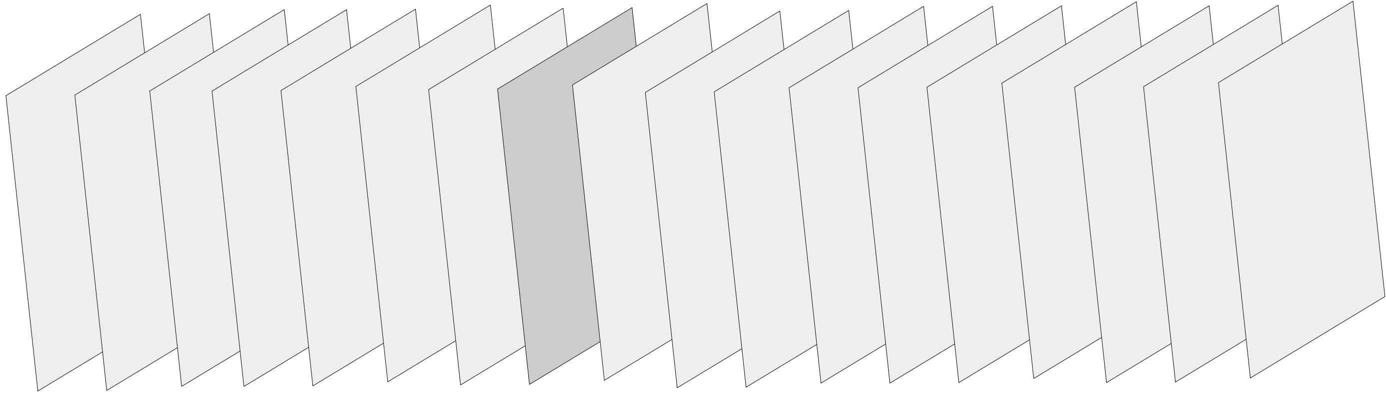
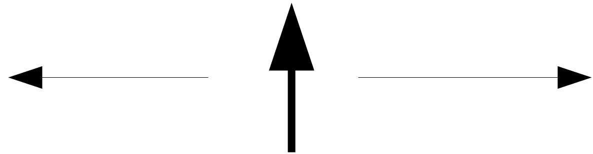
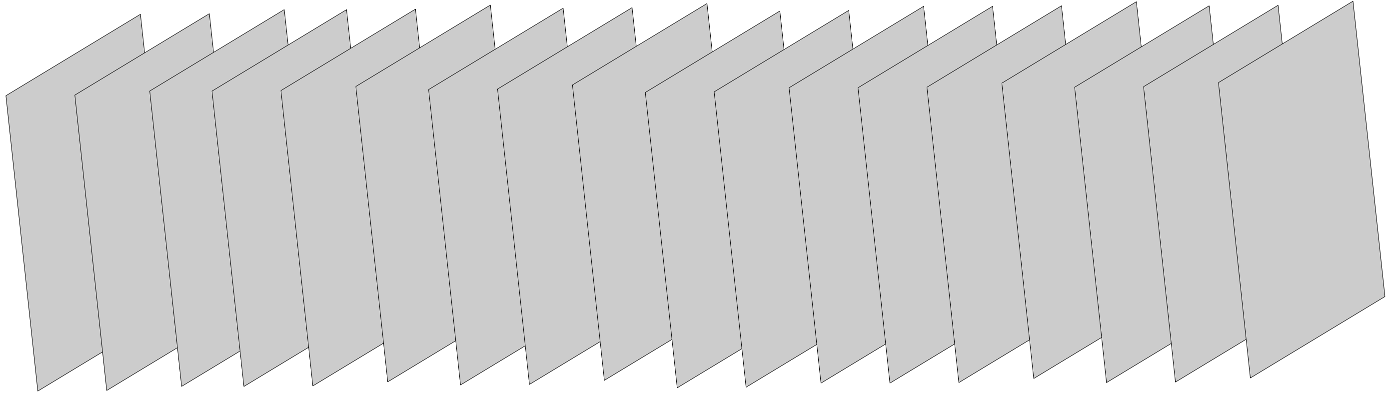
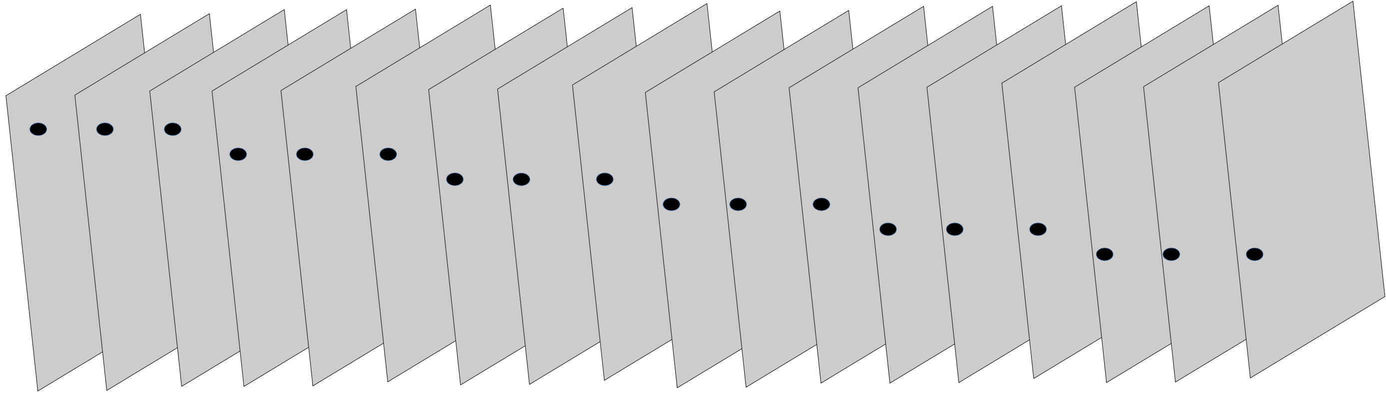
Building An Omniscient Debugger In Rust

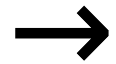
Robert O’Callahan
Pernosco





“Point in time” debugging



Omniscient data analysis and visualization!

Omniscient Debugging

- Build an efficiently searchable database of all program states
  - E.g. all memory and register writes
    ODB, Chronomancer, Chronon…
- How to achieve acceptable overhead in real-world debugging situations?
- What is the ideal debugger UI when you drop “point in time” implementation constraints?

Pernosco

https://pernos.co/debug/WV_SDEC1wkggMQGSnBsg9Q/index.html#f{m[AlDp,AQ_,t[AQ,B7c_,f{e[AlDn,TOM_,s{afw8kn

Toolbox▼ Search...

Stdout/stderr

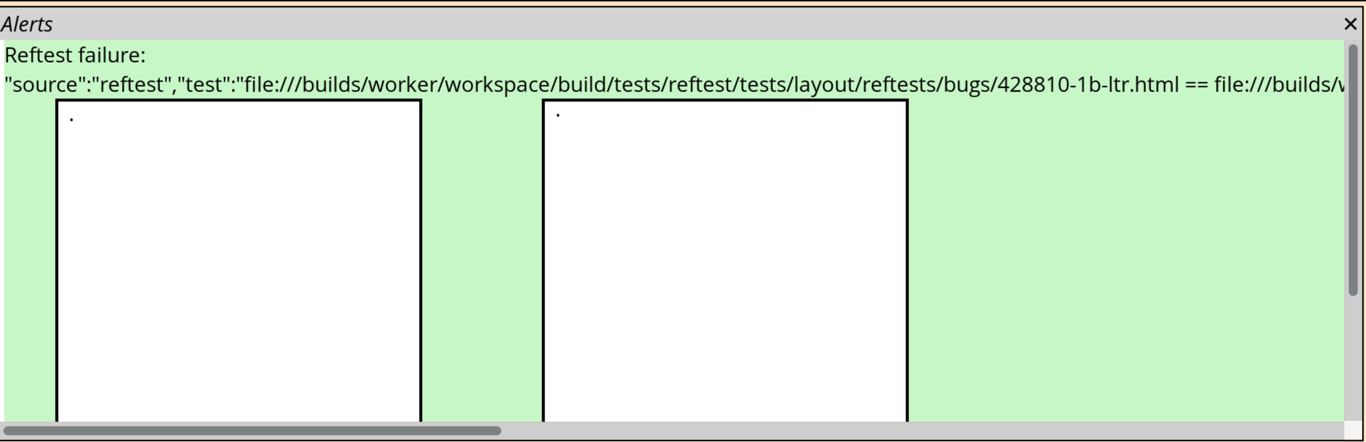
d":null,"source":"reftest","level":"INFO","message":"Saved log: Initializing canvas snapshot"}

{"action":"log","time":1555475111676,"thread":null,"pid":null,"source":"reftest","level":"INFO","message":"Saved log: DoDrawWindow 0,0,800,1000"}

{"action":"log","time":1555475111677,"thread":null,"pid":null,"source":"reftest","level":"INFO","message":"Saved log: [CONTENT] RecordResult fired"}

{"action":"log","time":1555475111677,"thread":null,"pid":null,"source":"reftest","level":"INFO","message":"Saved log: RecordResult fired"}

{"action":"test_end","time":1555475111678,"thread":nul



/builds/worker/workspace/build/src/dom/base/nsGlobalWindowInner.cpp

```
3356        c++;
3357      }
3358  #endif
3359
3360    if (cstr) {
3361      MOZ_LOG(nsContentUtils::DOMDumpLog(), L
3362              ("[Window.Dump] %s", cstr));
3363  #ifdef XP_WIN
3364      PrintToDebugger(cstr);
3365  #endif
3366  #ifdef ANDROID
3367      __android_log_write(ANDROID_LOG_INFO, '
3368  #endif
3369      FILE* fp = gDumpFile ? gDumpFile : stdo
```

*Stack of selected thread (**thread 1975 (firefox-bin)**)*

argv_=0x7ffcef6f4778 ▼ , argc_=1, constructing_=false, ignoresReturnValue_=false ▲ }) at /builds/worker/workspace/build/src/obj-firefox/dom/bindings/WindowBinding.cpp:7162

nsGlobalWindowInner::Dump (this=0x7f0f04606800 ▼ , aStr@0x7ffcef6f45c0.nsTStringRepr<char16_t>= { mData=0x7f0f0352500c ▼ , mLength=29788, mDataFlags=5, mClassFlags=0 ▲ }) at /builds/worker/workspace/build/src/dom/base/nsGlobalWindowInner.cpp:3370

__GI__IO_fputs (str@0x7f0f0576f000="\x A", fp=0x7f0f3750c620 ▼) at /build/glibc-LK5gWL/glibc-2.23/libio/iofputs.c:38

_IO_new_file_xsputn (f=0x7f0f3750c620 ▼ , data=0x7f0f0576f000, n=29788) at /build/glibc-LK5gWL/glibc-2.23/libio/fileops.c:1342

new_do_write (to_do=29788, data@0x7f0f0576f000="\x A", fp=0x7f0f3750c620 ▼) at /build/glibc-LK5gWL/glibc-2.23/libio/fileops.c:518

_IO_new_file_write (f=0x7f0f3750c620 ▼ , data=0x7f0f0576f000, n=29788) at /build/glibc-LK5gWL/glibc-2.23/libio/fileops.c:1263

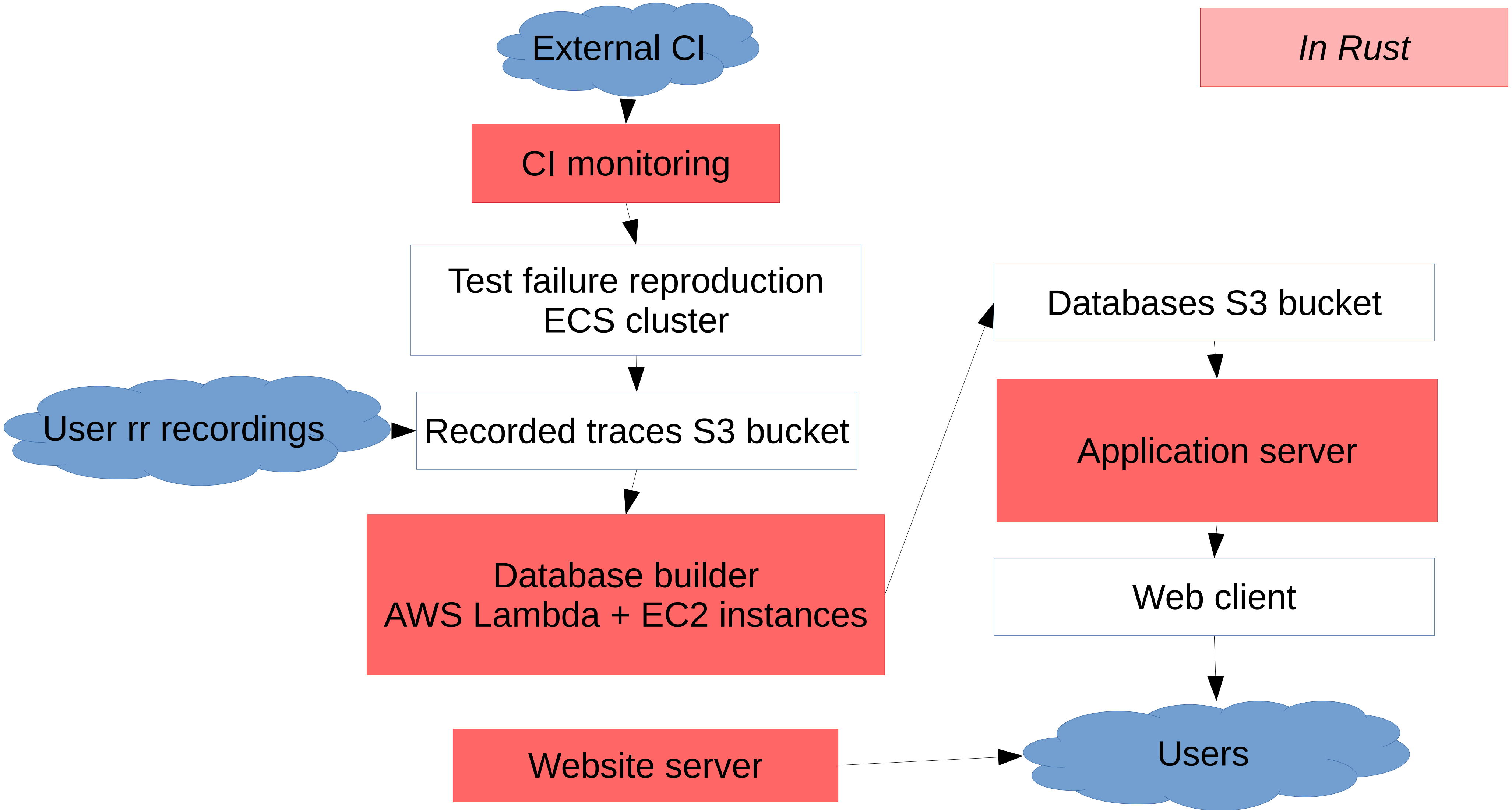
__GI___libc_write at /build/glibc-LK5gWL/glibc-2.23/sysdeps/unix/syscall-template.S:84

0x7f0f38631a7e

_syscall_hook_trampoline_48_8b_3c_24 at /home/khuey/dev/pernosco/rr/src/preload/syscall_hook.S:433

__morestack at /home/khuey/dev/pernosco/rr/src/preload/syscall_hook.S:417

/usr/lib/rr/librrpreload.so+b45


External CI
In Rust
CI monitoring
Test failure reproduction
ECS cluster
Databases S3 bucket
User rr recordings
Recorded traces S3 bucket
Application server
Database builder
AWS Lambda + EC2 instances
Web client
Users
Website server

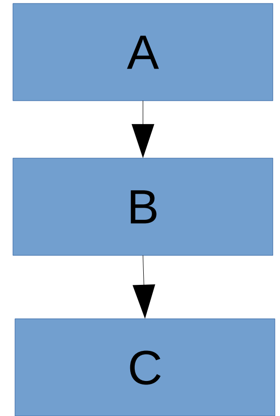
Project Organization

- Mono-repo: 118K lines Rust, 178K total
  - Gitlab
  - AWS-hosted CI runners
- 74 Rust crates
- 2 Cargo workspaces: “main”, “musl”
- 30 built executables, 34 examples
- 8 Docker containers

A Word About Microservices: “No”

- Each deployed service adds complexity
  - Version skew
  - Failure modes
- Only split out a service when it needs to start, stop, fail or update independently of other code

Taming Build Times



- Deep crate graphs → large binaries, slow build times, especially during linking
- Move non-deployed binaries (examples, tools, tests) to “toplevel” crate
- Fold all tests into a single “suite” binary
- Fold all other binaries into a single “multitool” binary
  - Use “exec stubs” that delegate to multitool

Taming Build Times

- Linkers do not handle DWARF debuginfo and Rust well
- If you only use a tiny part of a crate, ALL the debuginfo for that crate is linked in
  - Rust relies on –gc-sections to extract only the used functions from a library
  - --gc-sections does not affect DWARF linking
- My patch in latest LLD partially fixes this, will be in LLVM 9

Testing

- Rust makes it easy to write tests
- 9500 lines of Rust tests
  - Plus 4300 lines of “test subjects” code
- 1500 lines of Python+Selenium Web client tests
- 393 non-test “assert!”s
- 195 uses of “unsafe” in 18 (out of 74) crates
  - Almost all code is safe

Third Party Crates

- Massive advantage for Rust over C++
- We use lots of third-party crates
- Ability to (temporarily?) fork a crate and patch it is essential (and works well)
- Try to contribute upstream with bug reports and PRs

Third Party Crates: Base

- Popular Rust crates that are great:
  - serde (bincode, JSON, YAML)
  - nom parser
  - ring (crypto)
- Less great:
  - tokio (error prone, hard to debug)

Parallelism

- Huge Rust superpower
- Pernosco DB builds do a lot of heavy lifting
  - 10s to 100s of GB of data produced
  - Saturate a c5d.18xlarge for ~hour (36 cores)
    - Only USD 0.70 on spot!
- Only 1 data race bug in history of project
  - In an obsolete crossbeam version!

Third Party Crates: Parallelism

- Rayon is great for data parallelism but we don’t get much of that
- Crossbeam channels
- Scoped threads and thread pools
  - Have been using scopedpool
  - Moving to Crossbeam scoped threads and Rayon thread pools: separating the concepts works better

Third Party Dependencies: AWS

- Rusoto EC2, ECR, ECS, Lambda, S3, SES, SNS, CloudFormation
- Not very idiomatic Rust but gets the job done
- Need to layer on retry logic for network errors etc in practice (other AWS SDKs do this for you)
- lambda_runtime for AWS Lambda

Third Party Dependencies: APIs

- dkregistry (Docker)
- hubcaps (Github)
- jsonwebtoken for JWT/Oauth
- travis

Third Party Dependencies: Web

- actix-web for static site server
- hyper/http for simple dynamic serving
  - All behind a Traefik (Go) reverse proxy/TLS terminator
  - Traefik has issues but no better alternative known...
- tokio-tungstenite for Websocket server
- rustls (TLS) --- OpenSSL is horrible to deploy
  - Ok if you don’t talk to legacy servers/clients

Third Party Dependencies: Misc

- Servo “gaol” for sandboxing (forked)
- lapin for AMQP (annoying)
- capnp for Cap’n Proto (good)
- clap/structopt for CLI parsing (brilliant)
- xmas-elf for ELF parsing (obsolete, should use goblin)
- gimli for DWARF parsing (good)
- petgraph for graphs (unmaintained)

Rust Issues: Error Handling

- Error handling: using failure and error-chain, but no clear/easy path for implementing complex errors

Rust Issues: Async

- Lots of async code
- Mostly on futures 0.1 but some crates use 0.2
- Our nastiest code is async code
- Really looking forward to async/await and reunification of the async ecosystem

Rust Issues: Dependencies

- Need alerts when a dependent crate has known critical bugs or security issues
- Need crates to adopt stable 1.0 APIs

Rust Issues: Build Times

- Building is still far slower than C++
- Our project is not very big and still takes 10 minutes on a monster machine
- Build pipelining should help but there is much further to go to match C++
- Not entirely a fair comparison since C++ requires manual code structuring to achieve fast parallel builds

Rust Issues: Debugging

- Rust non-optimized builds are very slow
- Rust optimized builds are hard to debug
  - LLVM optimization passes lose track of variables
- The problem will get worse with async/await
- Pernosco can help but LLVM needs work

Rust Issues: IDEs

- If anything, has gone backwards over the last 3 years
- RLS does not scale to large multi-crate workspaces
- Not much seems to be happening :-(

Positive Impressions

- High probability “if it compiles, it works”
  - Can refactor with confidence
- Can design types to prevent many kinds of mistakes
  - E.g. String and numeric newtypes
- Easy deployment of self-contained binaries

Positive Impressions

- Confident handling of untrusted input
- Bugs are easier to reproduce and fix
  - Only one race-detection bug
  - Only a few memory corruption bugs
- Time and space efficiency
  - Fairly easy to profile and optimize at high and low levels

Conclusions

- It has been fun writing Pernosco in Rust
- The core language and library and the crate ecosystem are getting steadily better
- Even a very small team can get a lot done in Rust and get very solid results