LEVERAGING SYSTEM CALL INTERPOSITION FOR LOW-LEVEL PROCESS MANIPULATION

Omar S. Navarro Leija

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania

in

Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

2022

Supervisor of Dissertation

Joseph Devietti, Associate Professor of Computer & Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer & Information Science

Dissertation Committee

Linh Thi Xuan Phan, Associate Professor of Computer & Information Science
Sebastian Angel, Assistant Professor of Computer & Information Science
Boon Thau Loo, Professor of Computer and Information Science
Ryan R. Newton, Software Engineer, Meta

LEVERAGING SYSTEM CALL INTERPOSITION FOR LOW-LEVEL PROCESS MANIPULATION

Esté doctorado está dedicado a mis padres. No hubiera podido hacer esto sin su apoyo y amor. No puedo creer a dónde hemos llegado gracias a todo su trabajo. Ustedes me inspiran cada día. Le debo todos mis éxitos a ustedes.

ACKNOWLEDGEMENT

I would like to acknowledge all the people that made my doctorate possible.

My advisor, Joseph Devietti, for the opportunity to go to Penn and work with him. It has been a pleasure working and learning from your all these years. Joe is a great mentor, advisor, and incredible human being; he has been supportive and understanding all these years, specially when life got hard. I wish more of our projects would have panned out successfully.

My best friend and closest collaborator, Kelly Shiptoski. Meeting you and working along side was the highlight of grad school. I hope we always continue working together.

All the Penn graduate students who made grad school fun! Our long, late, slightly drunk, conversations during TGIF. Special shout outs to Caleb Standford, Solomon Maina, Zach Schutzman, Alyssa Hwang, and, Irene Yoon. Thank you to the Penn PL group for allowing a systems person like me to join in.

Special mention to Irene Zhang, Microsoft researcher extraordinaire. She is an incredible mentor and one of the coolest people I know.

My collaborators and mentors who gave me a chance and the opportunity to work with them: Ryan Newton, Alan Jeffrey, Leonid Ryzhyk, and Mihai Budiu.

Our research lab: Kelly Shiptoski (again!), Gautam Mohan, Bhavana Mehta, and Yuxuan Zhang.

Last but not least, my family and loved ones: My mother and father, Alma Leija Rea and Martin Barrera. My brother, Jose Navarro Leija. You guys supported me me all these years! And my lovely partner Dr. Hannah Joy Richter. Who's support was necessary to make it though the dissertation!

ABSTRACT

LEVERAGING SYSTEM CALL INTERPOSITION FOR LOW-LEVEL PROCESS MANIPULATION

Omar S. Navarro Leija

Joseph Devietti

Modern software continues to grow in size and complexity with no signs of slowing down. Program tracing allows us to observe the execution of a program. OS-level program tracing is useful, as it allows us to abstract over many details of program execution and view programs based on the IO operations they perform. Beyond read-only program tracing, this dissertation overviews low-level process manipulation. We argue process manipulation is a useful and general technique with many applications.

We show the utility of tracing and process manipulation by covering several projects which leverage these techniques. First, we describe Dettrace, a deterministic container abstraction. Dettrace provides a containerized environment where any computation inside the container is guaranteed to be deterministic. Next, we describe ProcessCache implements a system for automatically caching results of process-level computations. ProcessCache automatically infers inputs and outputs to a program and will only re-execute a process if its inputs have changed. Otherwise, it skips unnecessary recomputation by using the cached results. Finally, Tivo combines lightweight determinism enforcement with record and replay to suppress certain types of thread-level nondeterminism.

Finally, our future work proposes ChaOS, a fuzzing system for fault injection at system call sites. Lastly, we list key features and requirements for next generation program tracing and low-level process manipulation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

Introduction & Background

1.1. Program Tracing

Program size and complexity continues to grow with no signs of stopping. Developer tools allow programmers to reason about and understand the behavior of their programs. With program size and complexity growing, program abstractions are essential: abstractions allow programmers to reason about the relevant parts of the program without having to consider all details at all levels of the software stack. One useful way of viewing a program is based on relevant operations a program may execute during its lifetime. For example, we may only be interested in seeing which files a program accesses. This information is not easily available from the source code or program tests.

Program tracing is a useful way of getting this information. Tracing is a technique that allows us to observe the execution of a program. Furthermore, we can opt to only trace a subset of all operations, those relevant to our current task. Tracing allows us to focus on the high-level operations we are interested in, without having to consider the low-level details of program execution. While program tracing may be useful at at any level of the software stack, this dissertation focuses on OS-level tracing. We do not focus on other useful tracing methods, such as dynamic instrumentation tools [40, 73]. OS-level tracing offers several unique advantages: it allows us to treat programs as black-boxes, only focusing on the IO operations a program executes. This abstracts over many details of program execution: the programming language, compiler, libraries, runtime environment, etc. Furthermore, OS-level tracing does not require any source code availability or modification, allowing us to trace arbitrary executables.

Program tracing can be thought of as a type of dynamic program analysis. One example of such a tool is strace. strace is a Linux utility that traces and reports the system call and signals executed by a program. strace allows us to quickly specify system call events we are

intersted in. Some examples of operations that programmers may be interested in tracing include:

- **File system accesses**: Files created, read, or modified by a program. We can also track individual writes or reads to files and view the bytes written or read.
- **Networking**: We can view networking done by programs using the standard POSIX socket API.
- **Thread/Process Creation and Exit**: Threads or processes spawned and exit events.
- **System Calls**: System calls executed by a program to gain insight to its execution.
- **Signals**: Signals received by a program.

1.2. Low-level Process Manipulation

Program tracing can be thought of as a read-only operation. Tracing observes the behavior of a program but does not modify or affect (functionally) the execution. The same methods used for program tracing can be used to dynamically modify the state of a program. Beyond tracing, this dissertation focuses on *low-level program manipulation*. We argue program manipulation is a useful as a general technique with many applications to software systems. This dissertation presents various research projects, based on program manipulation methods, to build useful systems in various areas of systems. We focus on the main method for tracing and manipulating processes in Linux: ptrace. The future work section (4.8) gives an overview of other tracing and process manipulation techniques. This type of process manipulation works at a low level of the software stack, between the program executing in userspace but above the kernel. Manipulating a process' execution requires familiarity with many low-level OS concepts such as program registers, the system call ABI, signal delivery, etc. Therefore we use the term low-level process manipulation to refer to this type technique.

1.3. System Call Interposition

"The system call is the fundamental interface between an application and the Linux kernel" [28]. In Linux, executing nearly all IO operations requires a direct system call. Recent

advancements in kernel-bypass IO are a notable exception to this, as they allow data path IO operations to execute without calling into kernel space; however, their control plane operations still require calls into the kernel. Therefore, a natural way of implementing OS-level tracing utilities is via system call interposition/interception (we use both terms interchangeably in this dissertation). System call interposition allows us to introspect all (or some subset) of system calls made by a program. System call interposition is the main way we implement process-level manipulation techniques.

1.3.1. Use Cases

Program tracing is a powerful building block leveraged by various previous works. For example: to guarantee deterministic execution of batch processing [94], to precisely determine build dependencies [95], to find process-level races [67], and to build userspace record-and-replay systems [84]. System call interposition can emulate system call executions. For example, by intercepting Windows API calls, systems like Wine [22] and Proton [13] allow native Windows applications to run unmodified in a POSIX environment. Combining system call tracing, emulation, and modification allows us to build useful low-level systems. By intercepting, emulating, and modifying non-deterministic system calls, Dettrace [82] provides a deterministic container abstraction where any binary executed inside the container is guaranteed to be deterministic. ProcessCache (3) allows for automatic caching and skipping of redundant process execution, when none of the inputs to the computation have changed.

1.3.2. Security Implications

Program tracing and low-level process manipulation provide many benefits and use cases, as outlined above. There are also security implications and open questions raised by these methods. All popular tracing methods allow the tracer to read arbitrary program memory. While this may superficially seem like a security concern, Linux uses per-user capabilities to determine permissions for program tracing. So tracing does not allow for "more" access to processes than the user already had.

Tracing can be thought of as a "read only" operation which merely observes the state and IO

of a program. Contrast this to low-level process manipulation. Which allows us to change the way a program executes. This comes with security implications, as incorrectly manipulating a process could lead to security vulnerabilities in an otherwise correct program. Our process-level manipulation works at the OS-level, so we may circumvent language runtimes or compilers, which may have provided execution guarantees and invariants to the program. Future work in this area should address this question by considering how we can provide low-level process manipulation operations in a safe manner while allowing programs to execute correctly.

1.4. The ptrace API

Ptrace is a Linux mechanism for tracing the execution of another program. Ptrace is powerful enough to implement debuggers like GDB and system call tracing utilities like strace. Ptrace allows one process, the tracer, to trace the execution of possibly many processes or threads, the tracees. A tracee executes until some event specified by the tracer occurs. On such an event the tracee is stopped and the tracer receives an event message from the OS. Possible events include: system call execution, instruction execution, process spawn, process calls execve, signal arrival, process exit, and more.

While the tracee is stopped, the tracer can read and write to arbitrary memory and registers of the tracee. There are two system call interception events:

- Pre-hook events: The program is stopped before the system call is executed.
- Post-hook events: The program is stopped right after the system call is complete but before it returns control to the program.

We can combine ptrace interception events plus arbitrary reads and writes to memory and registers to create powerful process manipulation mechanisms. With some work, ptrace can be used to inject arbitrary system calls into a process, change a system call's arguments before it is called, skip or emulate system calls on behalf of the tracee, and more.

1.4.1. Improving Interception Performance with seccomp-bpf

By default ptrace stops the tracee for every system call twice (pre-hook event and post-hook) , but Linux's seccomp-bpf mechanism allows for selective system call interception, avoiding interception overhead on system calls we are not interested in. Futhermore seccomp also allows the interception code to dynamically decide whether to await a system call's post-hook event, or simply skip until the next system call. This allows us, in certain scenarios, to skip the post-hook event altogether, 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. Older Linux kernel versions (< 4.8) can still be supported in a backwards compatible manner by falling back to the slower implementation.

1.4.2. Building More Complicated Operations with ptrace

The ptrace primitive operations can be used as building blocks to build more interesting, useful process manipulation mechanisms. In this section we show how to implement various general process manipulation mechanisms.

System call modification. During a pre-hook system call event, we may write to the registers and memory of a tracee. Therefore we can observe the current system call about to execute using PTRACE_GETREGS and then write to the registers (via PTRACE_SETREGS) to change any argument to the system call. We may further change the system call itself by writing the corresponding system call number to the RAX register. Arguments to this system call may be specified via the other registers based on the Linux ABI.

System call replaying. We have access to a process' instruction pointer via the RIP register. This allows us to "rewind" execution of a process by setting the value of RIP. All instructions on x86-64 capable of calling a system call are two bytes long. So setting the RIP register to $RIP - 2$ allows us to replay a system call an arbitrary number of times at system call sites. Arbitrary arguments and memory may be modified between repeated calls to this system call for a flexible and powerful system call replaying mechanism.

System call injection. System call replaying can be viewed as a special case of system call injection. It is easiest to inject arbitrary system calls during a system call interception event. All we have to do is change the registers to specify system calls on the tracee. A more intricate approach allows us to inject system calls at arbitrary points in the program execution. For example, we may rewrite the current code pointed by RIP with a system call instruction. This process in nontrivial and there are various edge cases to consider [40] [1].

[1] An alternate approach would record the instruction pointer address of first use of a system call instruction, then to inject a system call at arbitrary points, we could save the current value of RIP and overwrite it with our recorded system call instruction address. This would allow arbitrary system call injection during any ptrace event without requiring rewriting of code. While simpler to implement, our performance overhead would worsen when executing instruction jumps from distant regions of code.

CHAPTER 2

Reproducible Containers

2.1. 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 [2, 15], reproducible data analytics [4, 9], and fault-tolerant distributed systems [38, 56, 58, 93]. Yet in spite of previous work on deterministic languages [35, 75, 94] and operating systems [31, 34, 61], 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 [31] does not provide a repeatable notion of deterministic time. dOS [34] 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 (subsection 2.7.3), 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 (ultimately 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 such as 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.

2.2. 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 [29] and enabling transparent fault recovery [56]. 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) [15] effort, target bitwise reproducibility of all their packages. Microsoft is pursuing reproducible software builds [88] with support in its C# and VB compilers [62]. Google's Blaze/Bazel build system [2] 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 [24] 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. At present, after more than 5 years of effort by dozens of DRB contributors, 5.2% of current Debian packages (1,289 in all) remain irreproducible. While tools exist to identify sources of irreproducibility [89], 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 irrepro-

ducibility.

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 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 [81] and can be seen in our community in the growing artifact evaluation movement. We find a common bioinformatics tool to be irreproducible (subsection 2.6.1).

Machine Learning There is growing interest in reproducible machine learning (ML) [91]. 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 [46, 63].

2.3. 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 [72], 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 filesys-

tem 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* [66]: 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 (subsection 2.5.8) or lower performance (subsection 1.4.1). 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.

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 2.1 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

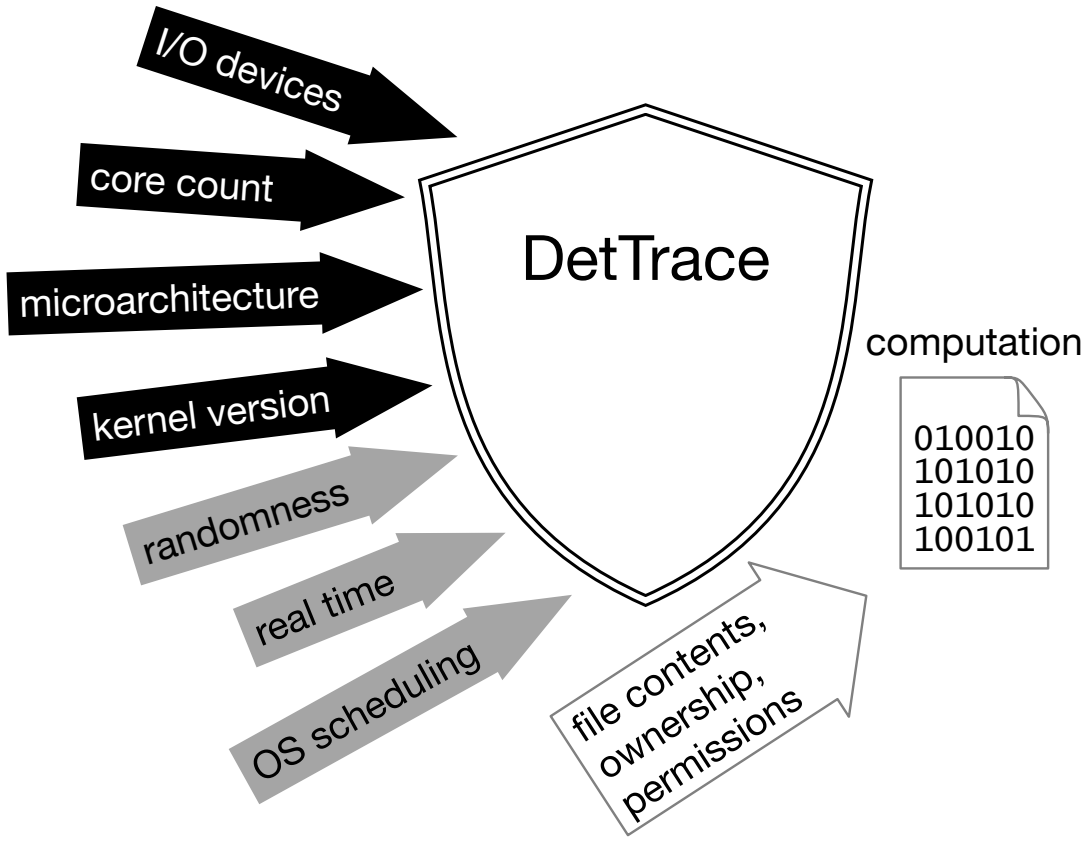

Figure 2.1: DetTrace containers abstract away both sources of nondeterminism (gray arrows) and nonportability (black arrows), making a DetTrace computation a pure function of its initial file state.

for domains like building and testing software where reproducibility is critical.

2.4. 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 [87] 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

[2]There are some undocumented x86-64 instructions [43]. Handling these would be an interesting avenue for future work.

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 subsection 2.5.3). 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 [7]. 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 (also noted by [85]). 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 [25] 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.

2.5. 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

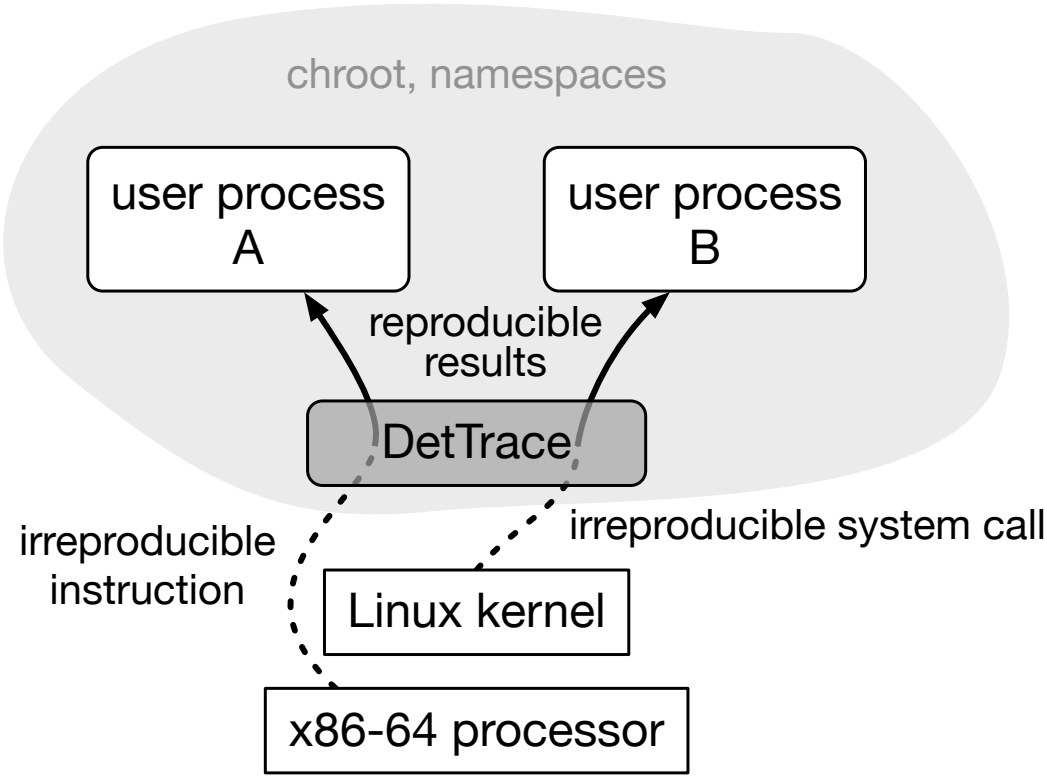

Figure 2.2: High-level overview of DetTrace's organization. The unshaded blocks (processor, kernel and user programs) are completely unmodified.

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 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.

2.5.1. 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.

2.5.2. 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 pseudorandom 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 randomness in a controlled way. User processes can also obtain randomness via the x86-64 instructions rdrand and rdseed, discussed later in subsection 2.5.8.

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.

2.5.3. 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 subsection 2.5.8. Filesystem timestamps are a final source of timing information which we discuss in subsection 2.5.5. With nondeterministic parallelism, racing threads can recreate high-resolution clocks, but our deterministic scheduling renders this moot [30].

2.5.4. 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 [86]. 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.

2.5.5. 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 2.1). 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

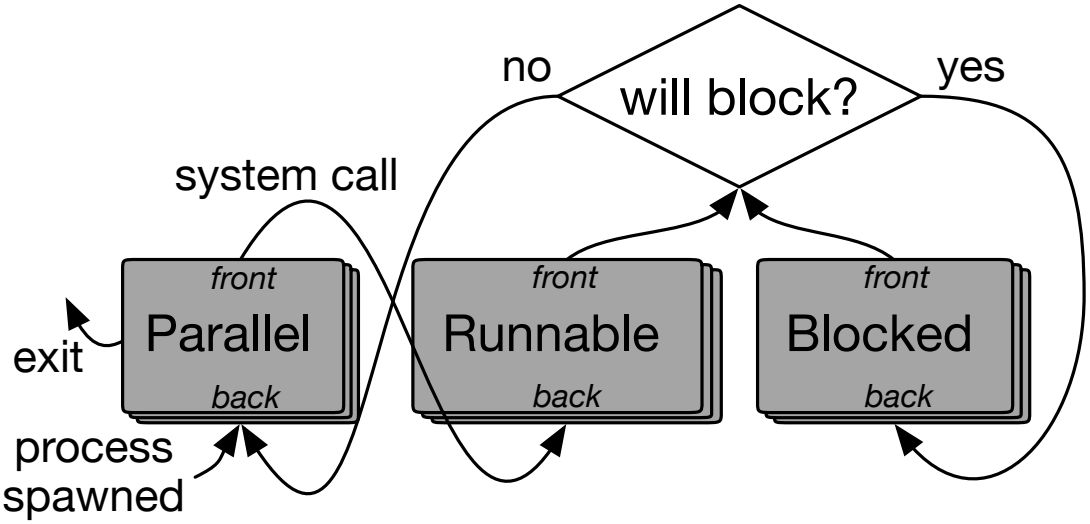

Figure 2.3: State transitions for a user process in the DetTrace scheduler.

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.

2.5.6. 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 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 2.3 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.

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.

2.5.7. 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 (Figure 2.5.6). 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 (subsection 2.5.9).

2.5.8. CPU Instructions

While irreproducible CPU instructions cannot be intercepted through ptrace, recent x86 hardware provides mechanisms for intercepting many irreproducible instructions (section 2.4). 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 2.4 (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.

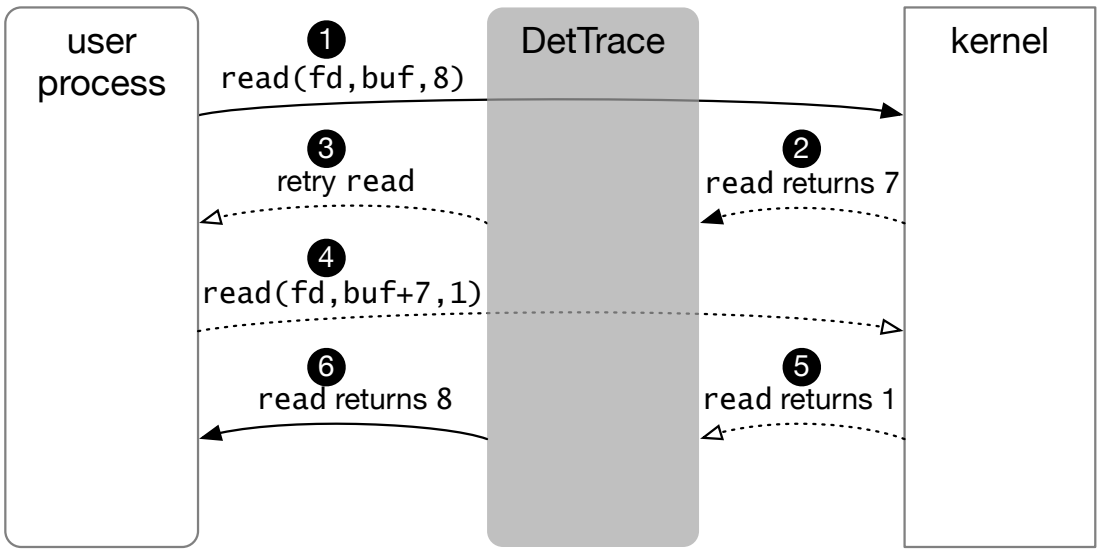

Figure 2.4: To render the read system call reproducible, DetTrace retries read operations that do not return the requested number of bytes. The solid arrows indicate what the user process perceives to have occurred. The dashed arrows indicate extra operations DetTrace undertakes to provide the illusion of reproducibility.

2.5.9. 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. subsection 2.7.1 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.

2.5.10. System Call Modification

DetTrace uses ptrace to intercept but then skip certain system calls, e.g., timer calls that DetTrace emulates internally (subsection 2.5.4). 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 2.4 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.

2.6. 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 (subsection 2.5.8). 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 [98], Clustal 2.1 in -ALIGN mode [42] and HMMER 3.1b2 [54]. We used TensorFlow v1.14 in our ML experiments, using the alexnet and cifar10 tutorials [99] 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.

2.6.1. 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 [16] 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

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

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 [19] 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 subsection 2.7.6), due to, e.g., randomization of the training set. DetTrace renders these workloads reproducible without any code changes.

Given	DT Reprod.	DT Irreprod.	DT Unsupported	DT Timeout
BL Irreprod. (11,958)	72.65% (8,688)	0% (0)	15.99% (1,912)	11.36% (1,358)
BL Reprod. (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 2.1: (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.

2.7. Evaluation

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

2.7.1. 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 subsection 2.6.1. *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 subsection 2.7.1.

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 2.1 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. Reassuringly,

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

The bottom table in Table 2.1 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.

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. Note our Java detection heuristic does not apply to other cases of busy waiting, which result in a *timeout* instead.

Comparison with DRB

407 of the packages that are reproducible under DetTrace are identified as *irreproducible* in the current stretch release by DRB [10]. 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.

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.

2.7.2. 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.

2.7.3. 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 2.6) 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.

2.7.4. 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 2.5 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 2.5 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

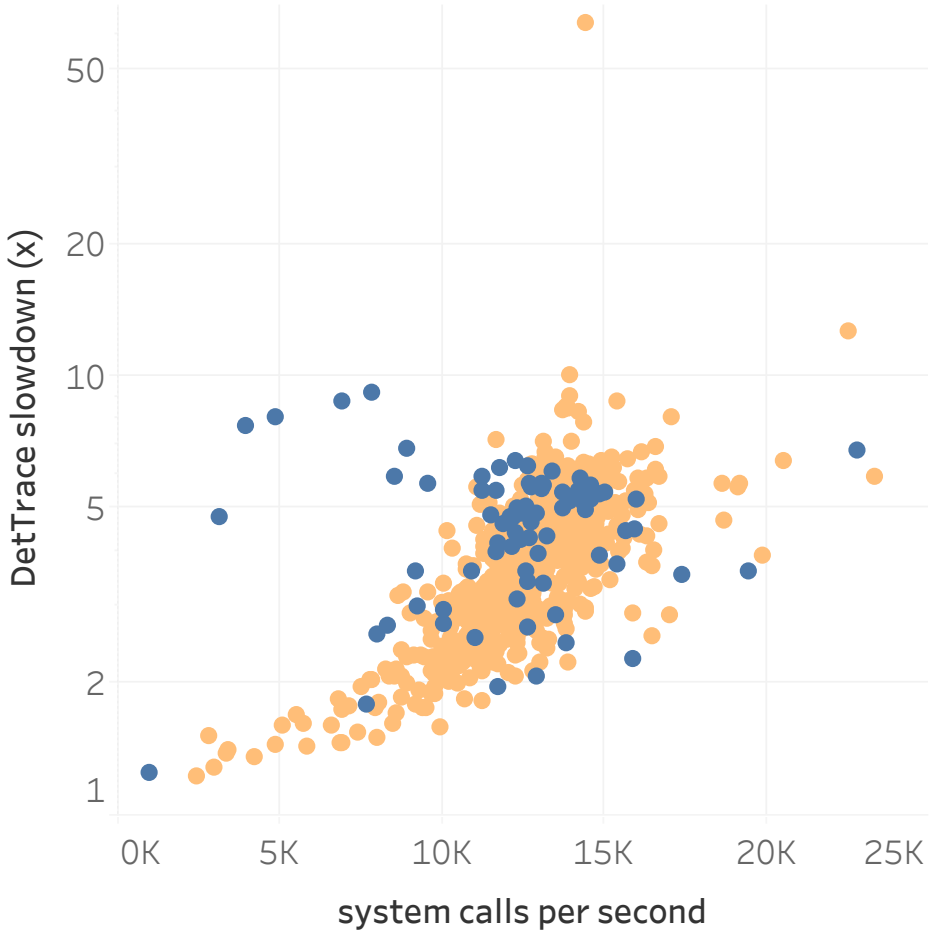

Figure 2.5: DetTrace overhead (y-axis, log scale) is largely driven by the rate at which system calls are performed (x-axis). Packages that use threads (dark blue dots) are typically slower than those that do not (light orange dots).

build (Table 2.2). 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.

2.7.5. Bioinformatics Workflows

Our three bioinformatics workflows use process-level parallelism for performance, and exhibit a range of overheads with DetTrace, dictated by the degree to which they are compute-bound. Figure 2.6 shows the speedup each workload observes with more parallel processes, normalized to sequential native execution. The highly compute-bound clustal performs

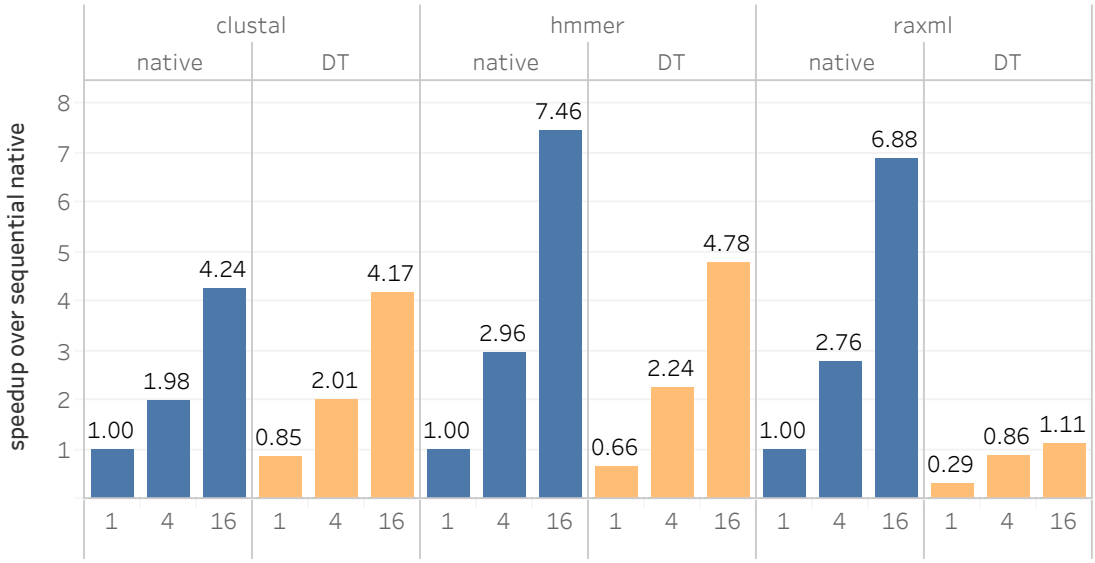

Figure 2.6: Speedup of bioinformatics workflows with 1, 4 & 16 parallel processes, normalized to sequential native execution (higher is better). Dark blue bars are native execution, and light orange bars are DetTrace.

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.

2.7.6. 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.

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 2.2: Per-package average number of events encountered by DetTrace.

2.8. 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 [47, 52, 53, 68, 76, 90, 100] and industry [12, 23, 84]. 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 [84] as it also relies on ptrace (a quantitative comparison with rr appears in subsection 2.7.1). 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 [31] provides new OS abstractions for deterministic fork-join parallelism, and DDOS [61] focuses on local network interactions, dOS [34] 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 [33, 49, 50, 60, 65, 70, 77, 78, 79], providing generality at a modest performance overhead. Other schemes leverage language support to provide determinism for Haskell [39, 66, 69, 74, 75, 83] or Java [35, 36] 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 [94] which provides deterministic parallel execution for batch jobs, though it lacks robust system call interception and requires a coordinator layer written in Haskell.

CHAPTER 3

ProcessCache: Automatic Process-Level Memoization

This chapter covers the ProcessCache project. In many ways, ProcessCache is a spiritual successor to DetTrace. ProcessCache furthers what is possible with low-level process manipulation by building on the methods we developed with DetTrace. The implementation of ProcessCache is informed by problems encountered in DetTrace.

3.1. Introduction

Existing systems already attempt to skip unnecessary recomputation. At the language-level, a runtime may attempt cache previously computed results, a technique known as memoization. This technique is most popular in purely functional programming languages like Haskell, where one can statically determine the IO and side effects of a function. Futhermore, programmers may implement their own memoization or caching in an ad-hoc manner for performance gains.

Build systems like Make [6] and Bazel [3] also skip unnecessary rebuilding when it determines none of the input sources to a build target have changed. The user must manually specify the input sources to a build step and the output build artifacts. Under-specifying or over-specifying dependencies or outputs can lead to erroneous builds [96], and the all too common need to `make clean` when software builds are not properly updating. Forward build systems [97] like Rattle [14] attempt to avoid these issues by automatically inferring build dependencies and build targets.

Incremental computing aims to only recompute what is strictly necessary when inputs have changed, thus saving computing time and energy. This technique can be implemented at the language level [59] or as a language-level framework/library [17]. Incremental computing can be combined with stream processing for low-latency result updates [80].

3.2. Caching/Memoizing Computation

We generalize exisiting ideas of caching and memoization to work at the whole-program granularity. We take a systems approach to avoiding unnecessary recomputation.

In this paper, we introduce ProcessCache a system for automatically skipping unnecessary recomputation. ProcessCache automatically determines all inputs an outputs to a computation and caches the results of any computation it has no seen before. Whenever the same program is executed with the same inputs, ProcessCache skips the actual execution of this program and uses the cached results instead. ProcessCache can be thought of as a general purpose make-like system, except no makefile is required and works on diverse sets of programs beyond build system.

Linux utilizes the execve system call to launch a new process image. This is the standard way a program is launched. We refer to a *program* as the root command that ProcessCache will execute on.

ProcessCache parses the command line arguments passed to it and attempts to execve the given program. ./process _cache [process cache params] – <exe> [exe params] A program is comprised of possibly many processes, those processes may themselves spawn new child processes and contain multiple threads, any child process may further call execve spawning new sub-programs. See Figure 3.1 for details. The root exec-unit will always exists, as it is created by ProcessCache based on the CLI arguments as described above.

We use a unique key to uniquely identify every program 3.3.6. Whenever ProcessCache sees a call to execve, it checks the program specified by the system call. ProcessCache does a lookup in our cache, a cache hit means we have seen this program before. The cache contains the outputs of this program. Instead of executing the program, ProcessCache copies the output file and write the correct output to stdout and stderr. ProcessCache essentially skips running the program but still produces the correct output of the program.

3.2.1. Detemining Program Inputs

Modern programs have many explicit but also implicit inputs. Explicit inputs include: command line arguments, input files, interactive input via stdin, network connections, pipes, etc. Implicit inputs include: any configuration files ready by the program, reading bytes from the network, environment variables.

Any state that could affect the execution of the program could be considered an input. Some of these inputs may not be so obvious, such as file metadata, time, signals, etc.

ProcessCache relies on the assumption that program output is a deterministic function of its inputs, see section 3.2.2. By considering all the inputs to a program we can define a unique program execution as the combination of the program binary and all of its inputs. As long as none of the program inputs have changed, it is safe to skip reexecuting a program, and simply use the cached results.

For correctness, ProcessCache attempts to consider as many inputs to a program. Process-Cache must know be able to observe all program inputs at "exec-time" to determine if any program input has changed. This is the main criteria for deciding which inputs ProcessCache considers: an program input must be observable before the program is executed.

3.2.2. Determinism

ProcessCache operates under the assumption that if a program is executed multiple times, with the same program inputs, it will produce the same outputs, that is, the program executes *deterministically*.

In practice this is not always the case due to various sources of non-determinism that sneak in [82]. ProcessCache is compatible with the orthogonal work on dynamic determinism enforcement at various levels. However, ProcessCache does not require a program to be deterministic or to use a dynamic enforcement mechanism to produce correct output. When ProcessCache encounters a never before seen program plus inputs combination, it captures the results of this program. This is a valid result for this program, subsequent executions of

this program and inputs will be skipped and the cached results will be used instead.

This works around issues of nondeterminism from thread and process scheduling, etc.

As long as we consider all inputs (even tricky ones like time) to a program, it is valid to cache and skip. ProcessCache could be combined with other systems like Dettrace, Arnold [47], Dthreads [71], if you do not like our approach.

3.2.3. Handling Program Side-Effects

A pure program would have no observable effects.

An effectful program will have some observable behavior on the system. Most commonly, a program will make changes to the filesystem, such as modifying, deleting, or writing to files. Write to stdout or stderr. A program may also write bytes to the network, or display results on the terminal or a graphic user interface.

Any obeservable change to the state of the world is known as a program side effect. To effetively simulate a program being skipped with ProcessCache we must be able to perform the same side effects a program would have done. ProcessCache handles common side-effects seen in batch workloads: writing to output files: stateful changes to the filesytem.

ProcessCache does not support networking. There is a long list of possible side effects a Linux program may support which are not implemented due to too much engineering effort, these include: sending signals, creating FIFOs, or many OS operations. These are not fundamental shortcomings of our design however, and can be implemented into our prototype with more engineering effort.

3.2.4. Correctness

When skipping, any caching system like ProcessCache should give the same results as if the computation was to actually be executed. ProcessCache is able to skip redundant programs and only re-execute programs whose inputs have changed.

As an example consider make. Running make from a "clean" state with no build files will

force all build rules to execute. Future runs of make will only execute the build rules where an input has changed. We expect make to produce the same results after a "make clean" versus running "make" and allowing make to automatically skip unnecessary re-computation. This property is known as from-scratch-consistency.

ProcessCache is designed in such a way to always maintain correctness. ProcessCache introspects all relevant IO events a program executes. So it is always able to precisely determine all actions required to later emulate the execution of the program. When ProcessCache detects IO events it cannot handle and skip, e.g. writing bytes to the network, sending signals to other processes, ioctl, ProcessCache can always maintain correctness by simply choosing not to skip the program and falling back to executing it.

3.3. ProcessCache Design

We present ProcessCache, a system to automatically cache and skip unnecessary reexecution of computation. By tracking program input and outputs, ProcessCache is automatically able to detect redundant computation. The next time the same exec is executed, ProcessCache checks if any input has changed. If no input has changed, we can skip re-executing the exec-unit and simply re-execute the program side-effects. An external observer should not be able to determine whether ProcessCache actually executed a exec-unit or skipped it, save for a noticeable speed-up of execution time.

ProcessCache is able to skip entire "exec trees", that is, programs made of multiple calls to processes and execs.

3.3.1. Determining Granularity of Memoization Units

We consider several candidates for cache-unit. A cache-unit is the smallest "unit of work" any system in this space may consider for caching and skipping. These can be small like regions of instructions. They can be domain specific like compilation-units for build systems, or entire programs. So, what is the right granularity of computation to cache? We consider several factors:

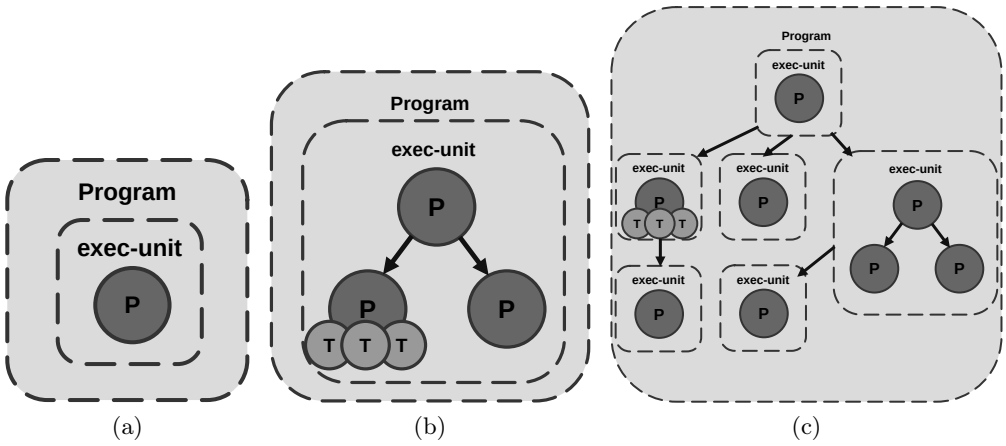

Figure 3.1: The exec-unit provides a useful abstraction over diverse program fork-exec structures. Processes are represented by darker circles with the letter P. Threads are represented by lighter circles with the letter T. (a) In the simplest case, a program contains one exec-unit with one process. (b) An exec-unit may internally contain multiple processes and those processes may be multi-threaded. (c) A program containing multiple exec-unit with their own possibly complex internal structure.

- Size of cache-unit.
- Ease of tracking cache-unit inputs.
- Number of opportunities to skip cache-unit

At one extreme end, we could cache and skip an entire program, but then any changes to an input would force us to reexectue the entire program, so our skipping wouldn't be very useful. On the other end of the spectrum, there are systems like Shortcut [51] which are able to skip "mostly-deterministic" regions of code. Shortcut is able to skip chunks of program instructions at a time. Shortcut's approach provides considerable opportunities to possibly skip, but this comes at the cost of complicated low-level machinery to track inputs and skip instruction-level computation. It is natural to consider the process as a candidate for our caching-unit. In a multi-processed program, we could then skip unnecessary processes, only executing those whose input has changed. Processes are difficult however. A cache-unit should be uniquely identified by its inputs, for a Linux process, the starting input includes the entire state of the parent process from which our process was forked/cloned from. This "input" process-image state is difficult to track. Instead, ProcessCache uses the

Linux execve system calls as our cache-unit. execve provides several advantages to caching. Unlike processes, execs have clear inputs: an exec can be uniquely identified by a tuple containing the: binary executable, command line arguments, environment variables, and any IO inputs read at runtime. This allows us to quickly and easily check if a exec-unit needs to be recomputed (one of its inputs has changed) or can be skipped.

Other benefits:

- Many Linux programs are written in a "fork-exec" style of computation. So execs provide a natural cache-unit.
- This design naturally supports multi-process execs (one exec call which spawns multiple processes) and multi-threaded programs. We only consider the execs and implementation details of a program like number of processes, threads, pipes, etc are abstracted over.

3.3.2. The exec-unit

ProcessCache works at the granularity of exec-unit. ProcessCache caches the results of exec-unit and at every exec IO event, determines whether it can skip this exec-unit. Figure 3.1a show the simplest case for a program: a single exec-unit containing one process. The exec-unit abstracts over the process and thread structure of a program. Just like processes form a natural tree structure for programs, exec-unit also create a tree structure as shown in Figure 3.1c. Every individual exec-unit in Figure 3.1c is a candidate for skipping. Furthermore, ProcessCache can also cache entire exec-unit trees or subtrees.

Programs with a fork-exec structure are the best candidates for ProcessCache, as this creates the most candidate exec-unit for skipping. Many Linux workloads follow the fork-exec paradigm, making our exec-unit a natural choice.

3.3.3. Tracing Program IO Events

Build systems could be consider a type of caching/memoizing system where the user is expected to explicitly list every dependency and output of a build. This process both error

prone and tedious. A user could over-specify (forcing the build system to rebuild unnecessarily) or under-specify (no rebuilding will happen even though an input has changed) dependencies. To avoid these issues, ProcessCache automatically determines all program dependencies by tracing all relevant IO events a program executes.

Our tracing happens at the system call level. We trace relevant IO system calls. This allows us to precisely determine the inputs and outputs to a program. It would be prohibitively expensive to trace every system call a program does. So ProcessCache only traces a subset of all system calls. For example, IO-bound processes can perform many calls to read and write. Instead for file events, we intercept calls to open, creat, openat, etc. By introspecting the mode argument to this system call we can determine whether this file was open for read (input file) for write, truncate, read/write etc. So we use the argument flags to conservatively determine file inputs and outputs.

System Call Events

We find only a small subset of all system calls are necessary to intercept for ProcessCache. Mainly, ProcessCache intercepts and analyzes only those system calls that actually modify the filesystem or informs the program about the starting state of the world. These are: access, clone, execve, exit, fork/clone, the stat-family, getdents, rename, and unlink.

In isolation, individual system call events are insufficient to correctly generate program preconditions and postconditions. System calls tell us part of the story but: system call ordering, parameters, and return values are also required. System calls, their arguments and return values are observed to generated higher-level constructs we refer to as SyscallEvents. This allows ProcessCache to work with high-level events which abstract over the low-level details of OS system calls. ProcessCache takes these ordered lists of SyscallEvents and generates preconditions and postconditions 3.3.6.

3.3.4. Cache Design

Our cache holds all the program executions that ProcessCache has traced. Our cache aims to provide quick cache look up to determine whether we have seen a program execute before. The cache must persist across many program executions, so it is persistent state on disk. Our cache maps keys which identify unique program executions to all metadata that ProcessCache generated during program tracing. This metadata is utilized by ProcessCache to determine several things: the inputs to this program execution and the state of the world before this program executed, the side-effects and state after this program executed, all output files the program generated.

3.3.5. Tracing New Executions

A cache miss means we have never seen this program plus inputs execute before. Or some preconditions for the execution fail, ProcessCache traces this as a new execution. ProcessCache will utilize its event tracer to introspect all IO events for this new execution. ProcessCache monitors all processes and threads for the current exec-unit and any child exec-unit. This creates an execution tree of exec-units. At the end of this program execution, ProcessCache generated IO event "facts" to create the preconditions and postconditions of the execution tree. The preconditions and postconditions are serialized written out to our persistent cache.

3.3.6. Uniquely Identifying A Program

In ProcessCache, a computation can uniquely be identified by the inputs to our exec-unit. Specifically a computation can be uniquely identified by the following tuple: (program binary, command line arguments, environment variables, current working directory, and input file).

Preconditions & PostConditions

Preconditions are predicates about the state of the world before a program is executed. For example, the content of all input files, but extends beyond inputs to a program to include current working directory, and state environment variables. Postcoditions are predicates

about side-effects that a exec-unit executes during its execution. Postcoditions tells us what the state of the world must look like after a program executes, e.g. file foo.txt now exists at a certain location in the filesystems with some specific contents and metadata.

Generating Preconditions and Postconditions

Once execution ends, Process Cache must analyze the list of system call events for each resource. Process Cache iterates through the list and adjusts the "current state" of the resource in a state machine type of way until a final "state" is produced when the full list has been enumerated. It does this twice: once to produce the final "preconditions" and once to produce the final "postconditions". Process Cache starts with a basic "starting state" of the resource, where no information is known, and as it examines the list of system call events, each event causes a different transition of the state of the conditions.

The state machine technique may be obviously necessary for postconditions, because post-conditions are determined by many events happening to the resource. It may seem redundant or unnecessary to iterate through all events to produce the preconditions. One might instead try to generate the preconditions of a resource by simply taking the union of the events that happened to a resource, or just looking at the first event that took place. This, however, is not a sufficient method for generating preconditions.

3.3.7. Checking & Skipping Unnecessary Program Executions

In this section we outline the steps necessary for skipping program executions.

Skipping Program Execution

When ProcessCache determines an exec-unit can be skipped 3.3.7 it does the following:

1. ProcessCache determines the current exec-unit can be completely skipped.

2. We could simply inject a exit/exit-group (exit-group is more correct as it handles exit of multithreaded programs properly) system call into the current process. This would not always work as program may rely on the close-on-exec semantics of programs to execute properly. So instead we execute the execve system call with an empty binary

which immediately call exit with the correct error code.

3. We consult the cache and execute all side effects this computation would have done. Namely, we copy all relevant output files to their correct location and write the correct bytes to stdout and stderr.

4. The rest of the program continues executing under ProcessCache.

Skipping Leaf Nodes

We start by explaining how ProcessCache skips a "leaf-node". A leaf-node is any exec-unit which does not have any further child exec-units. The next section generalizes this idea to caching entire exec-unit-subtrees, these are arbitrary exec-units which may have children. See Figure 3.1c for an example of a program with a exec-unit tree.

ProcessCache makes skipping decision at program execve-time. ProcessCache uses the arguments to execve plus additional context (see Section 3.3.6) to generate a unique key. This key is used for a lookup in our cache. The cache contains two important sets of predicates: the set of preconditions that must hold true for us to be able skip the execution, and the postcondition set. ProcessCache iterates through the set of preconditions asserting all of them hold true (See Section 3.4.2 for more information). If all the preconditions hold true, ProcessCache knows none of the inputs to the program have changed. Therefore it is correct to skip this program. ProcessCache skips the program execution as outlined in Section 3.3.7.

ProcessCache uses the postcondition set to replicate the side-effects this program would have made had it run. An outside observer should not be able to tell actually running the exec-unit versus ProcessCache skipping it, aside from a difference in execution time.

Skipping Entire Subtrees

Skipping leaf nodes is simpler as we only have to reason about the pre and postconditions to an individual exec-unit. Skipping exec-unit-subtrees requires us to reason about dependencies between exec-units we are attempting to skip. This means that ProcessCache assumes that different exec-units do not have any races with shared files, this assumption can be

checked by systems like RacePro [67]. The algorithm for computing the preconditions for a multi-exec-unit subtree is currently a work in progress. Note that simply unioning preconditions of all the child exec-units within a subtree is not enough.

3.4. ProcessCache Implementation

ProcessCache is implemented as a command line utility. Any program can run under ProcessCache by simply prepending ProcessCache to the command, for example `> ./process-cache ls -ahl`. ProcessCache is written entirely in Rust.

3.4.1. Tracing IO Events of Tracee(s)

ProcessCache uses the Linux ptrace system call for tracing IO events, skipping exec-unit, and redirecting IO streams. ProcessCache is implemented entirely as a userspace Linux program and does not require elevated privileges to execute beyond the CAP_SYS_PTRACE capability.

While our prototype relies on ptrace and Linux-specific implementation details, the methods presented in this paper may work with any IO tracing mechanism. Our methodology may be used with another OS (windows, MacOS) or some other interception mechanism (dtrace or in-kernel).

3.4.2. Asynchronous Handling of Ptrace Events

ProcessCache implements a reusable library and runtime for handling ptrace IO event. This provides an a high-level abstraction and allows code to be written in an asynchronous programming model, which mirrors the logical steps of the code. See Section 5.2.2 for a detailed explanation.

Precondition Checking

All preconditions must hold true if ProcessCache is to skip an exec-unit. ProcessCache iterates through the list of preconditions asserting they still hold true. For example, if a precondition states a some file `foo.txt` should exists with some contents and metadata, ProcessCache will perform the necessary system calls to assert these predicates hold true.

ProcessCache attempts to optimize this step in two ways: First, the precondition set is already the minimal set of predicates that must hold true (any redundant predicates are eliminated before this step). Second, we attempt to perform the minimal set of system calls to check preconditions.

3.4.3. Limitations

ProcessCache does not fully handle pipes. Intra-exec-unit pipes are easily handled by ProcessCache thanks to our exec-unit-level abstraction. An entire exec-unit containing a pipe will be entirely skipped or not at all. However, if a pipe spans multiple exec-units, problems may arise. For example, what if two processes in different exec-units communicate via a pipe. Could be skip the execution of one exec-unit but not the other? This exec-unit may be stuck waiting to read from the pipe because the other process was skipped. ProcessCache could support pipe operations by keeping track of the pipe file descriptors inherited across execve. Tracking the movement of file descriptors in this matter is not trivial.

Standard input is also not handled. ProcessCache is designed to handle non-interactive, batch processing style programs. In general, ProcessCache must know all inputs to a program at execve-time to make a decision about skipping a exec-unit.

ProcessCache does not handle networking IO. If ProcessCache detects the use of networking or socket related system calls, it simply stops tracing and lets the execution run without caching it. Networking computation is inherently based on back and forth communication (requests and responses), which is a similar paradigm to the problematic pipes described above. For example, if the serving process is skipped, there will be no one to respond to the requests. Sockets are similar in some ways to stdin. ProcessCache needs to validate that inputs have not changed at the start of execution, but sockets can constantly be receiving new inputs, and this model does not inherently work with ProcessCache.

3.5. Evaluation

The following evaluation was written entirely by Kelly Shiptoski with some minor revision by me. In our evaluation of ProcessCache, we hope to answer the following major questions:

- How well does ProcessCache perform? Does skipping *amortize* the cost of caching?
- Is ProcessCache *robust* to a diverse set of workloads?
- Can it be demonstrated that ProcessCache *correctly* skips executions, provided those programs do not violate the minimal assumptions ProcessCache makes (Section 3.4.3)?
- How does ProcessCache perform when some executions are skipped and others run?
- How much space does the cache use?
- What is the difference in performance when ProcessCache is running from a empty cache versus an existing cache?

The ProcessCache prototype is currently in its last stages, and we are working to identify slow parts of the system and optimize them. This evaluation covers optimizations up to this point, followed by our plans for the full paper evaluation.

To analyze how ProcessCache performs, we plan to employ a case study methodology to create a set of benchmarks with diverse types of programs. These include:

1. A set of **bioinformatics workflows**, all of which perform different types of biological computation with a different number of spawned processes performing the computations in parallel.

2. A comparison to a real world **build system**, to see how ProcessCache compares to an existing similar system.

3. A to-be-determined **batch workload**: current ideas are a process parallel program where each job compresses a file or converts the format of a photo.

Bioinformatics Workflows

Currently, we have been focusing on the bioinformatics workloads to develop our ProcessCache prototype to a minimum viable product. These workloads are highly parallel and CPU bound, so they are a perfect candidate to benefit from the caching provided by ProcessCache. Second, these workloads are highly diverse; they vary in how many processes they spawn to compute jobs, baseline runtimes, and the time it takes to compute a single job. Plus, as we began to run these workloads under ProcessCache, they provided quite diverse performance results, and so provided a good benchmarking system to analyze how the optimizations improved ProcessCache and to help us spot regressions.

We first trialed the benchmarks under ProcessCache when we felt the system was fairly stable and robust. We found that there was considerable overhead, with the average slowdown being 2.5x, raised especially by the `RAxML` workload which had a slowdown of 6.8x (see the **hashing files** results in Figure 3.3). Obviously, these slowdowns are undesirably high, as ProcessCache strives to be a lightweight system, and if caching is a lot of overhead, skipping will not make up for it and using the system will not be worth it.

To discover why ProcessCache was not performing to our expectations, we used `perf` [11] to examine which functions were being called the most frequently. We also effectively *turned off* sections of the system and ran the workloads under stripped down versions of ProcessCache, to examine which components were running for the most time. See Figure 3.2 for a detailed percentage overhead breakdown of the different components of ProcessCache for each bioinformatics workflow. The following components comprise ProcessCache and contribute to its overall overhead: system call interception via `ptrace`, system call fact generation at system call stoppages, the generation of preconditions and postconditions, hashing input files, and copying the output files to the cache.

It becomes quickly apparent that ProcessCache is spending an exorbitant amount of time hashing input files, and is doing so in every baseline. The reason ProcessCache is hashing

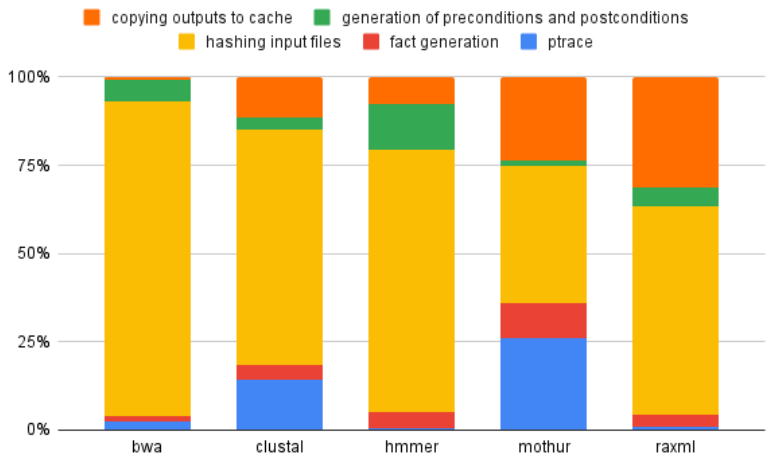

Figure 3.2: Percentage overhead breakdown of each of the different components which comprise ProcessCache for each bioinformatics workflows. The most obvious source of overhead is hashing input files, followed by copying output files to the cache.

input files is to provide a precondition for the program; ProcessCache hashes the input file before the process actually opens it, in case the process modifies it. This has to be done at that time, and cannot be offloaded to some background task, or the wrong version of the contents of the file can be hashed. But, as we can tell, the correctness of hashing comes at a steep cost.

In order to work around the limitations of hashing, two solutions were proposed: copying the input files to the cache, and checking the `mtime` of the resource instead of hashing it. Neither of these is perfect. Copying the files to the cache provides moderate improvement, but does not provide the performance gains we are looking for, and also adds to the space that the cache requires. Using the `mtime` mechanism provides much better speedups, and so was incorporated into the system as an option. Checking the `mtime` is fragile, and is not a portable solution. Luckily, ProcessCache can be run using whichever of these metrics the user chooses, depending on what they value most. The `mtime` checking mechanism provides quite a performance boost, but the `RAxML` workload, which has the most output files (by a significant margin) of the bioinformatics workflows, still has an overhead of 2.7x (see the **mtime + sync copying** results in Figure 3.3).

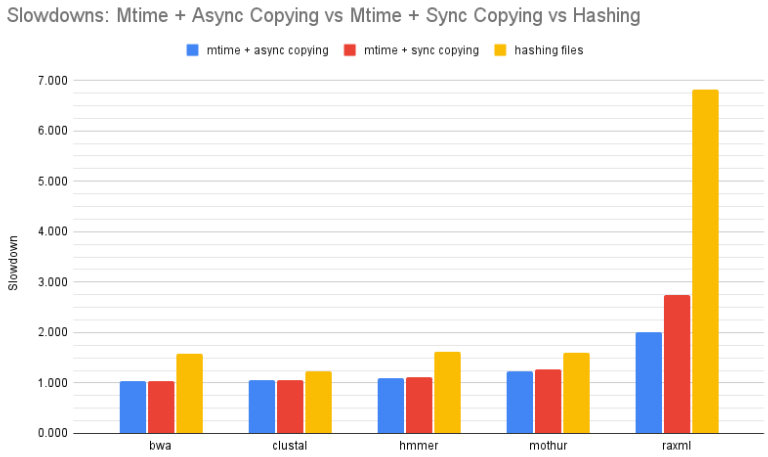

Figure 3.3: Slowdown (normalized to baseline execution) for each bioinformatics workflow. The blue bars indicate the slowdown over baseline of running under ProcessCache with the `mtime` checking mechanism and copying outputs files to the cache asynchronously (one background thread). The red bars indicate the slowdown over baseline of running under ProcessCache with the `mtime` checking mechanism and copying output files to the cache synchronously. The yellow bars indicate the slowdown overbaseline of running under ProcessCache with hashing input files as the checking mechanism and copying output files to the cache synchronously. Lower is better.

The second worst offender when it comes to overhead is the component of ProcessCache that copies the output files to the cache. In the original ProcessCache prototype, this was done at each process exit; this means that ProcessCache is effectively blocked (and so are all the executions it is tracing) until it finishes copying the output files to the cache. Fortunately, unlike hashing input files, copying output files can be moved off the critical path. Because ProcessCache has basic assumptions about how processes modifying the same file behave (Section 3.4.3), and because output files are not copied to the cache until the end of execution, output files can be copied to the cache by background threads. A rudimentary experiment with just one background thread shows promising results (Figure 3.3. We plan to increase the number of parallel background threads copying output files to the cache and incorporate this into the system to improve the performance even more.

CHAPTER 4

Tivo

So far, we have considered OS-level program tracing. In this chapter, we show tracing is useful at other levels of the software stack. This project traces messages at the language-level, this allows us to leverage language-specific and context-specific information not available with OS-based tracing. This allows us to work at the proper level of abstraction.

4.1. Introduction

This chapter presents Tivo, a system for lightweight record-and-replay (RR). In contrast to traditional "fully deterministic" RR solutions, lightweight RR focuses on handling nondeterminism arising from thread communication for programs with concurrent, message-passing architectures. By decreasing nondeterminism in programs, lightweight RR decreases the number of intermittent failures in program's test suites. We evaluated the effectiveness of lightweight RR on Servo, a highly concurrent web browser. Our evaluation shows lightweight RR is effective at greatly reducing intermittent failures for some tests, but not others.

4.2. Background

4.2.1. Concurrent Message Passing Channels

Slowdowns of single-core year-to-year performance gains in computing systems gave rise to the need for highly concurrent systems that take advantage of multiple cores. Which are now available in most CPUs. Concurrency is usually implemented using threads with shared-memory based communication; locks enforce exclusive writes to memory. This type of systems make it difficult to reason about useful properties of concurrent programs, e.g. data race freedom and deadlock avoidance. Furthermore shared data and the locks that protect it are often logically decoupled. That is, grabbing a lock before accessing shared data is not enforced by the programming language or runtime. Instead, in the worst case, it is an implicit programming convention buried in the documentation, or at best, enforced through the object API and encapsulation.

An alternative concurrency model popularized by languages like Go, Erlang, and Rust: message passing through channels. With message passing, isolated threads or processes communicate via channels, sending data as messages to each other. This avoids the need for the programmer to worry about data races, memory consistency models, or other low-level details. Furthermore channels come in many flavors: bounded, unbounded, blocking, non-blocking, and more. Channels can be implemented using shared memory, with lock or lock-free queues. Channels have comparable performance to threads and shared memory.

4.2.2. Intermittent Failures In Servo

Servo [18] is a parallel browser engine with the goals: highly concurrent, memory safe, and easily portable to various opearting systems and hardware. Servo achieves these goals thanks to its use of Rust: a novel systems programming language with stronger memory safety, data race freedom, and static guarantees. Servo also features a concurrent message passing architecture. This allows Servo to be highly concurrent, maximizing parallelism, while maintaining its components decoupled.

Nondeterminism is inherent to concurrent systems. Servo is no exception. Adding to the challenges of programming parallel systems, debugging is more difficult than in sequential programs. This difficulty is further exacerbated by nondeterministic interleaving of threads.

Servo runs a large suite of browser-agnostic tests known as web-platform-tests (WPT) [21] as part of the continuous integration system. WPT tests are expected to pass before changes to the code base are accepted. Unfortunately, Servo has hundreds of WPT tests that fail randomly for unknown reasons, known as intermittent failures. Since many intermittent tests seemingly pass or fail at random. There are unknown sources of nondeterminism in Servo which cause intermittent failures.

4.2.3. Existing Record-and-Replay Solutions

Record-and-replay has been widely studied in academia [47, 48, 55, 57, 61, 76, 92] and used in industry [101]. One such state-of-the-art system is Mozilla's `rr` project [84], which

serves as a robust RR debugging framework. It is successfully used in production with codebases consisting of millions of lines of code like the Firefox web browser. At a high-level, rr works by intercepting nondeterministic OS system calls and recording their values, as well as recording thread and process scheduling. So later it can faithfully replay them. During replay, rr uses its log to replay the system calls with the same values seen during record, while dynamically handling scheduling to schedule threads and processes in the same order. rr can fully determinize any program, so it must record a large amount of information about an execution, so its record logs end up being quite large. Furthermore rr must sequentializes execution of threads and processes, incurring particularly high performance overhead on parallel workloads.

While in principal rr is able to solve the problem with intermittent failures in Servo, we believe there can be a far more lightweight, high-level approach. rr's implementation is OS-dependent as it relies on many low-level OS details. Lightweight RR attempts tame some of the nondeterminism inherent to concurrent systems while operating at a much higher level: channel communication and thread scheduling. All while preserving parallelism in programs, and offering low overhead.

4.3. Lightweight RR

Lightweight RR records and replays multi-threaded systems where communication and coordination happens mainly through channel communication. Lightweight RR does not record all metadata required to perfectly replay a program execution. Instead, it aims to squash several sources of nondeterminism inherent to concurrent systems. This trade off allows lightweight RR several benefits: the implementation is far simpler than a full-fledged RR system, parallelism is maintained, only a tiny fraction of program execution is recorded making the implementation fast and log size small.

Lightweight RR handles the following sources of nondeterminism for channels [4]:

[4]The ideas presented here for lightweight RR are applicable to other areas of concurrency and program replay. See Section 4.9.1 for other possible use cases.

- **Message arrival order**. While messages sent by a thread down a channel are guaranteed to arrive in FIFO order, multiple threads may be sending messages to the same channel simultaneously. Thus, message arrival order can vary.

- **Select operation**. Channel implementations often have a select operation. A select allows for receiving one or more message(s) from any channel with a message at the time of the operation, or blocking until a message arrives. Which messages are ready on a given select is timing dependent, so this is a nondeterministic operation.

If the above were the only sources of nondeterminsm in a program, out approach would always successfully replay any recorded execution. Programs have nondeterminism outside the scope of lightweight RR, e.g. network communication, IO, thread locking [82]. Those programs may not replay successfully with lightweight RR. If the replay execution diverges from the recorded execution, we say the execution has *desynchronized*. We believe we could combine lightweight RR with a lightweight dynamic determinism enforcement technique as done in DetFlow [94].

Lightweight RR is robust to desynchronizations. When desynchonization is detected, lightweight RR can choose to stop replaying the recorded execution and falls back to allowing the program to run natively, or abort the program.

4.3.1. Lightweight RR for Intermittent Failures

Lightweight RR is designed to work in the context of software tests with intermittent failures. Any test that nondeterministically return: unexpected results, timeout, or crash, can benefit from lightweight RR. First, we must capture a execution of the program with an "expected" results, this may take many tries depending on the test. These "expected" logs are stored as part of the testing infrastructure. Later the tests run using lightweight RR in place of the standard channel library. The test run in replay mode where lightweight RR replays the "expected" execution. This allows lightweight RR to lower the nondeterminism and thus the number of intermittent failures for programs.

4.3.2. Supported Operations

For any message-passing channel implementation, we define the *receiver* as the reading end, and the *sender* as the writing end. We assume the channels are unbounded (do not have a maximum capacity), and may support multiple producers but a single consumer (MPSC). These assumptions fit the needs of most programs, However, this is not a fundamental short coming of lightweight RR. Nothing stops us from supporting bounded or multiple producers multiple consumer (MPMC) channels.

Lightweight RR supports all common operations on channels, including:

- **receive**: Block until we receive a message from a sender.
- **select**: Block on one or more receivers until one or more message(s) arrives from any channel.
- **try_receive**: Non blocking variant of receive. If no message is available, an error is returned.
- **timeout_receive**: blocks until a message arrives or the specified time elapses.
- **send**: non-blockingly send a message through a channel.

4.3.3. What is lightweight?

Lightweight RR does not reinvent the wheel by implementing deterministic channels from scratch, instead it is implemented as a wrapper around an existing channel implementation.

Furthermore, unlike fully deterministic approaches to RR, lightweight RR is implemented at the language-level as a user library, agnostic to the operating system and hardware details.

We highlight the following advantages between traditional RR methods and lightweight RR:

- Implemented as a language-level library, portable across operating systems and computer architectures.

- Threads run in parallel.
- Only channel communications events are recorded. No program data is recorded. Leading to smaller record logs.
- Robust to divergence of program execution during replay.
- The implementation wraps an existing channel implementation. Thus, the implementation is a straightforward transform.

4.4. Design

4.4.1. Assigning Deterministic IDs

In order to faithfully replay execution of a program, we must assign a deterministic thread ID (DTI) to all threads. DTIs must be unique and deterministic even in the face of racy thread spawning. Starting from the main thread, every thread spawn event is assigned an ID equal to the number of children that have spawned from this thread so far *child_id*. A single counter per thread keeps tracks of this information. Then, a DTI is generated as a vector of *child_id*s from the main thread, down to the new thread. The main thread has a DTI of `[]` (an empty list). As an example, the first thread spawned by the main thread is assigned a DTI of `[1]`, while the second great-grandchild of the main thread would have a DTI of `[1, 1, 2]`. Therefore, the length of a DTI is equal to the depth of the thread tree.

Analogously, every channel receiver/sender pair is assigned a unique, deterministic channel ID (DCI). The DCI consists of a `(DTI, N)`, where `N` is a per-thread counter, which is increased after generating a DTI.

DTIs are necessary for correctly replaying executions. Meanwhile DCIs are useful to detect desynchronizations, debugging, as well as understanding complex program interactions among threads, channels, and messages.

4.4.2. Recording Events

Given a channel message of type T, we transform it to a tuple (T, DTI). Where DTI is the DTI of the thread who sent this message. That is, a thread now sends its DIT down the channel along with the message, this allows us to track which thread the message originally came from.

During record mode, we let the program run with as little changes to the execution as possible. When the program calls a channel function, we allow the function to go through, and merely record some information necessary for replay later.

For all channel communication events, we record:

- Event Id: A per-thread integer that increases by one on every channel communication event. This should be thought of as the *logical time* that an event occurred at.
- Sender ID: The DTI of the sender of a message.
- Event Type: send, receive, try_receive, select, etc.
- Data Type: The type of data sent down the channel, e.g. string, integer, bool, etc.
- Channel Flavor: The channel variant for an event, e.g. ipc-channel, bounded channel, crossbeam channel, etc.
- Event Status: Possible return variant of an event (see more below).
- DCI: The ID(s) of the channel(s) involved in this event.

Notice our record log does not need to record the value of the data sent down the channel. However, it may be useful to record the bytes sent down channels. This can further aid in debugging or understanding when data values desynchronize.

Event status are the possible return variants of an event. For example, try_receive may either: timeout, get a receive error, or successfully receive a value. In case of a success we

record the DTI of the sender thread. The sender's DTI is important for faithfully replaying executions when there are multiple writers to the same channel.

Select operations have an ordering to their receiver set. Each receiver is given an index to its position on the select, based on the order in which receivers were added to the set. On select events, we record the index (or indices for multiple events) of the receiver(s) with messages ready. Since receivers on a select are themselves MPSC channels. We also record the DTI of the sender for all receivers which returned a message during a select.

Many of the items recorded above are not strictly necessary. Instead, the are useful for robust logging (debugging), and detecting desynchronizations of executions. For users interested in running lightweight RR in production environments, many of these values could be removed based on a compile-time flag.

4.4.3. Replaying Events

While record mode tries to affect the program's execution as little as possible, replaying requires wrangling in the program to force it to execute the same as the recorded execution. We may have to block threads and execute multiple channel receives per single channel receive requested by the user. All these operations are not directly observable to the user, and lightweight RR always maintains correct semantics with respect to the underlying channel implementation. Therefore, any program that works with a channel implementation will work equally well with lightweight RR.

Using (Sender ID, Event ID) we can deterministically identify an event for any thread. For a given event, we consult the log to see what actions are expected for this event. We compare the event type, DCI, and channel flavor to ensure we are still synchronized with the execution. Otherwise, a desynchronization error is returned.

Next, we consult the event status. If the status was e.g. timeout, we don't bother doing the actual operation on the channel, and instead immediately return a timeout. Even if the event status indicates the event succeeded, we may still not do the actual operation.

Instead, no matter the actual receive events e.g. timeout_receive, try_receive, we implement a rr_recv() as follows.

rr_recv() call the blocking receive() on the receiver directly. The blocking receive will eventually return a message of type (DTI, T). We compare the received DTI against the expected DTI from the log. If they match, we have found the correct message to return. Otherwise, this event came from another thread, so we add it to a per-receiver buffer of messages. We loop on receive() until the correct event is found. Next time there is a call to rr_recv(), the buffer is first checked for the expected event before looping on receive().

On select, we use the expected index/indices to retrieve the correct receiver(s) from the select set and call rr_recv() on each receiver. Notice rr_recv takes care of buffering all "wrong" messages from other senders, so the correct message is automatically returned.

We must be careful with our use blocking receive() as a desynchronization may cause the thread to block forever. See Section 4.4.5.

4.4.4. Running Off the End of the Log

When replaying an execution, it is common to fall of the end of the log. That is, there is no (DTI, N + 1) event in the log. Missing log entries usually mean that either: the program did not reach this point of the execution during record, or a desynchronization has occurred.

The former can happen for programs that race between exiting and communicating through channels. While one could argue this a source of nondeterminism outside the scope of lightweight RR, doing so would render lightweight RR useless for many programs. Instead, we handle end of the log events by blocking the thread on a conditional variable (this simulates the behavior of the thread not reaching this event in the recorded execution). See Section 4.4.5 for details on when this conditional variable is notified.

4.4.5. Handling Desynchronizations

We handle desynchronization events robustly by continuing to execute in a best effort mode (See Section 4.8 for possible extensions). Once a desynchronization is detected, initial user

input determines whether the program should error out or continue executing the program.

Programs desynchronize when there are sources of nondeterminism outside of what lightweight RR handles. For example network IO, timers, randomness, etc. These other sources of nondeterminism are outside the scope of this work. We expect many programs to have other sources of nondeterminism. So lightweight RR must be robust to desynchronization errors.

Once we detect that execution has failed. We switch from the replay method described above, to merely doing the operation the user has asked for. This allows us to continue the execution but with no determinism guarantees. We notify all threads blocked on the end-of-log conditional variable, this unblocks these threads and they continue running on desynchronization mode. Special care must be taken to flush any values buffered by our rr_recv() operation.

If a program replay desynchronizes, threads blocking on receive() may never never return. To avoid deadlocks, we use receive_timeout() instead of receive() in the implementation of rr_recv(). If the timeout time elapses, we consider this a type of desynchronization error and run the rest of the program in desync mode.

4.5. Implementation

We implemented lightweight RR as a Rust library rr-channel. Our Rust implementation wraps three popular Rust libraries for channel communication: ipc-channel and crossbeam-channel, and the Rust standard libray channel implementation. Rust does not have support for dependency injection or inheritance, so users must swap all instances of the channel library they are using to rr-channel. To make this process seamless, rr-channel exposes the exact same API as ipc-channel and crossbeam-channel. Therefore, the only change needed is editing the name of the dependency when it is imported[5].

[5]This does not handle the case where a program dependency *itself* uses any of these channels. For our evaluation, it was necessary to download the dependencies source code and manually patch the import. This is a current limiation of our approach.

We must assign DTI to all threads. So a user must also use our rr-channel thread spawn function to ensure the thread receives a DTI. A program may not always be able to use our thread spawn function: threads may spawn as in program dependencies outside the control of the user. rr-channel handles this case by giving these threads a None DTI. Events are still recorded and replayed for None DTI. However, if two threads, both with None DTIs are both writing to the same channel, we cannot distinguish these them, so the execution may not be deterministic. We print a warning to users if this case is detected.

It is straightforward for our rr-channel implementation to support any channel implementations as long as the underlying implementation supports one single channel operation: try_receive().

4.6. Evaluation

For our evaluation, we integrate rr-channel into Servo. Servo is a highly complicated program featuring a fully concurrent, message passing architecture. Servo represents a high target for lightweight RR: as Servo uses many complicated channels, to send many messages among its multitude of threads. Therefore, Servo is the perfect platform to stress-test lightweight RR and get a proper sense on the limits of our approach.

Thanks to the design of rr-channel, lightweight RR integration was fairly simple. Even though Servo is split into many libraries, with dozens of decoupled components. One day's worth of work is enough to have Servo using lightweight RR channels everywhere.

4.6.1. Reducing intermittent failures

To test the effectiveness of our approach we compiled a list of WPT tests known to fail intermittently in Servo. Using Github API we fetched all GitHub Issues for Servo marked as I-intermittent. We found 417 unique intermittently failing tests. We ran those 417 tests 100 times (baseline) to get the average number of times a test returned the following possible statuses:

- expected: The test returned an expected outcome.

- unexpected: The test returned an unexpected outcome.
- crash: Servo crashed unexpectedly while executing this test.
- timeout: The test timed out before returning.

Out of those 417 tests, only 43 displayed intermittent behavior. That is, they test would change from one state above to another. 9/43 never returned "expected" so they were ineligibly for lightweight RR.

To test our lightweight RR solution. We looped on the remaining 34 tests until we captured an expected execution. 2 tests were discarded because we were unable to record an expected execution after 100 tries. Even though these tests returned expected at least once in baseline. Note for the 2 tests in the previous sentence, as well as the 9 tests that never returned expected we could have recorded their unexpected executions as well, if the user is less interested in recording the expected execution, and just wants more consistent test results.

The remaining 32 tests make up our experimental results. Using the expected execution logs, we ran these tests 100 times in replay using our modified Servo.

Figure 4.1 shows the results for our 32 tests. The results are not great, but promising. While lightweight RR does seem to improve results, modestly to greatly depending on the test, sometimes lightweight RR actually returns less expected results. Many tests also failed to run all the way and timed out every time, these represent the missing rr bars in Figure 4.1. See Section 4.7 for discussion and interpretation of this results.

4.6.2. Performance Overhead

Lightweight RR only records channel communication events, a tiny subset of information recorded by traditional RR methods. Therefor we expect performance overhead to be quite smaller than traditional solutions.

We timed the our 32 tests using Servo's testing infrastructure reported elapsed time per test. The average time per tests is based off 100 executions of each test. Both baseline and

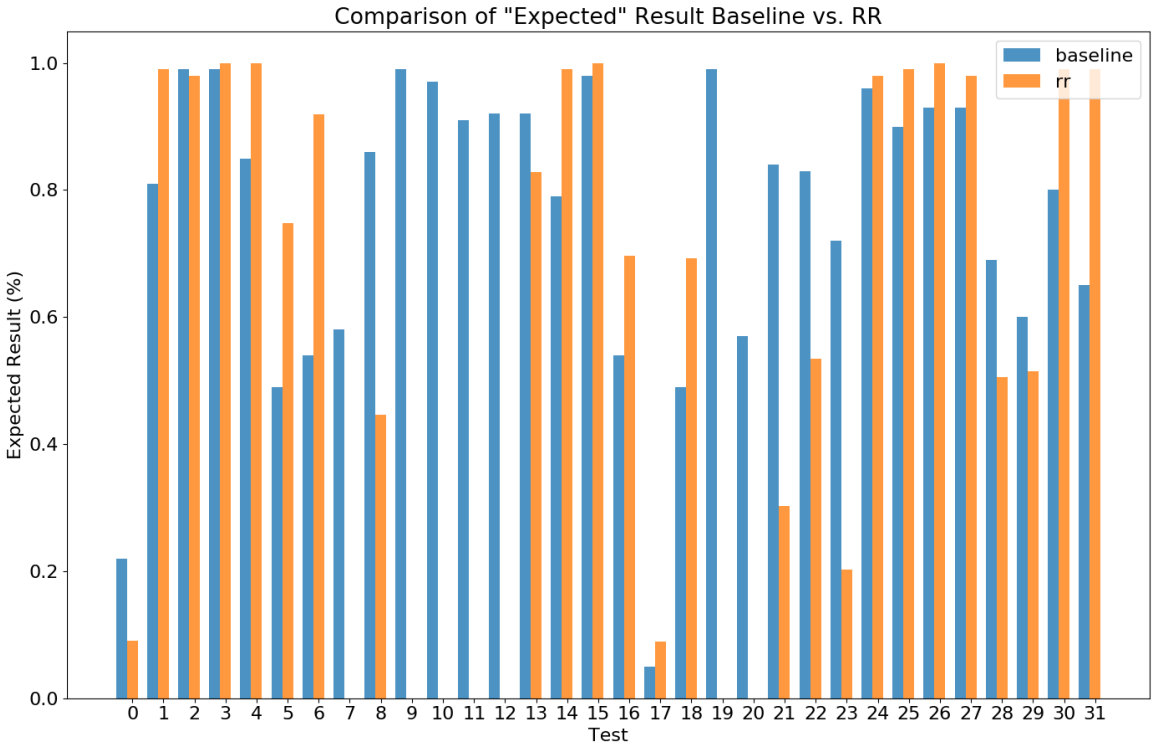

Figure 4.1: Results of comparing baseline with lightweight RR. Tests where rr is higher than the baseline represent cases where lightweight RR improved intermittent test's expected times. Missing orange bars represent a timeout during replay.

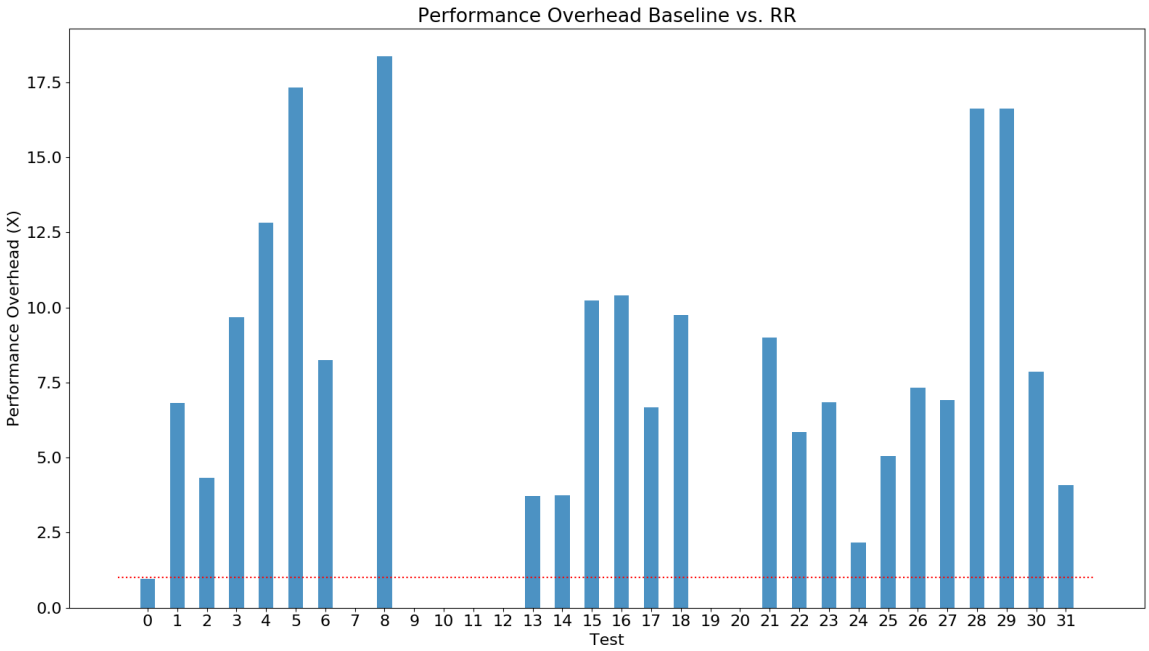

Figure 4.2: Performance of lightweight RR over baseline. Missing entries represent tests where rr-channels failed to run before timeout.

lightweight RR were measured in this manner. Figure 4.2 represents these results. As can be seen in this graph, the performance is terrible. We have identified the possible source of slowdown and hope to rerun the experiments soon.

4.6.3. Space overhead

For the evaluation, the logs were written out using a compact, binary serializer. Out of 30 recorded logs from our tests above, the median log size was 78.5KB. A more in depth comparison would be needed to compare log sizes versus existing systems like rr. However, we believe the log sizes are small enough to not add a storage burden, even for test suites with tens of thousands of tests.

4.7. Discussion

The current evaluation results are disappointing to say the least. Here we discuss some of the reasons why lightweight RR does not work well in its current state.

Several tests failed to run all the way in replay mode, and instead timed out. We attribute

these failures to logic bugs in the implementation. We believe, with further implementation effort, lightweight RR should never deadlock.

Servo is a complicated, highly concurrent program. Therefore, Servo represents a high mark to hit for lightweight RR. Simpler applications may fare better as far as lightweight RR is concerned.

Servo may be far too nondeterministic for lightweight RR to be useful: Servo may be desynchronizing early on, so the rest of the program runs without much benefits from lightweight RR. See Section 4.8 for proposed solutions to this problem. Servo has known sources of nondeterminism we are not capturing. Some examples include rayon, a library for high-performance data-parallelism. Since rayon threads are not spawned through our thread spawn wrapper, these threads are not assigned DTIs. Therefore any message which comes from a rayon thread cannot be properly determinized. We have observed such messages coming from rayon threads. With further implementation it should be possible to assign deterministic thread identifiers to Servo threads. Similarly, Servo uses Tokio library for asynchronous programming. Threads spawned through Tokio are not determinized. Even handling Tokio and rayon threads, Servo naturally has many nondeterministic IO operations. Network traffic, disk IO, and timers, all contribute to nondeterministic executions in Servo.

It may just be the case that Servo is too nondeterministic in its current state to benefit from our approach. Perhaps a combination of lightweight RR and some small modifications of Servo components may yield better results.

4.8. Future Work

The current work just barely scratches the surface: providing an implementation and doing a first attempt experiment. We believe with some iteration we could significantly increase how well lightweight RR works.

Lightweight RR has many possible extensions and future work. Currently, when the replay execution desynchronizes, we merely fall back to executing the program natively. A

more elaborate desynchronization recovery scheme could be robust to some desynchronized events. Some threads could be desynchronized while other threads continue to run synchronized. Similarly, we could look at real synchronization recovery by attempting to match the execution back at a later point.

Once a desynchronization is detected, it is indicative of a source of nondeterminism in some program thread. Using the current execution, and the log, it may be possible to help the programmer narrow down and identify the source of nondeterminism.

When we detect a desynchronization, we could start recording this alternate program execution. Later the two executions could be compared to better understand why they diverged. Furthermore, when considering continuous integration, and as programs evolve overtime, we expect changing part of the code will have changes in the execution, therefore, as new patches are tested for merging we could record the new executions, and use those for future record and replay testing.

The log we record is a full record of all communications and thread topology for a given execution. This log could create a visualization of threads, and channel communication for the program execution. This is extremely useful to a developer as a visualization tool to understand concurrent programs.

Servo is careful to use non-blocking, unbounded, channels. This helps Servo developers reason about deadlock-freedom, as long as there is no cycles in the communications among threads of Servo. Using the recorded logs, we could verify that cyclic dependencies don't exist for any given execution of Servo, thus increasing confidence in deadlock-freedom.

The evaluation is currently incomplete. More work is required to understand what kinds of applications or tests benefit most from lightweight RR (if any). Performance is also currently disappointing, work is needed to understand performance implications of lightweight RR.

4.9. Addendum

In this section we cover some follow up work and thoughts after further research on lightweight RR. Servo proved a hard program for diagnosing and evaluating the efficacy of lightweight RR. This is due to several reasons: Servo is massive and complicated, running scores of threads in parallel, Servo relies on a couple of Rayon parallelism and asynchronous frameworks: rayon and tokio. These frameworks have their own parallelism schemes and schedulers outside the current scope of lighweight RR, but they also present opportunities for future extensions, see Section 4.9.1.

4.9.1. Future Directions and Extensions

In this section I present thoughts, challenges, and possible future extensions for this line of work.

Narrowing Origin of Desynchronizations

The current implementation of Servo + Tivo suffers from a large amount of desynchronization errors. These errors are extremely difficult to track down, even with the robust logging currently present in our Tivo implementation. I believe an automated approach for tracking the origin point of desynchronizations would be useful not only for debugging the design and implementation of Tivo, but as a general tool for developers trying to squash nondeterminism in their program.

Tivo holds all the information necessary to understand how channels are moved between different threads and which channels send messages, from which thread, at any time. This information can be used to generate a "topology" or graph of channel communication for a program. Thinking about how messages flow through a program could help with automatically identifying sources of divergence.

Tivo aims to be lightweight by only recording metadata associated with channel operations. I believe it could also be useful to have a "verbose" Tivo mode, where Tivo also records the data being sent through channels, this could help diagnose and detect divergences earlier.

Generalizing Tivo Ideas

This chapter presents Tivo mainly in the context of channels. I believe these same ideas could be generalized to other contexts to extend the usefulness of this approach.

Atomic locks are useful in any concurrent program, even when channels are used, Servo still uses mutexes in several places. Ensuring deterministic access to data behind a lock is currently beyond the scope of Tivo. We could extend Tivo to support deterministic thread access in the same way we handle channel operations: during record, we use the DTI of a thread to record the ordering of lock acquisition for threads. During replay, threads many only access a lock in the same order as observed during replay. Threads that try acquiring a lock before their time are blocked until it is their turn.

Asynchronous IO is a paradigm currently gaining a lot of traction. Most languages now have some support for asynchronous IO. The Linux kernel recently gained a new API for better asynchronous IO [44]. Many asynchronous IO language implementations provide a programming model where programmers write async functions. These functions may be paused at user-defined yield points, where they wait for some IO operation to complete. A runtime is responsible for scheduling these async functions. The runtime also polls a lower level IO queue where new IO events will arrive. These IO events are then handed back to paused async functions, where they continue executing from their yield point. In some languages like Rust, these async functions are known as *futures* or *tasks*.

This async programming model suffers from analogous nondeterminism problems to those we have discussed in this chapter. I believe this is another context where the ideas presented by lightweight RR could help. We can slightly modify the async runtime to assign a deterministic task identifier to every future. Then lightweight RR records the scheduling of futures and creates a correspondence between an IO message and a specific future. In replay, lightweight RR could ensure IO events are given to the correct future as the record execution.

CHAPTER 5

Future Work

Based on our experience and familiarity with tracing and process-level manipulation mechanisms we outline future work directions. This section gives an overview of a a novel system for automatic fault injection at system call sites. This system, currently named ChaOS, may inject error codes at system call sites. We can used these faults to detect errors that are not being handled by a program and report these edge cases to the user. Then, we overview the key features, interfaces, and abstractions for a successor to ptrace.

5.1. ChaOS: Fault Injection at System Call Sites

5.1.1. Introduction

C and C++ remain the most popular languages for writing systems software. The Linux programming interface is defined entirely in C. So using many Linux APIs requires using these C interfaces. Meanwhile, the C type system is relatively weak and unexpressive. So instead of function invariants being expressed via types, where they can be machine checked, they are relegated deep into the man pages as documentation. Furthermore, C does not feature any language-level error handling support. So C programmers often implement their own ad-hoc error handling boilerplate. Without language or compiler support, C error handling is tedious and error prone. Testing these error handlers is also difficult or sometimes impossible. The lack of error handling support from the C language, combined with the relative rarity of specific system calls failing may lead to hard to find bugs. To address these shortcomings we propose ChaOS, a testing framework for system call failures. ChaOS injects faults at system call sites to automatically root out bugs in software caused by unhandled or buggy system call error handlers. This work is inspired by previous work on dynamic analysis [45], fuzzing [1, 37], and debugging error handling code [64]. Working at the system call level allows us to leverage low-level properties of programs in a novel way. Finally, we leverage our insights on deterministic code execution to develop a fast and sound

approach.

5.1.2. Background

System Call Failures

System calls failures are reported via the integer return value of the function call. A programmer may simply forget to check these return values. Developers are expected to familiarize themselves with the nuanced and rare ways system calls can fail. See 5.1.2 for an example. For example, a fork may fail due a `ENOMEM`, "failed to allocate the necessary kernel structures because memory is tight". While a write to a file may fail due to `ENOSPC` "the device containing the file referred to by fd has no room for the data". These failures can be so rare they never show up on testing environments or production systems for years.

Signals and System Calls

One example of this complicated design is blocking system calls. Any blocking system call can be interrupted at any time by a signal: for example, when a user asks the program to momentarily suspend, or any time a child process exits, a signal is delivered. If this happens, the system call returns with a `EINTR`, and the user is expected to restart the system call by calling it again. This is an unfortunate design quirk of Linux. So in practice, any program that performs a blocking system call must be ready to handle a signal interrupting a system call at any time. Anecdotally, it seems difficult to believe most developers handle this correctly. Even high-level languages like Python have not always handled this behavior correctly [41]. Thus we believe blocking system calls interrupted by signals are an ideal test target for programs.

5.1.3. ChaOS: Fault Injection Testing for System Calls

ChaOS injects a fault at S_n, the nth system call executed by the current process. Every system call has a different set of system call specific error codes. We plan to implement handlers for most common system calls. Choosing which system call to target and which errors to inject will be largely heuristic based. We suspect most programs do not handle specific error codes and instead handle all possible errors "coarsely", therefore the specific

error we inject is less relevant. We plan to experimentally validate this.

Forking At Fault Injection Sites

Consider injecting a failure at S_n where the tracee successfully handles this fault injection by reporting a message to the user and exiting. Now what if we wanted to test S_{n+1}? A naive implementation of ChaOS would have to reexecute the entire program from the beginning. This approach would likely be restrictively slow. Instead, before injecting the fault at S_n, on the ptrace prehook event, we inject a fork system call. The fork system call creates an identical copy of the process and all its resources (e.g. opened file descriptors). We leave the original process `P` in a stopped state and inject the failure into the forked copy `P'`. The copy-on-write (COW) semantics of fork makes this process both fast and lightweight. This will likely be the most technically challenging part of the project. Care must be taken to avoid various issues. Forking tends to be a mechanism with poor orthogonality [32]. Among many issues, fork is not thread-safe. Forking creates a deep copy so executing `P'` cannot modify the memory `P`. `P'` could still alter the state of the system via side effects. For example, writing to a file or reading from a socket. These side effects are the result of system calls execution. So our tracer intercepts system calls of `P'` and ensures no side effect will affect `P`. This way when `P'` is terminated, we can continue executing `P` in the correct program and system state.

Automatically Detecting Failures

Ultimately, ChaOS is designed to automatically detect unhandled system call failures. We defer making guarantees or giving a precise definition of handling system call failures. This is difficult to do as it is unclear what exactly it means for a program to properly "handle" a system call failure. We propose an automatic approach while avoiding (or at least largely minimizing) false positives. Let us consider the possible cases when injecting a fault at a system call site which would normally succeed (See 5.1.3 for ensuring a system call would succeed):

- Easy Case: The program reports an error and exits, we know this program successfully

handled the system call failure.

- Easy Case: The program aborts unexpectedly (e.g segmentation fault) due to our fault injection, we have found an unhandled system call error.
- Hard Case: The program continues executing. Did the program handle the error, recover, and continued executing? Or did it fail to handle the error and is now running in a state which the programmer did not intend?

We may handle the hard case as follows. As stated before, we inject faults into the forked version of the process, `P'`. After the fault injection, we can step through `P'` instruction-by-instruction observing the values of the program counter (PC) register. At the same time we also step through `P` (recall `P` was also stopped at S_n). `P` did not have a fault injected. If the values of the PC register are the same for `P` and `P'` we can conclude the program did not do any handling of the fault. It is currently unclear how many instructions to step through before concluding the error is unhandled. We expect to develop this heuristic based on observation of real world programs. The soundness of this approach relies on our previous work on deterministic program execution [82]. Given that P and P' have identical starting states (this is true by the semantics of the fork system call) stepping through both process executions is a deterministic function of the initial starting state. So any differences in program execution must be attributed to the fault injection.

System call Failures

Not all system call failures represent exceptional circumstances. It is routine for system calls to return error results. System calls like stat and access are commonly used to probe around the filesystem, e.g. attempting to locate where a shared library resides. For our automatic detection of unhandled errors to work as described above, ChaOS should only inject failures at system calls which would succeed in the untouched execution. We ensure we only inject failures at succeeding system call sites as follows: We inject a fork to spawn P' At this point P is in a stopped state. We allow P to execute its S_n system call and stop S_n at the system call post-hook event. In the post-hook event, we may observe the return value of S_n and

only inject a fault into P' if S_n succeeds.

Maintaining Program Invariants and Correctness

In general, programs may hold invariants we cannot know. Our fault injection may contradict programmer assumptions and lead to impossible scenarios. For example, a program executes the open system call to read some file f. Would it make sense to inject a fault on the open system call? Both yes and no seem to be sensible answers:

- Yes: A correct program should always check and handle all exceptional system call errors.
- No: This program may hold an invariant that the file should always exist. We have broken program invariants by injecting a fault so the program is still correct under its runtime assumptions.

While my personal opinion is that programs should handle even seemingly impossible scenarios, we will consult the relevant literature to see how this should be handled.

5.1.4. Evaluation

We will execute Chaos on a wide array of C programs. We are particularly interested in targeting systems which are expected to have high reliability and robustness to exceptional circumstances, such as databases and userspace filesystems. We hope to discover and validate many real bugs with this approach.

5.2. Better System Call Interposition Mechanisms

In this section I argue the need for better system call interposition (interception) mechanisms.

5.2.1. Current Approaches

Here we compare three popular approaches to Linux system call interposition: LD_PRELOAD, ptrace, and in-kernel.

LD_PRELOAD

LD_PRELOAD is an Linux environment variable specifying ELF shared objects to be loaded before all others [26]. This can be used to override functions which are dynamically linked into a program. For example, we want to trace every use of the open standard C library function in some program ./foo. The open function is dynamically loaded into ./foo via the shared object libc.so. We can create our own shared object, e.g. tracing.so which defines an open function with the same signature as the standard library version. Our new open function can execute arbitrary code to book-keep the number of function invocations, before executing the actual open system call.

Our tracing code can be injected by executing: LD_PRELOAD=tracing.so ./foo. In this section, we will use the term LD_PRELOAD as a shorthand for the LD_PRELOAD injection approach.

There are several advantages to LD_PRELOAD that make it widely used (even though it suffers from fatal drawbacks):

- **Ease of use**: it is straightforward to get something basic working quickly.
- **Familiarity**: The user is more likely to be familiar with C standard library functions and the Linux programming interface. This provides a familiar programming model for users.
- **Source/Binary Requirements**: LD_PRELOAD does not require any source code changes, recompilation, and will work on unmodified code. Only a dynamically linked binary is required.
- **Performance**: LD_PRELOAD adds no additional overhead beyond any custom code injected by the user. Our custom functions work just like any other dynamically loaded function invocation. This code is executed directly in the process address space and requires no additional context switches or system calls.

I believe the LD_PRELOAD approach should be avoided in every but the most trivial

use-cases as it suffers from several issues:

- **Mismatch between C standard library and systems calls**: C standard library functions like open, printf, fork are casually referred to as system calls. This is inaccurate and can lead to issues in situations with LD_PRELOAD. These functions should be thought of more as high-level[6] wrappers around actual OS system calls. For example, the printf function eventually calls the write system call.

  The C standard library often provides various convenience functions which all eventually call the same system call. We must ensure we are intercepting all such functions.
- **Statically Linked Binaries**: LD_PRELOAD only works for functions which are dynamically linked. If we are dealing with a statically linked binary, this approach will not work at all.
- **Use of C Standard Library**: LD_PRELOAD hinges on the erroneous assumption that languages and runtimes are implemented on top of the C standard library and libc.so. Go programs cannot be traced using LD_PRELOAD since the Go runtime calls system calls directly instead of using libc.so. Additionally, any code that calls system calls directly, e.g. via assembly, will not be traced by LD_PRELOAD.

These issues mean LD_PRELOAD is not a full-proof approach and its tracing may be missing system calls.

Ptrace

Please see Section 1.4 for an overview of the Linux ptrace API. ptrace provides the following advantages:

- **OS-Based Interception**: The OS handles interception of system calls and other IO events. This provides a full-proof approach which cannot be circumvented by programs.
- System Call Interface: Ptrace works at the system call level, the lowest layer for userspace programs. So ptrace works regardless of the language, technology, or runtime

[6] "High-level" being a relative term in this context.

of the program[7].

- **System call filtering**: Ptrace allows us to filter only those system calls we are interested in. The overhead of ptrace is proportional to the number of IO events intercepted [82]. Filtering allows us to specify the minimal set of system calls required for our use case.
- **Beyond System Calls**: Ptrace notifies us of IO events beyond just system calls. These events include: signals, process/thread spawn and exit, and calls to execve. These additional events are useful for program tracing.
- **Process Manipulation**: ptrace provides powerful process manipulation mechanisms. Ptrace allows for arbitrary reads and writes to the memory of the tracee. System calls may be emulated, substituted, injected, or modified by the tracer.
- **Dynamic Attachment and Detachment**: Ptrace allows the tracer to attach to arbitrary processes at any time. This is useful for tracing already running processes. Tracers may attach or detach dynamically to processes. This is in contrast to LD_PRELOAD which requires attaching at program execution time.
- **Centralized Tracing**: A single tracer may trace an arbitrary number of threads and processes. This is useful when tracing multi-threaded or multi-process programs. We can have a single centralized tracer which aggregates information from multiple tracees.

Ptrace has several issues:

- **Complicated API and Setup**: Ptrace's API and setup are difficult. This is most likely due to the complexity of the system call, and the archaic feel of many parts of Linux. There is also a general lack of documentation: The `man` page is not beginner friendly. All ptrace examples available on the web are simple and only handle the most trivial tracing needs.

  Ptrace must be used in conjunction with seccomp [27] to support system call filtering.

[7] One notable exception is VDSO system calls, in practice this is not a major issue.

This adds to the complexity. Even a minimal working ptrace + seccomp program requires a few hundred lines of boilerplate.

- **Low-Level Interface**: Ptrace requires knowledge of low-level program execution, specifically: familiarity with the Linux programming interface, systems programming, and the Linux system call ABI.

  To trace system calls events, we must first perform a PTRACE_GETREGS operation. This operation populates a struct representing the register state of the tracee. This struct consists of untyped values represented as 64-bit integers. Interpreting this struct requires knowledge of that specific system call's API and the Linux system call ABI [8]. The register values then need to be cast into the correct data type or pointer type. These pointers refer to the tracee's address space, but can be dereferenced via PTRACE_PEEK or the process_vm_read system call. Thus, this interface fails to abstract over implementation details, adding cognitive overhead to users.

- **Difficult Programming Model**: Ptrace (ab)uses the parent-child process relationship. Under the hood, a tracer becomes the parent of a tracee process. This enables the use of waitpid to receive notification of incoming ptrace IO events.

  Multiple tracees (additional threads or processes traced by our tracer) could be running in parallel. This is beneficial to amortize the overhead of ptrace, as we can serve events from one tracee while waiting for events from another. Therefore, ptrace IO event can arrive from any tracee at any given time.

  Often, it is useful to handle systems call events in a sequential manner. Consider the following example: We want to keep track of all file deletion events by a tracee, and we want to know the inodes of deleted files. To implement this with ptrace we would execute the following operations:

  1. Intercept call to the unlink system call prehook.
  2. Read the first argument to unlink as a string. This is the path of the file to be deleted.

[8] Notice the function signature between a C standard library wrapper and the system call API often differs in subtle ways.

3. Before allowing the unlink through, we inject a stat system call into the tracee to get the inode for this file [9].
4. Await for the posthook of stat.
5. Allow the unlink to execute.
6. Await the posthook for unlink to observe the return value.

Logically, these operations happen sequentially. We would like our code to reflect this. However, steps 3 and 6 require awaiting for posthook events. We could have arbitrarily many ptrace IO events from other tracees before the posthook events arrive. Ptrace does not allow us to wait for this specific event, we must serve events in their arrival order [10].

This is a real example from the Dettrace project. Dettrace handles this problem by having a per-tracee local state. On every IO event, we pop the corresponding state for that thread or process. Any time we areawaiting a specific IO event, we save the state for that tracee; In anticipation for receiving a IO event from a different tracee. While this approach worked, it was error prone as it was up to the user to ensure they pushed/popped the state. It also broke the sequential logical workflow of this operation. Futhermore, we could not factor out this code as a function, as we needed to return to the main IO event handler loop at any time.

Lastly, ptrace does not keep track of pre-hook and post-hook events per process or threads. It is up to the tracer (us) to manually remember the state of the tracee. The tracer may become desynchronized on the current ptrace event. This happens when the programmer has different expectations of what the next event should be, most commonly pre-hook and post-hook events. This leads to hard to diagnose, and unhelpful ptrace IO errors [11].

- **High Overhead**: Ptrace works on a tracer/tracee model. The tracer runs as a sepa-

[9] We must do this before the unlink, if the file is deleted, it will be too late for us to read the inode!

[10] We could instruct waitpid to wait on the PID/TID of this specific tracee, but then we cannot serve IO events which are ready from other tracees.

[11] The OS knows the current ptrace IO event we are handling, it is not clear why it does include pre-hook or post-hook information

rate process which awaits new notifications from the OS. Every time a relevant tracing events is executed, the tracee is put in a stopped state. Communicating between two processes in this manner incurs considerable overhead. Ptrace's overhead is directly proportional to the number of IO events intercepted [82]. IO-bound program may execute several times slower while being traced. CPU-bound workloads experience far better performance overheads. Based on our experiments, between 1%-2% on average.

- **Expensive IO**: Beyond the cost of stopping the tracee and context switching to the tracer, most ptrace operations require a system call for reading the state or memory of the tracee. Since the tracer operates on a different address space as the tracer, dereferencing any pointer requires further ptrace system calls. This additional IO likely accounts for a significant portion of ptrace's overhead.

Kernel

Previous work [47, 67] opts to trace in-kernel. These systems are making OS modifications beyond just tracing, so the kernel is a natural place for tracing system call events. This approach is quite fast, we have to enter kernel-space to service the system call anyways, so there is no additional overhead beyond any custom code that must be executed.

The kernel-modification approach comes with significant drawbacks:

- Kernel modifications are an intrusive change to any OS. For security and stability reasons, it is unlikely a modified kernel will be used in production systems. The majority of proposals are unlikely to be merged into the mainline Linux kernel. If a project relies on the low-overhead of an in-kernel implementation, their approach may never be useful in production. Rendering the viability of their research questionable.
- This approach relies on the internal kernel code instead of going through a public API. This approach usually pins the code to a specific kernel version. The code quickly becomes outdated as new kernel versions make changes to these internals.
- Kernel programming is a highly-specialized skill event among systems programmers and operating systems researchers. Most people, including myself, lack the deep do-

main knowledge for this approach.

5.2.2. Designing Better Systems

Given the issues with existing approaches, I believe there is a need for better mechanisms to fill this niche. In this section I outline some features a better system call interposition mechanism would have, as well as current work in this area.

Next Generation Tracing

In this section I outline the requirements and features a next generation solution should implement.

- **Better Abstractions**: I argue that system calls are actually the wrong level of abstraction for tracing mechanisms. System calls are an implementation detail of executing programs. I believe most users are interested in tracing higher level IO operations of programs. For example, instead of knowing the tracee executed a open system call with a certain set of flags, the user would rather work with a higher level concepts such as a FileOpened operation. There are many system calls which perform similar operations depending on the arguments, e.g. open with the O_CREAT, opentat, opentat2, creat, all perform similar function. Most users care less about the exact system call and more about the IO event, e.g. FileCreated. ProcessCache (See section 3) implements these concepts in-code to abstract over raw system call events.
- **Portability**: An ideal solution would be portable across different operating systems. This is probably the hardest feature to have. Different operating systems have different system calls making this infeasible. Using an intermediate, higher abstracted representation as described in the former bullet point would make this feasible.
- **Low-Overhead**: Overhead is an important factor for the feasibility of techniques built on top of system call interposition. A full-proof and low-overhead tracing would be be the best of all worlds. I believe such a system would require a userspace and in-process approach. We already see this trend on other areas of OS evolution [102], and I believe tracing should be done in userspace without the cost of the OS.

- **Programming Model**: As described in the Section 5.2.1, the current ptrace programming model is difficult and error prone. A more intuitive programming model could help cut through the complexity. Asynchronous programming has been popularized in recent years. I believe a asynchronous IO API is an excellent programming model for IO event tracing. See Section 5.2.2 for a detailed description.

Syscall User Dispatch

Progress is being made in this area. Linux now supports Syscall User Dispatch (SUD) [20]. Syscall User Dispatch is designed mainly to handle the use case of Wine and Proton: emulating system calls for a certain region of the process' address space. SUD allows filtering, trapping, and emulating system calls in userspace. SUD utilities a "selector" byte mapped to the application's memory. This allows for quick enable/disable of system call redirection via a write to memory. Notice this write does not require a mode switch to kernel space. When the selector is enabled, any system call performed in the application will generate a SIGSYS signal, which can be caught and handled in-process.

Additionally, SUD alows us to specify memory regions where any system call executed will not be intercepted, regardless of the state of the selector. This is useful for allowing fast system call execution, e.g. mapping libc into this memory region.

I believe the SUD mechanism could be used to engineer a faster system call interception mechanism than those currently available. SUD would not suffer from the shortcomings of other approaches. I expected the overhead to be low, as it works in-process. According to the documentation [20], handling signals on most architectures incurs a high cost but I expect this approach to be faster than other full proof solutions, i.e. ptrace. A SUD-based approach would work for all systems unlike the LD_PRELOAD approach. Reading and writing to the tracee's address space would be as fast as a pointer derefernce since the tracer shares the same address space as the tracee.

Note, this approach would not be suitable for sandboxing or hardening environments, as the

tracee runs in the same address space as the tracee.

Higher-Level Tracer On Top of Ptrace

We don't need a new in-kernel implementation to gain many of the benefits outlined above. It is possible to build a better higher-level interface on top of the ptrace. This is akin to the libseccomp library which makes it palpable to work with seccomp [8].

We implement some of these ideas for the ProcessCache project. ProcessCache implements a higher-level interface around ptrace, we will refer to this wrapper library as libptrace. libptrace is entirely written in Rust providing high-level methods for all ptrace operations.

For example, reading a string from the tracee is as simple as calling fn read_c_string(&self, address: *const c_char) -> Result<String>, where address is the pointer address in the tracee. Futhermore, we can read arbitrary values from the tracee via read_value<T>(&self, address: *const T) -> Result<T>. The type parameter T, and the sizeof(T) is automatically inferred by the compiler based on the context, neat!

libptrace abstracts over the underlying ptrace IO events by providing a higher-level enumeration. See 5.1 for the exhaustive list of events.

libptrace features a custom asynchronous runtime for handling ptrace IO events. This abstracts over the problem previously outlined of ptrace IO events arriving in any order, at any time. libptrace allows us to write code in a sequential matter while still allowing the runtime to handle events as soon as they are ready. See Figure 5.2.

The libptrace's implementation extends this concept beyond just handling a single system call. Every process traced by ProcessCache is handled by an asynchronous function which persists for the lifetime of that process. These asynchronous functions are paused at user-defined yield points via .await. The runtime is responsible for scheduling the next asynchronous function to run based on which ptrace IO event is received. This allows the programmer to write code to handle a single process/thread, the asynchrony is handled by the runtime. Based on our experience with ProcessCache this model works quite well.

```
pub enum TraceEvent {
    Exec(Pid),
    /// This is a stop before the actual program exit, this
    /// is our last chance to ptrace queries on the tracee.
    After this event, we expect a real program exit.
    PreExit(Pid),
    Prehook(Pid),
    /// This is the parent's PID, not the child.
    Fork(Pid),
    Clone(Pid),
    VFork(Pid),
    Posthook(Pid),
    ProcessExited(Pid, i32),
    ReceivedSignal(Pid, Signal),
    KilledBySignal(Pid, Signal),
}
```

Figure 5.1: libptrace TraceEvent enumeration. libptrace abstracts over the underlying ptrace IO events.

```
async fn handle_unlink(execution: _, tracer: _) {
    let regs = tracer.get_registers();
    let full_path = get_full_path(execution, name, tracer)?;

    let regs = tracer.posthook().await;
    // Get return value of system call.
    let ret_val = regs.retval::<i32>();
    ...
}
```

Figure 5.2: libptrace allows us to write code in a sequential style. The tracer object is a handle to various high-level tracing methods. The Execution object holds the context of the current process being executed. The .await call may yield the execution of this function until the posthook event arrives. Arbitrarily many IO events may arrive and be handled by the libptrace runtime for other processes or threads in the meantime. When the specific posthook we are waiting for arrives, the runtime will schedule our async function and continue executing the code from where we left off.

CHAPTER 6

Conclusion

This dissertation overviews techniques for low-level process manipulation. We argue process manipulation is a useful and generalizable technique with many applications in software systems. Our introduction gives an overview of program tracing and low-level process manipulation. Then, we summarize system call interposition and the Linux ptrace API. We show how the low-level primitive operations provided by ptrace can be used to build useful, reusable program manipulation constructs such as arbitrary system call injection, system call replay, and system call modification.

Afterwards, we see an application of low-level process manipulation towards solving an existing issue in software systems: software reproducibility. We describe 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. DetTrace shows the utility of low-level process manipulation by implementing a dynamic determinism enforcement mechanism on top of it. Process manipulation is orthogonal to other OS-level techniques. So it can be readily combined with other OS-level facilities, e.g. containerization. Dettrace further implements a userspace, reproducible, thread and process scheduler via ptrace.

DetTrace has several avenues for future work. DetTrace guarantees reproducible execution of threads by sequentializing thread execution within a process. This method does not always work, for example the Java virtual machine deadlocks during garbage collection under DetTrace. Furthermore, sequentializing thread execution incurrs high performance overhead due to the loss of parallel execution of threads. This motivates two avenues for future work: First, Dettrace could relax constraints such as thread sequentializing via command line options when the user knows a priori that thread execution will not cause issues. This could

be generalized to other areas of DetTrace to allow users to pick and choose which parts of DetTrace's determinism enforcement their application requires. This would minimize the performace cost incurred by our dynamic determinism enforcement while simultaneously allowing more programs to execute under DetTrace. Second, DetTrace could look into generalizing its methods to support parallel and deterministic thread execution. DetTrace may be combined with relevant work in this area.

DetTrace motivated many ideas and future work directions, among the the ProcessCache project. ProcessCache attempts to automatically cache and skip unnecessary process-level computations. ProcessCache traces the execution of programs to determine the program's input and outputs, caching the outputs. When this program is re-executed, ProcessCache will skip any processes whose inputs have not changed and use the cached output instead. ProcessCache represents the most advance use of the process manipulation techniques we have developed. Its implementation is informed by experience working on DetTrace. ProcessCache in turn has informed the future work section, specially Section 5.2.2.

There are several avenues for future work as a follow up to ProcessCache:

- ProcessCache could be generalized to support multiple users within the same machine, this would allow the cache to be shared and used by multiple users.
- An even larger leap would be to make ProcessCache fully distributed allowing cached results to be shared across machines and users, a la Google's distributed builds infrastructure [5].
- ProcessCache currently hashes file contents to detect if file contents have changed. This incurs a non-trivial performance penalty. ProcessCache can fall back to using mtimes on files for faster file change detection. mtimes do not always reliable and are not portable across machines. A better way to detect file changes would be to rely on file change mechanisms like inotify. ProcessCache could be modified to have a daemon responsible for keeping track of file changes.

Furthermore, ProcessCache is quite the engineering effort and requires further work to be turned into a useful production system. I suspect interesting and novel research questions would arise during this implementation effort.

Tracing is useful at many levels of the software stack. Working strictly with OS-level tracing and process manipulation can create difficulties. We find this is not always the right level of abstraction. This motivated Tivo and lightweight record-and-replay (RR). Lightweight RR is a hybrid approach between heavyweight, fully deterministic systems, and no determinism enforcement. Lightweight RR is designed to empirically lower the amount of intermittent test failures in a program's test suite, while enjoying minimal performance overhead and record-log sizes. Lightweight RR has exciting and promising future work avenues beyond the just our current use in Tivo. With more work, I believe lightweight RR could prove to be a useful tool for the development and debugging of concurrent systems and asynchronous programming models.

Finally, the future work chapter sketches the design for a novel system for automatic fault discovery. ChaOS utilizes low-level process manipulation techniques developed in our other work to implement a fault injection tool for system calls. ChaOS proposes using many of the process manipulation techniques developed during the course of my PhD. Including system call modification and system call injection. Specifically, ChaOS injects fork/clone system calls to make copies of the tracee. This allows us to effectively save the state of the currently executing tracee.

Our future work also overviews different tracing and process manipulation implementations commonly used. We compare advantages and shortcomings of each approach. Then, we propose key features next generation tracing and process manipulation implementation should support. It is unclear if any implementation could satisfy all the key features at the same time. As a stop gap, I propose a higher-level library, libptrace. libptrace is partially inspired by libseccomp a higher-level library build on top of Linux seccomp. libseccomp abstracts over many idiosyncrasies and low-level details of seccomp, creating a more accessible and useful

library. The design and implementation of libptrace has similar goals: providing a high-level easy to use tracing and process manipulation library built on top of ptrace. libptrace provides a asynchronous IO abstaction allowing code to be written in a sequential manner. It is my hope higher-level and abstracted tracing and process manipulation mechanism will bring accessibility and allow more people to use these powerful techniques.

BIBLIOGRAPHY

[1] american fuzzy lop. https://lcamtuf.coredump.cx/afl/.

[2] Bazel. https://bazel.build/.

[3] Bazel: a fast, scalable, multi-language and extensible build system. https://bazel.build/.

[4] Code Ocean homepage. https://codeocean.com.

[5] Distributed builds. https://bazel.build/basics/distributed-builds.

[6] Gnu make. https://www.gnu.org/software/make/.

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

[8] libseccomp main repository. https://github.com/seccomp/libseccomp.

[9] Pachyderm reproducible data science homepage. https://www.pachyderm.io.

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

[11] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

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

[13] Proton: Compatibility tool for Steam Play based on Wine and additional components . https://github.com/ValveSoftware/Proton.

[14] Rattle: Forward build system with speculation and caching. https://github.com/ndmitchell/rattle.

[15] ReproducibleBuilds. https://wiki.debian.org/ReproducibleBuilds.

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

[17] Salsa: A generic framework for on-demand, incrementalized computation. https://github.com/salsa-rs/salsa.

[18] Servo, the parallel browser engine project. https://servo.org/.

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

[20] *Sycall User Dispatch. The Linux kernel user's and adminitrator's guide.*

[21] web-platform-tests documentation. https://web-platform-tests.org/.

[22] Wine: Wine Is Not an Emulator . https://www.winehq.org/about.

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

[24] 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.

[25] 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.

[26] *ld.so(8) Linux Programmers's Manual*, August 2021.

[27] *seccomp(2) Linux Programmers's Manual*, August 2021.

[28] *syscalls(2) Linux Programmers's Manual*, August 2021.

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

[30] 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.

[31] 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.

[32] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. A fork() in the road. In *17th Workshop on Hot Topics in Operating Systems*. ACM, May 2019.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. *SIGPLAN Not.*, 45(3):167–178, mar 2010.

[38] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. *SIGOPS Oper. Syst. Rev.*, 25(5):152–164, September 1991.

[39] 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.

[40] Buddhika Chamith, Bo Joel Svensson, Luke Dalessandro, and Ryan R. Newton. Instruction punning: Lightweight instrumentation for x86-64. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 320–332, New York, NY, USA, 2017. Association for Computing Machinery.

[41] Victor Stinner Charles-François Natali. Pep 475: https://www.python.org/dev/peps/pep-0475/.

[42] 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.

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

[44] Jonathan Corbet. Ringing in a new asynchronous i/o api. Linux Weekly News, January 2019.

[45] Charlie Curtsinger and Emery D. Berger. COZ: Finding code that counts with causal profiling. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

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

[47] 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.

[48] David Devecsery, MIchael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. Eidetic systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 525–540, Broomfield, CO, October 2014. USENIX Association.

[49] 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.

[50] 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.

[51] Xianzheng Dou, Peter M. Chen, and Jason Flinn. Shortcut: Accelerating mostly-deterministic code regions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 570–585, New York, NY, USA, 2019. Association for Computing Machinery.

[52] 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.

[53] 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.

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

[55] Jakob Engblom. A review of reverse debugging. In *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, pages 1–6, 2012.

[56] 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.

[57] Zhenyu Guo, Xi Wang, Jian Tang, Xuezheng Liu, Zhilei Xu, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, page 193–208, USA, 2008. USENIX Association.

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

[59] Matthew A. Hammer, Jana Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael Hicks, and David Van Horn. Incremental computation with names. *SIGPLAN Not.*, 50(10):748–766, oct 2015.

[60] 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.

[61] 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.

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

[63] 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.

[64] Zu-Ming Jiang, Jia-Ju Bai, Kangjie Lu, and Shi-Min Hu. Fuzzing error handling code using Context-Sensitive software fault injection. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2595–2612. USENIX Association, August 2020.

[65] 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.

[66] 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.

[67] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 353–367, New York, NY, USA, 2011. Association for Computing Machinery.

[68] 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.

[69] 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.

[70] 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.

[71] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient deterministic multithreading. In *Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM.

[72] 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.

[73] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 190–200, New York, NY, USA, 2005. Association for Computing Machinery.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] 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.

[80] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

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

[82] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. Reproducible containers. ASPLOS '20, page 167–182, New York, NY, USA, 2020. Association for Computing Machinery.

[83] 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.

[84] 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.

[85] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability: Extended technical report,

2017.

[86] 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.

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

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

[89] 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.

[90] 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.

[91] 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.

[92] Yasushi Saito. Jockey: A user-space library for record-replay debugging. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging*, AADEBUG'05, page 69–76, New York, NY, USA, 2005. Association for Computing Machinery.

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

[94] 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), October 2017.

[95] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Build scripts with perfect dependencies. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[96] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Build scripts with perfect dependencies. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020.

[97] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Forward build systems, formally. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Pro-*

grams and Proofs, CPP 2022, page 130–142, New York, NY, USA, 2022. Association for Computing Machinery.

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

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

[100] 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.

[101] Min Xu, Vyacheslav Malyugin, Jeffrey Sheldon, Ganesh Venkitachalam, Boris Weissman, and Vmware Inc. Retrace: Collecting execution trace with virtual machine deterministic replay. In *In Proceedings of the 3rd Annual Workshop on Modeling, Benchmarking and Simulation, MoBS*, 2007.

[102] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S. Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, Pedro Henrique Penna, Max Demoulin, Piali Choudhury, and Anirudh Badam. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 195–211, New York, NY, USA, 2021. Association for Computing Machinery.