Recording and reproducing software executions with library interposition

Introduction

Lack of reproducibility in computational experiments undermines the long-term credibility of science and hinders the day-to-day work of researchers. Reproducing the end result, within a tolerance, using a domain-specific metric, on any sufficiently-powerful platforms is the end goal of reproducibility and the subject of reproducibility research. Being able to reproduce an execution that is identical, with specific exceptions, on the same CPU architecture is a necessary condition for that ultimate end.

The two categories of solutions that predominate today are (1) **sandboxed package management** (Pip, Conda [8], Spack [5], Guix [4], Nix [2]), (2) **virtualization** (Docker[1], CharlieCloud [9], Singularity [7], Vagrant, QEMU, VirtualBox).

Both of these types of solutions require significant additional effort from the user, when the user has already installed their software stack on their native system. A user will need to either **imperatively** install the software in a new environment (resulting in a Docker image, QCOW2 image) or **declaratively** write a specification (resulting in a Spack `package.py`, Nix/Guix derivation, Dockerfile, Vagrantfile, Singularity Definition File) that installs the software in a new environment[2].

Unfortunately, many practitioners of computational science may not know to use these methods when begininning new projects. Even if switching to virtualization or containerization would only take a few hours, many domain scientists are not willing to commit that amount of effort today.

A third category of solution for reproducibility is **record/replay** (rr, CDE [6], ReproZip [3], SciUnits [11]), which requires almost no user-intervention. In this paradigm, record and replay are two programs, where *record* runs a user-supplied program with user-supplied inputs and writes a **replay-package**, which contains all of the relevant data, libraries, and executables. The replay-package can be sent to any machine that the *replayer* supports. The replayer runs the executable with the data and libraries from the replay-package.

The only user-intervention required to achieve same-architecture portability is that the user must (1) run their executable within the record tool and (2) upload the replay-package to a public location. Compare these to re-installing the software imperatively or declaratively, as would be required with other methods.

[1]Note that for the purposes of this paper, we will consider "containerization" a form of virtualization, since container engines virtualize a separate filesystem, process tree, and Linux userspace (although they do *not* virtualize the Linux kernel or other resources).

[2]The "environment" can refer to a sandboxed software environment, such as that provided by Pip virtualenv or a virtual machine environment, such as that provided by Docker or Virtualbox.

The downside of record/replay is that is has historically been slow: recording adds significant overhead to the runtime of the user's program.

This work presents a novel technique for recording in order to accelerate it, and evaluates this novel technique against existing record/replay schemes for performance, portability, and degree-of-reproducibility.

We want to impact domain researchers who create computational scientific experiments and need a way to make those experiments reexecutable, but do not have the time or knowledge to use containerization or virtualization.

Background

Defining "degree of reproducibility"

The ACM defines **reproducibility** as

> the ability to obtain a measurement with stated precision by a different team using the same measurement procedure, the same measuring system, under the same operating conditions, in the same or a different location on multiple trials [1].

The ACM definition of **reproducibility** references a measurement procedure, measurement system, and operating conditions. In order to translate this to computational science, this work will define a "measurement procedure" as executing a program, a "measurement system" as a hardware platform and operating system, and the "operating conditions" as the machine state variables, on which the measurement depends. In this interpretation, operating conditions have to be set by the agent seeking to reproduce an experiment; in practice, these are difficult for the experiment-authors to identify and communicate, so operating conditions are often overlooked.

Each program, operating conditions, hardware platform, etc. either reliably meets or does not reliably meet the desired precision, giving reproducibility a strict binary value. However, it will be useful in researching reproducibility methods to define "more" or "less" reproducible, in a practically useful sense.

We will encode reproducibility as a proposition of a program and set of conditions, holding the hardware platform and operating system constant[3], for this analysis

$$\mathrm{Repro} : \mathrm{Progs} \times \mathcal{P}(\mathrm{Cond}) \rightarrow \{T, F\}.$$

Programs can have several, possibly unknown, set of conditions under which they will reliably meet the requisite precision, denoted here $\mathrm{Required} : \mathrm{Progs} \rightarrow$

[3]Note that "operating system" should be thought of very narrowly, as the syscall table, machine-code calling convention, and a few other unspecified things, e.g. "Linux" would be an OS, not specifically "Ubuntu". The parts of "Ubuntu" that are relevant for reproduction will considered as "operating conditions".

$\mathcal{P}(\mathcal{P}(\text{Cond}))$. This preserves the possibility that there is no such set (the program is always flaky) or that there are multiple sets (either the file system has to be a certain way or the network has to be a certain way). If the operator sets additional, unrelated conditions, without disturbing this set, then the result will still be reproducible.

$$\forall_{p \in \text{Progs}, c \in \text{Conds}} \exists_{c' \in \text{Required}(p)} c \supset c' \implies \text{Repro}(p, c)$$

A "reproducibility technique" is a way of augmenting the measurement system, which allows one to observe the same measurement, but requires fewer operating conditions. We will encode a reproducibility technique as $t \in$ ReproTechs where Wrap : ReproTech $\times$ Prog $\rightarrow$ Prog "wraps" the input program in the reproducibility technique, and Fixed : ReproTech $\rightarrow \mathcal{P}$(Conds) is a the set of conditions that the reproducibiltiy technique removes from the required set; then,

$$\forall_{t \in \text{ReproTech}, p \in \text{Progs}} \text{Required}(\text{Wrap}(\text{t}, \text{p})) = \{c' - \text{Fixed}(t) | c' \in \text{Required}(p)\}.$$

This is practically useful because it reduces the set of conditions an agent has to configure before the code will be reproducible, assuming there were some hypothetical—but unknown—conditions under which it was reproducible in the first place.

Even though there are an infinite number of programs and operating conditions which may or may not be reproducible with certain reproducibility techniques, we can still apply a partial order on reproducibility techniques. We will say a particular reproducibility technique t_0 is "more reproducible" than t_1 when $\text{Fixed}(t_0) \supset \text{Fixed}(t_1)$.

This is only a partial order, so a reproducibility technique that only fixes the filesystem is neither more nor less reproducible than a reproducibility technique that only fixes the network. However, one that fixes both *would* be more reproducible than either.

Defining operating conditions

Here are some sets of operating conditions that a reproducibility technique may or may not fix. Note that we will define several categories of state in terms of what is accessed by the program, "the contents of X that are read by the program". If the program only reads parts of X, only those parts would be elements of Required(p), and only those parts would need to be fixed by Fixed(t) in order to be totally removed in (Required(p) $-$ FixedConds(t)). The reproducibility techniques, since they augment the original program, will get a chance to monitor the program and learn what parts need to be saved. However, capturing all of X is also a valid option.

- **Environment variables**: a mapping of all POSIX environment variables which influence the program to their values, including `$LC_ALL` and `$PATH`.

Shi et al.[10] treat the locale as state, but we consider it a *part* of the environment variable state.

- **File-system contents**: The mapping of all files read by the program excluding procfs, sysfs, and device-files to the part of their metadata and contents read by the program. For every file, the metadata consists of the owner, group, permission bits (including sticky, setuid, and setgid bits), modification/access/creation times, other attributes, and the set of files which share the same inode[4], aka "hardlinks". For regular files, the content is the value returned by repeated `read(...)`; For FIFO (aka named pipes), the content is any buffered, unread data that will be read by the program; For symlinks, the content is the name of their referent; For sockets, the content is just their existence; For directories, the content is just the fact of their existence, since the files in it would be captured above, and directory listing order will be considered in a different category.
- **Directory listing order**: the sequence of entries returned by successive calls to `readdir`, `seekdir`, and similar functions. POSIX makes no guarantee on the order of entries returned by `readdir(...)`, and indeed Linux can return files in a different ordre.
- **File-to-inode mapping**: the mapping of file paths to inode, as seen by `ls -i` or `ls-id` (for directories).
- **Current time**: the current time returned by `gettimeofday` and related functions when they are called by the program.
- **`/dev/random` and `/dev/urandom`**: the sequence of bytes returned by these files up to a finite point. Exactly what point depends on how far the program reads, and their operating condition would include the bytes returned *up to that point.* This point could be arbitrarily large but known by the reproducibility technique during the first "record" run.
- **State of `/proc`**: the information accessed by the program directly or indirectly from files in the procfs. This includes the output of commands like `ps` and the virtual memory contents of every running process. TODO: technically this includes the virtual memory contents of every running process, which is quite a lot. Depending on how we use this category, we may want to exclude it.
- **State of `/sys`**: the information accessed by the program directly or indirectly from files in the sysfs. This includes `sysctl` queries, kernel parameters, kernel modules, and other kernel information.
- **`getrandom/getentropy`**: the sequence of bytes returned by these syscalls, like with `/dev/random`, up to the point that were read by the program.
- **State of device files**: a mapping from seek locations to bytes returned by read, like with `/dev/random`, up to the point that were read by the program.
- **`umask`**: the value of `umask`. `umask` controls the file permissions of files

[4]We will not consdier the actual inode a part of the file-system contents, because few program depend on it, as opposed to the equivalence-class of files with the same inode, and it is difficult for reproducibility methods to force the OS to assign a particular inode, so defining the state that way would be less discerning.

created by certain tools (only those which respect `umask`). `umask` is process-local and inherited by `fork`/`execve`, and different users might set it differently.

- **uid and gid**: the numeric value of the user ID and group ID.
- **`/etc/shadow` and `/etc/passwd`**: the contents in `/etc/shadow` and `/etc/passwd` that affect the program. Usually, this will just be the username of the current user, as accessed by `whoami`.
- **Dynamic library mapping**: the address at which dynamic, relocatable libraries are loaded into virtual memory. This, along with the actual library (see filesystem contents) fully determine the value of function pointers into the library.
- **Heap layout**: the sequence of addresses returned by `sbrk`/`brk`. This, along with the history of `malloc(...)` and `free(...)`, determines the value of pointers returned by `malloc(...)`.
- **Stack layout**:
- **Race conditions**: the resolution for all race conditions in the program. A race condition is when concurrent processes or threads execute operations whose effects overlap and contradict each other, for example two threads both assigning to the same pointer, `*ptr = 5;` and `*ptr = 6*;`. The resolution is what effects actually get applied, for example "after the preveious race, `*ptr` has the value 6". Data can be guarded by a lock, which removes the race on the data, but creates a race on the lock; in some contexts, a data race may induce undefined behavior, but a race on the lock will induce statically-unknown but defined behavior. It may be difficult to view this as state, but a reader should imagine all of the conditions which cause a race condition to resolve a specific way, including microarchitectural state, mapping of processes to cores, state of the scheduler. There are an extremely large number of true state variables that determine the resolution, and this category is a proxy for an equivalence class of states that all have the same observable result.
- **Accesses to the network**: the values returned by `read(...)` on sockets done by the program.
- **Non-deterministic machine instructions**: the intentional effects of instructions like `rdtsc` on the visible state, thereby excluding microarchitectural side-channels.
- **Performance counters**: the value of performance counters used by the program. Performance counters are a way for the microarchitecture to expose details to user-level programs.

However, we do not claim these state categories are exhaustive, in the sense that fixing all of these would always fix the result of every program.

Here are several things we do not consider system-level state:

- **`cpuid`**: The `cpuid` instruction is not truly state, since it is deterministic; it is just a operational condition that is difficult to reach. It is easy to get an x86_64 CPU in 2023; it is not easy to get an One might be able to trap the `cpuid` instruction and force it to lie about the CPU architecture.

- **ISA extensions**: ISA extensions are likewise not truly state, just difficult operational conditions to reach. If `cpuid` can lie about the CPU architecture, the operator can tradeoff between generality (disabling ISA extensions) and performance (enabling ISA extensions).
- **Uninitialized memory**: If pages are zero-initialized, and other sources of non-determinism are fixed up to a particular point, then uninitialized memory would have a deterministic value up to that point.
- **Language-level PRNG and hash seeds**: Many programs and programming languages use an environment variable to seed their pseudo-random number generator (PRNG) or salt their hashes (e.g., `$PYTHONHASHSEED`). Some reproducibility techniques set this manually to a fixed value, but each program may have a different variable to set, so this is not as general. On the other hand, when that variable is unset, programs often use some "upstream" source of randomness (e.g., Python uses `/dev/u?random`). If those upstream sources of randomness are fixed operating conditions, then the PRNG and hash-salt will be deterministic without having to set program-specific variables. Therefore, we will not consider language-level PRNG a separate operating condition.
- **Language-level iteration order of unordered collections**: The address-space layout and hash-salt determines map iteration order in Python, so fixing this "upstream" sources is sufficient to fix map iteration order. Other languages may work differently, but their non-determinism must still arise from system-level sources (OS, time, environment, architecture, etc.), so we will consider the system-level sources rather than language-level state.
- **State of swap file, used memory, and OOM-killer**: We will assume that the Linux out-of-memory (OOM) killer does not take effect. It would be strange for the original run to depend on the OOM killer for a scientific result, and we will simply require that any agent seeking to reproduce the run would have the same or greater amount of free RAM.

Of these, filesystem-contents and environment variables are often associated with reproducibility or portability. The contents of all software libraries, binaries, scripts are determined by filesystem-contents and environment variables, like `$PATH` and `$LD_LIBRARY_PATH`.

On the other hand, operating conditions like the state `/proc`, rarely matter to the results. The PID may change the names of intermediate files (e.g., a pidfile), but domain-relevant observation will probably still be within the relevant precision. Nevertheless, we will consider this state category to be as complete as possible.

Comparison of reproducibility in common reproducibility techniques

Technical Contribution

Library interposition

What does `LD_PRELOAD` do?

What if program does not use libc?

System call tracing

Evaluation

Compared to other record/replay tools, our evaluation shows that our tool has greater performance and portability of the recorder and replayer themselves but does not capture as many sources of non-determinism.

Performance

The performance of record/replay includes the wall time taken to record, wall time taken to replay, and size of the replay-package. If wall time taken to record is too great, users may not record frequently enough or at all. If the wall time taken to replay is to great, downstream users will avoid using the replay-packages in favor of native execution. If the size of the replay-package is too large, then it will not be easy to distribute.

[DSK: in fault tolerance, there is often a factor "expected faults" that is used to think about how important the cost of fault detection (overhead when running, whether or not a fault exists) is versus the cost of fault recovery (overhead when there is a fault). Is there anything similar that could be used here?]

All of the record/replay tools follow the same usage for any given `$program`, which can be written as:

```
$ $record $output_args /tmp/replay-package $program
...

$ $replay $output_args /tmp/replay-package [$program]
...
```

We simply measure time both of these commands and measure the recursive size of `/tmp/replay-package`, which may be a file or directory depending on the record/replay tool.

We use applications from previous publications on workflow provenance.

- Following PASS **?**, PASSv2 **?**, SPADE **?**, and LPM **?**, we use the NIH National Institute of Health's (NIH) National Center for Biotechnology Information (NCBI) Basic Local Alignment Search Tool (BLAST) **?**. BLAST is a genomics tool that finds regions of similarity between nucleotide or protein sequences. We use the test queries and data from real-world usage of the tool that were used by Bates' evaluation of LPM **?**; the other publications do not specific their data.

- Following PASSv2 **?**, SPADE **?**, Hi-Fi **?**, and LPM **?**, we use compiling Linux from source as a benchmark with the default configuration. [DSK: I expect that this depends on the version of Linux in some sense, which perhaps depends on the date of the work too]
- Following the First Provenance Challenge **?** and its 16 entrants (too many to list here), we use the fMRI workflow described by Moreau.
- Kaggle
- FIE and VIC from Sciunits
- Try building CentOS packages like [10]. Could leverage https://pypi.org/project/reprotest/

Portability of recorder and replayers

Rr's recorder requires the kernel parameter `kernel.perf_event_paranoid=1`, which requires superuser privelege to set. An approach which does not require superuser privelege would be more portable.

Using `ptrace` inside a container requires `CAP_SYS_PTRACE`, which is turned off by default.

- Record as normal user
- Record/replay in container

Sources of non-determinism

Prior work

- Library interposition
  - fsatrace
  - OPUS
  - Darshan
  - TREC
- Record/replay tools
  - TREC
  - Reprozip
    - We reuse their concept of multiple replay-targets, but we provide an unprivileged native replay target, and our container images do not depend on Docker or the internet.
  - rr
  - CDE
  - SciUnits

Future work

- Use zpoline or SABRe to reduce caveats
- Evaluate for multi-node HPC codes
- Other sources of non-determinism

[1] Artifact Review and Badging: 2020. *https://www.acm.org/publications/policies/artifact-review-and-badging-current*. Accessed: 2023-01-19.

[2] Bzeznik, B. et al. 2017. Nix as HPC package management system. *Proceedings of the Fourth International Workshop on HPC User Support Tools* (New York, NY, USA, Nov. 2017), 1–6.

[3] Chirigati, F. et al. 2016. ReproZip: Computational Reproducibility With Ease. *Proceedings of the 2016 International Conference on Management of Data* (New York, NY, USA, Jun. 2016), 2085–2088.

[4] Courtès, L. and Wurmus, R. 2015. Reproducible and User-Controlled Software Environments in HPC with Guix. *Euro-Par 2015: Parallel Processing Workshops* (Cham, 2015), 579–591.

[5] Gamblin, T. et al. 2015. The Spack package manager: Bringing order to HPC software chaos. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (New York, NY, USA, Nov. 2015), 1–12.

[6] Guo, P. and Engler, D. 2011. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. *2011 USENIX Annual Technical Conference* (Portland, OR, USA, Jun. 2011).

[7] Kurtzer, G.M. et al. 2017. Singularity: Scientific containers for mobility of compute. *PLOS ONE*. 12, 5 (May 2017), e0177459. DOI:https://doi.org/10.1371/journal.pone.0177459.

[8] Meurer, A. 2014. Conda: A Cross Platform Package Manager for any Binary Distribution.

[9] Priedhorsky, R. and Randles, T. 2017. Charliecloud: Unprivileged containers for user-defined software stacks in HPC. *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver Colorado, Nov. 2017), 1–10.

[10] Shi, Y. et al. 2022. An Experience Report on Producing Verifiable Builds for Large-Scale Commercial Systems. *IEEE Transactions on Software Engineering*. 48, 9 (Sep. 2022), 3361–3377. DOI:https://doi.org/10.1109/TSE.2021.3092692.

[11] Ton That, D.H. et al. 2017. Sciunits: Reusable Research Objects. *2017 IEEE 13th International Conference on e-Science (e-Science)* (Oct. 2017), 374–383.