RR

Leveraging Record-Replay for code exploration and debugging

Felix Klock (`pnkfelix@pnkfx.org`)

Rust Platform Team at Amazon Web Services

RR

Leveraging Record-Replay for code exploration and debugging

RR: Record all I/O events for a program, and then Replay them at the same points in time.

Me

Amazonian since October 2020; Mozilla before that.

Compiler hacker > 20 years (with years of garbage collector development mixed in).

Rustacean since 2013: allocator API, dropck, non-zeroing drop, non-lexical lifetimes (NLL)

Debuggers

Love/Hate relationship with debuggers for 25 years

- Debugger: Irreplaceable Tool? Crutch? Distraction?
- "A debugger is no substitute for thinking."   --John Guttag, and many others
- Counter: "Stop thinking, and look!"   --David Agans

Exploration: Toy program

```
dev-dsk-pnkfelix-1e-8e29ddd4 % ./target/debug/time-passages 2
line   0 now: 21:23:06 UTC, update freq: 2s
line   1 now: 21:23:08 UTC, 0 calls to sleep, ##
line   2 now: 21:23:10 UTC, 1 call to sleep,  ####
line   3 now: 21:23:12 UTC, 1 call to sleep,  ######
1
line   4 replaced: 2 with new frequency: 1
line   5 now: 21:23:14 UTC, 1 call to sleep,  ########
line   6 now: 21:23:15 UTC, 1 call to sleep,  #########
line   7 now: 21:23:16 UTC, 1 call to sleep,  ##########
```

Spelunking

tp-1: program

tp-2: gdb on program (no TUI)

tp-3: gdb TUI on program.

Spelunking

... with a typical debugger is not fun

Sometimes want to inspect state that's been overwritten.

Every debugger step is a one-way door.

Scientific Method of Debugger Usage

1. Hypothesize points of interest and place breakpoints accordingly

2. Resume program

3. At breakpoints: inspect data, step forward, think, and/or go to steps 1 or 2

4. On misstep: Curse, then restart program from start

Every debugger step is a one-way door.

Other exploration tools

Logs (and `println!`-based debugging) are useful in part because you can visualize multiple points in time at once in the output.

No such thing as "whoops I stepped too far" when reading a transcript.

(There *is* "whoops my instrumentation was broken.")

> "Clicking over statements takes longer than scanning the output of judiciously-placed displays. It takes less time to decide where to put print statements than to single-step to the critical section of code, even assuming we know where that is." --Kernighan and Pike

(unsurprising if you agonize on each step over whether you're *ready* to take it.)

But what if…

tp-4: rr record

tp-5a, tp-5b: rr replay

Talk Outline

Spelunking

Differentiation

Constraints, and how RR overcame them.

Practicalities

Jumping from GDB to RR; how to use RR in the cloud on AWS EC2.

Demo

Differentiation

How RR differentiates itself

Reversible Debugging is cool

... Record-Replay is *useful*

Origin of RR

Brain child of Robert O'Callahan while at Mozilla Research

Motivation: Intermittent test failures

e.g. failure occurs 1/1000th of the time → CI reports failures; engineers cannot reproduce; clearly "unrelated" to commit.

huge cost. lots of time wasted on fruitless investigations.

how to cope? (disable "flaky" tests? but they represent real bugs!)

(Old) Idea: record sources of non-determinism, then replay them.

Old Idea → *Many* predecessors...

...but nothing has had serious customer adoption.

Binary instrumentation (PinPlay, Nirvana, Chronomancer): High overhead

Host virtualization (ReVirt, PANDA): High overhead; whole host FBFW

"Best effort" (ReSpec, ODR, Jockey): Move goalpost (e.g. only reproduce output, not execution)

Hardware solutions (GHS Time Machine): Deployment difficult/costly at best

...

RR basics

Things work off the shelf

- Stock hardware (x86_64) and OS (Linux).
  - No kernel modification, not even kernel modules
  - (caveats re Intel vs AMD)
- Low overhead (record time **1.2x-1.5x**)
  - Single process (but includes spawned threads)
  - No code instrumentation!
    - (Too hard to build and maintain. High overhead.)
    - Good for use-case of JIT'ed code in Firefox.
- Works on real programs, not just toys

RR basics

Leverage "modern" features of hardware and Linux

- A Linux process has system call results and signals as inputs to deterministic user-space CPU execution.
  - Capture system call results and signals via `ptrace`
- No shared memory accesses: Limit process to single core.
  - (Also: no sharing memory with outside processes, nor GPU.)
  - Known major limitation of RR.
  - (Can record multiple threads! Just not truly parallel.)
- Use HW perf counters to time async event delivery (practically zero cost!)
- ...etc (seccomp-bpf, DESCHED events; see Robert O'Callahan's talk)

"A lot of these features are only working for us by accident..."   -- Robert O'Callahan

Practicalities

Platform compatibility?

How to use it?

RR-compatible platforms

- Intel x86_64 Linux: "just works"
- AMD Ryzen x86_64 Linux works too
    - (see epic rr-debugger/rr#2034), but:
    - you need to disable SpecLockMap Model Specific Register
        - rr provides a script to automate this.
- ARM Linux is under development (Graviton has been demo'ed)

All other non-Linux: Sorry

How do I know if its going to work?

Check if your dev machine supports Performance Events:

```
% dmesg | grep PMU
```

Yay: `Performance Events: Skylake events, Intel PMU driver.`

Boo: `Performance Events: unsupported p6 CPU model 85 no PMU driver, software events only.`

For x86_64 on AWS EC2

RR docs say it should work with one of: c5[d].9xlarge, c5[d]18.xlarge, m5[d].12xlarge, m5[d].24xlarge, or a bare metal instance type.

You need a dedicated socket

I've tested it. It does work on the cloud.

Jumping from GDB to RR

GDB cheat sheet

```
(gdb) break FILE:LINE
(gdb) break FUNCTION
(gdb) rbreak PATTERN

(gdb) continue
(gdb) next
(gdb) step
(gdb) finish

(gdb) watch EXPRESSION
(gdb) watch VARIABLE
```

(`up`, `down`, `p`); Enter: repeats last command; Ctrl-x a: enters TUI mode

RR cheat sheet

```
(rr) break FILE:LINE                (rr) when
(rr) break FUNCTION                 (rr) when-ticks
(rr) rbreak PATTERN

(rr) continue                       (rr) reverse-continue
(rr) next                           (rr) reverse-next
(rr) step                           (rr) reverse-step
(rr) finish                         (rr) reverse-finish

(rr) watch EXPRESSION
(rr) watch VARIABLE
```

(`up`, `down`, `p`); Enter: repeats last command; Ctrl-x a: enters TUI mode

Continued Demo

tp-6: replay without gdb via `rr replay -a`

(Did you notice: replay is faster than running the original?)

tp-7: replay with event-numbered output via `rr replay -M`

BTW have you noticed the error in the output yet? tp-8: event 569 is the event observing the error. Lets use `rr replay -g 569`

(in this case, using a wrapper around `rust-gdb` to pass in the argument)

tp-9: demo of emacs gud-gdb with rust-rr-replay to find the error

Workflows

Leverage deterministic replay!

- Run without the debugger: `rr replay -a`
- Hard-code memory addresses to examine

Leverage event numbering!

- `rr -M replay` annotates writes to stdio with `[rr PID EVENT]`
- `when` command returns RR's internal current event number
- `rr replay -g EVENT` tells RR to lauch debugger at that specific event
- Pass EVENT to gdb's run to tell RR to restart replay from that event

Workflows

Change Mindset!

- One-Way Door has been replaced with a Two-Way Door
- Debugger *can* be useful exploratory tool

FAQ

Q: How can (forward) replay yield "efficient" reverse execution?

Answer: `rr replay` also checkpoints program state periodically. In general, each reverse-execution runs *twice* from the most recent checkpoint.

Q: 1.2x time is nice; what about space? How big are recorded traces?

Answer: The one for this demo is 2.8 MB. A recent `rustc` one (for a ten line program) was 1.7 GB.

Q: if my program spawns another program, does `rr` trace the child program?

Answer: No!

Answer: More specifically:

`rr record` will capture any *events* that affect the traced parent process.

then `rr replay` will reproduce all those same events, on *just* that process. No need to replay the child!

(There are caveats; see namely `rr pack`)

IDE Usage

Known to work (at least for C): VS Code, CLion, QtCreator, Eclipse, Emacs, gdbgui

See RR wiki: https://github.com/rr-debugger/rr/wiki/Using-rr-in-an-IDE

- Most need debugger console to issue commands like `reverse-next` or `reverse-continue`
- CLion has reverse-* buttons via an UndoDB plugin.

References

RR Project: https://rr-project.org/

Robert O'Callahan's TCE 2015 talk https://youtu.be/H4iNuufAe_8 and Linux.conf.au 2016 talk https://youtu.be/ytNlefY8PIE

Epic reversible-debugging blog: https://jakob.engbloms.se/archives/tag/record-replay (Jakob Engblom)

"Give me 15min and I'll change your view of GDB": https://www.youtube.com/watch?v=PorfLSr3DDI (Greg Law)

"55,000+ lines of Rust code later: A debugger is born!": `rd`, a port of `rr` from C++ to Rust. (Sidharth Kshatriya) https://github.com/sidkshatriya/me/blob/master/004-A-debugger-is-born.md

Thanks