.2 |        0.4 |        0.7 |        1.2 |      147.3 |      484.1 |     168210 |      0.045
          next |        3.0 |        3.6 |        4.2 |        4.2 |        4.2 |        4.2 |        4.2 |          8 |      0.000
[ perf record: Woken up 68 times to write data ]
[ perf record: Captured and wrote 18.141 MB perf.data (2244 samples) ]
writing flamegraph to "flamegraph.svg"
```

heavy use of flamegraph crate



sled.rs

github.com/flamegraph-rs/flamegraph

@sadisticsystems

1 **b**illion operations in 57 seconds @ 95% reads / 5% writes / small working set

seriously though, it's beta

never use a database less than 5 years old

- site reliability engineering proverb

sled turns 5 this year, so 2020 will be an exciting year for the project

let’s see how it works!

sled architecture

- lock-free index loosely based on the Bw-Tree
- lock-free pagecache loosely based on LLAMA
- log structured storage loosely based on Sprite LFS
- io_uring on huge buffers for writes
  - ➢ io_uring functionality exported as rio crate
- cache based on W-TinyLFU
  - ➢ exported (soon!) as berghain crate

we avoid blocking while reading and writing

setting a key to a new value

1. traverse tree to find the key's leaf
2. modify the leaf to store the new key-value pair

but, we can't block readers or writers while updating

latency

we use a technique called RCU

Read-Copy-Update (RCU)

1. read the old value through an AtomicPtr
2. make a local copy
3. modify the local copy with the desired changes
4. use the compare_and_swap method to install the new version. goto #1 if we fail.
5. use crossbeam_epoch to delay garbage collection until all threads that may have witnessed the old version are finished

readers don’t wait for writers

writers procede optimistically

however, we need to also guarantee that our atomic operations are saved to disk in the same order

buggy solution

1. read
2. mutate local copy
3. CAS

thread descheduled here

4. log to disk

if the log message is delayed, other threads may perform their updates between 3 & 4. if the database crashes, it will load the last item in the log. we have to guarantee our log order matches our in-memory order

data loss

good solution (LLAMA trick)

1. read
2. mutate local copy
3. reserve log slot
4. CAS
5. only fill log reservation if CAS succeeded

by ordering our log reservations between the read and the CAS, we guarantee that the order on-disk will match what actually happened in memory, without using any locks.

how to de get fast io?

- we only write when we have 8mb of data to write sequentially
- we support out-of-order writes
- io_uring

io_uring is an interface for fully asynchronous linux syscalls

the old AIO interface forces O_DIRECT, isn't actually async sometimes, etc...

io_uring began as a response to that, but is far more ambitious

5.1	5.2	5.3	5.4	5.5	5.6
nop	sync_file_range	sendmsg	timeout	timeout_remove	send
readv		recvmsg		accept	recv
writev				async_cancel	fallocate
read_fixed				link_timeout	fadvise
write_fixed				connect	madvise
fsync					openat
poll_add					close
poll_remove					statx
					read
					write
					files_update

it’s 2 ring buffers

- submission
- completion

after setup, it can be run with 0 syscalls (SQPOLL)

io_uring is provided via the rio crate

```rust
let ring = rio::new().expect("create uring");
let file = std::fs::create("file").expect("openat");
let to_write: &[u8] = &[6; 66];
let completion = ring.write_at(&file, &to_write, at);

// if using threads
completion.wait()?;

// if using async
completion.await?
```

```rust
use std::{
    io::self,
    net::{TcpListener, TcpStream},
};

async fn proxy(ring: &rio::Rio, a: &TcpStream, b: &TcpStream) -> io::Result<()> {
    let buf = vec![0_u8; 512];
    loop {
        let read_bytes = ring.read_at(a, &buf, 0).await?;
        let buf = &buf[..read_bytes];
        ring.write_at(b, &buf, 0).await?;
    }
}

fn main() -> io::Result<()> {
    let ring = rio::new()?;
    let acceptor = TcpListener::bind("127.0.0.1:6666")?;

    extreme::run(async {
        // kernel 5.5 and later support TCP accept
        loop {
            let stream = ring.accept(&acceptor).await?;
            dbg!(proxy(&ring, &stream, &stream).await);
        }
    })
}
```

operations are executed
out-of-order

chained operations

connect + send + recv

PLs are DSLs for syscalls

io_uring changes this conversation

over time, BPF may be used to execute logic between chained calls, eg:

accept -> read -> write

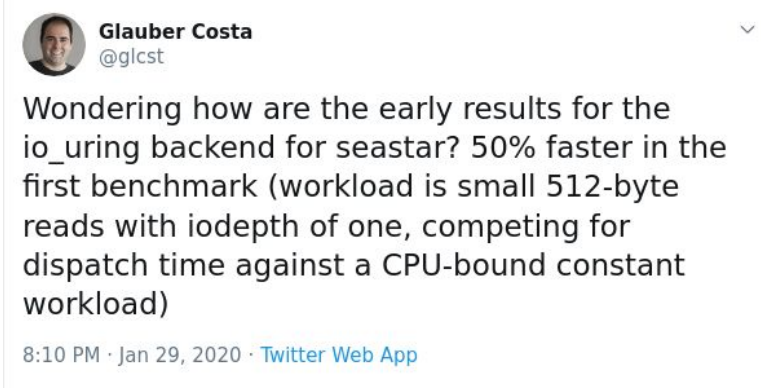
userspace: control plane

kernel: data plane

rio is misuse resistant

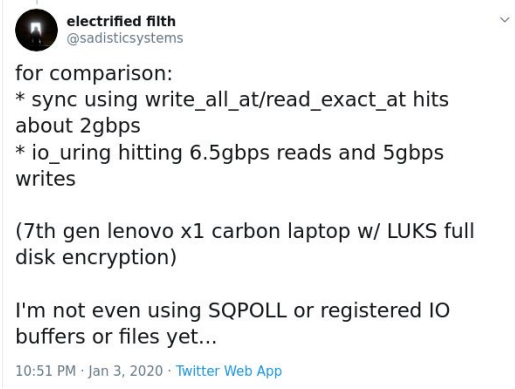
- guarantees Completion events don’t outlive the ring, the buffers, or the files involved.
- automatically handles submissions
- prevents ring overflows that can happen by submitting too many items
- on Drop, the Completion waits for the backing operation to complete, to guarantee no use-after-frees.

Basically all performance-conscious projects are getting ready to migrate to it, and they are measuring impressive results.


Glauber Costa
@glcst
Wondering how are the early results for the io_uring backend for seastar? 50% faster in the first benchmark (workload is small 512-byte reads with iodepth of one, competing for dispatch time against a CPU-bound constant workload)
8:10 PM · Jan 29, 2020 · Twitter Web App


You Retweeted
Mark Papadakis
@markpapadakis
Replying to @Sirupsen and @sadisticsystems
I have been using io_uring for network IO (need 5.5; subtle bugs in earlier versions), for accept, readv, writev. Now close to 80% increase in RPS over non io_uring based alt. Delta even higher the higher the load and connections multiplexed. It's incredible.
8:01 AM · Jan 14, 2020 from Geropotamos, Greece · Tweetbot for iOS


Jens Axboe Retweeted
frevib
@hielkedv
#io_uring vs #epoll: simple echo server. io_uring +99% performance, -45% cpu usage. Wow. @axboe @VincentFree. 🏆io_uring🏆.


electrified filth
@sadisticsystems
for comparison:
* sync using write_all_at/read_exact_at hits about 2gbps
* io_uring hitting 6.5gbps reads and 5gbps writes
(7th gen lenovo x1 carbon laptop w/ LUKS full disk encryption)
I'm not even using SQPOLL or registered IO buffers or files yet...
10:51 PM · Jan 3, 2020 · Twitter Web App

Try them out :)

docs.rs/rio

docs.rs/sled

Our Results To Date

- pure-rust io_uring functionality
- Modified Bw-Tree lock-free architecture (lock-free, log-structured)
- Millions of reads + writes per second (1 billion/minute)
- Minimal configuration
- Multiple keyspace support
- Reactive prefix subscription, replication-friendly
- Merge operators, CRDT-friendly
- Serializable transactions

Where We Want To Go

- Support for all io_uring operations
- Typed trees: cutting deserialization costs for hot keys
- Replication
- Make it more efficient
  - sled is currently a bit disk-hungry, we can dramatically improve this!
- Make it safer! This is the main point before 1.0
  - SQLite-style formal requirements specification & corresponding testing

Help Us Get There!

- Sponsorship allows me to focus all of my time on open source:
  - https://github.com/sponsors/spacejam

- Want to contribute to a cutting-edge and industry-relevant DB?
  - https://github.com/spacejam/sled
  - We love to mentor and teach people about databases!
  - Also check out our active discord channel

I also run Rust trainings!

Thank you :)