REPRODUCIBILITY AND PERFORMANCE OPTIMIZATIONS FOR UNMODIFIED LINUX PROGRAMS

Kelly R. Shiptoski

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

2023

Supervisor of Dissertation

Joseph Devietti, Associate Professor of Computer and Information Science

Graduate Group Chairperson

Mayur Naik, Professor of Computer and Information Science

Dissertation Committee

Boon Thau Loo, Assistant Professor of Computer and Information Science
Benjamin C. Lee, Profession of Computer and Information Science, Professor of Electrical and Systems Engineering
Vincent Liu, Assistant Professor of Computer and Information Science
Ryan Newton, Meta

REPRODUCIBILITY AND PERFORMANCE OPTIMIZATIONS FOR UNMODIFIED LINUX PROGRAMS

This doctorate is dedicated to my mother, Carmela, my sister, Maria, and my partner, Alex. I would never have made it here without your unwavering and unconditional love and support, through all the highs and lows. Thank you.

ACKNOWLEDGEMENT

I would like to acknowledge all the wonderful people I have been fortunate enough to have in my life during my graduate studies and without whom this dissertation would not have been possible.

I would like to thank the other graduate students in my program, especially my cohort and lab. Together, we grew together, we learned together, we shared triumphs and defeats, and ultimately, we became *researchers*. My love especially goes out to Charles Kazer, Pardis Pashakhanloo, Alyssa Hwang, Bhavana Mehta, and Caleb Stanford (the best TA Penn will ever see). We may have taken different paths, but I am so glad to have had the opportunity to get to know and work with each of you.

I would like to thank Pranoti Dhamal, for being the most incredible mentee I could ever have the pleasure of working with. Thank you for signing up to work intensely on a very established but immensely confusing project. Your leap of faith gave us the push we needed to finish ProcessCache. We might still be bickering about read-write semantics if it were not for you.

I would like to thank Leonid Ryzhyk, and Mihai Budiu for the opportunity to collaborate together at the VMware Research Group. This experience boosted my confidence and forced me outside of my comfort zone, and I am incredibly grateful.

I would like to thank Dr. Brian Stuart, who was my operating systems professor at Drexel University during my undergraduate studies. You were the first person to make systems programming *fun*, and you sparked a passion for low-level programming and a thirst for knowledge that ultimately led me to pursue a Ph.D. Thank you for being an inspiration and an excellent educator and mentor.

I would like to thank my advisor, Dr. Joseph Devietti. I know my research journey was anything but smooth, but no matter what happened (including a pandemic) we found a way to navigate the darkness. I am so proud of what we have built together. Thank you for having my back, and not using only academic metrics to evaluate my growth as a researcher. Thank you for believing in me when no one else did, especially myself. And a special thanks to the chair of my committee, Dr.

Boon Thau Loo, who was incredibly supportive and provided guidance and wisdom at a time when I really needed it.

I would like to thank Nikki Wolford, my best friend since sixth grade, who herself earned her Ph.D. in Chemistry in 2021. We may be far apart, and we may be doing totally different things, but we have always supported each other's pursuits. I think we did alright for a couple of kids from Berwick.

I would like to thank my best friend and my closest collaborator, Omar Navarro Leija. I would endure this entire painful yet rewarding experience again just to meet you. Our friendship has changed my life, and I am so glad that even post-Ph.D., we are closer than ever.

Thank you to my family, especially my mother, Carmela Shiptoski, and my sister, Dr. Maria Shiptoski. You always supported me even though I chose a path unfamiliar to our family. Mama, throughout my years in graduate school, during every call you would ask me to describe my work in detail, using proper engineering terminology, even though you do not come from a computer science background at all. Thank you for always taking so much interest in my passions and for supporting me through it all. Sissy, you make every achievement, no matter how small, feel like I just changed the world. You are the very best big sister.

Finally, to my best friend and loving partner, Alex: thank you for joining me on this wild ride. I am sorry you had to witness the ups and downs in person, but thank you for sticking with me through the good and the bad. This would never have been possible without your unconditional support. I love you.

ABSTRACT

REPRODUCIBILITY AND PERFORMANCE OPTIMIZATIONS FOR UNMODIFIED LINUX PROGRAMS

Kelly R. Shiptoski

Joseph Devietti

The demands of modern computing continue to escalate each year. Consumers expect increased performance from systems which must also operate perfectly and never experience issues. These goals are difficult to achieve for any single system, let alone systems in general.

System calls are the means by which applications interact with the OS (operating system). Program tracing that takes place at the system call level is an incredibly powerful tool, allowing us to abstract over many details of program execution and reduce programs to the system calls they perform. It allows us to work between the kernel and application levels, which is an easier level of abstraction to reason about, but still general enough that we can look at any program as a series of system calls.

In this dissertation, we take a step beyond read-only tracing, and delve into program manipulation via system call interposition, its many applications, and how it allows us to produce *program agnostic* systems. We fully analyze ptrace, a built-in Linux tool for program tracing and manipulation, and describe its strengths and shortcomings. We also explain how we developed an asynchronous wrapper around the ptrace API which allowed us to circumvent both programmability and performance issues inherent to ptrace. We demonstrate ptrace's utility as a key component of two systems we created: DetTrace and ProcessCache.

DetTrace is a reproducible container abstraction for Linux implemented entirely in userspace. All computation that occurs inside a DetTrace container is a pure function of the initial file system state of the container. Reproducible containers can be used for a variety of purposes, including replication for fault-tolerance, reproducible software builds, and reproducible data analytics. We use

DetTrace to achieve, in an automatic fashion, reproducibility for 12,130 Debian package builds, containing over 800 million lines of code, as well as bioinformatics and machine learning workflows. We show that, while software in each of these domains is initially irreproducible, DetTrace brings reproducibility without requiring any hardware, OS, or application changes. DetTrace 's performance is dictated by the frequency of system calls: I/O-intensive software builds have an average overhead of 3.49×, while compute-bound bioinformatics workflows are under 2%.

The ProcessCache system provides a generic facility for automatically memoizing the work of a broad class of multi-process Linux programs. ProcessCache caches results and transparently determines when cached results can be used and when re-execution is necessary. ProcessCache generalizes previous work on forward build systems, to go beyond software builds to other multi-process programs like shell scripts and bioinformatics workflows. ProcessCache supports unmodified Linux binaries, using the ptrace mechanism to trace system calls and determine program inputs. Our experiments show that ProcessCache can automatically provide incremental computation to existing programs, accelerating workloads from 1.06× to 65×.

We conclude with an in-depth analysis of future work directions for the two systems. We focus this section on ProcessCache because it comprises the bulk of this dissertation, but first we propose an addition to DetTrace that could alleviate performance and correctness issues it suffers when handling threads. For ProcessCache, we examine potential avenues to improve its performance, space utilization, and correctness guarantees, and also discuss why some previously proposed improvements are not viable solutions.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF ILLUSTRATIONS

CHAPTER 1

INTRODUCTION

1.1. Background and Motivation

The demands of modern computing continue to escalate each year. Consumers expect increased performance from systems which must also operate perfectly and never experience issues. These goals are difficult to achieve for any single system, let alone systems in general. It would be spectacular for programmers to have practical tools to use that provide some form of guarantees for arbitrary programs, such as improved correctness or better performance. Multiple obstacles exist on the path to building such tools. How can we create systems that not only work for most programs, but also improve most programs in a considerable way?

Let us examine the two goals we outlined for the systems we build: to increase correctness and to improve performance. There are many properties that contribute to the correctness of a program. For the purposes of this dissertation, we focus on a single correctness property: *determinism*. To be as pedantic as possible, the specific form of determinism we concentrate on is *dataflow determinism*. In the simplest terms, dataflow determinism guarantees that for a given program, the same set of inputs should produce the same set of outputs. Whenever we mention determinism in this work, the specific type of determinism we are referring to is dataflow determinism.

Achieving this feature in a program is harder to accomplish than many programmers realize, especially given the recent trends of scaling massive parallel programs and distributed systems. A typical approach to attaining correctness for programs is to spend engineering effort focusing on one program, rooting out the causes of nondeterminism. While this approach works to a degree, it almost always requires many development cycles to solve only one problem, and even then, the problem may not be truly fixed.

One real-world example of this is the Debian Reproducible Builds project [1], the objective of which is to render all Debian packages reproducible, *i.e.* deterministic across machines. Their

approach has been to proceed package by package, finding the individual sources of nondeterminism in each package and resolving them. This has taken years to accomplish, and the project is still ongoing. Worse, even if a package is made reproducible, there is no way of knowing if the change has compromised the reproducibility of other packages, even packages that have already been rendered reproducible.

Our other mentioned goal was better performance for systems. There are countless systems in existence that seek to improve the runtimes of programs. We see two main issues with these systems. The first issue is that they tend to be overfitted to work for a certain class of programs. One example of this is Ccache [41]. Ccache seeks to speed up compilation time, but it only works for C/C++ programs. The second issue is that for existing systems, if they *are* actually general, they are usually difficult to use. A prime example of this is Google's Bazel build tool [29]. It is a fairly general build tool, but it can be painful to work with as it leads many programmers into dependency hell.

Bazel is a build tool that works for all kinds of programming languages, provided the user specifies their own dependencies, and does so *correctly*. If the user over-specifies their dependencies, compilation time will suffer because unnecessary files are being included in the build. Worse, under-specifying dependencies can lead to erreneous builds, and the all too common need to clear the state and recompile the entire build when the build appears to not be properly updating. Finally, Bazel is a build system, and nothing more. While it is robust to many programming languages, Bazel is only capable of boosting the performance of builds, not any arbitrary program written in those languages.

Clearly, existing systems fall short of our ambitions. Now we must ask, how can we create *program agnostic* tools that benefit arbitrary systems without falling into the usual traps that existing systems do? One option is to drop to the kernel level to reason about and manipulate programs, in the form of a custom kernel module. This has the advantage of allowing us to reason about all types of programs on an even playing field, but it certainly has its drawbacks. For one, this level of abstraction is not natural for most programmers; kernel module programming is challenging for

even the most experienced systems programmers. In addition, adding a kernel module can introduce unintended consequences in terms of correctness or performance for other aspects of the kernel.

Requiring a custom kernel module also goes against our goal of generality. A user will have to pull in this kernel module to utilize our system, which, in addition to being an annoyance, locks the user to the exact version of the OS they are using. With each new kernel release, if our user wants to continue using our system, they must update the kernel module. This means it also becomes our responsibility to update our kernel module with every kernel release. Kernel modules are notoriously difficult to maintain; with each new release cycle the module may need to be updated if changes in the kernel could break the existing module. It would be better if we could reason about programs and implement a system for them at a higher level than the kernel, for ease-of-use and backwards compatibility, but at lower level than the individual programming language, to guarantee program agnosticism.

The level sandwiched between the kernel and applications is called userspace. The mode of interaction between the two is system calls. Whenever an application needs to communicate with the kernel, such as when it wants to access the file system, it issues a system call. Program tracing that takes place at the system call level is an incredibly powerful tool, allowing us to abstract over many details of program execution and reduce programs to the system calls they perform. It allows us to work between the kernel and application levels, which is an easier level of abstraction to reason about, but still general enough that we can look at any program as a series of system calls.

In this dissertation, we take a step beyond read-only tracing, and delve into program manipulation via system call interposition, its many applications, and how it allows us to produce *program agnostic* systems. We fully analyze ptrace, a built-in Linux tool for program tracing and manipulation, and describe its strengths and shortcomings. We also explain how we developed an asynchronous wrapper around the ptrace API which allowed us to circumvent both programmability and performance issues inherent to ptrace. We demonstrate ptrace's utility as a key component of two systems we created: DetTrace and ProcessCache.

1.2. Overview

This dissertation demonstrates the utilization of existing Linux mechanisms to build systems to automatically improve existing programs in many ways, from correctness to performance, *automatically*, *i.e.* without any effort required on the part of the user. A major goal of this work is to provide practical, easy-to-use tools that industry programmers can actually benefit from. We will examine two major projects in this dissertation: the first is a system which automatically provides determinism guarantees, and the second is a system that provides performance speedup through automatic, generic caching.

Chapter 2, presents DetTrace, a system that provides reproducibility for unmodified Linux programs. This system combines methodical dataflow determinism enforcement concepts with process manipulation via ptrace (an overview of which is provided in section 1.4 and section 1.5) to guarantee that all computation that occurs inside a DetTrace container is a pure function of the initial file system state of the container. Reproducible containers can be used for a variety of purposes, including but not limited to: fault-tolerance, reproducible software builds, and reproducible data analytics. The central component of the evaluation is our demonstration of DetTrace achieving automatic reproducibility for 12,130 Debian package builds, containing over 800 million lines of code.

Chapter 3 examines ProcessCache, a system that provides caching for arbitrary Linux processes. This system combines process introspection via ptrace (section 1.5) and asynchronous computing (subsection 1.5.2) to provide a generic caching system. This system is capable of caching almost any Linux program, from the simplest single process programs, to much more complicated parallel, multi-threaded, or incremental jobs. ProcessCache is always running in both caching and skipping mode, meaning that it can skip steps in an incremental build or skip some child process executions in a parallel program and cache others, if deemed appropriate. This chapter also demonstrates the usefulness of ProcessCache on a variety of workloads from a custom designed benchmark suite, and compares ProcessCache to related work that has come before it.

The next section describes my personal contributions to each project. The remaining sections of this chapter serve to familiarize the reader with important concepts that are used in one or both of the projects just introduced.

1.2.1. Contributions

Chapter 2 draws from joint work with Omar Navarro Leija, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti [94]. My major research contributions for this project include redesigning the sluggish, sequential scheduler to allow for parallel execution of processes in system-call-free regions, and implementing a parallel scheduler based on priority queues to ensure fairness and eliminate the risk of deadlock.

Chapter 3 is drawn largely from joint work with Omar Navarro Leija, Pranoti Dilip Dhamal, Ryan R. Newton, and Joseph Devietti. I was the primary doctoral student on this project, meaning that I drove the technical design and project management, and performed the bulk of the implementation. In order to succeed in this role, I taught myself Rust and asynchronous programming because I recognized the safety and performance improvements that Rust and its asynchronous computing framework could provide over C++.

My other responsibilities included organizing and leading all weekly meetings, conceptualizing all system components and coordinating their individual implementation timelines, creating and maintaining integration and end-to-end testing suites, and constructing a custom benchmark suite based on a deep-dive into a variety of real-world programs, which was then used to establish the breadth of programs that can benefit from using ProcessCache.

1.3. Program Tracing

As programs become increasingly complex, tools for examining programs as they run have emerged and become ubiquitous. Program tracing is the act of *tracing* a program to observe and report some metrics about it that may not be obvious from looking at the source code, tests, or outputs. Today, program tracing is used to provide a variety of developer tools to help programmers reason about their code. Program tracing is a form of dynamic program analysis, because it involves observing a

program *as it runs*. Many tools have been created for developers using program tracing including breakpoint debuggers, such as GDB, [6] and system call tracing systems, such as strace [14].

Program tracing can provide details at all levels of the software stack, but we focus on tracing at the system call level in this dissertation. Program tracing that takes place at the system call level is an incredibly powerful tool, allowing us to abstract over many details of program execution (the compiler, the programming language, the runtime environment, etc.), and reduce programs to the system calls they perform. There are many operations a program performs that can be of interest to trace, including but not limited to: file system accesses, networking, thread and process creation and exit, system calls, and signals. Being able to reduce any program to its system calls provides the ability to develop truly *program agnostic* systems, because we can effectively reason about the programs as black-boxes. We explicitly strive to avoid overfitting our system designs to a small set of programs, which would compromise the usability and practicality of the system.

Another strength of system call tracing is that we can choose to only intercept a subset of the operations we care about. For example, we can choose to only report metrics about accesses a program makes to a certain file. This can help improve performance, because program tracing does cause performance overhead (section 1.5). The Linux seccomp-bpf mechanism allow us to selectively intercept system calls (subsection 1.5.4).

1.4. Program Manipulation via System Call Interposition

Program tracing is very powerful, but it is simply a *read-only* operation. If our goal is to enforce certain correctness guarantees or improve performance, we need to be able to *manipulate* programs. Because this dissertation focuses on OS-level tracing, *i.e.* the form of program tracing that reports interactions the program has with the operating system, we begin with a discussion about system calls. System calls are the fundamental interface between the OS and userspace programs. Each time the program wants to interact with the operating system, it must execute a *system call*.

Even to simply report metrics about a program, we need to be able to intercept and examine all system calls a program makes during its execution. To achieve program manipulation, we need to

be able to change the arguments of system calls a program makes and the values of the program's registers. With system call *interposition*, we can stop the program at each relevant intercepted system call and either simply examine the system call or make changes to its arguments and/or return value. This seemingly simple mechanism provides the foundation to not only observe a program, but control it.

1.5. The ptrace API

Linux provides an API for tracing and manipulating the behavior of processes: ptrace. ptrace allows one process to trace various events of interest that may be happening in the execution of another process, such as system calls, signals, and process and thread creation and exit.

When working with ptrace, we use the following terms. There is always a singular single-threaded tracing process, called the *tracer*. The tracer can trace one or more threads or processes, called the *tracees*. Whenever a *tracee* executes an event of interest, the OS will stop the tracee's execution and inform the *tracer* of this event. There are two types of system call events, *pre-hook* events and *post-hook* events. A *pre-hook* event is when the process is stopped *before* the system call is executed. A *post-hook* event is when the process is stopped after the system call is finished but before it returns control to the program. In one of these stopped states, the tracer can perform various low-level operations on the tracee:

- Peek/poke tracee memory: the tracer may read or write arbitrary bytes to the tracee's memory.
- Signals: the tracer may inject arbitrary signals to the tracee.
- Registers: the tracer may read or write to the tracee's registers.

These operations are low-level and relatively *simple* on their own, but they are quite powerful primitives. We can implement high-level, useful operations with them, as explained in the next section.

Note that the ptrace API does not provide *any* arbitrary process with the ability to trace *any* other process. ptrace is designed to check various permissions before performing the requested ptrace

action to ensure the tracer process is indeed allowed to invoke ptrace with the given tracee process.

1.5.1. Powerful Operations from Simple Abstractions

With the tools of interception and manipulation of a tracee process's system calls, signals, memory, and registers, we can implement high-level, useful operations such as: injecting arbitrary system calls into the tracee, changing the arguments of a system call about to be performed, observing the results of a system call, and implementing user-level (albeit slow), scheduling for processes or threads. All of these and more are possible because ptrace's primitives can be combined to provide us with three fundamental and powerful high-level operations: system call injection, system call replaying, and system call modification.

- **System call modification.** The tracer can write to the registers and memory of a tracee during a pre-hook event. This means we can inspect the system call that is about to be executed (with the PTRACE_GETREGS flag) and then write to the registers (with the PTRACE_SETREGS flag) to modify any of the arguments to the system call. The system call itself can be changed to an entirely different system call by writing the appropriate system call number to the RAX register (though the exact register responsible for housing the current system call number may vary across Linux ABIs).

- **System call injection.** With the ability to change system call arguments and the system call itself, another powerful mechanism we can build with ptrace is system call injection. During a system call interception event, we can inspect the system call number and arguments to the system call, and then store this information. We can then change the system call number (by updating the RAX register, in most cases) and the arguments to change the current system call into a different one. But, we cannot just let the program run, because this would be replacing the current system call, not *injecting* a new one. We can modify the system call at the pre-hook, let the system call run, and then stop the process again at the system call's post-hook event. Then, we restore the original state of the registers for the original system call and "rewind" the execution by setting the RIP register to RIP $-$ 2. This is because all x86-64 instructions capable of calling a system call are 2 bytes long, and the RIP register

corresponds to the process's instruction pointer.

- **System call replaying.** A special case of system call injection is system call *replaying*. This mechanism involves stopping a process at the *post-hook* system call event and effectively "replaying" the system call. This is achieved by setting the RIP register to RIP − 2, as explained previously. Other arguments and registers can be edited between calls to this system call as a flexible and powerful replaying mechanism.

These mechanisms can be combined in many ways, such as: changing a system call from blocking to non-blocking, changing a process's file permissions, and manipulating the scheduling of multiple processes.

ptrace is implemented as a synchronous, low-level system call API. This means that if a tracer needs to trace multiple processes, the tracer needs to keep track of each process it is tracing. This tracer must be single-threaded, as that is one of the limitations of ptrace. The tracer is also responsible for restarting any traced process that is stopped by a ptrace event, every time one of these events occurs. A process is blocked from continuing its execution until it is restarted by the tracer. This can have performance and correctness implications, and demonstrates exactly how esoteric the ptrace API can be in practice.

A single-threaded tracer makes sense from an API design standpoint; it simplifies the API by assuming exactly one process will be examining the events of another process at any given time. An unfortunate side effect of this is exactly *one* process is in charge of receiving events, acting upon them, and then restarting the traced process after handling the event, for *every* process being traced. This is the main reason why ptrace is notorious for its overhead.

Scheduling of traced processes effectively becomes the duty of the tracer due to the sparsity of the ptrace API. Once a process is registered with the tracer, the tracer will issue a ptrace command for the process, such as *stop before executing the next system call event*, *i.e.* stop at the next pre-hook event for a system call. Once the traced process stops at this event, it will not start executing again until the tracer sends it a ptrace continue event, and it cannot run on its own without being traced

until the tracer issues a ptrace detach command. This means the traced process will continue to suffer a performance hit each time it stops and restarts after a ptrace event until it is released by the tracer.

Another perhaps unintended consequence of the tracer handling scheduling is that the tracer needs to handle scheduling *fairly*, which is in itself difficult to program, and even harder to program *correctly*. Even if a fair scheduling algorithm is implemented for the tracer, this feels like too much boilerplate, almost *kernel level programming*, needed outside the ptrace API just to trace system calls.

Implementing a fair scheduling algorithm for the tracer to utilize to manage its traced processes requires an extensive low-level programming effort, involving producing a scheduling framework around the existing ptrace API. This requires a substantial investment of time, energy, and an almost farcical amount of boilerplate necessary outside the existing API just to trace system calls.

This is because ptrace is meant to be an *asynchronous* API, but it is trapped in a synchronous system call world. In the next section, we discuss *asynchronous* computing and how building an asynchronous layer on top of the ptrace API can help with complexity, ease of programming, and performance.

1.5.2. A Primer on Asynchronous Computing

As we grievously learned while implementing DetTrace, ptrace introduces overhead, and quite a lot of it. Unlike DetTrace, ProcessCache's singular purpose is to improve performance. We could not approach using ptrace in the same manner we had for DetTrace, because the resulting overhead would make using the cache useless. When designing ProcessCache, we had to be clever. How could we work around the antiquated, synchronous nature of the ptrace API, while still utilizing ptrace to introspect system calls? Our solution was to implement an *asynchronous* ptrace runtime.

Review of Synchronous Computing

Before delving into our asynchronous framework, let us first formally define synchronous computing to clearly illustrate the fundamental distinctions between it and asynchronous computing. Synchronous computing is a *blocking* paradigm, and it is the traditional paradigm for computation. The reason for this is that until about a decade ago, CPUs were single core. A single core *seems* simple to work with, but from both a hardware and software standpoint, it poses challenges. How can we run many processes efficiently on one core, without them contending for resources?

Well before multicore processors arrived on the scene, computer scientists developed *virtualization*, where the system's resources are abstracted over to allow for multiple processes to execute on a single core. This is accomplished by giving each running process the impression that it has all the resources it could ever possibly need and exclusive access to the core. Virtualization is the mechanism of presenting a *virtual* version of the hardware to each process, that is mapped to the reality of the physical machine in a virtual mapping, and it is this virtual information that allows for multiple processes to multiplex on one core, unbeknownst to each other.

The other key to virtualization is that processes are assigned time slices, which is the time period during which they have exclusive access to execute on the core. When a process's time slice expires, or if it blocks on its operation, the process is preempted from the core and a different process is allowed to run. This gives a uniform priority and weight to each running process, but while fair, it does not prescribe time slices proportional to job size or which processes are actually *ready* to run at any given time.

Let us now describe a common scenario in which synchronous computing falls short. Imagine a server that handles many parallel requests, where each request involves serving the contents of a file. To accomplish this in a synchronous manner, the server must spawn an appropriately sized thread pool at startup or spawn a new thread for each new request. When a thread is assigned to handle a request, it reads the desired file. This operation could potentially block if the resource is not yet ready. The server is responsible for receiving requests as they arrive and dispatching them

to an available thread in the pool. This may sound simple, but it involves keeping track of which threads are free, which are currently blocked, and which are successfully performing their reads, in addition to managing incoming requests.

There are many ways in which overhead creeps into such a system. First, a nontrivial amount of overhead is incurred from spawning the thread pool at startup. This is, however, often favorable to spawning a thread at the time of a new request, so the user does not pay the price of thread spawning overhead. A thread pool has a set number of threads, and while the pool can possibly be resized at runtime, doing so is complicated in terms of programming, and adds another layer of responsibility to the server.

The scheduling management and context switching of threads and processes comes with its own overhead: it takes time to save one thread's state, remove it from the core, and replace it with another thread. It also takes time to iterate through the different threads as they run to keep track of whether they are available, blocked, or currently fulfilling a pending request. Finally, when a thread is blocked, it cannot make progress, and this in turn adds its own overhead in addition to the overhead incurred by the server for checking this thread to see if it has unblocked.

This simple example illustrates one common situation where synchronous computing is not the pragmatic choice. Synchronous computing is not well suited for circumstances when we have many I/O requests to honor or many short lived tasks to run. This is why many database systems that are used today (like Redis) do not exercise a synchronous workflow involving parallel processes or threads [12]. Synchronous computing is the ideal paradigm for parallel programs (on multicore machines) or multiplexed programs (on a single core) where each process is performing CPU-bound work. The cost of scheduling and managing synchronous parallel or multiplexed processes has only become more important in modern computing as workload complexity and dataset size both continue to grow exponentially, with no end in sight.

Introducing Asynchronous Computing

Asynchronous computing takes a radically different approach to process scheduling management by employing a *non-blocking* methodology. The asynchronous paradigm was inspired by the realization that many tasks are not CPU-bound, but I/O-bound, meaning they spend most of their computation time blocked waiting on the results of an I/O request. In programs with many tasks like this, such as web servers that employ a request/response workflow, a synchronous program has to manage a thread pool and continuously monitor thread state and handle scheduling. The asynchronous paradigm invites us to ask what should have been an obvious question: instead of us checking in on threads constantly to see when they are ready, wouldn't it be better for the threads to just notify us when they are?

There are many forms of asynchronous computing in different programming languages, but in this dissertation we concentrate on asynchronous programming in Rust. Asynchronous Rust centers on a unit of work called a *future* [73]. A future represents a value that may not be available yet, for example, a requested file from a server. Instead of blocking execution until the value is ready, futures allow us to continue executing and be notified when the value is available.

In Rust, a future is represented by a trait[1] comprised of the output it will return and a `poll` function that allows us to check if the value is ready (Figure 1.1). The `poll` function returns an enum, either `Poll::Ready(T)`, meaning the value (of type `T`) is ready, or `Poll::Pending`, meaning the value is not ready yet. This follows a higher level of abstraction: instead of a programmer reasoning about synchronizing OS threads, they can instead reason about their programs as being made up of tasks, where a central managing *executor* iterates over their asynchronous tasks, polling them and reporting if one of them is ready. The executor is essentially the task manager in the asynchronous runtime. There are two main keywords built into the Rust language that are central to its asynchronous mechanisms: `async` and `await`. The `async` keyword is added to a function to make it an asynchronous function whose return type is that of a Rust Future. The `await` keyword

[1] A Rust *trait* is similar to what is known as an interface in other languages such as Java. A trait defines the functionality a particular type has and can share with other types. Traits are more abstract and more powerfully generic than, for instance, Java interfaces, because interfaces can only be implemented for user defined types whereas traits can be implemented for any Rust type, even ones built into the standard library.

```rust
trait Future {
    type Output;
    fn poll() -> Poll<Output>;
}
```

Figure 1.1: Simplified code for the Rust future.

```rust
fn executor() {
    loop() {
        if let Some(new_request) = get_incoming_request() {
            add_new_task(new_request);
        }

        let next_task = get_next_task();
        match next_task.poll_task() {
            Poll::Ready(requested_content) => {
                serve(requsted_content);
            }
            // The request is not ready yet, we'll try again later.
            Poll::Pending => (),
        }
    }
}

// Function each async task runs.
async fn run_task(new_request) -> String {
    let file_contents = read_file(new_request).await;
    return file_contents;
}
```

Figure 1.2: Example of a simple asynchronous Rust program.

is used within an asynchronous function to retrieve the value of the future.

[TODO: Fix starting here]

Figure 1.2 is a simplified version of the asynchronous code for the example in the last section where we have a server that receives many concurrent read requests. It demonstrates Rust futures and Rust's async/await keywords. This example has two functions: one for the executor and one for asynchronous tasks. The executor essentially loops between receiving new incoming requests that are then delegated to new asynchronous tasks and checking for the next asynchronous task to finish so that its result can be served. If the requested file is not ready yet when an asynchronous task is polled, Poll::Pending is returned. If the task *has* successfully read from the file already, polling will return the content (Poll::Ready(requested_content)).

Figure 1.3, shows the same program expressed synchronously. There are again two functions: one for work threads and one for the main thread. In the main thread function, the main thread must spawn a worker thread pool, and is unable to alter the size once it gets them running. This means that if we want to change the size of the thread pool to keep pace with unexpected demand spikes, we have to add more implementation to our server to allow it to do so. Adding threads to the pool later also means we will incur overhead while the server is handling requests, as opposed to our original thread pool, which we only paid the price for at the start of execution of the server before we accept any requests.

Our asynchronous version, on the other hand, does not have to incur overhead by spawning an initial thread pool or adding to an existing one, it simply adds a new asynchronous task to the runtime whenever a new request comes in. Adding a new asynchronous task incurs minimal overhead compared to threads; it does not require the spawning of an entirely new OS thread and all the costs associated with such as memory allocation and context switching.

The loop that follows is a very simplified version of the actual loop the main thread would need to perform. First, it checks for a new incoming request. If one exists, it gets the next available worker thread, assigns the request to it, and moves that thread to the list of threads currently assigned a request (`assigned_threads`). If no available thread exists, this request is put in a list of requests that need to be handled at some point, but cannot be handled immediately. In order to have the ability to alter the size of the thread pool, this server's functionality would need to be extended.

Next, the main thread iterates through each thread that has been assigned a task in order to check its current status. There are a few reasons why a thread may not have completed its request yet: the thread may be blocked, or it may not have had its turn to run. In this synchronous setup, the main thread has to check them all, one at a time. Finally, the main thread has to iterate through the unassigned requests and try to assign them to threads. This version involves a lot of management on the part of the main thread, and there are many potential sources of overhead, including but not limited to: spawning the thread pool (and possibly altering the size of it while the server is running), not being able to handle a request immediately because no threads are available to do

```
fn main_thread() {
    // Threads waiting for work.
    let mut avail_threads = Vec::new();
    // Threads assigned tasks already.
    let mut assigned_threads = Vec::new();
    let mut unassigned_requests = Vec::new();

    for i in 0:n {
        spawn_thread();
    }

    loop {
        if let Some(new_req) = get_incoming_request() {
            if Some(avail_thread) = avail_threads.get_next_available() {
                avail_thread.handle_request(new_req);
                assigned_threads.push(avail_thread);
                avail_threads.remove(avail_thread);
            } else {
                unassigned_requests.insert(new_req);
            }
        }

        for thread in assigned_threads {
            // Check threads that are currently handling requests.
            // If we get None, the response is not ready yet.
            if let Some(request_to_serve) = thread.check_for_response() {
                serve(request_to_serve);
                assigned_threads.remove(thread);
                avail_threads.push(thread);
            }
        }

        for req in unassigned_requests {
            if let Some(avail_thread) = avail_threads.get_next_available() {
                avail_thread.handle_request(new_req);
                assigned_threads.push(avail_thread);
                avail_threads.remove(avail_thread);
            }
        }
    }
}

fn handle_request(new_request) -> String {
    let file_contents = read_file("foo.txt");
    return file_contents
}
```

Figure 1.3: The synchronous version of the program in Figure 1.2.

the job, and incoming requests waiting for the main thread to handle its other responsibilities.

Rust's asynchronous implementation differs from most other programming languages in the following ways:

1. Rust futures only make progress when polled. They do not waste any computation time on the CPU by blocking.

2. Futures are truly zero-cost. They do not require heap allocations or dynamic dispatch.

3. Rust does not have a built-in asynchronous runtime. Runtimes are provided by community maintained crates[2].

The final point is very important in the context of the ProcessCache system we will explore in this work. As a reminder, the ptrace API has numerous caveats, one of which is that the tracer must to be single-threaded. If Rust was a less flexible language with an asynchronous runtime that was essentially unchangeable, this runtime would probably be multi-threaded for performance reasons, destroying any chance we had at creating an asynchronous ptrace runtime with Rust futures.

Thankfully, this is not the case, and many open source runtimes exist that can easily be pulled in for use. However, when we embarked on our ProcessCache journey in 2019, asynchronous Rust was still in its infancy, and there were not many existing asynchronous runtime crates to choose from. In fact, async/await syntax was not standardized for the language until November of that year. The de facto asynchronous runtime available at the time was Tokio [17], which is still popular to this day.

We unfortunately could not use Tokio because it is a multi-threaded runtime. We needed a runtime that was single-threaded, where that single thread is the ptrace tracer. We were forced to get creative, and this led to us implementing our very own asynchronous runtime, one that is specifically designed for the ptrace use case. In subsection 3.4.1, we go into detail about the custom runtime.

[2] A Rust *crate* is a Rust package that can be pulled into Rust projects but does not come built-in with the standard library. Crates are typically open source and community maintained.

1.5.3. Security

As powerful as process tracing and manipulation is, it is natural to question what the security implications of using a mechanism like ptrace are. ptrace allows the tracer to observe the registers and memory of processes. This may appear dangerous, but in actuality, Linux uses per-user permissions to determine what tracing is allowed; therefore, tracing does not actually confer any extra access to processes than a user already has.

Tracing itself is a *read-only* operation, because it just involves reporting events from a program by examining memory. This means that tracing does not pose a risk of breaking the program, causing it to either run incorrectly or crash. Process manipulation, on the other hand, if done incorrectly, *does* have the potential to break a program. Therefore, purposely interfering with the execution of a process by using ptrace requires familiarity with many low-level OS concepts such as program registers, the system call ABI, signal delivery, and process scheduling and execution.

1.5.4. Performance Improvements with seccomp-bpf

The default behavior of ptrace is to stop the tracee for every system call twice, once for the pre-hook event and once for the post-hook event. This can be prohibitively expensive and is typically not the desired behavior when using ptrace. There are hundreds of Linux system calls, and a tracing system can usually accomplish its job without intercepting and examining every single one of them. Luckily, Linux has the seccomp-bpf mechanism, which allows for selective system call interception, avoiding overhead by allowing us to only intercept system calls that are relevant to the system and its goals. We utilize seccomp-bpf to improve our ptrace overheads in numerous ways for each project (as explained in subsection 2.5.11 for DetTrace and subsection 3.3.2 for ProcessCache).

1.6. Desirable Characteristics of Programs

There are countless desirable characteristics programs can have. Because one of our main goals for the designs of these systems is for the resulting implemented system to be program agnostic, let us limit our imagination to enhancements the *average* program can benefit from: determinism enforcement and performance enhancements.

Each system presented in this dissertation is designed and implemented to specifically provide either a determinism guarantee or to boost the performance of the program. Providing both of these in the same system is certainly possible. The drawback, however, is that enforcing determinism almost always diminishes performance improvements, because enforcing determinism is a costly operation. A system that attempts to accomplish both will see its performance gains curtailed by determinism imposition. This is the interesting trade-off between the two systems to be introduced in this work, and it is the reason we chose to focus on providing one useful enhancement in each system, instead of attempting to combine them into one system.

Before we scrutinize these projects thoroughly, let us first be exact with our definitions of the enhancements we aim to provide with each of our systems. Our aim with ProcessCache is to reduce the execution time of programs to less than that of their baseline execution times. Because this characteristic is simple enough to understand, we do not go into further detail here explaining it. Correctness, on the other hand, can take many forms and its many definitions have subtle differences. In this dissertation we focus on a single correctness property, *determinism*. Because determinism can take many forms and its many definitions have subtle differences, we devote the next section to a deeper analysis of what it means for our reproducible container system, DetTrace, to *enforce* determinism.

1.6.1. Correctness: Determinism and Reproducibility

There are many attributes of a program that contribute to its correctness, but in this work we focus on *determinism*. There are numerous definitions of determinism, and some are strict enough to require all facets of the computation performed (even seemingly less important aspects such as running time or power consumption) to be exactly the same. We leave the enforcement of these stricter forms of determinism to other systems. For us, determinism is *dataflow* determinism, which means that given the same set of inputs, a program always produces the exact same outputs.

Dataflow determinism implies many useful properties: the file system state after all processes have finished will be identical, as will the messages printed to standard output and standard error [94]. We choose to center on this precise type of determinism because it is an advantageous characteristic

for many types of programs to have. For example, a programmer may not care whether their program consumes *exactly* the same amount of power during each execution, but they would probably appreciate the guarantee that the outputs produced will be identical each time.

The property of dataflow determinism can be expanded it to include portability so that it extends across machines with varied microarchitectures or OS versions. The combination of dataflow determinism and portability is called reproducibility. We chose to pursue reproducibility because DetTrace is a container system, and containers are generally assumed to work equivalently across machines. Unfortunately, most if not all existing container systems, even Docker [94], are not actually deterministic, and therefore are also not reproducible. With cloud computing becoming the standard, we knew we had to design DetTrace to produce reproducible results. This is, however, much more challenging to accomplish than providing determinism on a single machine. This means that DetTrace must abstract over even more complicated low-level details, including the hardware on the machine itself.

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 [1, 7], reproducible data analytics [3, 10], and fault-tolerant distributed systems [40, 59, 61, 109]. Yet in spite of previous work on deterministic languages [36, 81, 110] and operating systems [27, 32, 67], 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 [27] does not provide a repeatable notion of deterministic time. dOS [32] provides a *deterministic process group* abstraction and can record-and-replay timing-related system calls. But dOS also uses record-and-replay for file system interactions, leaving the file system 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 2.7.5), 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 I/O 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 chapter and its corresponding work 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 [21] and enabling transparent fault recovery [59]. 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) [1] effort, target bitwise reproducibility of all their packages. Microsoft is pursuing reproducible software builds [105] with support in its C# and VB compilers [69]. Google's Blaze/Bazel build system [7] 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 [19] 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 a decade of effort by dozens of DRB contributors, 3.3% of current Debian packages (1,146 in all) remain irreproducible [4]. While tools exist to identify sources of irreproducibility [106], fixing a build is still a manual process. Even should DRB reach 100% reproducibility, vigilance would be required to ensure that errant code changes did not reintroduce irreproducibility.

Computational Science It is perhaps ironic that, while reproducing results is a cornerstone of the scientific method, many computational science tools are not reproducible. While chemical reactions and living organisms are intrinsically variable, there is no good reason for computation to behave similarly. Reproducibility in computational science would accelerate scientific advancement as 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 [93] 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) [108]. 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 [49, 70].

2.3. Reproducible Containers

In this work we aim to provide a reproducible container abstraction. The container itself is specified as an initial file system state and a program (from the file system) 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 file system 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 [79], 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 file system state after all processes have finished will be identical, as will the messages printed to standard output and standard error.

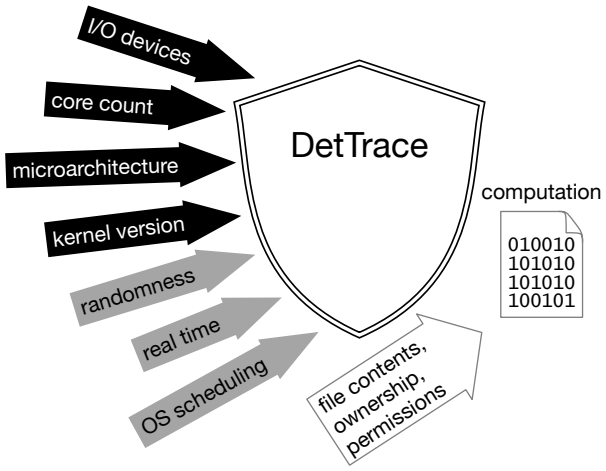

Figure 2.1: DetTrace containers abstract away both sources of nondeterminism (gray arrows) and nonportability (black arrows), making a DetTrace computation a pure function of its initial file state.

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* [74]: 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 1.5.4). 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 file system 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 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 [102] 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[3] and system call to see if it can be a source of irreproducibility, and under which conditions if so. Of particular importance is identifying *critical* members of an interface—those which permit irreproducibility but which cannot be reliably detected during execution. Any *critical* instruction or system call could silently introduce irreproducibility.

Our use of ptrace means that we see all system calls made from the container, so there is no potential for a critical system call (we also handle vDSO calls, see 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 [9]. Privileged

[3]There are some undocumented x86-64 instructions [45]. Handling these would be an interesting avenue for future work.

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 [98]). 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 [20] 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 instruc-

tions (*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. System Design

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

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

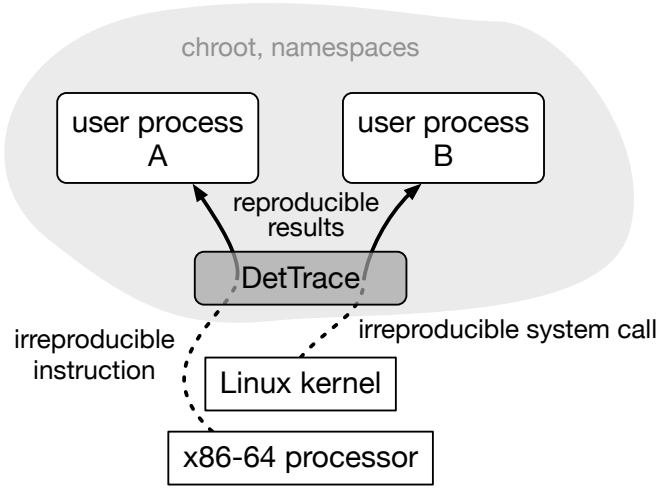

Figure 2.2: High-level overview of DetTrace's organization. The unshaded blocks (processor, kernel and user programs) are completely unmodified.

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. Operating System 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. Times 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. File system 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 [26].

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 [101]. 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 file system that a user process has, accomplished via the chroot system call. DetTrace can also be nested inside standard containers like Docker to provide stronger file system 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 file system 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 file system 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 repro-

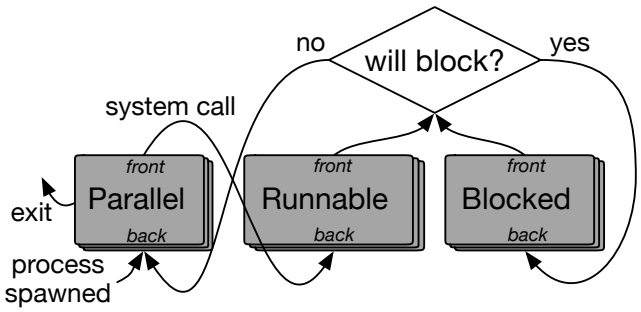

Figure 2.3: State transitions for a user process in the DetTrace scheduler.

ducible, 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 file system to obtain the path and real inode of the newly-created file descriptor. By examining the path both before the open call reaches the OS and afterwards, we can reliably identify when new files are created. If the file was newly created, we assign its mtime as the current virtual mtime, and increment the current virtual mtime. Otherwise, the file existed in the initial container image and we assign it a virtual mtime of 0. Writes to a file do not currently update its virtual mtime because we have not found this necessary in our workloads, however this could easily be added to provide more realistic-looking virtual mtimes.

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

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.

This is repeated until all processes have exited. When a process initiates exit under ptrace, it notifies the tracer. When the tracer receives this event, it detaches the process and removes the process from the *Parallel* set so that the process can exit normally on its own.

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. We have been able to convert every system call that we have encountered that may block into a non-blocking functional equivalent. 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. Because blocking queries resolve reproducibly, DetTrace can support ostensibly-nondeterministic system calls like poll and select operating on pipes and regular files.

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. A more sophisticated dependency tracking technique could track fine-grained resource dependency between processes, but we found our current implementation to be sufficient for the range of benchmarks we used to evaluate DetTrace.

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 biggest challenge for thread support is supporting the exit_group system call. When an exit_group is triggered, all threads in the current *thread group* will exit. A thread group TG_p is a set containing the parent thread (a process with PID p) and all its child threads. Just like the process exit procedure described previously, p cannot exit until all its child threads have exited. Therefore, we track thread group members. When exit_group is invoked, we schedule all child threads in TG_p for exit. When all child threads have exited, p is marked as *Finished*. When p's child processes have *Exited*, then p, too, can transition to the *Exited* state.

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.

Using the prctl system call, we ask the kernel to raise a segmentation fault whenever an rdtsc[p] instruction executes. To differentiate between a segmentation fault caused by a program memory error versus a faulting rdtsc[p], we use ptrace to check the instruction that triggered the fault. 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.

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 2.7.2 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.

The Linux kernel treats threads and processes quite similarly with some subtle but important differences. One disadvantage of ptrace is its poor support for threads, with a difficult to use API and little documentation. However, the scheduler outlined for processes can be extended to work with threads, with the following additions:

- Thread ID: The tid should be used instead of the pid, this information is readily available from the return value of ptrace.
- Thread Group ID (TGI): Threads in the same process share the same TGI. In a scheduler which may have threads from multiple different processes, the TGI is needed properly exit all threads due to certain events (see next bullet point).
- Tearing down threads: If any thread in a thread group calls either exit_group or execve, all threads within that TGI will exit immediately. Thus the scheduler must know to mark all these threads as ready for exiting and schedule them all to to have the kernel clean up their state.

Our prototype has minimal support for threads due to missing implementation for TGI and tearing down threads semantics. Thus we cannot run Java programs, which run to completion but appear to deadlock on a call to exit_group since the scheduler does not attempt to exit threads in the same TGI. Programs which do not tear down threads through exit_group or execve run deterministically as expected.

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

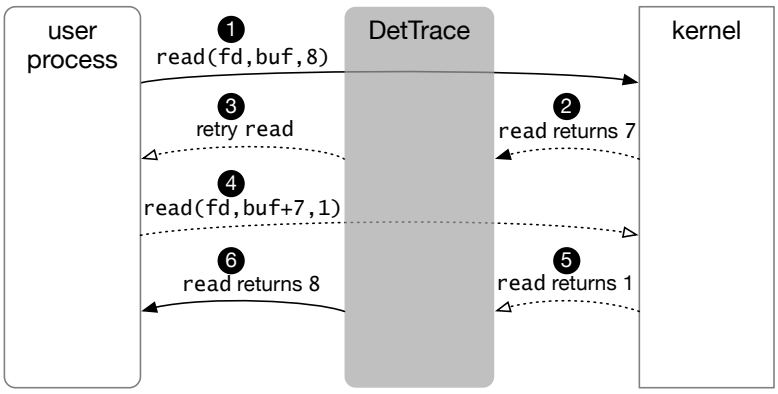

Figure 2.4: To render the read system call reproducible, DetTrace retries read operations that do not return the requested number of bytes. The solid arrows indicate what the user process perceives to have occurred. The dashed arrows indicate extra operations DetTrace undertakes to provide the illusion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

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.

2.5.11. Performance Improvements with seccomp-bpf

ptrace can be an expensive mechanism to use because by default it stops the tracee for every system call twice, as explained in subsection 1.5.4. We use seccomp-bpf to avoid intercepting system calls that are naturally reproducible in our environment. seccomp-bpf also allows the interception code, which runs before a system call reaches the kernel, to dynamically decide whether or not to intercept after the system call completes, further reducing overhead. Linux kernel versions $\geq$ 4.8 additionally optimize context switches by delivering a single event instead of separate pre-system-call and seccomp-bpf events. We support kernel versions $<$ 4.8 by falling back to the slower implementation.

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 [104], Clustal 2.1 in -ALIGN mode [46] and HMMER 3.1b2 [65]. We used TensorFlow v1.14 in our ML experiments, using the alexnet and cifar10 tutorials [114] 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 file system state.[4] Inside the container, we use a slightly modified version of the reprotest utility version 0.7.8 [13] from the DRB project. reprotest builds each package twice, varying the conditions for each build to exacerbate irreproducibility. We configure reprotest to vary environment variables, build path, ASLR, number of CPUs, time, user groups, home directory, locales, exec path, and timezone. We turn off domain host, kernel, and file ordering as they are not supported by the older version of Debian we're running our builds in. Similarly, the umask variation would randomize file permissions which DetTrace does not hide from user processes.

By default reprotest chooses variations randomly; we modified it to use a consistent configuration for the first build of all packages, and a different consistent configuration for all second builds, so that exactly the same environment is presented to DetTrace as in the baseline. We create a control-chroot of a minimal Wheezy installation, downloading the source via apt-get source, then installing a package's dependencies via apt-get build-dep (referencing an on-disk mirror to avoid network requests and ensure consistency across builds). Finally we copy the control-chroot to create an experiment-chroot, thus guaranteeing the same starting image for both builds. reprotest takes these starting chroots for running dpkg-buildpackage with or without DetTrace. When using DetTrace, everything dpkg-buildpackage does runs under DetTrace, which includes compilation, running tests (if the package is configured to do so), and creating the final .deb package. After both builds are complete, reprotest validates reproducibility with bitwise comparison of the two .deb packages. reprotest calls another DRB tool di□oscope which compares two directories, checking for bitwise identical contents. If di□oscope 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

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

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

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

Table 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.

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 [15] on the individual files, stripping timestamps. Finally, di□oscope 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 2.7.8), due to, e.g., randomization of the training set. DetTrace renders these workloads reproducible without any code changes.

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 2.7.2.

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.

2.7.2. 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 [5]. 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.

2.7.3. Comparison with Mozilla rr

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

2.7.4. 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.

2.7.5. 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 file system 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.

2.7.6. 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. Each baseline build runs pinned to a single core to allow us to measure DetTrace's overheads beyond that of process serialization.[5] 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.

Runtime events DetTrace counts a variety of relevant events on each run; the results are summarized in Table 2.2. Data is drawn from all 12,130 packages that build with DetTrace.

We find that system calls are frequent in package builds, with over 800,000 in an average build (Table 2.2). DetTrace also performs nearly 400,000 reads of user process memory on average, to extract pointer-based system call arguments. 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. Even simple software builds are rife with irreproducible operations.

2.7.7. 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

[5] In other experiments where the baseline is not pinned, it achieves only modest speedups because most builds do not use parallelism.

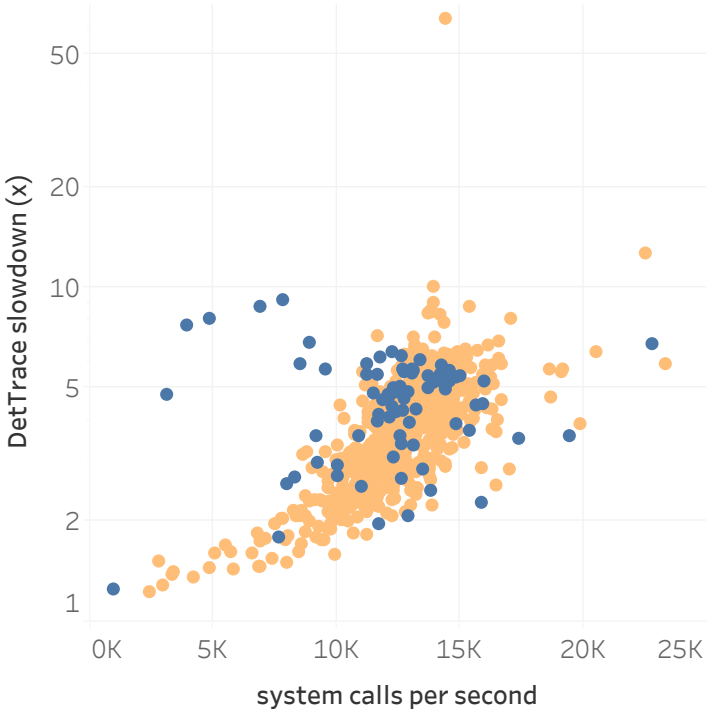

Figure 2.5: DetTrace overhead (y-axis, log scale) is largely driven by the rate at which system calls are performed (x-axis). Packages that use threads (dark blue dots) are typically slower than those that do not (light orange dots).

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

2.7.8. 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,

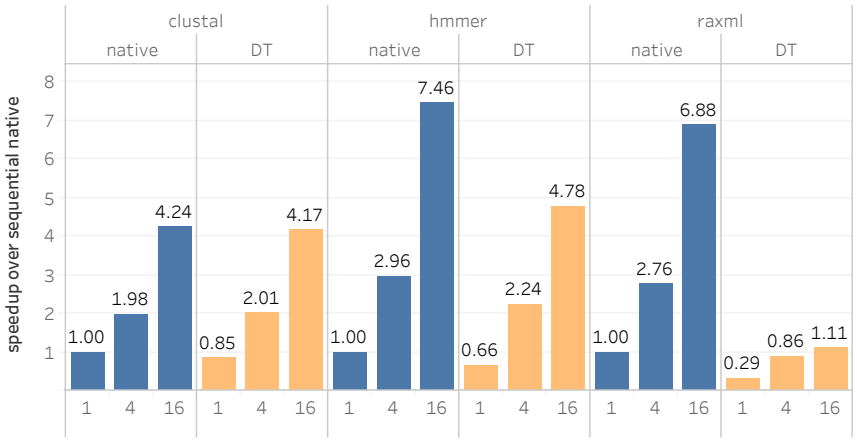

Figure 2.6: Speedup of bioinformatics workflows with 1, 4 & 16 parallel processes, normalized to sequential native execution (higher is better). Dark blue bars are native execution, and light orange bars are DetTrace.

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.

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.

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 [50, 54, 56, 75, 82, 107, 115] and industry [11, 18, 97]. 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 [97] as it also relies on ptrace (a quantitative comparison with rr appears in subsection 2.7.3). Many RnR systems target multi-threaded 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 chapter, providing deterministic versions of OS abstractions like processes and threads. While Determinator [27] provides new OS abstractions for deterministic fork-join parallelism, and DDOS [67] focuses on local network interactions, dOS [32] 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 file system interactions, defining the boundaries of its determinism abstraction too narrowly to be useful for software builds which interact extensively with the file system. 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 multi-threaded process, determinizing interactions through shared memory. Some schemes target arbitrary binary programs [31, 51, 52, 66, 71, 78, 84, 85, 86], providing generality at a modest performance overhead. Other schemes leverage language support to provide determinism for Haskell [42, 74, 76, 80, 81, 95] or Java [36, 37] programs. Whether language-agnostic or specific, these approaches eliminate the influence of thread scheduling, but do not determinize I/O interactions with the underlying OS and file system. The scope of their guarantees is thus too small to be useful for reproducible builds. One exception is DetFlow [110] which provides deterministic parallel execution for batch jobs, though it lacks robust system call interception and requires a coordinator layer written in Haskell.

2.9. Conclusion

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

Thus we conclude that user-space determinism enforcement is broadly useful for package build systems that seek bitwise-deterministic outputs. One clear avenue for future work is to apply reproducible containers to the other 21 package managers and Linux distributions that make up the reproducible-builds.org project.

Going forward, we plan to improve the generality and performance of reproducible containers, e.g., to reduce system-call interception overheads and to support parallelism. Looking further ahead, we envision reproducible containers finding many uses beyond reproducible builds. Reproducible containers can assist efforts, like the growing artifact evaluation movement, to make computer science research more reproducible. Reproducibility is a valuable property for many data analysis workflows, for validating and sharing results in the computational sciences, to enabling an audit trail for data-driven decisions taken in commerce and government. Reproducible containers can enable reproducibility in these domains with minimal perturbation to existing software.

Our sequential scheduler can be generalized to support process-level parallelism in service of, *e.g.* `make -j`. To mediate conflicts between processes, we might leverage ideas from deterministic multithreading systems. We could integrate DetTrace with a deterministic multithreading system such as Dthreads [77] for intra-process, shared-memory parallelism (although this incurs substantial overheads). Another approach for shared-memory parallelism is to integrate DetTrace with language-level determinism, as in DetFlow [110]. In such a scenario, binaries with language-level determinism could be privileged to "escape" the sandbox and avoid overheads. As with other sandboxes, future work can also improve the interception layer. There is substantial scope to optimize

our ptrace usage, following the example of Mozilla rr. Another approach would be to support a Windows reproducible container using the *pico process* abstraction for system-call interception—the same mechanism used for the Windows Subsystem for Linux (WSL) [2].

One method we've begun to explore is to sacrifice some portability and require a kernel module: *i.e.* use the Dune approach [30] to grant the user-space sandbox access to full Intel VT-x hardware-assisted virtualization support, enabling cheaper support for system call emulation and the ability to more broadly trap on instructions (like rdrand).

Finally, if reproducible containers eventually become trusted for critical applications—*e.g.* replicated state-machines that *must* stay in sync for a distributed algorithm to function properly—then it will become important to close any holes that adversarial programs could use to break determinism. For example, hardware-assisted virtualization methods can be used, or control-flow integrity (CFI) could be enforced in user-space so that a badly behaved binary cannot execute a disallowed instruction.

CHAPTER 3

AUTOMATIC MEMOIZATION FOR LINUX PROCESSES

3.1. Introduction

Memoization is an elegant model of computing that avoids unnecessary work when a program's input is partially, but not completely, changed. Work based on the unchanged part of the input can be cached and reused instead of being computed from scratch, saving time and energy. This core idea of reusing computation has been applied successfully in a range of domains, from dataflow systems [23, 43, 44, 89, 91, 92, 117] to programming languages [22, 34, 62, 63, 87, 103] to software build systems (like the venerable make and its many descendants [29, 38, 47, 90, 96]). Across these different domains, different terms are used to express the same core idea: *incremental computing*, *self-adjusting computation*, *reactive computation*, *caching* and so on.

While these different domains have important technical differences, in this chapter we focus on a systems context and aim to enrich the Linux process abstraction with transparent memoization at the granularity of Linux's execve system call. Our system, ProcessCache, provides automatic memoization for arbitrary Linux processes, accelerating multi-process workloads without the need for any code or configuration changes to the workloads themselves. ProcessCache is built on the Linux ptrace mechanism, which allows it to intercept system calls and comprehensively track inter-process dependencies to ensure correctness.

With our focus on the Linux process abstraction, ProcessCache is less similar to memoization systems that are specific to a particular domain (like dataflow graphs) or a particular programming language. The most closely-related work is in software build systems, which track dependencies among the various steps of a software build (like compiling a source file into an object file) to support efficient *incremental builds* that avoid re-compiling source files that have not changed, or more generally eschew re-running build steps whose (transitive) inputs have not changed. Most build systems rely on developers to specify dependencies, such as which source files are compiled together into an executable, though conventions about file layout can reduce the number of explicit

dependencies that must be specified in practice. Unfortunately, manually-specified dependencies are error-prone. An unnecessary dependency makes a build slower than need be, and a missing dependency can lead to erroneous builds [112].

To improve upon build systems with explicit dependencies, *forward build systems* [35, 48, 58, 112, 113] forgo the need to specify dependencies at all and instead ask developers to supply a *build script* that, when run, builds their software. The forward build system traces the build script as it runs to automatically discover dependencies among the build steps. Though the build script itself describes only how to build things from scratch, a forward build system uses these discovered dependencies to automatically provide incremental builds as well.

ProcessCache is perhaps best thought of as a generalization of a forward build system to support *arbitrary* software, not just software builds. We find that there are many process-level workloads, from shell scripts to bioinformatics workflows, that can benefit from memoization. Software builds, too, of course, benefit from reuse and ProcessCache readily handles them. For many of our workloads, however, there is no easy path to exploiting these reuse opportunities. Adding an engine to track dependencies, cache results, and determine when to reuse previous results would introduce substantial additional complexity to these workloads, and is orthogonal to their main algorithms. Alternatively, a workload could perhaps be re-expressed as the steps of a "software build" to take advantage of dependency tracking and allow for efficient incremental updates, though this would again be awkward and a distraction from the main focus of the software.

Instead of reinventing a caching engine in every software project, we propose ProcessCache as a cross-cutting approach that can automatically bring memoization to existing software. The contributions of this chapter are as follows:

- We describe ProcessCache, the first system to provide memoization for arbitrary Linux processes
- We describe the abstraction of the exec-unit as ProcessCache's fundamental unit of caching, and how to precisely reason about caching an exec-unit in terms of its preconditions and

postconditions

- We evaluate ProcessCache on a range of workloads, from shell scripts to bioinformatics workflows to software builds, showing that performance overhead is 1.69× on average even when no reuse opportunities exist, and that speedups up to 65× can be obtained when reuse is possible.

3.2. Caching for Linux Processes

In this section, we provide a high-level description of ProcessCache's operation, starting with various approaches to caching a process, then discussing how we determine whether a cache access is a hit or not, and finally showing how to use a cache hit to accelerate program execution. We elide several technical details in this section to speed the presentation; section 3.3 provides a more formal presentation of ProcessCache's operation.

For any cache, a key design decision is choosing the granularity of the cached elements. We are guided by three design criteria:

1. identifying cache hits (and misses) must be fast
2. cached elements must represent enough computation that cache hits can amortize the cost of cache lookups and provide significant additional performance benefit
3. it is always safe to re-run a program, so we don't have to cache all possible programs: we need only make the common case fast

Guided by our design criteria and our goal of caching Linux processes, we identify the **exec-unit** (explained next) as a useful granularity at which to cache computation. After an initial process p is launched via ProcessCache, exec-units within p and the process tree descending from p can be memoized.

An equally important consideration is correctness. Informally, ProcessCache provides the same results as running the program "from scratch" without ProcessCache, a property known as *from-scratch consistency* [64]. More precisely, ProcessCache has two key correctness criteria:

1. cached results provide indistinguishable behavior within the process tree managed by ProcessCache

2. cached results provide indistinguishable outputs to external observers after the process tree has terminated

Crucially, ProcessCache does not guarantee indistinguishability to external observers while processes are running: a motivated observer could examine `/proc` and see memoization at work. However, processes within the tree cannot tell when memoization has or has not occurred.

For processes that are especially nosy (*e.g.*, via `/proc`) or tightly bound to other processes via IPC, ProcessCache has the flexibility to choose caching boundaries to handle these cases or disable caching altogether. ProcessCache does not cache processes that are interactive (*e.g.*, reading from stdin connected to a terminal) or that consume explicit nondeterminism (like the current time); we discuss nondeterminism further in subsection 3.2.5. The current ProcessCache prototype also does not (yet) support networked programs, though we believe that networking support is feasible (see subsection 3.4.2). Next, we explain the concept of the exec-unit in the context of a single Linux process.

3.2.1. A Single Process

The simplest form of an exec-unit is the suffix of a process p's execution that begins with a successful `execve`[6] system call and continues through to p's termination where p spawns no child processes (we handle those next in subsection 3.2.2). The exec-unit encompasses all of the computation performed in this suffix by p, including all threads within p. At each successful `execve`, ProcessCache checks its cache to determine whether we 1) have cached results available and so can therefore *skip* an exec-unit, or 2) must re-run the exec-unit.

Consider a simple example where, from a shell, we run the command `/bin/echo hi`. Figure 3.1 shows the execution of the processes involved. First the shell forks a child process (Ê in the figure). The new process sets up signal handling, then runs the binary via `execve` (Ë), which will use a `write`

[6] We also handle `execveat` but discuss only `execve` for brevity

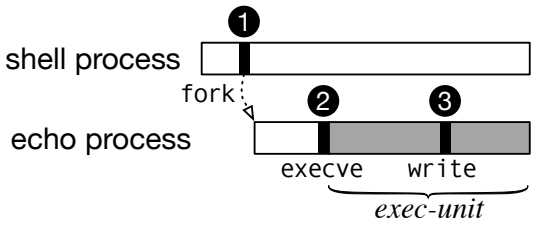

Figure 3.1: An example of an exec-unit (shaded gray)

system call (`write`) to write to stdout. The shaded part of the `echo` process is an exec-unit.

At first glance, `execve` system calls may seem like an odd place to define the start of our cacheable units. Caching an entire process p, from `fork` through termination, sounds more natural. But such a design would make it very challenging to identify cache hits, because the "inputs" to p include the entire address space of the parent process, the register state of the thread that called `fork`, and kernel resources like open file descriptors. Because these inputs can easily constitute gigabytes of state, checking for a cache hit will be slow. Moreover, some parts of the input state, like file descriptor flags which live in kernel data structures, need to be examined via expensive system calls. To determine whether we can skip p or if we need to re-run it, we must examine the entire state. If even a single bit of this state differs, it could in principle cause p to perform an arbitrarily different computation.

We could also choose cacheable units to be an arbitrary subset of p's execution. Here, the input state would be even bigger, adding in the registers for multiple threads and their accompanying kernel resources. The Shortcut system [53] actually does provide caching of arbitrary subsets of program execution along these lines, but incurs substantial implementation complexity (a custom OS kernel and C library, see section 3.6) to identify profitable regions to cache and to identify cache hits as a program executes.

Thus, there are many benefits to having every cached element begin with an `execve` system call. `execve` is notable for the state it clears away – the address space is reset, all threads but one are destroyed and the calling thread jumps to fixed initialization code. The "inputs" (explicit and implicit) to an `execve` system call are relatively small. The only explicit inputs are the binary

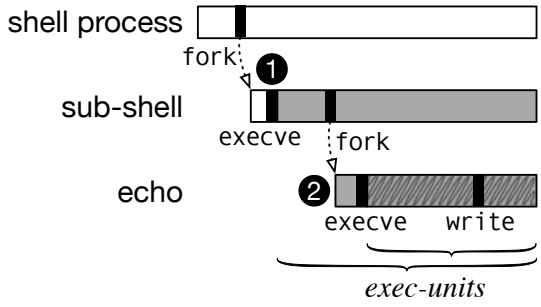

Figure 3.2: An example of nested exec-units

or shell script to run, command-line arguments and environment variables. There are additional implicit inputs which we consider later in Table 3.1. Regardless, execve's narrow set of inputs makes it easy to identify when we can use a cached exec-unit to skip ahead, and when we must re-run it.

3.2.2. Process Trees

Processes often spawn child processes. If a process p spawns a child process c, we can generalize our exec-unit definition above so that an exec-unit encompasses all of the computation performed by p's child processes as well. Now an exec-unit can start at an execve and continue through to the termination of all descendent processes. exec-units may also be nested within one another.

Consider the process tree that arises from running /bin/echo in a sub-shell instead of running it directly (Figure 3.2). We first spawn a process which execves the sub-shell binary (Ê), which in turn forks another process to execve the echo binary which finally writes to stdout. In this process tree we have two exec-units: one that includes just the suffix of the echo process (shown with the wavy line in Figure 3.2), and a larger exec-unit that contains the suffix of the sub-shell process and *all* of the echo process (the shaded areas), including the prefix of the echo process before its execve (Ë).

Within a process tree, each exec-unit is a sub-tree. ProcessCache does not support caching non-tree regions of the execution, e.g., the suffix of the sub-shell process from the execve but excluding the echo process. If we want to get the sub-shell from the cache but re-run echo (say its binary has changed), we don't have a parent process to launch the echo process or wait on it. The sub-

shell's execution also depends on its child's: if `echo` returns a different exit status, that impacts the sub-shell's execution and invalidates our decision to skip it. In practice, ProcessCache's proper nesting of exec-units avoids these difficulties.

3.2.3. Tracking Inputs

To determine whether we can skip an exec-unit e, we must have a precise understanding of the inputs that can affect e's execution. In ProcessCache we accomplish this via system call tracing with the Linux `ptrace` mechanism. We formalize the inputs to e as the *preconditions* for using a cached result instead of re-running e.

Let's consider the simple `echo` execution from Figure 3.1. The preconditions for this exec-unit are that the `/bin/echo` binary exists, is executable by the current user, and has identical contents to the recorded run. Shared libraries used by the program must similarly be accessible and identical. These properties are preconditions for being able to skip; if one of these properties does not hold (say the contents of the `echo` binary have changed) then the exec-unit cannot be safely skipped because running the new binary might produce different output.

As ProcessCache traces a program's execution, it is constantly both populating its cache with recorded exec-units *and* taking advantage of available skipping opportunities. ProcessCache does not have a "record" mode and a separate "skip" mode – it is always doing both. On the initial run of a program, the cache will be empty and so all cache accesses will miss. During this run, the cache will be populated with all of the exec-units that were observed, hopefully enabling skipping on future runs. During any given run, ProcessCache may both skip some exec-units it has seen before and also add new ones to its cache.

3.2.4. Skipping

Once ProcessCache has confirmed that the preconditions for an exec-unit e hold, it is possible to skip e. The actual skipping requires reproducing the functional outputs that e would have were it to be run. We refer to these outputs collectively as the *postconditions* for e.

Considering the simple `echo` execution from Figure 3.1, the postcondition is that a string is written to

stdout. ProcessCache records all output written to stdout in its cache so that it is available during skipping when we materialize the postconditions. In this case, the ProcessCache manager process will write that pre-recorded output to stdout itself, so that it appears to external observation after the process tree terminates that the `echo` process ran. Observation during the execution, however, could reveal that the child process never existed.

We limit our postconditions to what we deem to be functionally observable outputs. For example, when we skip a process sub-tree like that in Figure 3.2, only the process at the root of the sub-tree exists and we do not re-create child processes. A committed user could inspect the running processes via the `top` command and discover that `/bin/echo` never actually ran.

3.2.5. Nondeterminism and Parallelism

ProcessCache does not require a program to be deterministic. ProcessCache supports both parallel processes and parallel threads within those processes. Regions of execution free of system calls (the common case) can run in parallel. When ProcessCache experiences a cache miss it records the exec-unit e and adds it to the cache. This recorded version of e reflects one of e's possible executions when its preconditions were satisfied, though other outputs may be possible for programs that have, say, multiple threads or processes and output that is dependent on OS scheduling. While system call interception is currently serial (subsection 3.3.1), execution that does not involve an intercepted system call, which is most of the time, can run in parallel.

It may happen that ProcessCache records an execution of a nondeterministic program that happens to exhibit a bug; this buggy execution can be a source of cache hits causing the bug to manifest more often than it might under re-execution. Divining buggy executions is out of scope for this work, but the ProcessCache user can always force re-execution to try to record a successful execution instead.

Some processes explicitly consume nondeterminism, *e.g.*, via reading `/dev/random` or the current time. ProcessCache currently disables caching for such processes. A user can also tell ProcessCache to opt out caching and skipping programs they specify. When ProcessCache encounters

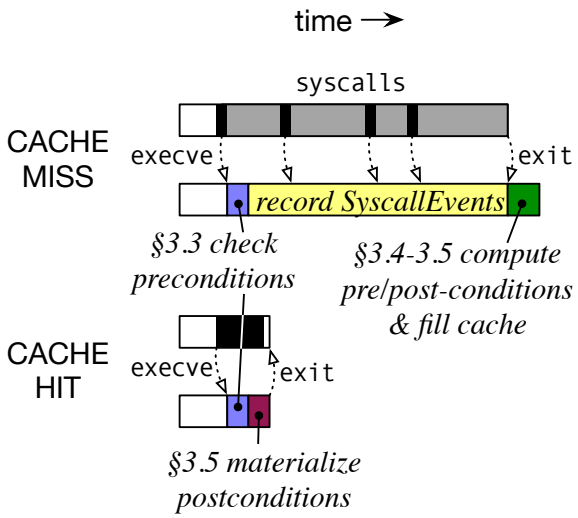

Figure 3.3: Execution timelines for a cache miss (top) and hit (bottom), showing the main operations of ProcessCache.

one such program, it will immediately detach from tracing the process and allow it to run normally. As an orthogonal approach, determinism enforcement schemes [28, 33, 77, 94, 111] can be used to enforce deterministic execution. While ProcessCache does not require determinism, having deterministic exec-units may be helpful in increasing the frequency of cache hits for programs that exhibit incidental nondeterminism.

3.3. ProcessCache Design

In this section we describe in detail how ProcessCache records exec-units and uses them to skip redundant computation. Since ProcessCache relies heavily on the Linux ptrace mechanism, we describe the ptrace interface next.

3.3.1. Primer on ptrace

Linux provides the ptrace API for tracing various events of interest in the execution of a *tracee* process such as the tracee's system calls, signals received, and thread and sub-process creation. These events can be intercepted and delivered to a *tracer* process. The tracer can also read and write tracee memory and registers.

For system calls, the tracer is notified when the tracee performs a system call but before the system

call reaches the operating system – we call this the *pre-hook* event. All ptrace events are handled synchronously, so the tracee is blocked while the tracer's event handler runs.

The tracer is notified again after the system call is handled by the OS but before it returns to the tracee; we call this the *post-hook* event. The post-hook allows the tracer to examine or even change the system call's return value. In addition, many system calls update one or more of their parameters, while their return value conveys little information besides success or failure. For example, the stat system call updates its *statbuf* struct with information about the resource specified by its *pathname* parameter; the real information is stored within the system call's updated parameter, not the return value. Thus, the post-hook event is useful in somewhat unexpected ways.

Signals directed at the tracee first trigger an event in the tracer, who can decide to suppress the signal, change it to another signal, or have it delivered intact.

These operations are quite low-level and simple on their own, but they are powerful primitives. We can implement high-level and useful operations from these primitives such as: injecting arbitrary system calls into a tracee, changing the arguments to a system call about to be performed, changing or emulating a system call in a process, observing the results of a system call, and implementing user-level, albeit slow, scheduling for processes or threads.

3.3.2. Usage of ptrace and seccomp-bpf

As explained in subsection 3.3.1, Linux provides the ptrace API for tracing various events of interest in the execution of a *tracee* process such as the tracee's system calls, signals received, and thread and sub-process creation. Due to synchronous event handling, and the separation of the tracer and tracee into separate processes, tracing with ptrace is somewhat slow.

Fortunately, a tracer can use seccomp-bpf to intercept only a subset of system calls, and unintercepted system calls run at native speed. ProcessCache only needs to intercept system calls related to file system interactions, and a few other bookkeeping things (like creation and exit, and other progeny this process may have spawned). This reduces the number of system calls actually intercepted by the system from hundreds to less than 35. We also avoid intercepting all read and

write system calls. These tend to be expensive and happen frequently, and do not convey any new information than the original open system call on the file gives us.

We also utilize system call injection in ProcessCache. One example is to record all the standard output from the process being traced. Immediately upon spawning and being registered with ProcessCache, ProcessCache injects a creat system call to create a new file on disk and and then a dup system call to duplicate the process's standard out file descriptor to the new file's descriptor. This allows ProcessCache to observe and cache all output from the process, so that if we choose to skip this process in the future, we can do so because we have the standard output stored.

3.3.3. execve and exec-units

A central part of ProcessCache's operation is computing the preconditions for system calls, to determine the preconditions for the encompassing exec-unit. No system call is more important to this effort than execve itself, which has a range of both explicit and implicit arguments, listed in Table 3.1.

We must first draw a distinction between a concrete argument like the C string holding the path to execute, and a higher-level set of preconditions that are implied by that argument. For example, when a path is passed to a successful execve call, there are three resulting preconditions: the path references an executable file, the file is executable by the current user, and the file's contents match those from when we cached the exec-unit. The preconditions for argv and envp are simpler – the strings that these pointers refer to must match those from when we cached the exec-unit. If an execve fails, that also contributes preconditions by implying the absence of various paths or permissions.

In addition to explicit arguments, execve also has a long list of implicit arguments, which are pieces of process state that survive across an execve; Table 28-4 in [72] describes them in detail. While ProcessCache does not yet track all of these pieces of state, they can in general be tracked through a combination of system call tracing (*e.g.*, tracking fcntl to see if the close-on-exec flag gets cleared), system call injection (*e.g.*, to find the current umask as a process may never itself call umask), and ptrace features (*e.g.*, to get the set of pending signals). Table 3.1 shows the current

```
struct StructEU {
  String binary_path,
  Vec<u8> binary_contents,
  Vec<String> argv,
  Vec<String> envp,
  u32 exit_code,
  Vec<SyscallEvent> syscall_events,
  Set<Precondition> preconditions,
  Vec<Postcondition> postconditions,
  Vec<StructEU> children
}
```

Figure 3.4: The data recorded for each exec-unit

subset of implicit arguments that ProcessCache tracks, which are sufficient for our workloads.

To record information about an exec-unit, ProcessCache uses StructEU objects (Figure 3.4). These are an important part of the actual cache implementation (subsection 3.3.6), and they live in memory while ProcessCache is tracing a program and then are serialized to disk for use on subsequent runs. Concretely, StructEU objects contain a set of preconditions and postconditions (see subsection 3.3.4 and subsection 3.3.5, respectively), and also nested StructEUs to capture information about child processes (see subsection 3.3.7).

Returning to our discussion of execve, we have only covered how to handle execve when it is successful. The system call may fail for a variety of reasons, *e.g.*, if the supplied path does not exist. There is no performance benefit to caching failed execves, so a failed execve does not begin a new exec-unit. While we could attempt to determine a priori whether the execve will fail or not, by checking the path and so on, doing so would be redundant and likely incomplete with respect to the real Linux implementation. Instead, we can use ptrace's post-hook event to identify when an execve has failed. These failed execves contribute to the preconditions by implying the absence of various paths or permissions.

To complicate matters further, successful execves do not have a proper post-hook, though ptrace does provide a "successful execve" event in its place. Because in the ptrace pre-hook we do not know whether an execve will be successful or not, and in the case of success the tracee's address space is reset so the memory referenced by system call arguments like argv and envp is lost, we speculatively

Explicit	binary/script to execute
	command line arguments argv
	environment variables envp
Implicit	open file descriptors without close-on-exec
	mask of ignored signals
	pending signal set
	current working directory
	umask
	timers from alarm

Table 3.1: Arguments to execve that are tracked by ProcessCache.

create a StructEU object in the pre-hook. If the execve turns out to be unsuccessful, the StructEU is destroyed.

More concerning is state that persists across an execve but which the tracee can access without a system call, or without making a system call that ProcessCache is intercepting. For example, if a file descriptor is not marked close-on-exec then it will remain open after the execve succeeds, and the tracee can read or write the underlying file. For performance reasons, ProcessCache does not intercept frequent read and write system calls. In our workloads, the only instances of non-close-on-exec file descriptors arose from shell pipelines, and ProcessCache aggregates the resulting exec-units in this case anyway (see Table 3.3.3). By intercepting fcntl system calls and doing extra bookkeeping in parent processes, we could in general track file descriptors that persist across an execve.

While we did not observe any usage of signals (outside of SIGCHLD for child process termination) among our workloads, we do record the current signal mask via a PTRACE_GETSIGMASK system call. A particular signal being masked (or not) could affect tracee execution should that signal be received, so the signal mask is part of the precondition. As Linux continues to evolve, additional pieces of state are likely to cross the execve boundary, but we are confident that ProcessCache's design can be extended to record them as needed.

Choosing Friendly exec-unit Boundaries

To handle sources of inter-process communication like pipes, ProcessCache will adjust exec-unit boundaries to reduce implementation complexity. Consider a command like `cat foo | wc`, where one process pipes data to another. Creating an exec-unit solely for the `wc` process would require all bytes received on standard input to be part of its precondition, which could be a significant amount of state. Checking that `cat`'s output matches that recorded in a previous run is also expensive, as we must buffer all its output to do so, eliminating the concurrency and streaming benefits of real pipes. Thus, we avoid placing an exec-unit boundary around `wc`, and in general require that processes that communicate via pipes are part of the same exec-unit. This requirement extends transitively, so an entire pipeline will be in the same exec-unit and skipped or re-run together. Other sources of inter-process communication like Unix domain sockets or shared memory could be handled similarly by placing all communicating processes within the same exec-unit.

3.3.4. Computing Preconditions

In computing the preconditions for an exec-unit, we seek to capture the conditions under which its behavior would be the same were we to re-run it. This is the key correctness criterion for ProcessCache:

Precondition Correctness. *If the preconditions of exec-unit e are satisfied, when re-running e it will execute the same sequence of system calls and each will return the same values as during the recorded execution.*

In the simplest examples, there is a clear 1:1 mapping from system calls to preconditions. However, in even mildly sophisticated programs the interactions between system calls give rise to complex and subtle preconditions. Consider the execution in Figure 3.5, where a program successfully stats, truncates, writes to and then reads from a file f. For simplicity we show only the system calls that ProcessCache intercepts, so the actual file `write`s (between `Ë` and `Ì`) and `read`s (after `Ì`) are elided.

First, we consider the preconditions for each system call in isolation, if that system call were the

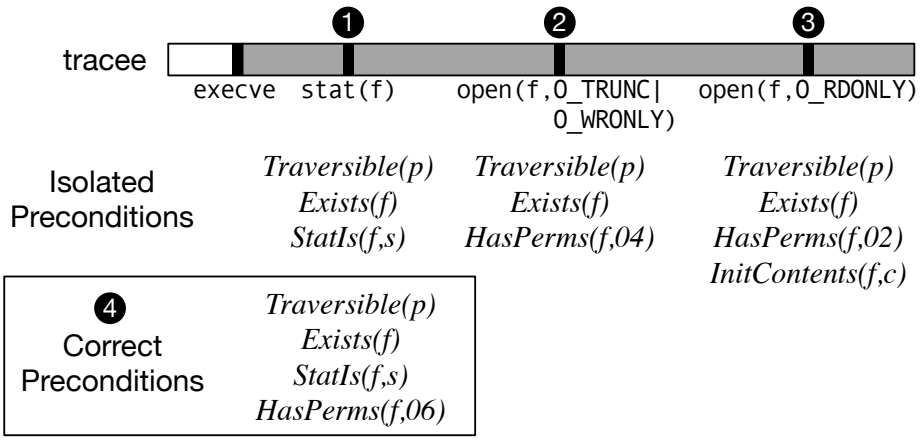

Figure 3.5: An execution with non-trivial preconditions.

only one in the execution. For the stat (Ê), the requirements for the system call to be successful and return the same results are that f's parent directory p is traversible, f exists, and its struct stat contents are given by s. For the open system call that truncates the file (Ë), it must be that p is traversible, f exists (since O_CREAT is not used) and the current user has write permissions (04 in octal) to f. When f is opened for reading (Ì), it must be that p is traversible, f exists, f is readable and it has initial contents c (which are returned by future reads not intercepted by ProcessCache).

In Figure 3.5, Í shows the correct preconditions. We illustrate their correctness by showing how simple approaches to computing preconditions fail to capture important subtleties.

For instance, we could ignore all but the first system call that accesses f (in this case, stat). Using only stat's preconditions, we would not check the permissions on f, and skip even if f were unwritable, which would diverge from the recorded execution at Ë when the open fails.

If we were to union all preconditions together, we would require that f have initial contents c. Actually, the initial contents of f don't matter (except for the file size, which is captured by the data s returned from stat) because f is truncated before it is read. All of the bytes of f read during the execution are a result of writes that occurred during the execution itself. Imposing a constraint on f's initial contents would inhibit skipping even when it is sound to do so.

system call	SyscallEvent
access	Access
creat, open, openat	Create
unlink, unlinkat	Delete
mkdir, mkdirat	DirectoryCreate
getdents64	DirectoryRead
execve, execveat	FailedExec
execve, execveat	SuccessfulExec
creat, open, openat	OpenFile
repc, repcat	Repc
stat, fstat, lstat, newfstatat	Stat
statfs	StatFs

Table 3.2: System calls and corresponding SyscallEvents

The correct preconditions capture precisely that the stat must return the same value s, that we need both read and write permissions to f, but that the contents of f are irrelevant. To compute such preconditions in general, we must carefully consider the type and values returned from each system call. Order is also critical: *e.g.*, switching Ë and Ì would make the initial contents c part of the precondition. We describe our algorithm for computing preconditions next.

Recording system calls

As ProcessCache intercepts system calls of interest, it records information about each one as a SyscallEvent (Figure 3.3). SyscallEvents provide a thin abstraction over each intercepted system call, recording its arguments and return value. Table 3.2 illustrates this translation into SyscallEvents. The mapping is many-to-many. Sometimes a range of system calls (like stat*) can all be represented with a single SyscallEvent. In other cases, like open, the arguments used can result in different SyscallEvents. SyscallEvents are retained in a list (reflecting their order in the execution) until a process terminates (Figure 3.3). At termination, the SyscallEvent list can be processed into a list of preconditions (explained in Table 3.3.4).

In some cases, preconditions require information from throughout process execution. For example, if a precondition requires that a file have some initial state, that initial state may be lost to overwriting by the time the process terminates. Thus, ProcessCache must also record information during execution in two cases, in anticipation of future precondition generation. First, if an existing file is

Precondition	Description
DirectoryEntries(dir,[(pc,FileType)])	directory has contents with the given names and types (file, directory, symlink)
DoesNotExist(path)	path does not exist
Exists(path)	path exists
Traversible(path)	all parent directories are traversible (executable)
HasPermissions(path,perms)	path has at least the given octal permissions
InputFilesMatch(path)	mtime/hash matches cached contents
NotTraversible(path)	some parent directory is not traversible
LacksPermissions(path,perms)	path is missing some of the given octal permissions
InitialContents(path,bytes)	file at path has given initial contents
StatfsIs(path,s)	statfs on path returns struct statfs s
StatIs(path,s)	stat on path returns struct stat s

Table 3.3: ProcessCache's precondition vocabulary

opened (without O_TRUNC) for the first time in the exec-unit, we record its SHA256 hash. Second, we check the link count of a file before it is unlinked, to tell whether the file was removed from the file system or not.

From system calls to preconditions

Immediately after a process has terminated, ProcessCache uses its list of SyscallEvents to compute a set of preconditions in a forward (program order) pass over the SyscallEvents list. While ordering is critical to SyscallEvents, preconditions are inherently unordered as each precondition must hold independently for an exec-unit to be skipped. In addition to the semantics of each SyscallEvent, preconditions must also take into account the initial and current state of each resource being accessed. For simplicity we only consider files as resources in the following discussion, but ProcessCache also supports directories.

The initial state can either be that the file exists, does not exist, or we don't care (*e.g.*, the file was created and overwritten with O_CREAT | O_TRUNC). We can infer the initial state based on the first SyscallEvent that targets a file. As additional SyscallEvents are processed, we maintain the most recent change to the file, which can be that it was created, deleted, modified, renamed (discussed more below), or it has not yet been changed. We also record whether a file was ever deleted, as any subsequent accesses to the file means that it must have been recreated by the process p, and its contents no longer depends on its initial state before p was launched.

Table 3.3 lists the set of preconditions that ProcessCache currently supports; conjunction and

```
fn Access(path, flags, syscallResult,
          initialFileState, fileWasDeleted) {
  pre = set([])
  if initialFileState == DidNotExist || fileWasDeleted {
    return set([]) // add no preconditions
  }
  // file existed and was not deleted
  if syscallResult == Success {
    pre.add(Traversible(path))
    if flags.contains(F_OK) {
      pre.add(Exists(path))
    } else {
      pre.add(HasPermission(path,flags))
    }
  } else {
    pre.add(LacksPermissions(path,flags))
  }
  return pre
}
```

Figure 3.6: Pseudocode for computing preconditions for the access system call event

disjunction of preconditions are also supported. Both successful and failed system calls generate preconditions. For example, if an open(p,O_RDONLY) system call fails with EACCESS then we know that either we could not traverse a directory somewhere in p, or we do not have read permissions on p. We would represent this as: NotTraversible(p) || LacksPermissions(p,02).

While ProcessCache has only 11 SyscallEvents, and files have only a handful of initial and current states, the cross-product is quite large and results in hundreds of cases to consider across our workloads. Figure 3.6 shows one representative rule for computing the preconditions for the access system call, which checks whether the calling process has the requested permissions (read, write and/or execute) for a given path.

One additional virtue of representing preconditions as an unordered set is that redundancy among preconditions is naturally eliminated. If an execution repeatedly opens the same file for reading, *e.g.*, we will only record the need to check file existence once. The stat system call presents a slight wrinkle in that we must retain the information from the *first* stat call for each file. Subsequent stat calls are redundant but also may return information about the file that reflects changes to the file from within the exec-unit; the precondition must capture only the state of the file at the start of the exec-unit.

File and directory renaming

File and directory renaming present a challenge to computing precise preconditions. Consider the case of renaming a file from A to B. Suppose an exec-unit reads A, then writes to it, then renames it to B, and then reads B. We must understand that A and B represent the same resource and avoid a precondition that requires a particular initial state for A *and* an initial state for B. Such a precondition is incorrect since the exec-unit doesn't know or care about the initial state of B, and might inhibit skipping.

The correct preconditions will only mention that A exists, is readable and writable, and has certain initial contents. B is created from A entirely by the exec-unit, and belongs only in the postcondition (which, conversely, should not mention A). Directory renaming presents a similar issue where everything in and underneath the renamed directory is affected.

ProcessCache models the effects of renames by updating the paths of the files and directories being tracked as we traverse the SyscallEvents list. This ensures that paths always reflect the "current" path at that point in the execution, which supports arbitrary renaming and allows for easy post-condition generation.

3.3.5. Computing Postconditions

When a process terminates, the list of SyscallEvents, along with information about a file's initial and current state, is also used to generate postconditions (Figure 3.3). Postconditions are significantly simpler than preconditions, consisting of the following operations on files and directories: creating, deleting, adjusting permissions, and (for files only) modifying contents. Similar to preconditions, when a process terminates we compute an exec-unit's postconditions for use later on during skipping (subsection 3.2.4): the postconditions are a recipe for how to recreate the functional output of the exec-unit. Note that while preconditions are inherently unordered, postconditions do require partial ordering – parent directories must be created before they can be populated.

3.3.6. Cache Design and Skipping

Our cache data structure holds all the exec-units that ProcessCache has recorded. The cache consists of a serialized data structure that maps from (binary-path, arguments) tuples to StructEU objects, and also holds files referenced by those objects. After a process terminates and pre- and postcondition generation completes, the resulting StructEU can be written to disk (Figure 3.3).

We perform cache lookups in the pre-hook of execve system calls, using the path and argv arguments to index into the cache. Note that multiple exec-units may exist for a given (path,argv) tuple as that binary may have been run with the same arguments but, say, differing initial file contents. The cache is thus properly a multimap. If there are multiple StructEUs for the current execve, we evaluate the preconditions for each one to see which one holds. At most one StructEU's preconditions will be satisfied. Two StructEUs a and b with equivalent preconditions can never be resident in the cache simultaneously. Suppose a was added first; when the exec-unit corresponding to b runs we will skip it instead of adding it to the cache.

There is a trade-off between the complexity of indexing into the cache and the number of hits we find at a particular index. The indexing function itself is in a sense precondition checking - by looking only at the binary path and argv we are checking a subset of the proper precondition. We could generalize this checking to the full precondition, but it would be expensive and clumsy to do so. Nevertheless, if cache collisions become an issue then incorporating a richer (but still approximate) precondition into cache indexing that takes, say, the current working directory into account, might prove beneficial for reducing cache collisions.

As Figure 3.3 shows, if we do not find a StructEU whose preconditions hold, the execve system call is allowed to run as normal. If we do find a StructEU whose preconditions hold, however, we have an opportunity to skip the exec-unit. First, we materialize the postconditions for the exec-unit, copying output files into place and writing the correct bytes to stdout and stderr. If our pre- and postconditions have been computed correctly, skipping satisfies *from-scratch consistency* in that it has the same effects as re-running the exec-unit.

Finally we must take care of the process that called execve. We edit the arguments to the execve system call to run an empty binary which immediately calls exit with the exit code from the StructEU. This makes the skipped process behave as expected from the point of view of its parent process, which is likely waiting on it.

3.3.7. Nested exec-units

As mentioned in subsection 3.2.2, exec-units may be nested within each other. Previously, we discussed precondition and postcondition generation only for non-nested exec-units. To create nested exec-units, we first splice together the SyscallEvent lists from all constituent exec-units, all of which must be terminated. Simple list concatenation does not suffice, as we must account for ordering among the exec-units via forking and waiting. Once the combined SyscallEvent list is created, we use the same precondition- and postcondition-generation algorithms as in the non-nested case. Unfortunately, due to stateful interaction among SyscallEvents, it is not possible in general to reuse pre/postconditions from an inner exec-unit when computing the conditions for an outer exec-unit. In some cases, like when disjoint resources are being accessed, it should be possible to compose pre/postconditions more quickly. However, pre/postcondition generation is not a significant bottleneck in our current prototype.

3.4. ProcessCache Implementation

ProcessCache is written in 5,200 source lines of Rust, and will be open-sourced upon publication. ProcessCache is a command-line utility; programs can be run under ProcessCache via a command like ./process-cache ls -ahl. ProcessCache is implemented entirely as a userspace Linux program and does not require elevated privileges to execute beyond the CAP_SYS_PTRACE capability.

3.4.1. Asynchronous Handling of Ptrace Events

In ptrace, events are delivered to the tracer by waiting on a single file descriptor. If there are multiple tracee processes, the events from those tracees are interleaved with one another as they occur in real time. It is more natural, however, to consume events in a per-tracee manner, as each StructEU accumulates events from a particular tracee. Threads would be a natural solution to this problem,

```rust
struct AsyncPtrace {
    pid,
}

impl Future for AsyncPtrace {
    type PtraceEvent;
    fn poll() -> Poll<PtraceEvent>;
}
```

Figure 3.7: Simplified code for our custom Rust Future.

with one thread per tracee, but as we stressed many times previously, all ptrace system calls and event waits must originate from the same (tracer) thread.

As described in subsection 1.5.2, Rust offers an elegant solution via asynchronous programming. Rust's standard library does actually have a asynchronous runtime; it deliberately contains only the async/await keywords. This allows us to implement our own asynchronous runtime with a single-threaded executor. We launch one asynchronous task per tracee, which can await events and perform ptrace system calls freely since all tasks are multiplexed onto one kernel thread. As shown in Figure 3.7, our futures are identified by the pid of the traced process. The poll function returns the ptrace event the process is stopped on, if it is indeed stopped. In a custom executor, we receive ptrace events and dispatch them to the appropriate task. This keeps the task code simple and focused on the operations of just one tracee. It is also a natural way to reason about ptrace, as it is at its core asynchronous. We ask a process to run until it stops at another ptrace event, which mirrors Rust futures perfectly.

Optimizations

Copying output files into the cache, to enable postcondition materialization, can be time-consuming since several of our workloads produce thousands of output files. To make matters worse, output files can only be copied once a process terminates. For the last straggler process in a batch workflow, all of the latency of copying its output files ends up on the critical path. To reduce this latency, ProcessCache copies output files to the cache using parallel threads. Parallel copies are used when materializing postconditions as well.

ProcessCache's hierarchy of exec-units further compounds the problem of copying output files

into the cache. Processes toward the top of the tree often require copies of many child outputs. ProcessCache avoids some of this slowdown by hardlinking child output files to a parent instead of copying them, because the cache needs to maintain just one physical copy of each output file.

For many workloads, multiple processes utilize the same executable. For example, with builds many jobs run `/usr/bin/gcc`. ProcessCache takes advantage of the fact that this executable's contents will almost certainly not change during the execution by not hashing the executable every time a process runs it. Instead, as the program runs, ProcessCache maintains an executable hash cache, and when a process calls `execve`, checks the cache for the executable's hash instead of immediately hashing it. Cache entries are invalidated should a process ever open the executable for writing.

If a process opens a file for reading, ProcessCache hashes the file's contents in a background thread. This is safe to do because the contents of the file are not being changed. Files opened for writing are hashed synchronously instead. If a file is first opened for reading, and later opened for writing, ProcessCache hashes the file immediately on the `write` call, and this hash is used as the precondition.

3.4.2. Limitations

ProcessCache currently does not cache programs that read from standard input, with the exception of pipe-based communication encompassed within a single exec-unit where it does not contribute to preconditions or postconditions. ProcessCache is designed to handle non-interactive, batch-processing programs. In general, ProcessCache must be able to validate all inputs to a process p at the time of an `execve` to make a decision about skipping. If p is consuming stdin, we will have to wait for the sender process s writing to stdin to terminate so that we can check whether p's stdin matches what we have recorded. Deadlock could arise if s is itself blocked waiting for p to terminate.

ProcessCache handles pipes by encompassing in one exec-unit all processes within a running program which are communicating via pipes. ProcessCache could, in theory, 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.

ProcessCache does not currently handle networking, but could be extended to handle client network requests. A request to a remote server would contribute a precondition: if the server responds the same way to the request, skipping is allowed. Thus, requests and responses would be logged in the StructEU. Some protocol-level reasoning might be desirable to, *e.g.*, identify HTTP responses that vary only with respect to cache-control headers. Running ProcessCache on a networked server, however, is tantamount to running an interactive program, which is not supported.

3.5. Evaluation

To evaluate ProcessCache, we seek to answer the following questions:

- Can ProcessCache support a useful range of programs?
- How much time can ProcessCache save with various rates of opportunities to skip? (subsection 3.5.3)
- What are ProcessCache's overheads when there are no opportunities to skip computation? (subsection 3.5.2)
- Is ProcessCache correct? (subsection 3.5.4)
- What are ProcessCache's space overheads? (subsection 3.5.5)

We address these questions across the remainder of this section. We show the breadth of programs that ProcessCache can support by measuring performance across multi-process workloads from bioinformatics, shell scripts and software builds. subsection 3.5.5 gives additional metrics like space consumption.

3.5.1. Experimental Setup

We evaluate ProcessCache with a variety of different workloads to provide a case study of its performance. For our bioinformatics workloads we used Bwa 0.7.10, Clustal 2.1 [46], HMMER 3.1b2 [65], Mothur 1.39.5, and RAxML 8.2.12 [104]. For our shell scripts we ran 55 shell scripts from the

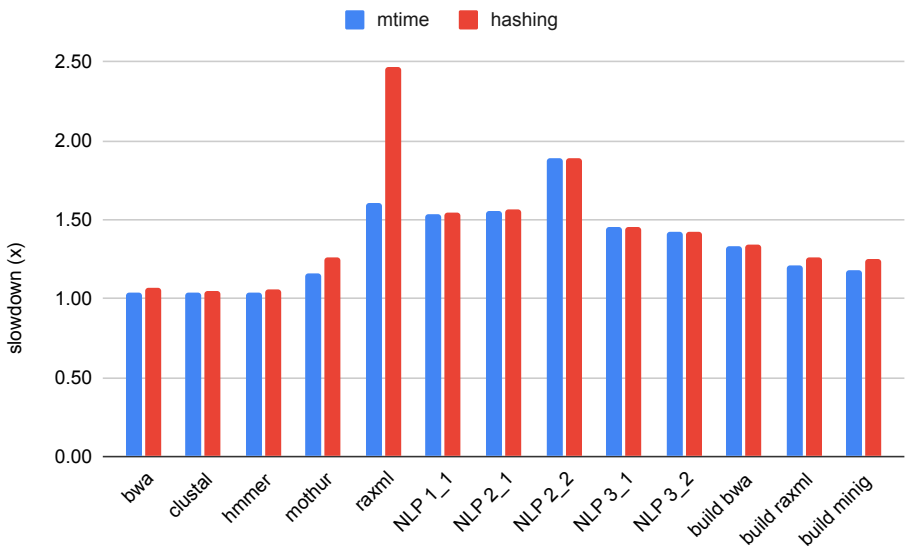

Figure 3.8: Slowdown when populating an empty cache. The blue bars represent slowdown when mtime is used as the input checking mechanism, while the red bars represent slowdown when hashing is used.

PaSh [116] benchmark suite, including scripts from their NLP, unix50, oneliners and analytics-mts collections, though we show performance results for just 5 representative scripts from NLP. We include only NLP scripts from the PaSh benchmark suite because the other scripts do not admit opportunities for speedup via incremental computation. We include only 5 of the NLP scripts because they all behaved similarly in our experiments, and including them all would crowd the graphs with somewhat redundant data. For our software builds comparison, we built multiple C projects of varying complexity and size, including: the Minigraph [88] DNA sequence mapper, and our Bwa and RAxML benchmarks. We ran our experiments 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 20.04 with Linux 5.4.0. Our workloads triggered no cache collisions (subsection 3.3.6).

3.5.2. Overheads With Empty Cache

Figure 3.8 shows the performance overheads (lower is better) that ProcessCache incurs when populating an initially-empty cache normalized to native runtime without ProcessCache. For

each benchmark, the graph includes the slowdown when the input checking mechanism is mtime and when it is SHA256 (ProcessCache's default). For compute-bound workloads (bwa, clustal, hmmer and mothur) the overhead of populating the cache is relatively low, as there is plenty of computation to amortize ProcessCache's system call interception and analysis. Workloads like the NLP shell scripts in particular have very little computation for many of their processes (often simple utilities like cat and tr) which leads to higher overheads of up to 1.89× (for both mtime and hashing).

raxml is an outlier in that it runs for a very short amount of time but produces thousands of output files, leading to the maximum caching overhead of 2.47× with hashing as the input checking mechanism. raxml represents a type of program that could benefit from using mtime as the input checking mechanism, as the caching overhead is then reduced to 1.61×. In this case, the loss of some soundness may be worth it to reduce the overhead by .86×. The software builds are in between, with moderate overheads of 1.25-1.34× with hashing and 1.18-1.33× with mtime. We also reproduced the from-scratch builds with Riker [48], which produced overheads of 1.04-1.07×, for both hashing and mtime. Generally, with respect to caching overhead, we did not find mtime to provide significant speedups when used as the input checking mechanism.

3.5.3. Performance Benefits of Skipping

To measure how much ProcessCache can exploit caching opportunities, we run each of our benchmarks (with a fully-populated cache) under four configurations: when none of the input files are changed (and thus all exec-units will be skippable), and when 5%, 10% and 25% of inputs are changed. We determined these percentages by examining the commit history of the Minigraph, Bwa and RAxML GitHub repositories, in which an average of 3-5% of files are changed per commit. Thus, we expect 5% changes to be a common case for how much inputs shift in practice, with 0% representing re-running a program when a user forgets that nothing has changed, and 10% and 25% showing ProcessCache's performance under atypically large sets of changes.

Benchmarks were measured with both mtime and hashing as input checking mechanisms. We compared both mechanisms in the case of pure caching overhead and found that the choice of

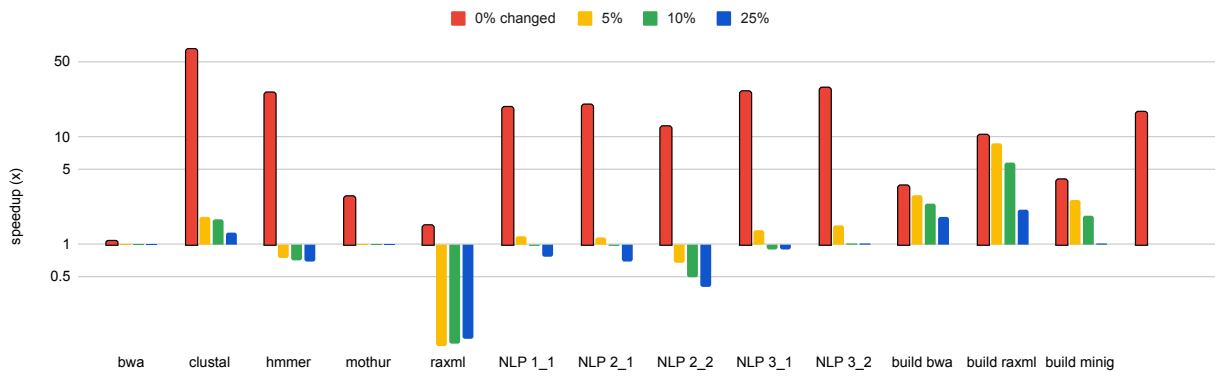

Figure 3.9: Speedups achieved when zero to 25% of the input files are changed when the checking mechanism is hashing. Note the log-scale y-axis.

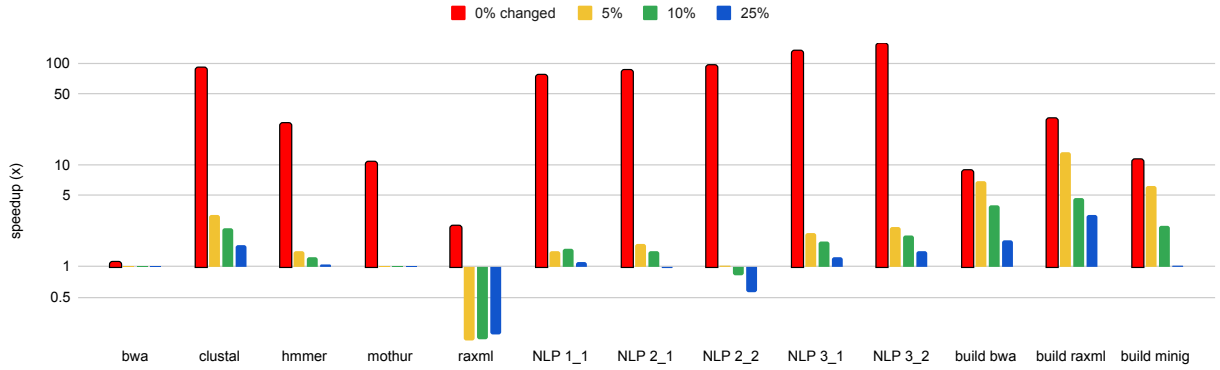

Figure 3.10: Speedups achieved when zero to 25% of the input files are changed when the checking mechanism is mtime. Note the log-scale y-axis.

input checking mechanism did not impact performance (subsection 3.5.2), and so we compare the mechanisms in the case of skipping to see if a similar pattern emerged.

Figure 3.9 shows the results of skipping our benchmarks with a varied number of input files changed, using hashing as the input checking mechanism. Figure 3.10 shows the results of the same experiment, but with mtime as the input checking mechanism. As Figure 3.9 shows, when inputs are unchanged, ProcessCache with hashing is able to provide dramatic speedups of up to 65× on NLP 3_2, and 17× on average across our benchmarks. However, as Figure 3.10 shows, mtime provides even more impressive performance in the case of unchanged inputs: up to 159× on NLP 3_2, and 26.3× on average across benchmarks.

With 5% of inputs changed, ProcessCache is not able to skip large, outer-level exec-units and must traverse the process tree, which dents the achievable speedups significantly but is still able to provide an average speedup of 2.05× when hashing is used. On the other hand, the average speedup increases to 3.64× when mtime is used instead. With 10% or 25% of inputs changed, ProcessCache with hashing sees far fewer opportunities to skip, incurring net slowdown (speedup below 1.0) for several workloads. For ProcessCache with mtime, even with 10% of inputs changed, almost all benchmarks still benefit substantially from significant speedup. Just two of the benchmarks (NLP 2_2 and raxml) incur net slowdown. Speedup drops off at 25% for ProcessCache with mtime, but as explained previously, we do not expect this to be the common case for users. A user can weigh the pros and cons of both input checking mechanisms. It may be worth some loss of soundness to enjoy the dramatic speedups mtime provides in the common case. Generally, the more skipping opportunities present themselves, the better ProcessCache performs. Since small changes to inputs are commonplace in practice, we expect ProcessCache to be useful in a range of real-world deployments.

While very little slowdown is incurred when using hashing over mtime for *caching*, we did not find this to be the case for *skipping*. There are a few reasons for this. In many ways, caching is easier to optimize. As explained in section 3.4, there are multiple opportunities to hash inputs in a smarter, faster way than hashing all input files on the critical path. ProcessCache hashes read-only files in the background, and utilizes an executable hash cache to reduce redundant hashing of shared executables.

For skipping, we cannot easily move the hashing of input files off the critical path. In the case when 0% of the inputs have changed, ProcessCache must check all the preconditions of the entire execution tree. All those preconditions are nested into the root execution, and so ProcessCache does not need to traverse the tree to get all the preconditions of the entire execution. As explained in subsection 3.3.7, the preconditions of the progeny are recursively folded into the parent's preconditions, up the execution tree, leading to the root execution. Also, this mass precondition check only happens once. This is not the case when *any* of the inputs have changed.

0% of inputs changing is the ideal case, but it is not the common case. As previously explained in this section, we have found that a common percentage of files changed in many code repositories is 3-5%. Even in the common case of 5% of inputs being changed ProcessCache must make the caching/skipping decision at *every* execve, and checking the preconditions has to be done at the time the process is stopped, and cannot be sent to a background task. The results in figures Figure 3.9 and Figure 3.10 show how quickly process stoppages for precondition checking past the root execution lessens the overall benefits of skipping with ProcessCache. Each time the preconditions are checked, input files must be hashed.

In addition, there are other processes running besides the one ProcessCache has stopped for precondition checking. These other processes can end up blocked from running because they have stopped on a system call event that ProcessCache must handle. None of these other processes can run until the preconditions are checked and a decision to cache or skip is made about the original process. As explained in subsection 3.4.1, one limitation of ptrace is that the tracer must be single-threaded. Therefore, this blocking is not avoidable. The mechanics of skipping make optimization difficult. Precondition checking is on the critical path, and cannot be removed from it. In a well-behaved program (one that does not perform race-prone parallel updates to resources), ProcessCache can serve outputs in parallel, speeding up skipping.

As Figure 3.10 shows, mtime does not suffer the same skipping overheads in most cases, especially the extremely important common case of 5% of inputs changed. This is because the cost of doing the system call to check a resource's mtime is *much* lower than the overhead of hashing the contents of a file with a cryptographic hash function like SHA256. The only outlier is raxml, which has a very short runtime and generates thousands of output files.

Generally, mtime is significantly faster to use as an input checking mechanism when skipping. For some benchmarks, even with only 5% of input files changed, the benefits of using ProcessCache are diminished when hashing is used as the input checking mechanism. Conversely, mtime improves most of the benchmarks significantly, and not only in the ideal case. For example, when using hashing, NLP 3_1 has a speedup of 26.29× with 0% inputs changed, and when using mtime, that

speedup jumps to *134.82*×. For NLP 3_2, the speedup increases from 28.43× to *159.06*×. In the ideal case, when 0% of inputs are changed, the average speedup of skipping increases from 17× to *56.53*×. In the most common case, when 5% of inputs are changed, the average speedup of skipping increases from 1.89× to *3.24*×.

mtime is not perfectly sound, but many users may prefer to trade perfect soundness for performance improvements in the most common cases. One potential route to improve skipping performance is to combine mtime and hashing. Hashing done during caching has already been amortized away through optimizations (section 3.4). We also know from our experiments that retrieving the mtime of inputs introduces very little overhead. During the caching phase, ProcessCache can record both the hash and the mtime for each accessed resource. During the skipping phase, ProcessCache can first check the cached mtime against the current mtime, and if they do not match, it can proceed by checking the hash. This may be a happy medium between soundness and performance; we explore this idea further in subsection 4.3.4.

To explore the performance of specific benchmarks further, we leverage the performance metrics presented in Table 3.4. A key factor in ProcessCache's performance is the granularity of exec-units. raxml's exec-units, on average, run for just 6.6 milliseconds which makes it hard for ProcessCache to amortize per-exec-unit costs like precondition checking. raxml also has the highest rate of system calls, imposing high recurring costs for ProcessCache as well. NLP 2_2 is similar, with the 2nd smallest exec-units and the 2nd highest rate of system calls. Smarter exec-unit batching would likely help for workloads like these, as well as more efficient system call interception strategies like LD_PRELOAD. bwa has favorable metrics but a short overall running time of 4.03 seconds, making it difficult to absorb ProcessCache's fixed costs.

Two of our benchmarks, bwa and mothur, do not have 5%, 10% or 25% bars in Figure 3.9 and Figure 3.10. bwa uses thread-level, instead of process-level, parallelism, and ProcessCache cannot skip at the thread level. mothur does use parallel processes but its input files do not readily admit incremental changes, though we plan to investigate this further.

Skipping extraneous Makefile dependencies

```
1 foo.bin: foo.c bar.c
2     gcc foo.c -o foo.bin
```

Figure 3.11: Makefile with extraneous dependence on bar.c

To evaluate ProcessCache's ability to compose with existing build systems, we constructed a small Makefile (shown above in Figure 3.11) where a binary is built from just one source file foo.c, but the Makefile includes an extraneous dependency on bar.c as well. Whenever bar.c is updated, make will re-build the binary since its declared (but not actual) dependencies have changed. However, simply by running make under ProcessCache, even though make still launches the compiler, when we reach its exec-unit it is skipped because ProcessCache can see that its true dependencies have not changed. This shows that ProcessCache can bring some of the benefits of forward build systems, like accurate dependency tracking, to a legacy build system with zero developer effort.

3.5.4. Correctness

To validate the correctness of ProcessCache, we check that the output produced by ProcessCache with skipping matches that of a native run where everything is re-run from scratch. All of the experiments presented here have been validated in this way, and ProcessCache has produced bit-identical output in each case. We also validated ProcessCache on dozens of additional shell scripts from the PaSh [116] benchmark suite, which we elided from our performance evaluation since they behave very similarly to other scripts we did include. While bugs surely remain in our implementation, we are confident that ProcessCache is currently robust enough to handle a useful range of workloads.

3.5.5. Additional Metrics

Table 3.4 shows additional performance metrics for our benchmarks. All data is collected from a ProcessCache run with an empty cache. Column 1 shows the size of the cache on-disk after the run completes. While we have not investigated cache eviction strategies in this work, simple policies like LRU are likely to work well while preventing excessive cache consumption. Column 2 shows the average duration of an exec-unit, in milliseconds; short exec-units are harder for ProcessCache to handle efficiently. Column 3 lists the average rate at which system calls intercepted by Process-

benchmark	1 cache (MiB)	2 millis/ ex-unit	3 system calls/sec	4 Syscall Events/sec	5 pre/post conditions
bwa	0.2	122.1	530	192 (2.8x)	200 / 32
clustal	22	118.7	329	152 (2.2x)	6,358 / 1,810
hmmer	161	33.9	1,060	353 (3.0x)	5,450 / 905
mothur	358	275.2	346	102 (3.4x)	854 / 300
raxml	33	6.6	21,070	13,084 (1.6x)	14,284 / 5,322
NLP 1_1	363	46.1	4,811	714 (6.7x)	5,344 / 1,061
NLP 2_1	350	45.6	5,795	842 (6.9x)	5,350 / 1,002
NLP 2_2	13	26.3	10,049	1,459 (6.9x)	5,353 / 1,061
NLP 3_1	365	59.2	4,509	666 (6.8x)	5,354 / 1,060
NLP 3_2	370	67.9	3,931	580 (6.8x)	5,300 / 1,055
bld bwa	5.6	109.1	6,605	311 (21.2x)	2,180 / 1,002
bld raxml	3.4	287.2	2,646	94 (28.1x)	1,422 / 306
bld minig	4.2	123.0	3,424	282 (12.1x)	1,938 / 1,100

Table 3.4: Key performance metrics

CACHE are performed, and Column 4 shows the rate at which SyscallEvents are recorded, with the number in parentheses showing the reduction when translating from system calls to SyscallEvents. This shows that SyscallEvents are an effective way to compress the system call trace. Finally, Column 5 shows the total number of pre- and postconditions recorded for each benchmark, which represents another integer-factor reduction (not shown) from the number of SyscallEvents.

Figure 3.12 shows where PROCESSCACHE spends its time when populating an empty cache and using hashing as the input checking mechanism (from top to bottom in each column): copying outputs, generating pre/postconditions, hashing input files, generating system call facts, and ptrace overhead. The benchmarks in Figure 3.12 are sorted from highest overall slowdown on the left (raxml) to lowest on the right (build raxml). The main bottleneck varies significantly across the slower benchmarks.

PROCESSCACHE uses SHA256 hashing by default to detect file changes, but we also experimented with checking the file's mtime. Figure 3.13 shows the slowdown incurred when using PROCESSCACHE's default SHA256 hashing to check for file changes instead of using mtime. Using mtime provides a performance boost of just 6% on average thanks to the performance optimizations we implemented for hashing (section 3.4). As explained in subsection 3.5.3, hashing does introduce more overhead when PROCESSCACHE isn't caching a brand new execution or skipping from the

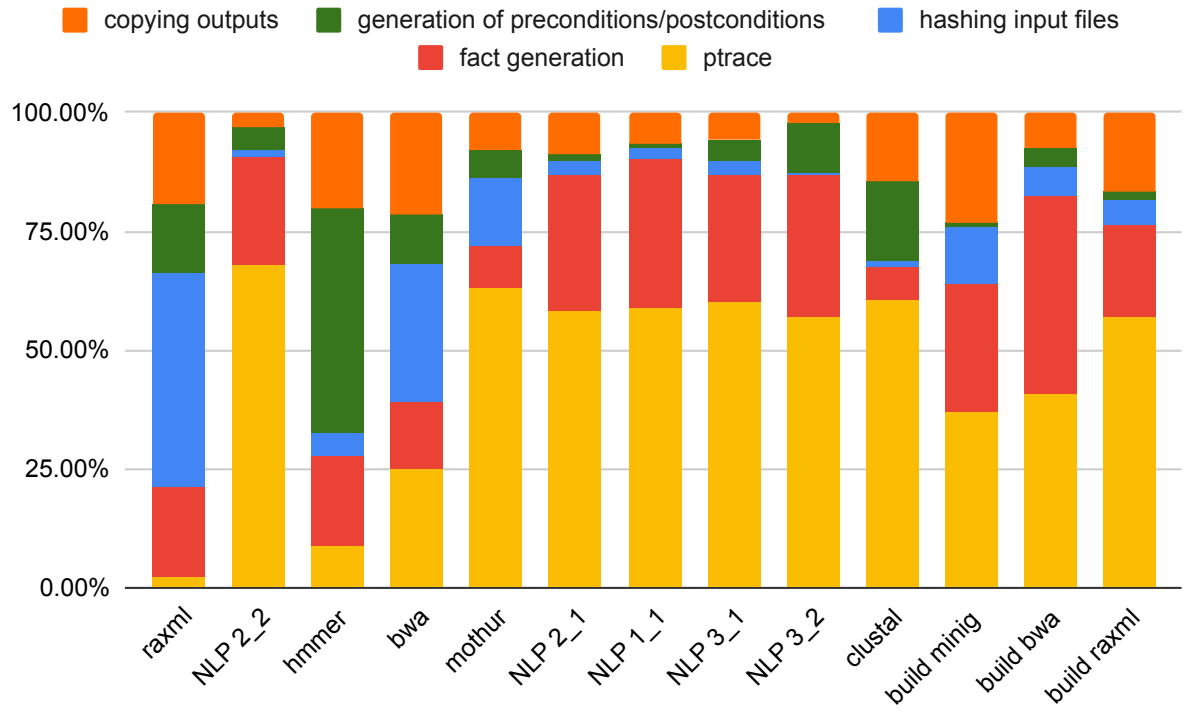

Figure 3.12: The major components of ProcessCache's caching overhead.

root execution. A combination of hashing and mtime could provide a compromise of soundness and performance. Alternatively, file system notifications (*e.g.*, Linux's inotify) could provide the speed of mtime with the soundness of hashing.

3.6. Related Work

Here we discuss the projects that are most closely related to ProcessCache. Shortcut [53] is a system that exploits computational reuse at very fine-grain: sequences of machine instructions within an execution. Through sophisticated kernel, compiler and dynamic binary instrumentation support, Shortcut is able to identify opportunities to "jump ahead" in an execution when a series of dynamic instructions have been seen before. While ProcessCache operates at the OS process abstraction instead, Shortcut's computational reuse is orthogonal to ProcessCache's and they could be profitably composed. ProcessCache could perform coarse-grained skipping of entire exec-units while Shortcut identifies additional skipping opportunities within processes that ProcessCache would otherwise re-run in their entirety.

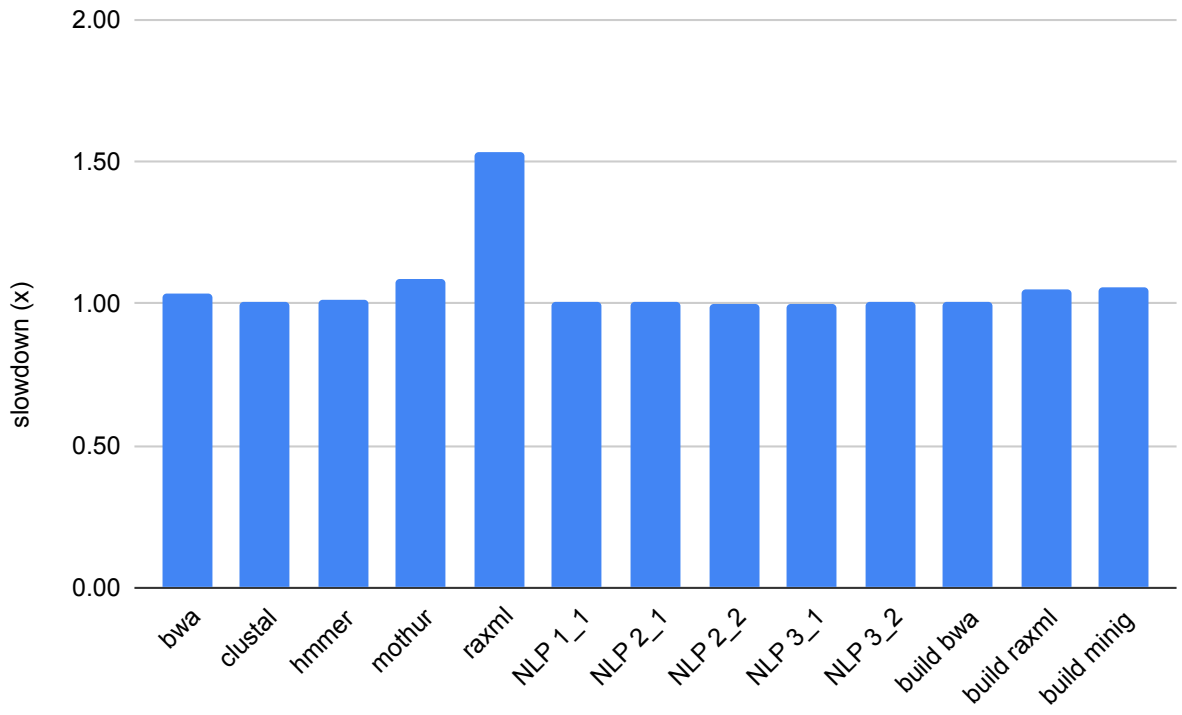

Figure 3.13: Slowdown when using hashing instead of mtime to check for file changes

We attempted a quantitative comparison against Shortcut, but were unable to build the custom Linux 3.5.0 kernel and glibc it requires. We were able to build the dynamic instrumentation tools (based on Intel Pin) which Shortcut uses for preprocessing to identify skipping opportunities, but they require the Shortcut kernel to run. While Shortcut's skipping capabilities are extensive, ProcessCache offers a much more deployable approach that runs in userspace on modern Linux machines (*e.g.*, we also ran ProcessCache experiments on Linux 4.18, a different major version, without issue).

Build Systems All software build systems, from the classic make to its many successors, support *incremental builds*, which avoid rebuilding parts of the software (*e.g.*, a subset of object files) that do not need to be rebuilt because their inputs have not changed. While many build systems cache results only from local builds, the build cache can also be shared across machines with systems like CCache [41]. This allows a team of developers to reap the rewards of caching as every build both populates the cache and is potentially accelerated by cache hits. Build systems in the vein of make

require developers to specify dependencies between source code files and build products (*e.g.*, an output executable). As a result, dependencies can be specified incorrectly. Spurious dependencies lead to unnecessary rebuilding, and missing dependencies can lead to incorrect builds where the output fails to reflect the source code. These challenges with conventional explicit-dependency build systems have motivated a newer generation of *forward build systems*.

Forward Build Systems Build systems like Memoize [35], Fabricate [58], Rattle [112] and Riker [48] do not rely on explicit, error-prone dependencies but instead ask developers to write a build script (often a shell script) that builds their software from scratch. The system traces the execution of the build script to identify dependencies between the steps of the build, allowing the build to, transparently, be run incrementally [48] or even in parallel [112]. Forward build systems, since they focus on accelerating build steps, which are typically collections of OS processes, have a similar focus to ProcessCache. However, ProcessCache is not restricted to software builds, requires no user effort, and employs a principled mechanism of preconditions and postconditions to allow memoization of a more general class of programs.

Record and Replay ProcessCache shares underlying technical underpinnings with system-level record and replay (RnR) systems like Mozilla rr [99, 100], Arnold [50] and others [24, 25, 39, 55, 57, 60]. For example, rr also uses Linux's ptrace mechanism to intercept system calls at runtime. However, RnR systems strive to provide detailed instruction-level replay of a recorded execution, to detect bugs or otherwise analyze software behavior. ProcessCache, on the other hand, only reproduces the visible side-effects of an exec-unit, which makes "recording" much cheaper. Nevertheless, techniques from high-performance RnR systems (like using in-process system call interception via Linux's LD_PRELOAD) could reduce ProcessCache's tracing overheads even further.

Determinism Enforcement Another area of related work that often uses system call tracing techniques is schemes for deterministic execution. DetFlow [111] provided a lightweight deterministic process group abstraction for fork-join parallel process workflows. DetTrace [94] provided a more robust and general-purpose deterministic container abstraction, but at a higher performance overhead.

Full deterministic operating systems have also been proposed [28, 33]. Currently, ProcessCache neither requires nor enforces determinism. However, even partial determinism enforcement might be useful to help increase the probability of cache hits. For example, some software builds inject the time of day into a source file [94] which can cause spurious build cache misses.

Incremental Computation and Memoization Re-executing only the parts of a computation that are affected by a given input change is the goal of many incremental computation systems. Language-level solutions in this area rely on *memoization*, which is the mechanism of storing the results of expensive function calls and returning the cached result when the same inputs appear again. These techniques are often utilized by functional programming languages, such as Haskell [8], which employs lazy evaluation to provide memoization. Lazy evaluation is a strategy which strives to delay the evaluation of an expression until it is explicitly needed and which also attempts to avoid repeated evaluations by storing evaluated results in a lookup table for future use.

Solutions outside of programming languages also exist. iThreads [34] is a threading library for parallel incremental computation that supports unmodified multi-threaded programs and can be used as a replacement for pthreads. ProcessCache, on the other hand, treats a multi-threaded program as one big execution, as its goal is to speed up *multi-process* programs, and while it can handle multi-threaded programs, it was not designed specifically for them. iThreads can only be used if a program already relies upon pthreads (unless the user is willing to rewrite their program), whereas ProcessCache caches programs regardless of the language they are written in.

Algorithms have been proposed to provide incremental computation such as differential dataflow [83, 92], which has been developed into a fully functional library called timely dataflow [16]. Essentially, timely dataflow is similar to a distributed data-parallel compute engine, which scales a program from one thread to distributed execution across clusters of computers. This work differs from ProcessCache in that it is a generalized algorithm and targets a different class of programs than ProcessCache. ProcessCache seeks to speed up programs on a single node, while timely dataflow strives to increase performance in distributed clusters of nodes by allowing them to perform iterative and incremental computations.

3.7. Conclusion

We have described the design and implementation of ProcessCache, a system for automatic memoization of Linux processes. ProcessCache automatically traces a program's execution, records portions of that execution to its cache, and skips ahead in the execution when it has a cached result for reuse. ProcessCache brings the benefits of incremental computation to process-parallel programs without requiring any software changes. In our experiments, we found that ProcessCache's performance overhead (with SHA256 hashing) is less than 2.47× in the worst-case (and 1.69× on average) when no skipping opportunities present themselves. We explored the trade-offs of different input checking mechanisms and found that their impact on performance varied greatly between pure caching, pure skipping, and a combination of caching and skipping. The worst-case overhead using mtime is less than 1.89× and is 1.34× on average. If there is redundancy to be exploited, we found that ProcessCache can accelerate workflows by anywhere from 1.06× with hashing (1.1× with mtime) to 65× (159× with mtime). In the next chapter (chapter 4), we explore multiple potential paths for future work for ProcessCache, including optimizations to improve skipping speedup, reduce space usage, and strengthen ProcessCache's correctness guarantees.

CHAPTER 4

FUTURE WORK

4.1. Introduction

After a detailed examination and analysis of our two systems, DetTrace and ProcessCache, we now explore future avenues of work. As ProcessCache was the larger of the two projects and comprises the bulk of this dissertation, we choose to focus the majority of the future work analysis on it. Before delving into the many future work possibilities for ProcessCache, we provide a high level analysis of one possible improvement for DetTrace that solves its biggest issue: sequentialization of thread execution.

For ProcessCache, there are three main areas for enhancement: performance, space utilization, and system correctness. As previously discussed in section 3.4, ProcessCache has already been optimized in many ways, so we choose to center its future work proposal on correctness. However, we first discuss potential ways to improve both performance and space. Even with the many ways ProcessCache has been optimized in order to speedup both caching and memoization, overhead still remains an issue, particularly for memoization (subsection 3.5.3).

4.2. DetTrace Future Work

DetTrace has several avenues for future work, but we elect to concentrate on improving how DetTrace handles threads. There are a number of issues with the current way that DetTrace proceeds when it encounters threads in the program running under it. First, DetTrace always sequentializes threads, and this does not always work. In fact, in some cases, it breaks the program running in the DetTrace container. For example, Java programs deadlock in a DetTrace container. This is because the Java virtual machine relies upon a *multi-threaded* garbage collector, and due to the nature of its implementation, the garbage collector deadlocks when DetTrace sequentializes its threads. Other languages may have multi-threaded garbage collector implementations that differ from Java's. Furthermore, there are certainly garbage collected languages such as Python which are single-threaded, and therefore will not suffer from the same issues.

It is very possible, however, that this issue and others like it affect multi-threaded programs outside of garbage collectors; this could be confirmed by expanding DetTrace's evaluation to a wider breadth of multi-threaded programs. Realistically, applications are only going to rely on multi-threading more as performance demands increase, which could result in issues similar to Java's deadlocking garbage collector manifesting themselves in other applications when running under DetTrace.

Moreover, even when sequentializing threads does indeed work, it introduces substantial performance overhead. Not only do the threads lose their parallel speedup, they also have the additional cost of ptrace starts and stoppages to contend with. Multi-threaded programs are just as common as multi-process ones, and therefore DetTrace *should* be able to handle them, keeping in line with its goal of program agnosticism.

We propose this approach to alleviate the issue: DetTrace could allow a user to opt for a less strict thread scheduler, via a command line argument. One can easily imagine a user of DetTrace whose programs simply do not need the absolute stringency of reproducible thread tids and execution order. For example, DetTrace could schedule threads in a similar fashion to how it schedules processes, letting them run in parallel outside of system calls. This solution also has the added benefit of being reasonably straightforward to add to the existing DetTrace prototype because it already contains a specialized parallel process scheduler that can be expanded to handle threads.

4.3. ProcessCache Future Work

4.3.1. Time Performance

In this section, we will explore two potential speedup optimizations for ProcessCache. First, we consider replacing ptrace with inotify, and the exact ramifications of doing so (subsection 4.3.2). Then, we discuss a smarter hashing design for ProcessCache, one that involves using a combination of SHA256 hashing *and* mtime to check inputs (subsection 4.3.4).

4.3.2. inotify versus ptrace

Because ProcessCache is built using the Linux ptrace API, there is unavoidable overhead from all the system call stoppages. To improve the *time* performance of ProcessCache, we could replace the ptrace mechanism with a different Linux mechanism, inotify. inotify is a Linux API for monitoring file events that provides "real time" updates for files in the registered *watch list*.

There are drawbacks to inotify, which is why we ultimately chose to use ptrace. inotify is significantly harder to work with, and requires the user to implement a *watcher daemon*, which then reports events to ProcessCache. This is not an insignificant programming challenge. Also, inotify comes with many caveats. We choose a select number of them to present here, but there are many more [68]. These were chosen because of their direct impact on important components of ProcessCache functionality.

1. inotify provides no information about which process triggered the inotify event. Even worse, there is no definite way for a process that is monitoring other events via inotify to tell the difference between events it triggered itself and events triggered by other processes.

2. Because inotify identifies events via watch descriptors, the user is responsible for maintaining a mapping between watch descriptors and pathnames, including when pathnames are changed. Directory renamings may affect multiple cached pathnames.

3. inotify identifies affected files by filename. By the time the monitoring process processes an inotify event, the filename may have already been deleted or otherwise changed. In the event of a filename changing (by deleting the file or renaming it), no additional notifying events are triggered by inotify. It is the responsibility of the user to update their cached mapping of watch descriptors and pathnames.

4. inotify does not monitor directories recursively. Subdirectories under a watched directory need their own watch descriptors. This can take a significant amount of time for large directory trees.

While the ptrace API can be described as "bare bones", its simplicity does not prevent the programmer from building powerful tools from the building blocks it provides. For example, with ptrace, we are able to stop the execution of a process before it deletes a file, to examine the number of hardlinks the file has. ptrace may be a "stop-the-world" approach, but at least we know that the resource in question could not have been further modified by the process because we have stopped the process's execution. With inotify, we would receive an inotify message informing us that the file has been deleted. But, we do not know *who* changed the file, or if any other events have happened to the resource in the interim before we received this file delete event. Even worse, there is no way to work around this. This is just one example of the pigeon-holing characteristics of the inotify API. inotify is simply too inflexible to be worth the potential performance savings.

4.3.3. Space Optimizations

ProcessCache is exactly that, a cache. But, the current prototype of the system does not have any particular traditional cache optimizations implemented. Currently, ProcessCache only evicts entries from the cache when they are invalidated. ProcessCache could be extended to employ "smarter" caching. We could utilize heuristics to maintain what are considered *hot* entries in the cache, and evict what are deemed to be *stale* (not necessarily invalidated, but least recently used).

ProcessCache could have flags to set the maximum physical size of the cache, the maximum number of entries the cache can hold, and even the maximum size a cache entry is permitted to occupy. Diverse workloads on different machines may benefit from the ability to tune their cache sizes and layouts to their unique needs. All of these metrics could be set and updated as ProcessCache is running. This could benefit users with increased space (and possible time) performance, in addition to the speedup they are already getting from default ProcessCache.

4.3.4. Smarter Hashing

There are two modes that ProcessCache seamlessly shifts between, caching and skipping, depending on the process it is currently tracing. ProcessCache will run in caching mode if this is a new execution and no cache hit is found, or if a cache hit is found but the inputs have changed. ProcessCache will run in skipping mode if this execution produced a cache hit, and the inputs

match.

Extensive effort has already been made to optimize ProcessCache to boost its performance, but this effort has concentrated principally on improving its caching mode overhead. Based on the results of our evaluation, we have found that hashing inputs is not a major source of overhead for ProcessCache running in caching mode, thanks to revamping the tracer to cleverly remove hashing of read-only inputs from the critical path, instead doing this in the background in parallel.

Lamentably, hashing read-only input files remains a pain point for ProcessCache's skipping mode. While skipping 90-100% of jobs (which is indeed the most common case) performs generally well, once the percentage of jobs to skip dips to 75%, many of the benchmarks suffer from worse performance than their baselines. This leads to a conundrum: we want to utilize hashing because it provides the soundness we desire. Unfortunately, hashing comes with inherent overhead that mtime simply does not have.

Why is it easier to optimize the hashing of read-only inputs in caching mode? In caching mode, we generate the hash of the input file for future use when we get a cache hit and need to know whether we should skip this execution. If the file is opened for reading only, we don't need to generate the hash *right this moment*, because the contents of the file are not going to change. Unless this file is later opened for writing, it is safe for us to "put off" hashing this read-only file. This allows us to move read-only input hashing off the critical path in caching mode, meaning that we can continue tracing the execution while we wait for the results of the hash.

We are unable to use this clever method in skipping mode. When we hash an input in skipping mode, the purpose is to check its precondition, to inform us whether this execution is safe to skip. This has to be done immediately because we have to decide, right now, are we skipping this process or caching it? This does not mean we have not optimized ProcessCache's skipping mode in any way. In fact, we have boosted skipping by implementing parallel output file serving. This enhancement is helpful, but serving outputs is not the biggest source of overhead for skipping. To see considerable speedup in skipping mode performance, we need to try to fix the issue of synchronous input file

hashing.

mtime may not be perfectly sound, but many users may prefer to trade perfect soundness for performance improvements in the most common cases. As we found in our experiments, mtime is almost always faster than hashing. We propose combining the two checking mechanisms in a way that capitilizes on their individual strengths and bypasses their weaknesses. During the caching phase, ProcessCache can record both the hash and the mtime for each accessed resource. During the skipping phase, ProcessCache can first check the cached mtime against the current mtime, and if they do not match, it can proceed by checking the hash. This may be a happy medium between soundness and performance. It allows us to generally benefit from the speed of mtime, and utilize hashing when we need to in order to ensure correctness. Obviously, only checking the hash when the mtime changes is not perfectly sound, but it is much better than only ever checking the mtime, and much faster than always checking the hash.

4.3.5. Correctness

In this section, we examine methods for enhancing ProcessCache's correctness guarantees. We define correctness as ProcessCache producing an accurate ordering of system calls that reflects the precise sequence of events on the accessed resources. This ordering then produces preconditions that uniquely identify this computation and can be used to verify whether it can be skipped, and postconditions that can be applied instead of rerunning the computation, to achieve better performance but with the guarantee that the outputs are equivalent to rerunning the computation.

First, we conduct an in-depth analysis of the assumptions the current prototype makes, and the dangers that can arise when these assumptions are broken (subsection 4.3.5). Then, we describe promising extensions that can be implemented for ProcessCache to strengthen its correctness guarantees (subsection 4.3.6).

Current Assumptions

ProcessCache operates by generating system call events for each unique execution in a program, and then computing preconditions and postconditions based on those events. These allow Pro-

cessCache to identify an execution in the cache and cache the correct results for the execution. As explained in subsection 3.3.7, ProcessCache can handle multi-process programs with multiple execve calls, and it does so by recursively inserting the child's system call events into those of its parent. When a parent process spawns a child process *c*, ProcessCache adds a ChildExec(*c*) event to each of its system call event lists. Upon exit of a child process *c*, ProcessCache iterates through the parent process's system call events, and replaces each ChildExec(*c*) with the child's system call events if any exist for the corresponding resource. Otherwise, the ChildExec(*c*) event is removed.

As explained in Section 3.4.2, one limitation of ProcessCache is that the event ordering it produces is a *total ordering*, which is then used to produce the factual preconditions and postconditions of a resource. This total ordering is constructed based on two assumptions:

1. Child processes of the same parent process will not modify the same file while running in parallel (child-child conflict).

2. A parent process will not modify the same file as a child process while the child is running (parent-child conflict).

If either of these assumptions is broken, a program can exhibit dangerously nondeterministic behavior. Some nondeterminism is perfectly fine. For example, imagine a program with many child processes where each performs some separate computation, the results of which are then written to individual output files. This program is nondeterministic in that the order in which the child processes will run cannot be predetermined. ProcessCache can capture a valid ordering of the events, however, because we know that no resources are being modified at the same time, the order in which the processes are scheduled does not really matter. This program follows our two requirements despite being nondeterministic. A program that exhibits nondeterminism by breaking either of the two assumptions is dangerous for the following reasons:

1. It is difficult for ProcessCache to recognize these behaviors in an arbitrary program. ProcessCache must infer what the program is doing solely through system call introspection.

2. The employment of these behaviors in programs can lead ProcessCache to **store incorrect cache results**, and/or **serve incorrect cache results** if they are not detected and handled.

Arguably, a parallel program in which the worker processes race to update the same resource is a prime example of what *not* to do when writing parallel programs. A program which exhibits this behavior will produce different results on each run. It is difficult to imagine a situation in which one wants a program that has multiple processes editing the same file with no guarantees about the ordering. More often than not, these behaviors exist in programs without the programmer's knowledge.

While scaling paradigms are evolving with the widespread usage of asynchronous programming (even in ProcessCache itself), many systems still exist today that are written to run in parallel using OS processes and bare-bones OS primitives for process management. The needs of the modern world require every computation to run as fast as possible, as soon as possible. This leads to the rapid production of very complex code, leaving very little time for design. These problematic programs exist in the real world, and because ProcessCache is built to be practical and help programmers, it must be robust to programs that exhibit these kinds of behaviors.

A natural question is what *exactly* goes wrong if ProcessCache does not recognize dangerous nondeterminism in a program it is caching. As explained in subsection 3.2.5, ProcessCache caches one valid result of the execution, causing it to be robust to many forms of nondeterminism. But, this result is only valid if ProcessCache has generated a *correct* total ordering of events.

Child-Child Conflicts

The first potentially unsafe behavior ProcessCache must detect is when sibling processes (sharing a *parent* process) modify the same resource in parallel. This kind of conflict is nuanced, and hard to detect.

ProcessCache operates by recording one run of an execution, and produces preconditions and postconditions based on that. This is fine because if we see the *same* executable invoked with *di erent* inputs, we know this is a different *line of execution* and we cache it as such. This relies on

```
1  void do_calculation(input_file) {
2    int fd = open(input_file, O_RDWR | O_APPEND);
3
4    int* buf[];
5    read(fd, buf);
6    if buf == NULL {
7      printf("No input provided");
8    } else {
9      int calculation = buf + 5;
10     printf("Result: %d\n", calculation);
11   }
12 }
```

Figure 4.1: Pseudocode for do_calculation() function.

us producing preconditions from a total sequence of system call events that reflects the true order in which the interactions with the accessed resources took place. If our total ordering does not reflect the actual sequence in which the resources were accessed, we will produce preconditions for some other line of execution, effectively tying this set of preconditions to the incorrect postconditions.

Table 4.1 shows an example program where two processes modify the same resource (foo.txt) in parallel, resulting in a generated system call ordering that does not reflect the actual sequence of accesses to the shared resource. Figure 4.2 provides the total order of system call events that ProcessCache will generate for the execution.

Child-child conflict		
P_1	C_1	C_2
fork() fork() wait()	sleep() open(foo.txt, O_TRUNC) exit()	do_calculation(input_file) exit()

Table 4.1: Example program in which two sibling processes modify the same file (foo.txt) in parallel.

Figure 4.2: Totally ordered system call events for foo.txt based on the program in Table 4.1. Note that we simplify

```
Parent: [ChildExec(1), ChildExec(2)]
        [Open(Trunc), Open(Append, Read)]
Child 1: [Open(Trunc)]
Child 2: [Open(Append, Read)]
```

Let us carefully examine the behavior of this program. First, P_1 forks C_1 and C_2, and then waits on its two child processes. C_1 first sleeps, then opens and truncates foo.txt, and finally exits. C_2

opens foo.txt for reading and writing, in append mode. It reads from the file, and uses the contents of the file to perform a calculation. Finally, it exits. While obvious to us, the race condition in this program was probably not obvious to the original programmer. They might have used O_APPEND and O_RDWR for C_2's open call because they are not fully aware of how the open call works, and because they ran their program and received the output they expected, they chose not to refactor it. They might have chosen to use the sleep() call because they thought it was more efficient for the thread whose only job is to clear a file, unwittingly making a choice that will probably cause C_1 to run after C_2.

From our bird's eye view of the program, it appears obvious that the desired behavior is indeed C_2 running before C_1. C_2 uses foo.txt as an input file and its contents as the parameter for the calculation it is performing, and even outputs a warning to the user if it has not been provided an input. C_1 only exists to clear the input file; it does not seem likely that its purpose is to clear the other process's input file before it is able to access it. One can imagine the programmer testing their program with a few different inputs that produced the correct outputs, and, based on this limited testing, assuming the program is working correctly.

Unfortunately, ProcessCache is unable to analyze a program to understand its behavior and context the way that a human can, which is why ProcessCache goes awry when trying to cache this. As Figure 4.1 shows, the do_calculation() function prints a warning to the user in the case where the input file contents are empty. We have already established that C_2 is probably scheduled before C_1 on most runs of this program. The problem is that ProcessCache assumes that if more than one child process writes to the same resource during execution, they perform the writes in the order they are spawned, *i.e.* C_1 will write to foo.txt before C_2.

Because C_1 opens foo.txt with the O_TRUNC flag, the file's contents are truncated upon successful opening of the file. The preconditions for foo.txt are dictated by the first successful modifying system call performed on it. Because ProcessCache orders C_1 first, its truncating open call is the first modifying system call. A successful open for truncation call contributes the file name and its existence as preconditions, meaning the full path must match and it must exist already when we

check the preconditions. But, because the file is truncating, the starting contents do not matter, so we do not check those as a precondition.

This means that if we see this execution again, we will only check that the file exists and that its file name matches what we have cached; we will not check the contents of the file. The trouble with this is that the program's behavior is dictated by the contents of the file. Most likely, in the run of the execution we cached, C_2 *does* access the file first, which means that it will indeed use the input of the file for its calculation.

Let's say that in the particular run that ProcessCache is tracing, foo.txt contained the number 5 at the start of execution, then the program will output 10. Remember, ProcessCache will assume C_1 runs before C_2, and so the generated system call sequence will match this output with the case when the file is truncated first. These preconditions are associated with an output that would not result from it. Additionally, for this particular program, ProcessCache's assumptions about process scheduling result in ProcessCache treating all runs of this execution the same, because it always assumes C_1 runs before C_2. Regardless of the contents of foo.txt, ProcessCache will skip as if it is the truncate-first case.

This is the type of behavior ProcessCache must detect so it can discontinue caching. We began with child-child conflicts because they are actually the simplest kind to identify; they can be recognized using existing ProcessCache faculties, as we will explain in subsection 4.3.6.

Parent-Child Conflicts and Beyond

The second assumption ProcessCache makes about the programs it caches is that, within those programs, parent processes will not update the same resources as their children in parallel. This is essentially assuming that the program follows the traditional fork/wait paradigm in which the parent process invokes the wait system immediately after forking its child processes. Table 4.2 shows an example program where the total ordering of events can be generated *incorrectly* in a program where a parent process and its child modify the same file in parallel. This leads to ProcessCache caching incorrect results. Figure 4.3 provides the total order that ProcessCache will generate

from the program.

Parent-child conflict	
P_1	C_1
fork() open(foo.txt, O_APPEND) wait()	sleep() open(foo.txt, O_TRUNC) exit()

Table 4.2: Example program in which the parent process and the child process write to the same file (foo.txt) in parallel.

Figure 4.3: Totally ordered system call events for foo.txt based on the program in Table 4.2.

```
Parent: [ChildExec, Open(Append)]
        [Open(Trunc), Open(Append)]
Child:  [Open(Trunc)]
```

Note that ProcessCache assumes that the parent process is not writing to the same file as the child process at the same time. First, P_1 forks C_1, resulting in a ChildExec event added to P_1's system call event list. Then, P_1 opens the file in append mode and writes to it, resulting in the Open(Append) event. Finally, it waits on C_1. C_1 sleeps, then opens the file in truncation mode and writes to it. Thus, the only event in C_1's system call event list for foo.txt is Open(Trunc). ProcessCache recursively adds C_1's file events to P_1's list, replacing ChildExec with Open(Trunc).

To ProcessCache, the file was truncated and written to (by C_1), and then appended to again later (by P_1). In reality, the most likely ordering is reversed, with P_1 appending to the file and then C_1 truncating it and writing to it, because C_1 process calls sleep() before it opens the file. The other order is just as possible, because P_1 does not invoke the wait system call immediately after forking C_1. ProcessCache does not examine the code, and cannot know that this is what is happening. These two orders of execution result in different contents for foo.txt at the end of execution. If C_1 runs first, which is what ProcessCache decides, the file contents include C_1's bytes and P_1's, in that order. If P_1 runs first, the file contents include just C_1's bytes, because P_1's are removed when C_1 truncates the file.

The issue here is that ProcessCache generates a set of preconditions and postconditions for the program that are only valid for one possible scheduling of the processes involved. According to

the event list generated by ProcessCache, the child's truncation and writing of the file happens before the parent appends to it. The truncation of the file as the first modifying event means that ProcessCache assigns only search permissions to the directory in which the file resides and write permissions to the file itself as the preconditions. But in reality, the true order of events is undecidable. If the order is indeed parent appending to the file first and then the child truncating the file and writing to it (which is the more likely ordering), then the file's starting contents should also be a fact in the preconditions set.

If ProcessCache checked only the permissions, and that the file exists with the same name, but the input file had different contents, it would serve cache outputs for the wrong computation. For example, imagine at the start of the caching run, the file contained the bytes "hello". Then on another round, the input file is still named foo.txt but the contents are now "42". Not only could ProcessCache serve cached outputs for the wrong execution, ProcessCache has produced incorrect preconditions *and* postconditions, and therefore is never correctly identifying the cached program. This example may seem innocuous, but, imagine if this program performed different calculations based on the contents of that input file. This would result in ProcessCache serving output files containing the wrong results. As explained in the last section, similar issues can arise when child processes modify the same file.

This parent-child conflict case can be generalized beyond just the direct parent-child process relationship. Realistically, any two processes at any two levels in the execution tree can modify the same resource in parallel, and ProcessCache needs to be able to identify conflicts that arise beyond sibling processes which share a level or processes which are 1 level apart (parent-child) in the process hierarchy. In subsection 4.3.6, we will first introduce a solution for recognizing parent-child conflicts that can be accomplished with a simple addition to the existing ProcessCache prototype, and then we will explain how this method can be tweaked slightly to detect general conflicts between processes across arbitrary levels of the process hierarchy. But first, let us consider an example of a general conflict we would want ProcessCache to be able to identify.

Consider the program in Table 4.3. Figure 4.4 provides the total order that ProcessCache will

generate from the program. First, G_1 (the grandparent process) forks P_1 (the parent process). Then it opens the file foo.txt for appending, and writes "grandparent" to the file. Finally, it waits on the process it spawned. P_1 simply forks its child process, C_1, waits on it, and then exits. C_1 opens foo.txt, truncating the file in the process, writes "child" to the file, and finally exits. It is impossible to know whether the grandchild process or grandparent process opened foo.txt first. If it was the grandchild, then the preconditions generated by ProcessCache are correct. But, it is just as possible that grandparent wrote to foo.txt first; we cannot be certain because the grandparent process does not immediately wait on the parent process after spawning it. Instead, it goes forward with editing a file, which happens to be the same file its progeny is modifying. The preconditions generated from Figure 4.4 for foo.txt will not include the file's starting contents as a precondition, because the first modifying event is truncation. But, if the grandparent indeed is the first process to write to foo.txt, then the file's starting contents should also be checked as a precondition.

Grandparent-grandchild conflict		
G_1	P_1	C_1
fork() open(foo.txt, O_APPEND) write("grandparent") wait()	fork() wait() exit()	open(foo.txt, O_TRUNC) write("child") exit()

Table 4.3: Example program in which a grandparent process and a grandchild process write to the same file (foo.txt) in parallel.

Figure 4.4: Totally ordered system call events for foo.txt based on the program in Table 4.3.

```
Grandparent: [ChildExec, Open(Append)]
             [Open(Trunc), Open(Append)]
Parent: [ChildExec]
        [Open(Trunc)]
Child: [Open(Trunc)]
```

4.3.6. Practical Determinism Validation for ProcessCache

Now that we have established the precise types of nondeterministic behavior ProcessCache must detect and handle, we can chart a path toward a solution. Enforcing determinism is expensive as we know from our previous work in deterministic containers [94]. Because ProcessCache is a performance *enhancing* system, this potential method for improved correctness is not viable. We

also cannot guarantee the user that we will *always* recognize all nondeterministic behavior in their programs. Therefore, a solution will have to balance retaining performance benefits and improving ProcessCache's correctness guarantees.

Instead of enforcing determinism, ProcessCache can be extended to identify the behaviors described in subsection 4.3.5, and proceed the same way as with other programs it specifically cannot cache (such as interactive programs): first, ProcessCache logs an error explaining the nature of the conflict to the user. Next, it detaches itself from the execution to allow it run on its own and not be cached. Finally, ProcessCache removes output files that were cached in the background that are associated with this execution. As explained in section 3.4, one optimization for ProcessCache is background output file caching. ProcessCache does not serialize the execution's cache entry into the cache until the end of execution. Therefore, if ProcessCache discovers one of these behaviors in the execution it is caching, it does not have update the serialized cache map, but it will need to remove any output files copied to the cache in the background before this behavior was recognized.

While this solution is not perfect, it is certainly preferred to ProcessCache silently ignoring the problem or raising a runtime error that exits or crashes the traced program, leaving the execution in some nonrecoverable state. This is, however, more complicated than handling other behaviors ProcessCache already detects such as networking, which only involves recognizing when networking system calls like `socket` are invoked.

Child-Child Conflict Detection

We begin by discussing child-child conflict detection, because these conflicts are the simplest to spot. This behavior can be detected using existing ProcessCache data structures and methods, with the addition of a simple flag. In order to identify child-child conflicts, we can add a flag as metadata to each resource's system call event list called `MODIFIED_BY_CHILD`, which is initially set to false.

We return to our child-child conflict example from Table 4.1 to demonstrate the use of this new flag.

If we examine the events ProcessCache produces in Figure 4.2, we see that P_1 will recursively fold the event lists of its children into its own. It first comes across ChildExec(1), and then looks up the child's events for foo.txt, and ultimately replaces ChildExec(1) with Open(Trunc). Because this is a system call event that *modifies* the resource, ProcessCache sets MODIFIED_BY_CHILD to true, and continues iterating over the list. The next and final element is ChildExec(2). ProcessCache looks up this child's events for foo.txt. Before replacing ChildExec(2), it checks the value of MODIFIED_BY_CHILD, which was set to true on the last iteration. ProcessCache then notifies the user of the conflict, and detaches from the program, discontinuing its caching of it.

General Conflict Detection

Similar to child-child conflicts, we can utilize the existing nesting of system call events in a program's execution hierarchy to enable ProcessCache to detect parent-child parallel modifications to resources. For this particular case, we also will implement a new system call event. This event is ChildExit, and it corresponds to when a parent process is notified that its child process exited successfully. ProcessCache already intercepts the wait system call, but now the system call will also spur ProcessCache to append a ChildExit event to each of the parent's resource event lists, in much the same way as when it appends ChildExec to each resource event list when a child process calls execve.

When ProcessCache iterates through the resource events to fold the child's events into the parent's by replacing its ChildExec events with the corresponding child's events for the resource, if any *modifying* system call events appear between a ChildExec and ChildExit, this means the parent process modified the resource while the child was running. ProcessCache can then examine the child's events for this resource, if any exist, and if any of them are also modifying, then this is a parent-child conflict on this resource. This solves the issue of identifying when parent and child processes update the same resource in parallel.

We can use our example program from Table 4.2 and the system call event lists ProcessCache generates for the program in Figure 4.3 to demonstrate this method. ProcessCache recursively folds C_1's events into P_1's event list for foo.txt. Figure 4.5 presents the generated resource event

lists for the program, using the new ChildExit event.

Figure 4.5: Resource event lists for foo.txt based on the program in Table 4.2. The parent's event list clearly shows a modifying system call event between the ChildExec and ChildExit.

```
Parent: [ChildExec, Open(Append), ChildExit]
Child: [Open(Trunc)]
```

As a reminder, first, the parent process forks the child process, then it opens foo.txt for appending, and finally it waits upon its child process. The child process calls sleep(), then opens (and truncates) foo.txt, and then finally exits. The parent's resource event list reveals that the process modifies the file while the child process is running, thanks to the context provided by the ChildExit event. The parent's Open(Append) event occurs directly between the ChildExec and ChildExit events in its system call event list for foo.txt. This would potentially be permissble, provided the child process did not actually modify the resource during its execution. The next step, however, reveals the conflict: ProcessCache examines the child's system call event list for foo.txt and discovers a modifying system call event: Open(Trunc).

When it is a parent process and a child process updating the same resource in parallel, detection is simple using ChildExec and ChildExit events, as we just explained. The beauty of this solution is how precise we can be: not only do we know when parent and child processes write to the same resource in parallel, we can also detect when a parent process and child process write to the same file, but it *is* properly ordered, and thus not dangerous nondeterministic behavior. We can apply this idea recursively to identify conflicts between processes across any number of levels in the hierarchy tree.

Let us return to our example program from Table 4.3 to demonstrate how we can detect general conflicts between processes. We again utilize the new ChildExit event, and Figure 4.6 shows the resulting events generated by ProcessCache. As a reminder, this program has a potential conflict because the grandparent modifies a file immediately after forking the parent instead of waiting on the parent. The parent forks a child process, which writes to the same file as the grandparent process.

As ProcessCache folds the events together from the child to the parent, it will find no issue

Figure 4.6: Totally ordered system call events for foo.txt based on the program in Table 4.3.

```
Grandparent: [ChildExec, Open(Append), ChildExit]
Parent: [ChildExec, ChildExit]
Child: [Open(Trunc)]
```

between the parent and child processes. Indeed, no conflict exists, as the only thing the parent process does is fork the child and immediately wait on it. When ProcessCache folds the parent's events into the grandparent's, however, the conflict becomes obvious. The parent's events are replaced with the child's event, Open(Trunc), and when the grandparent replaces its ChildExec event with the parent's, ProcessCache recognizes that some child process of this parent has altered the same file as the grandparent in parallel.

CHAPTER 5

CONCLUSION

This dissertation examines various techniques for providing reproducibility and performance optimizations for arbitrary, unmodified Linux programs. We establish that process manipulation through system call interposition is a powerful and general technique by demonstrating the variety of ways we harness the ptrace API to improve the correctness or performance of programs. A major goal of the works found in this dissertation is to *automatically* provide benefits to programs without requiring changes to the program's code, the OS, or the hardware the program is running on.

The introduction in chapter 1 provides an overview of the two systems explored in this work, DetTrace and ProcessCache. Then we examine process tracing and manipulation, and introduce the built-in Linux mechanism that allows for it: the ptrace API. We discuss the strengths and weaknesses of ptrace, and how its ostensibly primitive methods can be used to build capable process manipulation mechanisms, such as system call injection and modification. We formally define the goals of our two systems: for DetTrace to enforce reproducibility for all computation performed in the container, and for ProcessCache to provide generalized caching by assuming weak determinism is a characteristic of the programs it is caching. We conclude with an explanation of our asynchronous wrapper for the ptrace API and how it allows us to bypass both programmability and performance issues innate to ptrace.

In chapter 2, we presented DetTrace, a system that provides reproducibility for unmodified Linux programs. This system combines determinism enforcement with process manipulation via ptrace to guarantee that all computation that occurs inside a DetTrace container is a pure function of the initial file system state of the container. Reproducible containers can be used for a variety of purposes, including but not limited to: fault-tolerance, reproducible software builds, and reproducible data analytics. Our proudest result from our DetTrace evaluation is our demonstration of DetTrace achieving automatic reproducibility for 12,130 Debian package builds.

DetTrace motivated us to find other ways to utilize ptrace. We saw how we could use this

unassumingly simple API to construct powerful mechanisms, and we wondered if this could be used to provide performance instead of correctness. This is how we arrived at the beginning ideas for ProcessCache. In chapter 3, we delve into the ProcessCache project, a system for automatic caching of arbitrary Linux programs. ProcessCache traces executing programs to ascertain their inputs and cache their outputs. When the same program is run under ProcessCache and a cache hit is found, ProcessCache will serve its cached outputs instead of running the program, provided its inputs are unchanged.

With ProcessCache we developed our most sophisticated use of process tracing and manipulation, in the form of a custom Rust asynchronous runtime developed specifically to cater to ptrace's inherently asynchronous nature. This runtime improved our experience as developers using ptrace by simplifying the abstraction, and also lessened the overall cost of using ptrace. Our ProcessCache experiments confirmed this; we never found ptrace to be a major source of overhead for any ProcessCache benchmark. The runtime was inspired by the excessive slowdowns we suffered with DetTrace due to its sequentialization of system call regions of execution, on top of the expected overhead ptrace introduces. The current prototype of ProcessCache is quite robust, but as with all systems, there are potential areas of improvement and we have identified some of them in section 4.3.

This dissertation is unique in that it overviews just two systems, but each system is expansive and complex in its own right. Each system is also an excellent example of how a simple idea can turn into a very complex implementation, and this cannot always be foreseen in the initial project design cycles. There are many similarities and also many differences between the systems. In terms of goals, both systems strive to dynamically improve programs with no extra work on the programmer besides running their program under the system: DetTrace aims to provide reproducibility while ProcessCache aims to improve overall performance.

They both aim to provide their unique benefit without requiring changes to software, hardware, or the OS, and they both achieve this goal through the utilization of Linux's ptrace API. The actual use of ptrace varies between the systems, as we developed a smarter way to use ptrace in the

context of our caching system. In addition, for ProcessCache, ptrace was only used for tracing, and occasionally manipulating system calls. DetTrace's use case, on the other hand, involved so much more, such as the reproducible scheduler and manipulating many system calls to enforce determinism. This caused there to be inherently much more overhead in DetTrace from ptrace than in ProcessCache.

We have demonstrated the use of program tracing and its counterpart program manipulation for use in building practical systems. Through our anaylsis of DetTrace and ProcessCache, we established that the construction of powerful program agnostic systems can be accomplished using existing Linux mechanisms, such as ptrace, instead of needing to introduce new kernel or hardware components. We have shown that as simple as ptrace appears, it is in actuality very dynamic. We also realized that introducing new ideas can help navigate an existing bottleneck; we applied a new paradigm, asynchronous computing, to the somewhat archaic ptrace API, resulting in better performance and programmability. Developing these systems has shown us the potential of the tools within the existing Linux kernel, and how, when combined together with determination and a little creativity, the possibilities for building useful, powerful systems with them are endless.

BIBLIOGRAPHY

[1] Reproducible Builds. https://wiki.debian.org/ReproducibleBuilds.

[2] Windows Subsystem for Linux (WSL). https://docs.microsoft.com/en-us/windows/wsl/about.

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

[4] Packages in Bookworm/Amd64 Which Failed to Build Reproducibly. https://tests.reproducible-builds.org/debian/bookworm/amd64/index_FTBR.html, .

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

[6] GDB: The GNU Project Debugger. https://www.sourceware.org/gdb/.

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

[8] Haskell: an advanced fully functional programming language. https://www.haskell.org/.

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

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

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

[12]

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

[14] strace: Linux Syscall Tracer. https://strace.io/, .

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

[16] Timely dataflow. https://github.com/TimelyDataflow/timely-dataflow.

[17] Crate Tokio. https://docs.rs/tokio/latest/tokio/.

[18] VMware: VMware workstation zealot: Enhanced execution record / replay in workstation 6.5, April 2008. URL http://blogs.vmware.com/workstation/2008/04/enhanced-execut.html.

[19] 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.

[20] 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.

[21] Abadi, Daniel J. and Faleiro, Jose M. An Overview of Deterministic Database Systems. *Communications of the ACM*, 61(9):78–88, August 2018. ISSN 0001-0782. doi: 10.1145/3181853. URL http://doi.acm.org/10.1145/3181853.

[22] Umut Acar, Guy Blelloch, Matthias Blume, Robert Harper, and Kanat Tangwongsan. A library for self-adjusting computation. *Electron. Notes Theor. Comput. Sci.*, 148(2):127–154, mar 2006. ISSN 1571-0661. doi: 10.1016/j.entcs.2005.11.043. URL https://doi.org/10.1016/j.entcs.2005.11.043.

[23] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, aug 2013. ISSN 2150-8097. doi: 10.14778/2536222.2536229. URL https://doi.org/10.14778/2536222.2536229.

[24] Gautam Altekar and Ion Stoica. Odr: Output-deterministic replay for multicore debugging. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 193–206, 2009.

[25] antithesis. Antithesis - autonomous testing. https://antithesis.com.

[26] 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. ISBN 978-1-4503-0089-6. doi: 10.1145/1866835.1866854. URL http://doi.acm.org/10.1145/1866835.1866854.

[27] 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.

[28] 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*, OSDI'10, page 193–206, USA, 2010. USENIX Association.

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

[30] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos

Kozyrakis. Dune: Safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 335–348, Berkeley, CA, USA, 2012. USENIX Association. ISBN 978-1-931971-96-6. URL http://dl.acm.org/citation.cfm?id=2387880.2387913.

[31] 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.

[32] 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.

[33] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, page 177–191, USA, 2010. USENIX Association.

[34] Pramod Bhatotia, Pedro Fonseca, Umut A. Acar, Björn B. Brandenburg, and Rodrigo Rodrigues. Ithreads: A threading library for parallel incremental computation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 645–659, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450328357. doi: 10.1145/2694344.2694371. URL https://doi.org/10.1145/2694344.2694371.

[35] Bill McCloskey. Memoize. https://web.archive.org/web/20100905092103/http://www.eecs.berkeley.edu/~billm/memoize.html, June 2008.

[36] 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. doi: 10.1145/1640089.1640097. URL http://portal.acm.org/citation.cfm?doid=1640089.1640097.

[37] 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. ISBN 978-3-642-22654-0. URL http://dl.acm.org/citation.cfm?id=2032497.2032519.

[38] buck. Buck: A fast build tool. https://buck.build.

[39] Anton Burtsev, David Johnson, Mike Hibler, Eric Eide, and John Regehr. Abstractions for practical virtual machine replay. In *Proceedings of The12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '16, page 93–106, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450339476. doi:

10.1145/2892242.2892257. URL https://doi.org/10.1145/2892242.2892257.

[40] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of munin. *SIGOPS Oper. Syst. Rev.*, 25(5):152–164, September 1991. ISSN 0163-5980. doi: 10.1145/121133.121159. URL http://doi.acm.org/10.1145/121133.121159.

[41] ccache. Ccache — a fast C/C++ compiler cache. https://ccache.dev.

[42] 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. ISBN 978-1-59593-690-5. doi: 10.1145/1248648.1248652. URL http://doi.acm.org/10.1145/1248648.1248652.

[43] Badrish Chandramouli, Jonathan Goldstein, and David Maier. On-the-fly progress detection in iterative stream queries. *Proc. VLDB Endow.*, 2(1):241–252, aug 2009. ISSN 2150-8097. doi: 10.14778/1687627.1687655. URL https://doi.org/10.14778/1687627.1687655.

[44] Raymond Cheng, Ji Hong, Aapo Kyrola, Youshan Miao, Xuetian Weng, Ming Wu, Fan Yang, Lidong Zhou, Feng Zhao, and Enhong Chen. Kineograph: Taking the pulse of a fast-changing and connected world. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 85–98, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312233. doi: 10.1145/2168836.2168846. URL https://doi.org/10.1145/2168836.2168846.

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

[46] clustal. Clustal. http://www.clustal.org/download/2.1/.

[47] cmake. CMake. https://cmake.org.

[48] Charlie Curtsinger and Daniel W. Barowy. Riker: Always-Correct and fast incremental builds from simple specifications. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 885–898, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-29-70. URL https://www.usenix.org/conference/atc22/presentation/curtsinger.

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

[50] 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. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685090.

[51] 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. doi: 10.1145/1508244.1508255. URL http://portal.acm.org/citation.cfm?doid=1508244.1508255.

[52] 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.

[53] 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. ISBN 9781450368735. doi: 10.1145/3341301.3359659. URL https://doi.org/10.1145/3341301.3359659.

[54] 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. ISSN 0163-5980. doi: 10.1145/844128.844148. URL http://doi.acm.org/10.1145/844128.844148.

[55] 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, dec 2003. ISSN 0163-5980. doi: 10.1145/844128.844148. URL https://doi.org/10.1145/844128.844148.

[56] 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. ISBN 978-1-59593-796-4. doi: 10.1145/1346256.1346273. URL http://doi.acm.org/10.1145/1346256.1346273.

[57] 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*, pages 121–130, 2008.

[58] fabricate. fabricate. https://github.com/brushtechnology/fabricate.

[59] 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.

[60] 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 *OSDI*, volume 8, pages 193–208, 2008.

[61] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: Practical accountability for distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(6):175–188, October 2007. ISSN 0163-5980. doi: 10.1145/1323293.1294279. URL http://doi.acm.org/10.1145/1323293.1294279.

[62] Matthew A. Hammer, Umut A. Acar, and Yan Chen. Ceal: A c-based language for self-adjusting computation. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 25–37, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605583921. doi: 10.1145/1542476.1542480. URL https://doi.org/10.1145/1542476.1542480.

[63] Matthew A. Hammer, Khoo Yit Phang, Michael Hicks, and Jeffrey S. Foster. Adapton: Composable, demand-driven incremental computation. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 156–166, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450327848. doi: 10.1145/2594291.2594324. URL https://doi.org/10.1145/2594291.2594324.

[64] 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. ISSN 0362-1340. doi: 10.1145/2858965.2814305. URL https://doi.org/10.1145/2858965.2814305.

[65] hmmer. HMMER. http://eddylab.org/software/hmmer3/3.1b2/hmmer-3.1b2-linux-intel-x86_64.tar.gz.

[66] 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.

[67] 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. doi: 10.1145/2451116.2451170.

[68] inotify. inotify. https://man7.org/linux/man-pages/man7/inotify.7.html.

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

[70] 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.

[71] 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.

[72] Michael Kerrisk. *The Linux Programming Interface*. 2010.

[73] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. 2022.

[74] 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.

[75] 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. doi: 10.1145/1736020.1736031. URL http://portal.acm.org/citation.cfm?doid=1736020.1736031.

[76] 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.

[77] 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. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043587. URL http://doi.acm.org/10.1145/2043556.2043587.

[78] 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. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043587. URL http://doi.acm.org/10.1145/2043556.2043587.

[79] 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. ISBN 978-3-642-24099-7. URL http://dl.acm.org/citation.cfm?id=2075029.2075086.

[80] 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.

[81] 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.

[82] 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. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037751. URL http://doi.acm.org/10.1145/3037697.3037751.

[83] Frank McSherry, Derek G. Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research*, CIDR'13, 2012.

[84] 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. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465365. URL http://doi.acm.org/10.1145/2465351.2465365.

[85] 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. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741960. URL http://doi.acm.org/10.1145/2741948.2741960.

[86] 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. ISBN 978-1-4503-6240-5. doi: 10.1145/3297858.3304047. URL http://doi.acm.org/10.1145/3297858.3304047.

[87] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, page 1–20, New York, NY, USA, 2009. Association for Computing Machinery. ISBN 9781605587660. doi: 10.1145/1640089.1640091. URL https://doi.org/10.1145/1640089.1640091.

[88] minigraph. Minigraph. https://github.com/lh3/minigraph/releases/tag/v0.20.

[89] Christopher Mitchell, Russell Power, and Jinyang Li. Oolong: Asynchronous distributed applications made easy. In *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450316699. doi: 10.1145/2349896.2349907. URL https://doi.org/10.1145/2349896.2349907.

[90] Neil Mitchell. Shake before building: Replacing make with haskell. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming*, ICFP '12, page 55–66, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450310543. doi: 10.1145/2364527.2364538. URL https://doi.org/10.1145/2364527.2364538.

[91] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. Ciel: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 113–126, USA, 2011. USENIX Association.

[92] 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. ISBN 9781450323888. doi: 10.1145/2517349.2522738. URL https://doi.org/10.1145/2517349.2522738.

[93] National Academies of Sciences, Engineering, and Medicine. *Reproducibility and Replicability in Science*. The National Academies Press, Washington, DC, 2019. ISBN 978-0-309-48613-2. doi: 10.17226/25303. URL https://www.nap.edu/catalog/25303/reproducibility-and-replicability-in-science.

[94] 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. ISBN 9781450371025. doi: 10.1145/3373376.3378519. URL https://doi.org/10.1145/3373376.3378519.

[95] 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. ISBN 978-1-4503-4092-2. doi: 10.1145/2851141.2851142. URL http://doi.acm.org/10.1145/2851141.2851142.

[96] ninja. Ninja: A small build system with a focus on speed. https://ninja-build.org.

[97] 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. ISBN 978-1-931971-38-6. URL https://www.usenix.org/conference/atc17/technical-sessions/presentation/ocallahan.

[98] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability: Extended technical report, 2017.

[99] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, page 377–389, USA, 2017. USENIX Association. ISBN 9781931971386.

[100] Robert O'Callahan, Chris Jones, Nathan Froyd, Kyle Huey, Albert Noll, and Nimrod Partush. Engineering record and replay for deployability: Extended technical report, 2017. URL https:

//arxiv.org/abs/1705.05937.

[101] 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. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508256. URL http://doi.acm.org/10.1145/1508244.1508256.

[102] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361073. URL http://doi.acm.org/10.1145/361011.361073.

[103] W. Pugh and T. Teitelbaum. Incremental computation via function caching. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, page 315–328, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897912942. doi: 10.1145/75277.75305. URL https://doi.org/10.1145/75277.75305.

[104] raxml. RAxML. https://github.com/stamatak/standard-RAxML/releases/tag/v8.2.12.

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

[106] 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. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180224. URL http://doi.acm.org/10.1145/3180155.3180224.

[107] 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. ISSN 07342071. doi: 10.1145/312203.312214. URL http://portal.acm.org/citation.cfm?doid=312203.312214.

[108] 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.

[109] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990. ISSN 0360-0300. doi: 10.1145/98163.98167. URL http://doi.acm.org/10.1145/98163.98167.

[110] Ryan G. Scott, Omar S. Navarro Leija, Joseph Devietti, and Ryan R. Newton. Monadic composition for deterministic, parallel batch processing. *Proc. ACM Program. Lang.*, 1 (OOPSLA):73:1–73:26, October 2017. ISSN 2475-1421. doi: 10.1145/3133897. URL http://doi.acm.org/10.1145/3133897.

[111] Ryan G. Scott, Omar S. Navarro Leija, Joseph Devietti, and Ryan R. Newton. Monadic com-

position for deterministic, parallel batch processing. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017. doi: 10.1145/3133897. URL https://doi.org/10.1145/3133897.

[112] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Build scripts with perfect dependencies. *Proc. ACM Program. Lang.*, 4(OOPSLA), nov 2020. doi: 10.1145/3428237. URL https://doi.org/10.1145/3428237.

[113] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. Forward build systems, formally. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2022, page 130–142, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391825. doi: 10.1145/3497775.3503687. URL https://doi.org/10.1145/3497775.3503687.

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

[115] 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. ISSN 0018-9340. doi: 10.1109/TC.1987.1676929. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1676929.

[116] Nikos Vasilakis, Konstantinos Kallas, Konstantinos Mamouras, Achilles Benetopoulos, and Lazar Cvetković. Pash: Light-touch data-parallel shell processing. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 49–66, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383349. doi: 10.1145/3447786.3456228. URL https://doi.org/10.1145/3447786.3456228.

[117] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 423–438, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323888. doi: 10.1145/2517349.2522737. URL https://doi.org/10.1145/2517349.2522737.