Repeatable execution and why operating systems should support it

(. . . maybe)

Stephen Kell

stephen.kell@cl.cam.ac.uk

Computer Laboratory
University of Cambridge

Repeatable execution: what and why?

```
$ ./meaning <universe
41
Segmentation fault
```

Repeatable execution: what and why?

```
$ ./meaning <universe # "live execution"
41
Segmentation fault
$ replay-it <log       # "replayed exec." <- implement!
41
Segmentation fault
```

Many uses!

- during development: bug reproduction
- “science”: distributing reproducible behaviour
- more! (ask me later)

State of the practice

“Poor person’s” approach: virtual machine images

- bundle whole fs image $\rightarrow$ repeatable? (unsound)
- cumbersome
- requires anticipation

Hypervisors can also record-replay…

- e.g. VMware Workstation
- (er, Xen maybe?)

… with some neat added value (e.g. Aftersight)

State of the art: `rr`

```
$ rr --record ./meaning <universe # records "input"
$ rr --replay                     # replay from recording
```

Problem solved? Not quite:

- opaque to tools (gdb server; profiling?)
- (can’t record all processes/executions)
- logs are fairly large
- $\rightarrow$ definitely not “on by default”

How it works

Record all “input”, a.k.a. sources of nondeterminism

- input data (≈ outputs of system calls)
  - ptrace() system calls
- thread scheduling & descheduling points
  - more ptrace()
  - clever sampling of perf counters
- memory interleaving [, weak memory effects. . .]
  - serialize the program!
  - i.e. no solution (needs hw changes)

Note: no reduction in nondeterminism during live execution

Why in the OS?

The OS seems the Right Place to implement repeatability

- hypervisor: wrong granularities!
  - want to replay processes (/trees), not VMs
  - want logs in terms of files, not blocks
- user-level: OS interfaces need fixing, at best
- user-level: logs are too big

Replay sits well alongside the *file* abstraction

- file ≈ human-meaningful unit of storage
- log ≈ human-meaningful unit of replay

Practical step 1: is “on by default” feasible?

I have no idea. Need experiments! Gut feeling:

- most desktop applications: yes
- a few desktop applications: no (network I/O)
- server-side applications: mostly no (network I/O again)
- long-lived apps might need checkpointing (hybrid)

Synergy is with *local storage*:

- OS has exclusive oversight of local block devices
- same affordance as CoW and shared libs
- $\rightarrow$ “shared logs”

Pause. I can go on (and on. . .).

Storage and computation: “it’s complicated”

Von Neumann: “programs are data!”

Unix: “programs on disk are files!”

T.J. Killian: “processes are files!”

me: “files are the outputs of processes!”

n ways to store a file

0. Remember the data, i_0
   - what ordinary OSes do
1. Remember i_1, input to an execution generating i_0
   - code is “input” too!
   - i_1 is a closure computing i_0
2. Remember i_2, input to an execution generating input…
   - and so on…

Conjecture: sometimes the best n is not 0:

- (obvious) data larger than the closure that generates it
- (subtle) when we want tighter control of input variation

Evidence?

```
LOCALE-GEN(8)                                              LOCALE-GEN(8)

NAME
       locale-gen - compile a list of locale definition files

SYNOPSIS
       locale-gen [options] [locale] [language] ...

DESCRIPTION
       Compiled  locale  files take about 50MB of disk space, and most
       users only need few locales.  In order to save disk space, com-
       piled  locale files are not distributed in the locales package,
       but selected locales  are  automatically  generated  when  this
       package is installed by running the locale-gen program.

       If  a  list  of  languages and/or locales is specified as argu-
       ments, then locale-gen only generates these particular  locales
       and  adds  the  new ones to /var/lib/locales/supported.d/local.
       Otherwise it generates all supported locales.
```

Wanted: OS design tightly integrating computation and storage.

Controlled variation (1)

More ambitious "scientific" use-case: modify and re-run

```
$ ./meaning <universe
41
Segmentation fault
$ nano meaning.c          # make a small edit
$ <replay-compilation> # same build, modulo small edit
$ <replay-run>            # same run, modulo input binary
42
```

This is why *files* are important

- users must control allowable variations
- files make sense to user, cf. blocks

Controlled variation (2)

What I'm calling "controlled variation" means

- knowing what input is changing between executions
- another reason why logs should be *human-meaningful*
  - visualise logs ("what input is going in?")
  - *edit* logs? ("take *live reading* of x, rest from log")?

What happens if variation causes *divergence*?

- program takes a different path than before
- … doing different input actions
- … requiring data that isn't in the log!?

One answer: "set it free", and start recording!

Bad approximations of controlled variation

- software build
- software deployment
- software linking & loading
- performance analysis
- other dynamic analyses
  - “same input, instrumented program”
  - this case shouldn’t diverge

A linking perspective (1)

```
$ ld -o myprog prog.o -l1 -l2
```

Static linking means

- embed lib1.a and lib2.a into myprog
- (then garbage-collect)
- i.e. “record” lib1.a and lib2.a as inputs

Dynamic linking means

- “interpreter” / loader loads lib1 and lib2
- i.e. “play live”

A linking perspective (2)

Linking is building an "environment" (collection of libs)

- which can be seen as *program input*...
- ... sufficient for a *prefix* of execution
- i.e. to reach to the end of loading ($\approx$ before main())

Libraries are just collections of loadable segments

- segments are files
- ... either manifest, or described (bss, relocations...)
- like Multics all over again

Wanted: unify linking/loading segments with files

- "statically linked" $\rightarrow$ replayable closure

A linking perspective: I’m not insane

README.md

gotar is a drop-in replacement for `go build` that also includes any static files (e.g. html, templates, and javascript) within the resulting binary.

Installation

```
go install github.com/ConradIrwin/gotar
```

Usage

Instead of running `go build` , run `gotar` . That's it!

Concrete things to work on

Feasibility experiment

- can we make the logs small-ish? for what applications?

Linker/loader integrating closer with “input files”

- sections/segments are overlay fs?

`rr` more flexible about divergence

- first target: support instrumented replay

software deployment/build system exploiting replay

- “log edit” summarise allowable variations in build

“Containers” angle. . .

Divergent replay

Sometimes we want only *near*-replay:

- same input, different program
- slightly modified input, same program
- … or any combination (programs are input!)

This is problematic because of *divergence*

- log no longer matches program input actions

Current `rr` can't deal with any divergence. Research challenge!

- ill-posed problem in general (timing, interactivity)
- sensible for many specific use cases (timing-insensitive, batch)

Relationship to partial evaluation

Lots of steroids

A replayable execution is

- a tarball on steroids
- a closure on steroids
- `make` on steroids
- “functional” (for the cool kids)