Reproducible Containers

Omar S. Navarro Leija
omarsa@seas.upenn.edu
University of Pennsylvania

Kelly Shiptoski
kship@seas.upenn.edu
University of Pennsylvania

Ryan G. Scott
rgscott@indiana.edu
Indiana University

Baojun Wang
wangbj@gmail.com
Indiana University

Nicholas Renner
University of Pennsylvania
nrenner@seas.upenn.edu

Ryan R. Newton
Indiana University
rrnewton@indiana.edu

Joseph Devietti
University of Pennsylvania
devietti@cis.upenn.edu

Abstract

We describe the design and implementation of DetTrace, a reproducible container abstraction for Linux implemented in user space. All computation that occurs inside a DetTrace container is a pure function of the initial filesystem state of the container. Reproducible containers can be used for a variety of purposes, including replication for fault-tolerance, reproducible software builds and reproducible data analytics. We use DetTrace to achieve, in an automatic fashion, reproducibility for 12,130 Debian package builds, containing over 800 million lines of code, as well as bioinformatics and machine learning workflows. We show that, while software in each of these domains is initially irreproducible, DetTrace brings reproducibility without requiring any hardware, OS or application changes. DetTrace's performance is dictated by the frequency of system calls: IO-intensive software builds have an average overhead of 3.49×, while a compute-bound bioinformatics workflow is under 2%.

CCS Concepts. • **Software and its engineering** → **Multiprocessing / multiprogramming / multitasking**.

Keywords. reproducibility; determinism; Linux; Docker; software containers

ACM Reference Format:
Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. 2020. Reproducible Containers. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3373376.3378519

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03…$15.00
https://doi.org/10.1145/3373376.3378519

1 Introduction

In data-processing contexts, it is often important to repeatably map each input to a unique, deterministic output. Determinism is useful in software builds [1, 2], reproducible data analytics [3, 4], and fault-tolerant distributed systems [5–8]. Yet in spite of previous work on deterministic languages [9–11] and operating systems [12–14], it is challenging to enforce deterministic output in practice. Thus we seek a practical *container* abstraction to isolate running software and execute it against clearly delimited input data, achieving end-to-end reproducible handling of data. For deployability, it is furthermore essential to provide this guarantee on commodity hardware and software.

Prior work on deterministic operating systems is neither necessary nor sufficient to meet our definition of repeatable data processing, as additional encapsulation is needed to ensure the program starts in the *same state*, without differences in system time or identifiers such as pids. For example, Determinator [12] does not provide a repeatable notion of deterministic time. dOS [14] provides a *deterministic process group* abstraction and can record-and-replay timing-related system calls. But dOS also uses record-and-replay for filesystem interactions, leaving the filesystem outside of the "deterministic box". Thus dOS cannot determinize a data-processing job that necessarily includes file I/O.

Ultimately, determinism is also a weaker property than what we desire – determinism guarantees the same result for repeated runs on a given machine, but in this work we seek identical results *across* machines (Section 7.3), a property we term *reproducibility*. In this paper, we describe the *DetTrace* system that makes strides towards

a *reproducible container* abstraction for x86-64 Linux programs. All code running within the container is forced to run reproducibly, without needing any source code changes. DetTrace encapsulates a Linux process tree and the IO it performs, and runs on commodity hardware and stock Linux distributions. The DetTrace runtime uses a combination of Linux namespaces, bind mounts, and ptrace facilities to intercept system calls and x86 instructions with irreproducible semantics. While DetTrace supports process-level parallelism, threads within a process are currently serialized. While many prior deterministic execution systems support thread-level parallelism, we focus on providing a robust container implementation for complex multi-process workloads.

DetTrace exports the same POSIX API that the process tree inside the container would otherwise see—in each case we simply select one valid behavior out of many to ensure reproducibility. For example, DetTrace provides a reproducible notion of time so the timestamps added to archives by the stock tar utility (stemming from a system call like time) are accordingly reproducible. By enforcing reproducibility at the system call and ISA level, we can transparently export reproducibility to all higher levels including arbitrary language VMs.

This paper makes the following contributions:

- We present the design of DetTrace, the first *reproducible container* abstraction which runs in user-space and supports unmodified programs.
- We give the first taxonomy of the sources of irreproducibility within Linux system calls and x86-64 instructions. For sources we don't handle, we describe the challenges involved in doing so.
- We use DetTrace to run bioinformatics workflows, train TensorFlow models, and build 12,130 Debian packages reproducibly, including large packages like llvm, clang and blender. Much of this software runs irreproducibly by default, but DetTrace is able to render it reproducible.
- We show that DetTrace's performance overhead is correlated with the frequency of system calls in a given workload: e.g., compute-intensive process-parallel bioinformatics workflows can see overheads under 2%, while system-call-intensive software builds see overheads of 3.49× on average.

2 Why is Reproducibility Important?

Reproducibility confers many advantages for software development. Reproducibility is crucial during debugging; bugs that can't be reproduced are much harder to fix. In distributed systems, reproducibility ensures that all replicas behave the same way, accelerating consensus [15] and enabling transparent fault recovery [6]. Reproducibility also has more specific benefits in a range of software domains, which we explore next.

Reproducible Builds. Bitwise-reproducible builds confer many advantages. Builds can run faster thanks to more hits in caches of build artifacts, and builds can be confidently distributed knowing that the same artifact will be produced on any node of a cluster. Reproducible builds also increase software integrity, boosting confidence that a given binary originated from a particular source code release. For these reasons, many Linux distributions, catalyzed by the Debian Reproducible Builds (DRB) [2] effort, target bitwise reproducibility of all their packages. Microsoft is pursuing reproducible software builds [16] with support in its C# and VB compilers [17]. Google's Blaze/Bazel build system [1] encourages a reproducible build ecosystem, to prevent spurious changes due to irreproducibility causing massive additional downstream rebuilds in Google's unified internal software repository.

To achieve reproducibility, every piece of the software build toolchain needs to be reproducible: preprocessors, compilers, scripts used in the build process, and so on. For example, to deal with timestamps that tar records for each file in the tarball, tar was extended with the --clamp-mtime flag [18] to force these timestamps to a fixed value. The modified tar program then needs to be packaged and distributed, and build scripts updated to use the new flag, before reproducibility is achieved.

Whacking one irreproducible mole at a time is predictably laborious. After its first year, DRB had 3,193 of 5,151 supported packages building reproducibly [19], and 18,800 reproducible packages (of 21,782) after two years. After two years of development, we have achieved reproducibility for 100% of our 12,130 supported packages.[1] At present, after more than 5 years of effort by dozens of contributors, 5.2% of current Debian packages (1,289 in all) remain irreproducible. While tools exist to identify sources of irreproducibility [20], fixing a build is still a manual process. Even should DRB reach 100% reproducibility, vigilance would be required to ensure that errant code changes did not reintroduce irreproducibility.

Computational Science. It is perhaps ironic that, while reproducing results is a cornerstone of the scientific method, many computational science tools are not reproducible. While chemical reactions and living organisms are intrinsically variable, there is no good reason for computation to behave similarly. Reproducibility in computational science would accelerate scientific advancement as

[1] The total number of packages differs between our effort and DRB's current set, because we target a fixed set of packages to ensure clean experimental conditions (see Section 6), while new packages are constantly being added to Debian.

scientists could more easily share, reproduce, and build upon one another's work. Improving the reproducibility of scientific results is a key focus for funding agencies [21] and can be seen in our community in the growing artifact evaluation movement. We find a common bioinformatics tool to be irreproducible (Section 6.1).

Machine Learning. There is growing interest in reproducible machine learning (ML) [22]. Reproducibility enables auditing of models to see why they made certain decisions. It also makes it easier to see whether performance changes are attributable to, e.g., conscious design changes or incidental randomness like sampling of the training set. We apply DetTrace to the popular TensorFlow framework, which is well-known to be irreproducible [23, 24].

3 Reproducible Containers

In this work we aim to provide a reproducible container abstraction. The container itself is specified as an initial filesystem state and a program (from the filesystem) to run. This program may in turn launch other programs, e.g., if it is a shell. The programs running in the container may attempt to execute arbitrary x86 instructions and Linux system calls, though we do not guarantee that all such attempts succeed. In our initial prototype, containerized code can interact only with its filesystem and other programs running concurrently in the container. However, in the future we envision limited forms of external interaction being permitted if they preserve reproducibility, e.g., downloading files with known checksums.

Our reproducibility goal can be decomposed into two sub-properties: determinism and portability. For us, determinism is *dataflow* determinism [25], which means that, on a given machine, each read returns the same value on every run. This hides sources of irreproducibility like time and explicit randomness. Determinism implies many useful properties: the filesystem state after all processes have finished will be identical, as will the messages printed to standard output and standard error. Strictly speaking, due to the possibility of external errors that cannot be determinized, e.g., running out of disk space, our guarantee is one of *quasi-determinism* [26]: any two runs are either dataflow deterministic, or at least one run crashes due to an external failure.

Portability means that dataflow determinism extends across machines as well, with varied microarchitectures or OS versions. Our container hides these details by always reporting a simple x86-64 uniprocessor and Linux 4.0 kernel. To be practical, our container can only abstract away from a limited number of hardware or OS details: we do not emulate an x86-64 chip when running on an ARM microcontroller. DetTrace also requires certain hardware and OS support to provide this abstraction, in particular at least an Intel Ivy Bridge processor and Linux 4.12. DetTrace can run on older processors and Linux versions, though with fewer portability guarantees (Section 5.8) or lower performance (Section 5.11). DetTrace also offers a measure of *forward compatibility*. While a future Linux version might introduce new irreproducible APIs that DetTrace would grow to support, today's software using existing Linux APIs cannot access these, and so if software works with DetTrace today it will remain reproducible going forward.

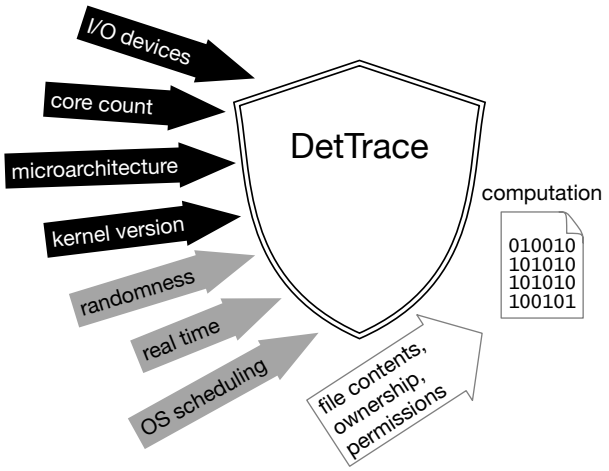

Figure 1. DetTrace containers abstract away both sources of nondeterminism (gray arrows) and nonportability (black arrows), making a DetTrace computation a pure function of its initial file state.

Ultimately, a DetTrace container runs as a pure function of the container configuration and initial filesystem state. File contents affect the computation, but file metadata is only partially visible. Two runs where only the mtime of a file varies will produce the same output, but a permissions change can affect output. Figure 1 illustrates what constitutes an *input*, i.e., what can induce output changes in a DetTrace computation.

Existing container technologies (like Docker) do not provide reproducibility: they are neither deterministic nor portable, as many details of the host OS and processor microarchitecture are directly visible inside the container. Virtual machines offer stronger hardware abstraction but lack determinism and are also quite heavyweight. We believe that the DetTrace reproducible container abstraction delivers significant advantages over existing approaches for domains like building and testing software where reproducibility is critical.

4 Reproducibility Requirements for Linux and x86-64

Code running inside our user-space reproducible container has access to two major interfaces: the x86-64 instruction set and the Linux system call API. Because we place no restrictions on code in the container, it can contain arbitrary instructions and attempt arbitrary system calls. Inspired by the Popek and Goldberg virtualization requirements [27] which define the requirements to provide a virtual machine abstraction, we define the set of requirements for reproducibility. We analyze each documented x86-64 ISA instruction[2] and system call to see if it can be a source of irreproducibility, and under which conditions if so. Of particular importance is identifying *critical* members of an interface—those which permit irreproducibility but which cannot be reliably detected during execution. Any *critical* instruction or system call could silently introduce irreproducibility.

Our use of ptrace means that we see all system calls made from the container, so there is no potential for a critical system call (we also handle vDSO calls, see Section 5.3). If a given system call is a source of irreproducibility, there are many potential mitigations: wrapping the syscall or replacing it entirely with a deterministic counterpart (like time calls), converting it into a nop (like sleep calls), or not supporting it and throwing a (reproducible) container-level error.

There are many sources of irreproducibility within the latest x86-64 instruction set [29]. Privileged instructions are often irreproducible but will raise an exception in our user-level container. Some irreproducible user-level x86-64 instructions are difficult, though possible, to trap. rdrand and rdseed return random bits from a hardware entropy source, and can be trapped at the hypervisor level via the VT-x extensions, but not from ring 0. Instructions like rdpmc (read from performance counter) are sometimes accessible from user-space but can be configured to cause traps via appropriate kernel settings.

Some floating-point instructions like cvtsd2si (which converts a double to an integer) are documented as having "unpredictable behavior across different processor generations" with certain instruction encodings. We have not investigated the extent of this behavior, but, by compromising portability, it is a potentially critical source of irreproducibility.

TSX Irreproducibility. Ultimately, we found just one family of definitively critical instructions: the TSX instructions used for transactional memory and lock elision. A transaction can abort for a variety of reasons, some of which—like the arrival of a timing interrupt—are highly irreproducible. A program can monitor its own aborts via the abort handler registered with the xbegin instruction, and perform irreproducible computation as a result. While the presence of TSX can be hidden by crafting the return value of cpuid, an invalid or adversarial program can ignore cpuid and run these instructions anyway. We are not aware of any ability to trap on the execution of TSX instructions, though Intel's microcode updates that disabled prior buggy versions of TSX [30] show that software configurability does exist on some level. Hardware support for trapping critical instructions is necessary for efficient and complete detection, because the hardware knows definitively what instructions a program is executing. Detecting the presence of xbegin in an adversarial program is impractical: the program may jump into the middle of an otherwise-valid instruction or employ self-modifying code to obfuscate its behavior beyond the reach of static binary analysis. Dynamic analysis or emulation can in principle catch such behavior, but only at a prohibitive runtime cost.

Because current hardware does not allow our DetTrace prototype to trap *all* irreproducible instructions, we rely on programs being well-behaved enough not to execute illegal or missing instructions (i.e., respecting the output of cpuid). Nevertheless, our characterization of Linux system calls and x86-64 instructions is a useful yardstick for work towards 100% reproducible containers that are robust against even adversarial programs.

[2]There are some undocumented x86-64 instructions [28]. Handling these would be an interesting avenue for future work.

5 DetTrace Design

DetTrace combines a lightweight sandboxing container with system call interception to achieve reproducibility enforcement for arbitrary Linux programs. DetTrace achieves this function while meeting our design goals: a pure-software user-space solution, supporting unmodified binaries, requiring no privileged (root) access, and requiring no record and replay. DetTrace uses standard Linux container features: user, PID, and mount namespaces, bind mounts and chroot. These mechanisms help to insulate programs in the container from programs and files outside it.

DetTrace uses ptrace to intercept all system calls made by code running in the container. The Linux ptrace mechanism allows one process (the *tracer*) to monitor the execution of another process (the *tracee*). The tracer can intercept the tracee's system calls (both before they reach the kernel and before they return to the tracee), signals, and more. The tracer can also read and write tracee memory and registers. Since the tracer is its own process, it is well-isolated from tracee faults (and vice-versa). However, extra context switches are required on intercepted events to jump to the tracer each time. In DetTrace, system calls with reproducible semantics

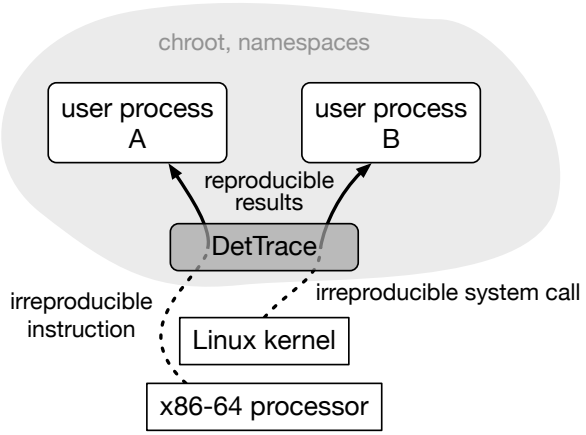

Figure 2. High-level overview of DetTrace's organization. The unshaded blocks (processor, kernel and user programs) are completely unmodified.

are permitted through, while those with irreproducible effects are either wrapped reproducibly or are identified as unsupported, triggering a runtime error.

Next we detail sources of irreproducibility and describe how DetTrace renders each one reproducible. For simplicity, we use the term "user process" to refer to a process running inside a DetTrace container.

5.1 Process, User and Group IDs

Thanks to our process namespace, processes inside our container receive unique PIDs that are independent of the world outside the container. A user process cannot name any process outside the container. As user processes are created and terminated deterministically, and Linux allocates PIDs in each namespace sequentially, PIDs inside the container are naturally deterministic. We similarly leverage uid and gid namespaces to similar ends. The first user process starts with root privileges, and can change identity via setuid.

5.2 OS-Generated Randomness

A Linux user process can request randomness from the OS via the getrandom system call, or by reading from the special /dev/random or /dev/urandom files. DetTrace intercepts getrandom system calls and fills the specified user buffer with values generated from a simple LFSR pseudo-random number generator. Similarly, /dev/random and /dev/urandom are named pipes to which DetTrace writes values from our PRNG. The PRNG seed can be specified when invoking DetTrace, to introduce "true randomness" in a controlled way. User processes can also obtain randomness via the x86-64 instructions rdrand and rdseed, discussed later in Section 5.8.

Some applications require true randomness for security reasons. DetTrace can provide such applications with direct access to, e.g.,, the real /dev/urandom and optionally log the values read to preserve reproducibility.

5.3 Time and Clocks

A variety of system calls return some form of timing information. For system calls that report wall clock time directly (like gettimeofday) DetTrace reports instead reproducible logical time values. For logical time, DetTrace uses a count of the number of time calls performed by a user process. This ensures that time monotonically advances between calls, which is important for some user programs which check timing behavior.

To enable high-resolution timing, Linux uses the virtual Dynamic Shared Object (vDSO) mechanism to implement timing system calls like gettimeofday. For performance reasons, these system calls are implemented as library calls and are thus not intercepted by ptrace. While Linux's LD_PRELOAD mechanism is a natural choice for intercepting library calls, it is incomplete in small but important ways. First, it doesn't support statically-linked binaries. Second, a process can find the vDSO library within its address space (via getauxval) and directly call a vDSO function; indeed, libc does just this in its mkstemp function. To ensure airtight interception of vDSO calls, DetTrace instead, just after each execve system call, replaces the vDSO library code with our implementation where each vDSO function makes a direct system call—which is duly intercepted via ptrace. We furthermore make the vvar page unreadable to prohibit any access to the raw nondeterministic data that vDSO timing calls use. While replacing vDSO calls with normal system calls incurs a performance penalty, we plan to extend our vDSO library to handle the timing calls directly in a future version of DetTrace.

The x86 rdtsc instruction returns timing information in the form of the current cycle count. Fortunately, rdtsc can be trapped and emulated reproducibly, see Section 5.8. Filesystem timestamps are a final source of timing information which we discuss in Section 5.5. With nondeterministic parallelism, racing threads can recreate high-resolution clocks, but our deterministic scheduling renders this moot [31].

5.4 Signals and Timers

Signals are a prime source of irreproducibility as their arrival is typically asynchronous. In principle, signal generation and delivery can be made fully reproducible via a reproducible logical clock, as with deterministic shared memory synchronization [32]. However, we have not found this necessary for our current workloads. Instead, DetTrace provides reproducibility for a subset of Linux signals. First, DetTrace does not support sending signals between user processes. It is important, however, that a user process can send itself signals. Some such

signals are naturally reproducible: SIGSEGV, SIGILL and SIGABRT act like "precise exceptions" that halt program execution at a well-defined, reproducible state.

Timers, requested via system calls like alarm, are another common source of self-signals. To render timer expiration reproducible, timers in DetTrace expire "instantaneously," invoking a signal handler if appropriate. We convert signal-generating timer calls (like alarm) into a pause system call that blocks the user process. Then, the tracer sends the necessary signal to the user process, invoking a registered signal handler if appropriate. This causes the pause call to return, and the user process resumes execution. The timer call never reaches the OS, but is instead emulated by the tracer.

5.5 Files and Directories

Files and directories are a rich source of irreproducibility, due to a complex API and extensive metadata. Our first step in providing a reproducible abstraction for files and directories is to isolate the view of the host filesystem that a user process has, accomplished via the chroot system call. DetTrace can also be nested inside standard containers like Docker to provide stronger filesystem isolation from the host.

File and directory **ownership and permissions** are inputs to a DetTrace computation (Figure 1). The Linux namespace controls the mapping from uid/gid inside the namespace to uid/gid on the host machine; this mapping is also part of the input to DetTrace. By default, we map the current user account to root inside the container, and all others to nobody/nogroup.

The order in which **directory entries** are returned is under the control of the filesystem implementation. To make the getdents system call reproducible, DetTrace sorts directory entries by name before returning them to the user process.

The **read** and **write** system calls have irreproducible semantics, as they may read/write arbitrarily fewer bytes than requested. While in practice we have never seen such "partial" operations on regular files, they do regularly arise when accessing pipes. To render these system calls reproducible in all cases, DetTrace automatically retries partial reads and writes until they process the requested number of bytes, or a read returns EOF. This is accomplished by decrementing the user process program counter to rerun the system call instruction, and adjusting the arguments to, e.g.,, tell the current read to continue where the previous read ended.

Inodes are unique identifiers for a file or directory within a filesystem mount. The stat family of system calls report inodes to a user process, and simply reporting a fixed value is insufficient as many user processes compare inode values to quickly identify identical files. Instead, DetTrace maintains a mapping from real (irreproducible) inodes to reproducible virtual inodes. Special care is needed to identify when a new file f is created, as the OS may recycle a real inode for f but DetTrace must allocate a new virtual inode to preserve reproducibility (see file timestamp discussion, next).

File timestamps present a notion of time to user processes which, unfiltered, could be used to reconstruct an irreproducible clock. Thus, DetTrace virtualizes file timestamps. On Linux, each file or directory has three associated times: time of last content modification (mtime), time of last access (atime) and time of last content or metadata modification (ctime). In DetTrace, we always report atime and ctime as 0. However, we found that always returning a fixed value for mtime falls afoul of sanity checks in many programs. For example, configure from GNU Autotools checks for clock skew by creating a new file, then comparing its mtime to that of an existing file, raising an error if the mtimes don't make sense.

DetTrace implements a mapping between real inodes and virtual mtime, allowing for a reproducible, but sensible, response from system calls like stat that report mtime. Whenever a user process opens a file, before the open call reaches the kernel we check whether a file exists at the specified path. Before the open call returns to the process in the container, we identify the underlying real inode by examining the /proc filesystem to obtain the path and real inode of the newly-created file descriptor. By examining the path both before the open call reaches the OS and afterwards, we can reliably identify when new files are created. If the file was newly created, we assign its mtime as the current virtual mtime, and increment the current virtual mtime. Otherwise, the file existed in the initial container image and we assign it a virtual mtime of 0. Writes to a file do not currently update its virtual mtime because we have not found this necessary in our workloads, however this could easily be added to provide more realistic-looking virtual mtimes.

For stat calls, we consult our *real inode→virtual mtime* map to report mtime appropriately. Any inode without an entry in the table gets a virtual mtime of 0, as it must have existed as part of the initial container image. Our lazy population of inode maps assigns reproducible virtual inodes and mtimes to every file in the container, while avoiding the need to index the entire container image at launch.

5.6 Reproducible Scheduler

DetTrace supports multiple concurrent processes by sequentializing system call execution, and allowing processes to run in parallel for other operations. Our tracer makes scheduling decisions at system calls, process spawn, and process exit.

DetTrace implements a reproducible scheduler, which consists of three queues. The *Parallel* queue contains the

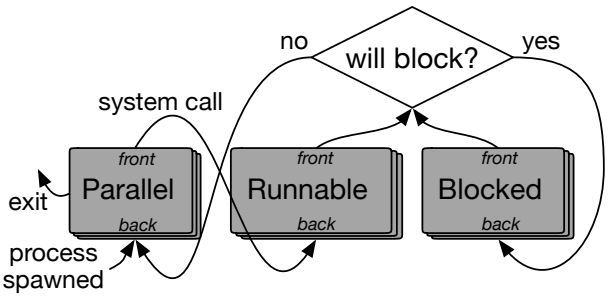

Figure 3. State transitions for a user process in the DetTrace scheduler.

processes currently running in parallel, and the other queues contain the processes that currently need to be scheduled for sequential system call execution. As Figure 3 shows, processes begin their lives at the back of the *Parallel* queue. The process at the front of the *Parallel* queue moves to the back of the *Runnable* queue when it needs to do a system call. The process at the head of the *Runnable* queue is allowed to perform a system call next, if this system call will not block then the process returns to *Parallel*, if it will block then the process moves to the end of *Blocked* queue and will be revisited later. The process at the front of the *Blocked* queue is then consulted to see if its system call will still block, and it moves to *Parallel* or back to *Blocked* accordingly.

5.6.1 Blocking System Calls. System calls that may block exhibit a potential for deadlock with DetTrace's sequential system call execution. DetTrace avoids deadlock by identifying in advance (and, of course, reproducibly) whether a system call may block. On any given potentially-blocking system call s from a process p, s can either succeed immediately, or p must wait until some event in another process enables s to complete. If the former, we execute s, move p to the *Parallel* set, remove it from the queue it was on, and resume p in parallel. If the latter, we preempt p by moving it to the *Blocked* queue.

To detect whether a system call will block or not, we transform blocking calls into non-blocking ones, e.g., a wait4 call is modified to use the WNOHANG flag. When the non-blocking system call returns and indicates the resource is not available, we preempt the process and move it to the end of the *Blocked* queue. We reset the process state to retry the system call in the future.

Some system calls, like a write to a pipe, may unblock one or more other processes. We do not track such dependencies between processes; when process p writes to a pipe we do not know precisely which *Blocked* processes (if any) this will unblock. But, because the scheduler iterates fairly over *Runnable*, *Blocked* and *Parallel* processes, any unblocked process will eventually run.

5.7 Threads

The ptrace API for threads and processes are identical, allowing DetTrace to support threads with few extensions to the scheduler. Threads within a process are sequentialized to render shared memory interactions reproducible.

The futex system call is Linux's implementation of fast, userspace locks. We treat futex wait calls like any other blocking system call (Section 5.6.1). If threads busy-wait instead of blocking, our sequential scheduler fails to make progress, which is one reason a program may be incompatible with DetTrace (Section 5.9).

5.8 CPU Instructions

While irreproducible CPU instructions cannot be intercepted through ptrace, recent x86 hardware provides mechanisms for intercepting many irreproducible instructions (Section 4). Our current DetTrace implementation intercepts the rdtsc and rdtscp instructions, which return a count of current cycles, via the prctl system call. For rdtsc[p], we overwrite their nondeterministic result with a linear function of rdtsc[p] instructions executed so far.

Additional irreproducible instructions include TSX instructions, rdrand, rdseed, and cpuid. Serendipitously, the latter provides a solution to the former: we use cpuid interception to report the absence of TSX and hardware randomness support, as described in Section 4 (while adversarial programs can try running them anyway, supporting such programs is not our target). While hypervisors have long been able to intercept cpuid, Intel's Ivy Bridge microarchitecture introduces a ring 0 mechanism that the Linux kernel (starting with 4.12) exports to user-space.

With an Ivy Bridge or newer machine, we can achieve forward-portability when rerunning a job: pinning the reported system information, while supporting subsequent processors. We also simplify the hardware details presented to the user process, for example listing a single core and canonical cache size. This further increases the equivalence class of machines which *must* observe the same answer for a job.

Older Intel architectures, such as Sandy Bridge, lack user-space cpuid interception, but they *also lack* rdrand and TSX. Therefore DetTrace can still run reproducibly on these older machines, but the portability guarantee ranges over a much smaller class of machines because we cannot hide cpuid information.

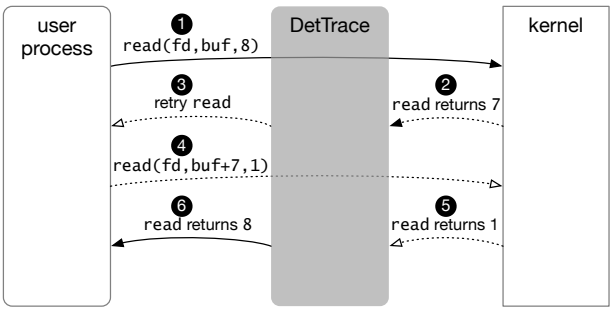

Figure 4. To render the read system call reproducible, DetTrace retries read operations that do not return the requested number of bytes. The solid arrows indicate what the user process perceives to have occurred. The dashed arrows indicate extra operations DetTrace undertakes to provide the illusion of reproducibility.

5.9 Unsupported Operations

Here we describe some limitations of our current DetTrace prototype. If a user process attempts to use one of these features, DetTrace raises an error. Section 7.1.1 evaluates in more detail the number of Debian packages that fail to build due to these reasons.

DetTrace does not support **busy-waiting** threads because our scheduler performs context switches for threads only at thread creation/exit and system calls. **Sockets** are also not supported, as arbitrary socket use for network communication is a significant reproducibility challenge. We plan to investigate limited forms of socket communication, e.g., as interprocess communication within our container, that can be rendered reproducible.

5.10 System Call Modification

DetTrace uses ptrace to intercept but then skip certain system calls, e.g., timer calls that DetTrace emulates internally (Section 5.4). While one cannot directly skip a system call with ptrace, one can indirectly skip it by replacing the system call number before it is examined by the kernel. We use time as a convenient "NOP" system call that takes no arguments and always succeeds.

We can leverage system call interception to arbitrarily modify, replay or inject new system calls. As a more involved example, Figure 4 illustrates the system call injection we perform when a user process performs a read system call that requests 8 bytes though the kernel initially returns only 7. DetTrace adjusts the read arguments to fill in the user buffer with the remaining bytes and resets the PC to perform another read. Once the user buffer is full (or we reach EOF), the user process is allowed to continue past the read call, with the buffer seemingly filled on the first try.

Sometimes a system call requires that we allocate memory in the tracee address space. For example, the utime system call sets the atime and mtime for a file at a given path. If the times are specified as null, then the kernel sets the atime/mtime to the current time. To avoid the kernel setting irreproducible timestamps, DetTrace needs to allocate a timestamp struct in the tracee address space, initialized with reproducible timestamps, and call utime with this struct as an argument. To this end, DetTrace allocates a page of memory in each tracee's address space after each execve system call. Our custom timestamp struct is allocated from this page, to avoid perturbing the tracee's heap or stack.

5.11 Improving performance with seccomp-bpf

By default ptrace stops the tracee for every system call twice, but Linux's seccomp-bpf mechanism allows for selective system call interception, avoiding overhead on system calls that are naturally reproducible in our environment (like getcwd). seccomp also allows the interception code, which runs before a system call reaches the kernel, to dynamically decide whether or not to intercept after the system call completes, further reducing overhead. Linux kernel versions >= 4.8 additionally optimize context switches by delivering a single event instead of separate pre-system-call and seccomp events. We support kernel versions < 4.8 by falling back to the slower implementation.

6 Experimental Methodology

We ran our package build evaluation using Debian 7 (Wheezy) packages, a stable version first released in May 2013 which contains 17,145 packages total. We chose this version of Debian to avoid confounding effects from the efforts of the Debian Reproducible Builds project, which began in late 2013. We wanted to capture an accurate pre-DRB picture of the Debian package ecosystem.

We build our packages on CloudLab c220g5 nodes, where each node has two Intel Xeon Silver 4114 Skylake processors, each with 10 cores (20 threads) running at 2.2GHz, and 192GB of RAM. These processors support interception of the cpuid instruction (Section 5.8). We use the full seccomp-bpf optimizations. Each node runs Ubuntu 18.04 LTS with the Linux 4.15 kernel.

For our bioinformatics workflows, we used RAxML 8.2.10 with AVX support [33], Clustal 2.1 in -ALIGN mode [34] and HMMER 3.1b2 [35]. We used TensorFlow v1.14 in our ML experiments, using the alexnet and cifar10 tutorials [36] to perform model creation, training and inference. Bioinformatics and ML workloads run on a machine with two Intel Xeon E5-2618Lv3 (Haswell) processors each with 8 cores (16 threads) running at 2.3GHz, and 128GB of RAM. The machine runs Ubuntu 18.10 with Linux 4.18.

6.1 Verifying Reproducibility

Package builds. We build packages, both with and without DetTrace, inside a fresh Docker instance to easily control filesystem state.[3] Inside the container, we use a slightly modified version of the reprotest utility version 0.7.8 [37] from the DRB project. reprotest builds each package twice, varying the conditions for each build to exacerbate irreproducibility. We configure reprotest to vary environment variables, build path, ASLR, number of CPUs, time, user groups, home directory, locales, exec path, and timezone. We turn off domain host, kernel, and file ordering as they are not supported by the older version of Debian we're running our builds in. Similarly, the umask variation would randomize file permissions which DetTrace does not hide from user processes.

By default reprotest chooses variations randomly; we modified it to use a consistent configuration for the first build of all packages, and a different consistent configuration for all second builds, so that exactly the same environment is presented to DetTrace as in the baseline. We create a control-chroot of a minimal Wheezy installation, downloading the source via apt-get source, then installing a package's dependencies via apt-get build-dep (referencing an on-disk mirror to avoid network requests and ensure consistency across builds). Finally we copy the control-chroot to create an experiment-chroot, thus guaranteeing the same starting image for both builds. reprotest takes these starting chroots for running dpkg-buildpackage with or without DetTrace. When using DetTrace, everything dpkg-buildpackage does runs under DetTrace, which includes compilation, running tests (if the package is configured to do so), and creating the final .deb package. After both builds are complete, reprotest validates reproducibility with bitwise comparison of the two .deb packages. reprotest calls another DRB tool diffoscope which compares two directories, checking for bitwise identical contents. If diffoscope reports no differences the package is deemed *reproducible*, otherwise the package is deemed *irreproducible*.

Under this Debian/reprotest configuration 15,761, or 91.9%, of the total available packages build completely, whereas 40 time-out after 30 minutes and 1,344 fail to build. For the evaluation in the next section, we focus on the set of 15,761 packages that build in the baseline, whether reproducibly or irreproducibly. In fact, in a stock Wheezy system, *zero* packages build reproducibly because of timestamps embedded by tar. So we adjust our driver script to unpack the deb packages using dpkg-deb, then run strip-nondeterminism [38] on the individual files, stripping timestamps. Finally, diffoscope can do a meaningful bitwise comparison. The DetTrace builds do not require this workaround, as they are naturally robust to timestamps. With the tar-timestamp workaround, 3,803 (24.1%) packages are reproducible in a stock Wheezy system. The other 11,958 packages require additional manual intervention to achieve reproducibility.

Bioinformatics. While we did not leverage an adversarially-irreproducible environment like reprotest for the bioinformatics tools, using hashdeep on the outputs from HMMER and RAxML revealed irreproducibility across consecutive runs on a single machine. We confirmed (using hashdeep) that the irreproducibility is removed when running under DetTrace. The clustal workflow appeared reproducible, both natively and with DetTrace.

Machine Learning. To check the reproducibility of our TensorFlow workloads, we recorded the value of the loss function at each step during training. Unsurprisingly, these values are irreproducible when running natively, even with serialized TensorFlow (see Section 7.6), due to, e.g., randomization of the training set. DetTrace renders these workloads reproducible without any code changes.

7 Evaluation

In this section, we describe our results using the DetTrace system with software builds, and bioinformatics and machine learning applications.

7.1 Package Build Reproducibility

Package builds can fall into one of four categories when building under DetTrace. Some package builds are *reproducible* or *irreproducible* as described in Section 6.1. *Timeout* packages do not finish building within 2 hours. We allot a high timeout for DetTrace to account for its performance overheads and to avoid eliding high slowdowns from our performance evaluation. Lastly, a package may be *unsupported* for a variety of reasons we discuss in Section 7.1.1.

Of the 12,130 packages that DetTrace supports (i.e., the build with DetTrace is neither *unsupported* nor does it timeout), DetTrace is able to render every single package reproducible. This represents over 800 million (non-comment/non-blank) lines of code from over 3.3 million source files building under DetTrace.

Table 1 shows how package status changes when moving from the baseline to DetTrace and vice-versa, focusing just on those packages that build (reproducibly or irreproducibly) in the baseline. The top table shows what happens to baseline packages when run with DetTrace. For example, the first row shows that of the 11,958 packages that are irreproducible in the baseline, 8,688 of them are automatically rendered reproducible by DetTrace.

[3] DetTrace can also provide an isolated filesystem environment, but Docker provides easy image distribution across our cluster. DetTrace nests within Docker without issue.

Given	DT Reproducible	DT Irreproducible	DT Unsupported	DT Timeout
BL Irreproducible (11,958)	72.65% (8,688)	0% (0)	15.99% (1,912)	11.36% (1,358)
BL Reproducible (3,803)	90.51% (3,442)	0% (0)	3.60% (137)	5.89% (224)

Given	BL Reproducible	BL Irreproducible
DetTrace Reproducible (12,130)	28.38% (3,442)	71.62% (8,688)
DetTrace Timeout (1,582)	14.16% (224)	85.84% (1,358)
DetTrace Unsupported (708)	7.91% (56)	92.09% (652)

Table 1. (Top) How build status changes moving from the baseline (BL) to DetTrace (DT), and from DT to BL (bottom). DetTrace automatically renders reproducible 72.65% of packages that are irreproducible in the baseline.

Reassuringly, packages that are reproducible in the baseline never become irreproducible under DetTrace.

The bottom table in Table 1 shows, for a package with a given DetTrace status, what happens in the baseline. Packages that timeout or are unsupported by DetTrace are very commonly irreproducible in the baseline, suggesting these are more complicated builds.

7.1.1 Unsupported Packages. A total of 1,912 packages failed to build due to known DetTrace limitations. The most frequently encountered issue was busy waiting, which arose for 876 Java packages (45.8% of failures) that fail to build. The next most common reasons are socket operations (302 packages, 15.8%), and sending intra-process signals (79 packages, 4%) The rest form a long tail of miscellaneous system calls DetTrace does not yet support. Other cases of busy waiting result in a *timeout*.

7.1.2 Comparison with DRB. 407 of the packages that are reproducible under DetTrace are identified as *irreproducible* in the current stretch release by DRB [39]. While those packages are newer than the Wheezy versions we use, the DRB effort has also categorized why these packages are irreproducible. Common reasons include build paths being captured in a build artifact, timestamps embedded in files and randomness affecting build artifacts. Though these issues have been resolved in hundreds of other packages, each package requires analyzing the cause of irreproducibility and getting patches accepted by maintainers. In contrast, DetTrace automatically makes a build immune to such variations.

7.1.3 Comparison with Mozilla rr. Record-and-replay (RnR) systems are similar to DetTrace in needing to intercept sources of nondeterminism. However, record-and-replay systems do not directly facilitate reproducible builds, as opaque recording files do not enable one to inspect the source code of a package. Recordings also require storage, typically much more than pure source code. We undertook a small experiment with the latest version (5.2.0) of the rr tool, as it is the most robust RnR system we are aware of. We selected 81 packages that build from source natively in Ubuntu 18.04 (to provide a more modern build environment for rr than Debian Wheezy), and tried building them with rr. Unfortunately, rr crashed on 46 of them due to a known bug with unsupported ioctl calls. Of the 35 packages that build with rr, the average runtime overhead was 5.8× (ranging from 3.3-22.7×), comparable to DetTrace. Unlike RnR, DetTrace avoids opaque recordings and provides a human-readable audit trail from inputs to outputs.

7.2 Package Build Correctness

To validate the functional correctness of DetTrace, we used several of the packages built using our system to ensure they work correctly. For example, we built the popular 3D graphics package blender with DetTrace, installed the resulting .deb on a Debian wheezy virtual machine, and used the UI to render a sample project. We built the core TeX/LaTeX packages using DetTrace and used them to build the paper you're reading.

To validate DetTrace's correctness on a complex software system, we first built the LLVM 3.0 compiler from source without using DetTrace. We ran LLVM's test suite via the make check finding that 5,594 tests pass, 48 expectedly fail and 15 are unsupported in this baseline configuration. We then ran the LLVM build under DetTrace (using a version of clang built with DetTrace as well) and received the same test outcomes. Given the complexity of the LLVM source code, we find these results with "self-hosting" LLVM encouraging evidence that software built using DetTrace functions correctly.

7.3 Package Build Portability

To evaluate DetTrace's portability, we perform package builds on two different machines with different microarchitectures and OS versions. One machine is our standard CloudLab node (described in Section 6) and the other has Intel Xeon E5-2620 v4 processors (Broadwell instead of Skylake) running Ubuntu 18.10 (instead of 18.04, Linux versions 4.18 and 4.15, respectively). We use the same reprotest-based build methodology to perturb the environment, and ensure that each build on each

machine produces a bitwise-identical package. Due to time constraints, we randomly selected 1,000 packages reproducible with DetTrace. Every one built identically across the two systems.

Achieving portability for these packages required one extension to DetTrace. We found that the size of a directory (returned by stat) varied across machines, though the directory contents were identical, were created via extraction from the same tarball, and the filesystem type and block size were the same. This behavior had not arisen across any of our previous experiments which used a single machine type, empirically illustrating the distinction between portability and determinism. DetTrace implements reproducible directory sizes by reporting sizes as a deterministic function of the number of directory entries.

7.4 Package Build Performance

DetTrace is designed for reproducibility, but is only moderately optimized for performance overheads. Considering builds in aggregate, DetTrace incurs an total 3.49× slowdown in wall clock time. Figure 5 shows a scatter plot for 860 randomly-selected DetTrace-supported packages, showing DetTrace's slowdown over the baseline (log scale) against the build's rate of system calls per second (as measured by DetTrace). We exclude builds that run for less than 5 seconds in the baseline, and we run just one package build per machine to avoid performance interference. We crop a few outliers from the plot to make it easier to read: 4 packages that perform more than 25,000 syscalls/second (the max is 82,533 for the) and exhibit slowdowns from 3.97-30.11×, and 3 packages that run about twice as fast with DetTrace than in the baseline—though they appear to build correctly, e.g., their internal tests all pass at the end of the build.

The light orange dots in Figure 5 show packages that do not use threads, while the dark blue dots show threaded packages. Overall, there is a positive correlation between DetTrace overhead and system call rate. Though there are just 76 threaded packages in this sample, they exhibit some of the highest slowdowns due to common futex operations being converted from blocking to non-blocking.

We find that system calls are frequent in package builds, with over 800,000 in an average build (Table 2). We also find many potential sources of irreproducibility in *all* of our packages. rdtsc instructions are used by the loader ld for internal profiling, and by libc to generate temporary file names for gcc. gcc also reads from /dev/urandom to produce unique symbol names.

7.5 Bioinformatics Workflows

Our three bioinformatics workflows use process-level parallelism for performance, and exhibit a range of overheads

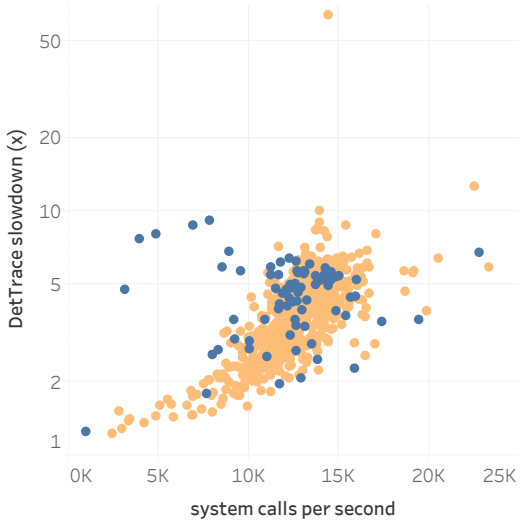

Figure 5. DetTrace overhead (y-axis, log scale) is largely driven by the rate at which system calls are performed (x-axis). Packages that use threads (dark blue dots) are typically slower than those that do not (light orange dots).

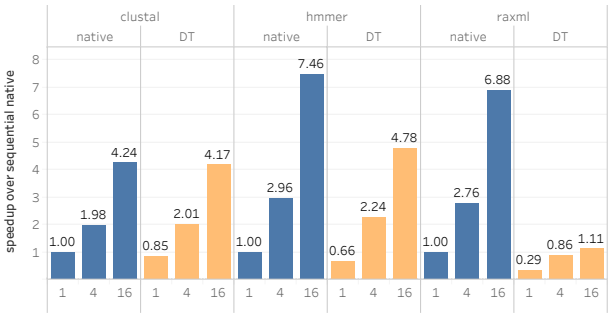

Figure 6. Speedup of bioinformatics workflows with 1, 4 & 16 parallel processes, normalized to sequential native execution (higher is better). Dark blue bars are native execution, and light orange bars are DetTrace.

with DetTrace, dictated by the degree to which they are compute-bound. Figure 6 shows the speedup each workload observes with more parallel processes, normalized to sequential native execution. The highly compute-bound clustal performs the best, scaling well with additional processes and exhibiting under 2% overhead with 16 processes. In contrast, hmmer and raxml execute system calls at a rate 19× higher (over 55,000/second on average), incurring more serialization. raxml in particular writes to stdout frequently, which are potentially-blocking operations that are more expensive for DetTrace, resulting in 6.2× overhead with 16 processes. hmmer has more non-blocking system calls which enable better scaling, and just 1.56× overhead with 16 processes.

System call events	843,621.53
User process memory reads	396,474.88
rdtsc intercepted	33,487.55
Requests for scheduling next process	6,049.51
Replays due to blocking system call	1,283.72
Process spawn events	2,377.54
read retries	141.28
/dev/urandom opens	159.92
write retries	113.98

Table 2. Per-package average number of events encountered by DetTrace.

7.6 TensorFlow

We ran the alexnet and cifar10 programs in three configurations, each of which run exclusively on the CPU: 1) natively in parallel, 2) natively but with TensorFlow configured to use a single thread and 3) with DetTrace. Since TensorFlow uses thread-level parallelism via OpenMP, DetTrace's serialized threading incurs a large slowdown over native parallel execution on 16 cores: it is 17.49× slower on alexnet and 11.94× on cifar10. Compared to serialized native execution, however, DetTrace fares much better with slowdowns of 1.51× and 1.08×, respectively, reinforcing that DetTrace exacts a small performance price for non-threaded compute-bound workloads.

8 Related Work

DetTrace's unique reproducible container abstraction takes inspiration from many previous systems. We categorize this previous work into record-and-replay systems, and deterministic execution systems.

Many **record and replay** (RnR) systems have been proposed both from academia [40–46] and industry [47–49]. These systems record a trace of one nondeterministic execution to enable subsequent replay of that execution, typically for debugging purposes. These systems have broadly similar interception requirements as DetTrace, since system calls are a prime source of irreproducibility that must be recorded in the trace. DetTrace borrows some implementation techniques from Mozilla's rr [47] as it also relies on ptrace (a quantitative comparison with rr appears in Section 7.1.3). Many RnR systems target multithreaded workloads, as those are very challenging to debug without RnR support, and provide high-performance parallel recording and replaying.

Deterministic execution schemes enforce determinism during program execution. Deterministic operating systems tackle several of the systems issues we describe in this paper, providing deterministic versions of OS abstractions like processes and threads. While Determinator [12] provides new OS abstractions for deterministic fork-join parallelism, and DDOS [13] focuses on local network interactions, dOS [14] is closer to our work in offering a deterministic process group abstraction. The *shim* abstraction in dOS bears similarity to Linux's ptrace API. Unlike DetTrace, dOS supports parallel execution of both threads and processes. However, dOS uses RnR for filesystem interactions, defining the boundaries of its determinism abstraction too narrowly to be useful for software builds which interact extensively with the filesystem. More generally, a custom OS is a heavyweight prerequisite to perform deterministic computation, and existing deterministic OSes have not evaluated portability across different microarchitectures.

Other deterministic execution schemes focus on a single multithreaded process, determinizing interactions through shared memory. Some schemes target arbitrary binary programs [50–58], providing generality at a modest performance overhead. Other schemes leverage language support to provide determinism for Haskell [10, 26, 59–62] or Java [9, 63] programs. Whether language-agnostic or specific, these approaches eliminate the influence of thread scheduling, but do not determinize IO interactions with the underlying OS and filesystem. The scope of their guarantees is thus too small to be useful for reproducible builds. One exception is DetFlow [11] which provides deterministic parallel execution for batch jobs, though it lacks robust system call interception and requires a coordinator layer written in Haskell.

9 Conclusions

We have described the design and implementation of DetTrace, which provides a new *reproducible container* abstraction. DetTrace automatically provides reproducibility for software builds, bioinformatics processing and ML workflows without requiring any changes to the hardware, OS, or application code. To facilitate further experimentation with DetTrace, we plan to open-source its code upon publication.

Acknowledgments

This work is supported by the National Science Foundation under grant #1703541. Opinions or findings in this material are the authors' and do not necessarily reflect the views of the NSF.

References

[1] Bazel. https://bazel.build/.
[2] ReproducibleBuilds. https://wiki.debian.org/ReproducibleBuilds.
[3] Pachyderm reproducible data science homepage. https://www.pachyderm.io.
[4] Code Ocean homepage. https://codeocean.com.
[5] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. *SIGOPS Oper. Syst. Rev.*, 25(5):152–164, September 1991.

[6] Jonathan Goldstein, Ahmed Abdelhamid, Mike Barnett, Sebastian Burckhardt, Badrish Chandramouli, Darren Gehring, Niel Lebeck, Umar Farooq Minhas, Ryan Newton, Rahee Ghosh Peshawaria, et al. Ambrosia: Providing performant virtual resiliency for distributed applications. Technical report, Technical report, 2018. https://aka. ms/amb-tr, 2018.

[7] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[8] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(6):175–188, October 2007.

[9] Robert Bocchino, Mohsen Vakilian, Vikram Adve, Danny Dig, Sarita Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, and Hyojin Sung. A type and effect system for deterministic parallel java. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA '09*, page 97, Orlando, Florida, USA, 2009.

[10] Simon Marlow, Ryan R. Newton, and Simon Peyton Jones. A monad for deterministic parallelism. In *Proceedings of the 4th ACM symposium on Haskell*, Haskell '11, pages 71–82. ACM, 2011.

[11] Ryan G. Scott, Omar S. Navarro Leija, Joseph Devietti, and Ryan R. Newton. Monadic composition for deterministic, parallel batch processing. *Proc. ACM Program. Lang.*, 1(OOPSLA):73:1–73:26, October 2017.

[12] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[13] Nicholas Hunt, Tom Bergan, Luis Ceze, and Steven D. Gribble. DDOS: taming nondeterminism in distributed systems. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, volume 48, pages 499–508, March 2013.

[14] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven Gribble. Deterministic process groups in dOS. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010.

[15] Abadi, Daniel J. and Faleiro, Jose M. An Overview of Deterministic Database Systems. *Communications of the ACM*, 61(9):78–88, August 2018.

[16] Raymond Chen. Why are the module timestamps in Windows 10 so nonsensical? https://blogs.msdn.microsoft.com/oldnewthing/20180103-00/?p=97705.

[17] Jared Parsons. Deterministic builds in Roslyn. http://blog.paranoidcoding.com/2016/04/05/deterministic-builds-in-roslyn.html.

[18] tar: please add –clamp-mtime to only update mtimes after a given time. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=790415., June 2015.

[19] Jeremy Bobbio. Reproducible Builds for Debian, a year later. https://summit.debconf.org/debconf14/meeting/78/reproducible-builds-for-debian/.

[20] Ren, Zhilei and Jiang, He and Xuan, Jifeng and Yang, Zijiang. Automated Localization for Unreproducible Builds. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 71–81, New York, NY, USA, 2018. ACM.

[21] National Academies of Sciences, Engineering, and Medicine. *Reproducibility and Replicability in Science*. The National Academies Press, Washington, DC, 2019.

[22] Rosemary Nan Ke and Alex Lamb and Olexa Bilaniuk and Anirudh Goyal and Yoshua Bengio. Reproducibility in Machine Learning: An ICLR 2019 Workshop. https://sites.google.com/view/icml-reproducibility-workshop/home.

[23] Daniel Maskit. Problems Getting TensorFlow to behave Deterministically. https://github.com/tensorflow/tensorflow/issues/16889.

[24] Jennifer Villa and Yoav Zimmerman. Reproducibility in ML: why it matters and how to achieve it. https://determined.ai/blog/reproducibility-in-ml/, May 2018.

[25] Li Lu and Michael L. Scott. Toward a formal semantic framework for deterministic parallel programming. In *Proceedings of the 25th International Conference on Distributed Computing*, DISC'11, page 460–474, Berlin, Heidelberg, 2011. Springer-Verlag.

[26] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. Freeze after writing: quasi-deterministic parallel programming with lvars. In *POPL*, pages 257–270, 2014.

[27] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[28] Christopher Domas. Breaking the x86 Instruction Set. Black Hat, 2017. https://www.youtube.com/watch?v=KrksBdWcZgQ.

[29] Intel 64 and IA-32 Architectures Software Developer Manuals. https://software.intel.com/en-us/articles/intel-sdm.

[30] Intel Xeon Processor E3-1200 v3 Product Family. https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e3-1200v3-spec-update.pdf, July 2018. HSW136: Software Using Intel TSX May Result in Unpredictable System Behavior.

[31] Amittai Aviram, Sen Hu, Bryan Ford, and Ramakrishna Gummadi. Determinating timing channels in compute clouds. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, page 103–108, New York, NY, USA, 2010. ACM.

[32] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient deterministic multithreading in software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 97–108, New York, NY, USA, 2009. ACM.

[33] Alexandros Stamatakis. Raxml version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312, 2014.

[34] Ramu Chenna, Hideaki Sugawara, Tadashi Koike, Rodrigo Lopez, Toby J. Gibson, Desmond G. Higgins, and Julie D. Thompson. Multiple sequence alignment with the clustal series of programs. *Nucleic Acids Research*, 31(13):3497, 2003.

[35] Sean R. Eddy. Profile hidden markov models. *Bioinformatics*, 14(9):755–763, 1998.

[36] tensorflow. Models and examples built with TensorFlow. https://github.com/tensorflow/models/tree/master/tutorials/image. Commit 583408.

[37] Reprotest GitLab page. https://salsa.debian.org/reproducible-builds/reprotest.

[38] strip-nondeterminism Debian Package Description. https://packages.debian.org/sid/strip-nondeterminism.

[39] Packages in Stretch/Amd64 Which Failed to Build Reproducibly. https://tests.reproducible-builds.org/debian/stretch/amd64/index_FTBR.html.

[40] Thomas J. Leblanc and John M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on*

Computers, C-36(4):471–482, April 1987.

[41] Michiel Ronsse and Koen De Bosschere. RecPlay: a fully integrated practical record/replay system. *ACM Transactions on Computer Systems*, 17(2):133–152, May 1999.

[42] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 525–540, Berkeley, CA, USA, 2014. USENIX Association.

[43] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: efficient online multiprocessor replayvia speculation and external determinism. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems - ASPLOS '10*, page 77, Pittsburgh, Pennsylvania, USA, 2010.

[44] Ali José Mashtizadeh, Tal Garfinkel, David Terei, David Mazieres, and Mendel Rosenblum. Towards practical default-on multi-core record/replay. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 693–708, New York, NY, USA, 2017. ACM.

[45] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, December 2002.

[46] George W. Dunlap, Dominic G. Lucchetti, Michael A. Fetterman, and Peter M. Chen. Execution replay of multiprocessor virtual machines. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '08, pages 121–130, New York, NY, USA, 2008. ACM.

[47] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 377–389, Santa Clara, CA, 2017. USENIX Association.

[48] VMware: VMware workstation zealot: Enhanced execution record / replay in workstation 6.5, April 2008.

[49] Program Record/Replay Toolkit. https://software.intel.com/en-us/articles/program-recordreplay-toolkit.

[50] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: deterministic shared memory multiprocessing. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems (ASPLOS '09)*, page 85, Washington, DC, USA, 2009.

[51] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 53–64. ACM, 2010.

[52] Derek R. Hower, Polina Dudnik, David A. Wood, and Mark D. Hill. Calvin: Deterministic or not? free will to choose. In *Proceedings of the 17th International Symposium on High-Performance Computer Architecture (HPCA)*, 2011.

[53] Joseph Devietti, Jacob Nelson, Tom Bergan, Luis Ceze, and Dan Grossman. RCDC: a relaxed consistency deterministic computer. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011.

[54] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011.

[55] Timothy Merrifield and Jakob Eriksson. Conversion: multi-version concurrency control for main memory segments. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 127–139, New York, NY, USA, 2013. ACM.

[56] Kai Lu, Xu Zhou, Tom Bergan, and Xiaoping Wang. Efficient deterministic multithreading without global barriers. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, February 2014.

[57] Timothy Merrifield, Joseph Devietti, and Jakob Eriksson. High-performance determinism with total store order consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 31:1–31:13, New York, NY, USA, 2015. ACM.

[58] Timothy Merrifield, Sepideh Roghanchi, Joseph Devietti, and Jakob Eriksson. Lazy determinism for faster deterministic multithreading. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 879–891, New York, NY, USA, 2019. ACM.

[59] Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel haskell: A status report. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 10–18, New York, NY, USA, 2007. ACM.

[60] Simon Marlow, Patrick Maier, Hans-Wolfgang Loidl, Mustafa K Aswad, and Phil Trinder. Seq no more: better strategies for parallel haskell. In *ACM Sigplan Notices*, volume 45, pages 91–102. ACM, 2010.

[61] Ryan R. Newton, Ömer S. Ağacan, Peter Fogg, and Sam Tobin-Hochstadt. Parallel type-checking with haskell using saturating lvars and stream generators. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 6:1–6:12, New York, NY, USA, 2016. ACM.

[62] Daan Leijen, Manuel Fahndrich, and Sebastian Burckhardt. Prettier concurrency: Purely functional concurrent revisions. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 83–94. ACM, 2011.

[63] Robert L. Bocchino and Vikram S. Adve. Types, regions, and effects for safe programming with object-oriented parallel frameworks. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 306–332, Berlin, Heidelberg, 2011. Springer-Verlag.

A Artifact Appendix

A.1 Abstract

This artifact description gives an overview of the command line interface for DetTrace as well as common DetTrace workflows. We describe the steps necessary to build DetTrace, software dependencies, hardware requirements, and compiler toolchains. Furthermore, we show various example use-cases for DetTrace and using chroot environments.

A.2 Artifact check-list (meta-information)

- **Program:** DetTrace: A Deterministic Container.
- **Compilation:** Any C++ compiler with C++14 support. Verified to compile with both g++ and clang++.

- **Run-time environment:** Linux x86-64. Optimized for kernel versions $\geq$ 4.8. Works on $\leq$ 4.8 kernel version using slower methods (see paper for more information). Kernel version $\geq$ 4.12 required for CPUID interception. DetTrace is untested for kernel version $\geq$ 5.13 but should work with minor modifications or future updates. Linux namespaces enabled required.
- **Hardware:** Intel x86-64 CPU. To intercept CPUID, the CPU must support this feature.
- **Output:** DetTrace will produce a deterministic, reproducible output of whichever command runs under it. (See more below).
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** MIT
- **Archived (provide DOI)?:** 10.5281/zenodo.3562526

A.3 Description

A.3.1 How delivered. The latest version of DetTrace can be found on GitHub. A more permanent but perhaps older version exists at
https://zenodo.org/record/3562526.

A.3.2 Hardware dependencies. The DetTrace prototype currently only works for x86-64 Intel CPUs. While not strictly necessary, the CPU must have support for intercepting certain nondeterministic CPU instructions, e.g. CPUID.

A.3.3 Software dependencies. DetTrace works well with kernel versions 4.8 and $<$ 5.3 (only minor modifications should be necessary to allow DetTrace to work in newer kernel versions). Kernel version $\leq$ 4.8 use a slower ptrace implementation (more details in paper above), making overall execution of DetTrace slower. Kernel version $\geq$ 4.12 are required for OS support for CPUID interception. Currently DetTrace has a few dependencies:

- libssl: Hashing implementation for bytes from read system call (Useful for debugging nondeterministic bytes read from pipe).
- libseccomp: Helper library for finer-grained system call filtering using seccomp-bpf.
- libarchive: Unpackaging archive files used by DetTrace.
- libelfin: ELF/DWARF parser.
- libfuse: Running tests only. Used for filesystem tests.

These dependencies are fairly common and should be easy to install from your Linux distro's package manager. For Apt these are libarchive-dev, libssl-dev, and libseccomp-dev, libelfin-dev, and libfuse-dev.

DetTrace relies on various Linux namespaces for isolation and determinism. Some systems disable user namespaces by default. You may encounter a *clone failed: operation not permitted* error. If so, please enable user namespace access (the exact method may depend on your distro and OS version). Furthermore, Linux does not allow user namespaces for processes whose root is not / (the chroot system call was executed at some point).

A.3.4 Data sets. While DetTrace is designed to enforce reproducibility for arbitrary computation, the prototype implementation was overfitted to work with Debian packages and chroot environments. The Debian Apt Package Repository and chroot images available through debootstrap may be of interest. Our prototype built mostly Debian Wheezy packages, newer Debian version may execute system calls not currently implemented in DetTrace.

A.4 Installation

DetTrace uses a standard Makefile for compilation. Assuming you have the all the software dependencies (above), and a C++ compiler supporting C++14, running `make` in the root directory of the repository will generate the DetTrace binary under *bin/*. DetTrace may also be statically linked using `make static` assuming all the correct dependencies are present as statically linkable objects. DetTrace may be installed anywhere on the system by moving the *bin/* directory along with the *root/* directory. This directory structure must be maintained as the DetTrace binary always expects *root/* to be located *../root* relative to the location of the DetTrace binary (see reasoning below).

The word *chroot* is unfortunately overloaded, both referring to the Linux system call which changes the location of the root directory for a process, and for the concept of a directory tree which "look like" a Linux filesystem, containing all the binaries, libraries, and special files (e.g. */proc*, */dev*) necessary for a Linux system to function.

The Debian command *debootstrap* is an easy way to get such a chroot environment. For more information see:
https://wiki.debian.org/chroot.

If you're using a different Linux distribution, you should consult the documentation to find the appropriate way to install a chroot. You can install debootstrap from the Apt repository. *sudo apt install debootstrap*.

One can install a Stretch Debian chroot by running the commands:

```
$ mkdir stretch
$ sudo debootstrap stretch stretch/ \
http://deb.debian.org/debian
```

Unfortunately, this requires root to work. Once you have downloaded the chroot, you should have a 'stretch' directory which we can 'ls':

```
$ ls stretch
bin/   dev/  home/  lib64/  mnt/  proc/  run/
boot/  etc/  lib/   media/  opt/  root/  sbin/
```

As you can see, this looks just like a Linux filesystem tree. DetTrace accepts any such chroot to be used as the starting image. As DetTrace will be using the files in 'stretch' we need to ensure the permissions are correct (more information can be found at https://wiki.debian.org/chroot):

```
sudo chgrp -R $USER stretch/
sudo chown -R $USER stretch/
```

You are now ready to use DetTrace and the chroot (see below).

A.5 Experiment workflow

DetTrace is a command line tool which makes it easy to make computation reproducible. You can simply prepend the command you are interested in determinizing with *./path/to/dettrace*. For example:

```
$ ./bin/dettrace ls -ahl
-rw-r--r-- 1 root ... Jan 1 1970 initramfs.cpio
drwxr-xr-x 1 root ... Jan 1 1970 root
-rw-r--r-- 1 root ... Jan 1 1970 shell.nix
drwxr-xr-x 1 root ... Jan 1 1970 src
...
```

Note the dates of all files are now deterministic! DetTrace will determinize the specified program along with any subprocesses that are spawned. Furthermore, DetTrace containerizes the program to guarantee a reproducible starting filesystem.

Similarly we may create a file and then *stat* the file with DetTrace:

```
$ touch foo.txt
$ ./bin/dettrace stat foo.txt
  File: foo.txt
  Size: 0  Blocks: 1  IO Block: 512 ...
Device: 1h/1d Inode: 8   Links: 1
Access: (0644/-rw-r--r--)  Uid: ...
Access: 1970-01-01 00:00:00.000000000 +0000
Modify: 1970-01-01 00:00:00.000000000 +0000
Change: 1970-01-01 00:00:00.000000000 +0000
 Birth: -
```

Notice a lot of the file's metadata is now deterministic!

We didn't specify a container when calling DetTrace with these simple commands. Yet, when running commands:

```
$ ./bin/dettrace pwd
/build
$./bin/dettrace ls /
bin/ build/ dettrace/ dev/ etc/ lib/ ...
```

We see the current working directory (CWD) is */build* and listing the root directory looks vastly different from the host's root directory. This is because when no container is specified DetTrace will create a minimal one based on the *root/* directory found at the top level of the DetTrace repository.

Running:

```
$ ls root
bin/ build/ dettrace/ dev/ etc/ lib/ ...
```

We can see this directory holds what looks like a minimal Linux filesystem. For this reason, DetTrace needs to be able to find the *root/* directory, which DetTrace expects will always be *../root/* relative to the location of the binary.

While running the command without a chroot is easy, it does not guarantee reproducibility since the starting filesystem state may be different from run to run. Therefore we use the chroot flag to specify the chroot image created in the build step (see above). We can use this chroot as the starting image for DetTrace:

```
./bin/dettrace --chroot path/to/stretch/ ls
```

This command will now run deterministically using the chroot as the starting image. DetTrace also supports running inside Docker containers, see the Makefile and Dockerfile to see how this is done for our own tests.

A chroot or containerized environment is standard and recommend way to use DetTrace.

A.6 Evaluation and expected result

Any program running under DetTrace is expected to produce the output of the program with a deterministic version of the output. So running the *date* command multiple times will produce the same result.

```
$ ./bin/dettrace date
Sun Aug  8 22:00:00 UTC 1993
$ ./bin/dettrace date
Sun Aug  8 22:00:00 UTC 1993
```

DetTrace comes with dozens of sample nondeterministic programs meant to stress test different sources of nondeterminism found in programs. These programs are designed to be nondeterministic, and have an expected, deterministic, output when running under DetTrace. You may test your installation by running `make test` to run all our sample programs and integration tests. If docker is installed you may run `make test-docker` to run the same tests under a Docker environment instead. Docker is the preferred way to run the tests as it handles hard to reproduce sources of unreproducibility seen in the wild.

Let's do a reproducible build! For simplicity we will use DetTrace itself as the program to build reproducibly. From the root directory of the DetTrace repository:

```
./bin/dettrace make -C src/
```

This will build DetTrace deterministically under DetTrace. You can use a program like *hashdeep* to hash the binary outputs of programs and ensure the results are deterministic across executions.

A.7 Experiment customization

Dettrace has several command line options which may be useful:

- *–debug N*: Verbose debug output for DetTrace. Prints extra information about system calls, instructions, and OS events that DetTrace is intercepting. Useful for debugging failures or determinism bugs.
- *–working-dir*: By default DetTrace "bind mounts" the CWD inside the container, this makes working without chroots easy and seamless. However, when working with chroots we probably don't want the arbitrary CWD to be moved inside the chroot. Instead we want to specify a directory inside the chroot. This flag allows you to specify a CWD inside the chroot.